**Middlesex University**

Final Year Project   
Report

**A Comprehensive Analysis of Traffic Load Balancers in Software Defined Networking**

­­­­­­­

*Author Supervisor*

*Salon Ghalan Tamang Clifford Sule*

April 28, 2024

**Table of Contents**

**1 Declaration 2**

**2 Acknowledgement 3**

**3 Abstract 4**

**4 Introduction 7**

4.1 Problem Definition 7

4.2 Report Structure 7

4.3 Aims 8

4.4 Objectives 8

4.5 Deliverables 9

**5 Literature Review 9**

5.1 Introduction 9

5.2 Relevant Literature 10

5.3 Conclusion 14

**6 Requirements specification 14**

6.1 Functional Requirements 14

6.2 Non-functional Requirements 15

**7 Design and Implementation 15**

7.1 Design Overview 16

7.2 Network Topology 16

7.3 Technological Stack 19

7.4 Load Balancing Algorithms 21

7.4.1 Round Robin 21

7.4.2 Weighted Round Robin 23

7.4.3 Least Connection 25

7.4.4 Dynamic Threshold based Load Balancing 27

7.4.5 Neural Network based Load Balancing 29

7.5 Implementation Details 32

**8 Discussion and Evaluation 34**

8.1 Performance Analysis and Evaluation 34

8.2 Challenges and Limitations 43

8.3 Future Recommendations 45

**9 Conclusion 46**

**10 References 47**

1. **Declaration**

**Declaration of Authenticity**

I, Salon Ghalan Tamang, hereby declare that the work presented in this report and all other associated materials is wholly my work. The information derived from article papers has been referenced and appropriately accredited and a list of references is provided. No part of this dissertation was previously presented for another degree or diploma at this or any other institution.

Signature Date: April 28, 2024

Salon G. Tamang

1. **Acknowledgement**

I would like to acknowledge and give thanks to my supervisor and networking tutor Prof. Clifford Sule for allowing me to work on this project. His guidance, support and directions were crucial throughout every stage of this project. I’ve gained valuable insights on load balancing algorithms in software defined networking and how they perform in real life situations. I would also like to thank my friends and family for their support and help during this period.

**3 Abstract**

This study is a comprehensive effort to assess the effectiveness of various traffic load balancing algorithms in the dynamic landscape of software-defined networking (SDN) systems. Given the exponential expansion of network traffic and the rising complexity of modern networks, effective load balancing has emerged as a vital aspect in optimizing resource utilization and improving overall network performance. Using Mininet and the Opendaylight controller, this study carefully investigates a variety of load balancing algorithms, analyzing their performance, scalability, and applicability in tackling contemporary network difficulties. This study attempts to discover the benefits and limitations of each load balancing strategy through a rigorous comparison analysis, providing significant insights into their practical implications and applicability for a variety of network scenarios. Furthermore, by providing actionable recommendations, this study aims to inform network design and management practices, allowing stakeholders to make informed decisions to improve the resilience and efficiency of their network infrastructure. Finally, the study's findings are expected to add significantly to a better knowledge of load balancing mechanisms in SDN, providing practical assistance for both network practitioners and regulators.

1. **Introduction**

Software-defined networking (SDN) has emerged as a transformative approach to network management, offering unparalleled flexibility and programmability in contrast to traditional networking methods. However, with the increasing dynamism and complexity of networks, efficient traffic load balancing is becoming indispensable to ensure optimal resource utilization and performance. This introduction outlines the problem statement, provides an overview of the report's structure, articulates the overarching aims and objectives, and identifies the expected deliverables of the project.

* 1. **Problem Definition**

In today's networking world, the constant surge in network traffic and diverse application needs presents a tough challenge for network admins. Traditional networking setups struggle to keep up with these changing demands, often leading to poor resource management, congestion, and slower performance. SDN steps in as a solution by separating the control plane from the data plane, making management centralized and more flexible. But, even with SDN, efficiently balancing traffic within networks remains a big issue. The main challenge is to create load balancing methods that can adjust traffic flow across the network in real-time, ensuring everything runs smoothly and can handle any changes or growth.

* 1. **Report Structure**

The report is split into five sections. The first section introduces the problem statement, outlines the objectives, and identifies the anticipated deliverables.

The second section is the literature review section. Here, I have examined existing research and discoveries on traffic load balancing in software-defined networking (SDN), including an introduction, relevant literature review, and concluding remarks that summarize key insights.

The third section outlines the functional and non-functional requirements of the system or solution being developed.

The design and implementation segment are the fourth section that provides an overview of the design approach, detailing the network topology, listing the technological stack used, discussing various load balancing algorithms investigated, providing insights into the implementation process, and outlining the testing and validation procedures.

The last section of the report consists of challenges and limitations of the project, performance comparison and future recommendations regarding load balancing algorithms.

* 1. **Aims**

The primary aim of this research is to thoroughly assess and compare the different traffic load balancing methods' performance in SDN environments. To thoroughly evaluate the effectiveness, scalability, and application of different load balancing algorithms in tackling modern network difficulties, carefully selected experiments and simulations utilizing Mininet and the Opendaylight controller have been carried out.

* 1. **Objectives**

To accomplish the overall goal, the project's distinct objectives are as follows:

* Implementing various traffic load balancing algorithms into practice in an SDN setting.
* Evaluating each load balancing algorithm's performance in terms of resource usage, network latency, and throughput through experiments and simulations.
* Analyzing the performance of various load balancing strategies using pre-established metrics.
* Provide analysis and suggestions for improving resource usage and network performance in SDN systems.
  1. **Deliverables**

The following are the project's deliverables:

* Comprehensive documentation of the experimental setup and load balancing strategies used.
* Reports from comparative analyses that show how well each load balancing method performs.
* Recommendations based on the project's findings for researchers and network operators.

1. **Literature Review**

The literature review section provides an overview of existing research and findings relevant to traffic load balancing in software-defined networking (SDN).

**5.1 Introduction**

Software-defined network (SDN) is an emerging technology in computer networks. By partitioning the current network into a centralized control plane (CP) and a remotely programmable data plane (DP), SDN streamlines the design, control, and management of next-generation networks, such as 5G, cloud computing, and big data. A SDN southbound interface (SBI) links the DP and the CP (Lamiae Boukraa et al., 2022). Previously, in a traditional network, DP (the actual forwarding elements) and CP (the logic underlying the forwarding functionality) are packaged into a single box and connected tightly. The overall design remains complex and expensive due to its intricate and tight integration. In contrast to this, SDNs are flexible, dynamic, cost-effective, and customizable, making them ideal for the dynamic nature of today's applications (Rukmini Bhat B et al., 2021). However, the ever-growing avalanche of network traffic has offered many challenges to SDNs in terms of offering reliable connectivity, quality of service and scalability. Hence, load balancing (LB) has a crucial role in SDN. In this literature, I will be reviewing publications on several LB techniques that allow for dynamic traffic management and efficient utilization of network resources in SDN.

**5.2 Relevant Literature**

The rapid growth of the internet with advancements in communication technologies has flooded the networks all over the world. This increasing demand coupled with high-speed data transmission requirements needs LB and proper management of server resources. (Fakhrun Jamal & Tamanna Siddiqui, 2021) in their paper defines LB as the practice of reallocating load to various nodes of common infrastructure to develop resource proficiency and increase the job's answer period whereas similarly eliminating a condition in which some nodes are overloaded, and others are underloaded. In the journal, they have classified LB mechanisms into three categories namely process initiation based, system state-based and spatial distribution of node-based. They discovered that static LB is less efficient than dynamic LB, but dynamic LB is less productive than the hybrid technique, which incorporates QoS and performance requirements (Response time, Reliability, Resource utilization, fault tolerance, and Scalability).

A table of text with black and white text

Description automatically generated

Figure 1: comparison table of LB techniques from Fakhrun Jamal & Tamanna Siddiqui, 2021.

Another study by (Fancy et al., 2019) compared traditional networks with SDN networks. The SDN networks have shown great advantages in many aspects, but if the load distribution is uneven in the SDN networks, it will greatly affect the performance of the network. The three load balancing algorithms considered for analysis purpose are least connection, Round Robin and Weighted Round Robin algorithm. Based on throughput analysis, it showed that round robin algorithm could only provide the least number of transactions. Also, it is evident that the least connection algorithm (a dynamic LB algorithm) provides the highest throughput when compared to the Weighted Round Robin and Round Robin algorithms (static LBs). Furthermore, a recent study by (Rukmini Bhat B et al., 2021) shows how a derived method using least connection algorithm with Dijkstra’s algorithm could be used in a single SDN controller to reduce round trip time (RTT) in overall network. The purposed solution claims that the average standard deviation for sending 200 packets 10 number of times reduced from 0.9449ms to 0.429 after applying the algorithm. However, this solution failed to consider traffic volume in real time.

A solution considering traffic volume and distributing traffic equally to avoid data congestion and link overloading has been proposed by (M.C. Nkosi, A.A. Lysko, S. Dlamini, 2018). The proposed system uses multipath-based LB in a single centralized Opendaylight controller’s data plane. This LB algorithm takes source and destination as input and calculates alternative short paths. Moving on, the paths are pushed down to the flow table. The algorithm then computes path costs for all defined paths using network statistics and load balancing is repeated until cost for all paths are equal. As a result, the load balancer showed improved network performance in transfer rate and response time. However, it was found that for better network improvement, the data plane should have multiple alternative links so that multiple paths can be defined for a routing path. Conventional networks use centralized controllers to coordinate traffic distribution. In a study by (P. Dharam and M. Dey, 2021) mentions that a single controller can no longer support the application requirements of SDN networks due to the growth of network applications and continuing scale expansion. Additionally, a single controller is vulnerable to failure issues and overload, which is detrimental to the scalability and reliability of networks. Hence, large SDN networks with numerous controllers in the control plane have emerged. Each controller oversees a group of switches, and the SDN network is managed by a control plane that is both physically and logically distributed.

(Kai-Yu Wang, Shang-Juh Kao and Ming-Tsung Kao, 2018) in their paper proposes a solution to challenges of using a single dedicated controller. They implemented a distributed approach with multiple controllers to avoid a single point of failure. Their proposed system focuses on balancing load across multiple controllers and shifts packet flows to a controller with a lighter load. In the proposed system, three logical components were included in each controller, namely a load collector, which periodically activates to collect loading statuses from other collectors and share its loading information, a load balancer which records its identification, loading information, associated switches and most recent update time and a switch migrator, which notifies its associated switch to redirect the forwarding path to the target controller once load of a controller exceeds the threshold. Moreover, to avoid simultaneous migration assignment to same controller, the balancer module processes information of one controller at a time and the controller’s sequent actions were delayed for 5 seconds when an overloaded controller registered another overloaded controller on its load record. The outcome of this proposed system confirmed that the standby controller solved the reliability problem. But delaying controller actions for a brief period involves a tradeoff of bandwidth conscription. In addition, the controller has predefined static threshold which could not align parallel with network changes and, the controller assignment does not consider the dynamic loads of network.

(Songzhou Li et al, 2023) outlines the problems associated with fixed threshold setting and introduces controller LB algorithm based on dynamic threshold (CLBDT), which has LB and threshold adjustment functions. The experiment is simulated in Mininet using Floodlight controller and Cbench were used to simulate flow requests from the switch to send Packet-In messages to controller. From the simulation results, they found that the load standard deviation of the fixed threshold load balancing algorithm is calculated to be 0.098 and the load standard deviation of the CLBDT algorithm is 0.010. Hence, they showed that CLBDT algorithm has better load balancing effect than fixed threshold LB algorithm.

Another research paper by (Zhihao Shang et al, 2019) illustrates the dynamic controller assignment problem as NP-complete problem. In their proposed solution, they designed a heuristic for solving the controller assignment problem named LANS (Late Acceptance Neighbor Search). It accepts the current controller assignment matrix and provides a new controller assignment matrix that can minimize the weighted sum of the flow setup time and switch migration time. They used a greedy algorithm that generates a feasible controller assignment matrix which is later fed to LANS. This algorithm sorts the controllers by their utilizations and tries to balance load in the controller with least migrations. They compared their solution with dynamic controller assignment algorithm DCP-SA in terms of flow setup time, migrations, and CV (coefficient of variation). They measured the flow setup time of a controller and modeled each controller as an M/PH/1 queue to capture its performance. The queueing model is used in the heuristic for the fitness function. The results showed that their solution can balance the controller better, reduce the flow setup time and make less migrations less than DCP-SA.

Controller failure is a critical challenge in distributed SDN. A research paper by (Poonam Dharam & Mithila Dey, 2021) implements two dynamic solutions namely Random Weight Load Balancing and Progressive Assignment Load Balancing discarding proactively assigned pre-partition which do not consider current network state. Their solution setup consists of multiple controllers and a LB. The LB keeps track of global topology network and has a monitoring module that periodically sends heartbeat messages to all connected controllers. Results from their simulation showed that proposed solutions successfully reassigned all the orphan switches to other active controllers such that the load of controllers after assignment is close to each other. Nevertheless, it is important to note that the LB is a single entity that handles the critical task of assigning orphan switches to available controllers. Hence, failure of such centralized component would freeze the network in turn affecting QoS.

Finding the shortest path between source and destination is detrimental in maintaining performance of distributed SDN. In a recent paper by (Dmitry P. et al, 2023) proposes an intelligent multipath routing method based on artificial neutral network that allows the controller to configure data transmission policies quickly and efficiently. Hyperparameters when designing the neural network model are optimized by artificial bee colony algorithm. The accuracy of the model predicting the shortest paths is about 90%. However, the authors fail to show integration of this solution in SDN controller with simulation making real-time routing decisions.

**5.3 Conclusion**

The purpose of this review was to analyze recent research papers on traffic load balancing in SDN. Most research papers gathered show that dynamic distributed approaches have superior performance over static load balancing algorithms. Based on various studies considered, multiple techniques still lack consideration of various performance metrics. Nevertheless, more exploration of hybrid LB balancing techniques should be encouraged. Furthermore, as technology is advancing and networks are congesting on daily basis, I believe that extensive research on load balancing schemes integrating artificial intelligence is also required to refine reliability, scalability, and efficiency of SDN in the future.

1. **Requirements Specifications**

This section describes the functional and non-functional criteria for a traffic load balancing system in software-defined networking (SDN). The requirements are divided into functional and non-functional categories to provide an in-depth overview of the system's specifications. As I am going for distributed architecture of load balancing system, following are the requirements for the system:

* 1. **Functional Requirements**

Decentralized Control: For load balancing decisions, the system should use a decentralized control mechanism that distributes decision-making processes over numerous controllers or nodes.

Consistency and Coordination: To prevent conflicts and maintain coherent behavior across the network, the system must ensure consistency and coordination across distributed components.

Fault Tolerance and Resilience: The system should be fault tolerant, smoothly accepting node failures, network partitions, and other failure scenarios without jeopardizing system availability or performance.

Interoperability and Compatibility: The system should be able to communicate with a variety of SDN controllers, switches, and network devices, allowing for smooth integration and compatibility among distributed components.

* 1. **Non-functional Requirements**

Performance and Latency: The system should maintain low latency and high throughput across distributed components while reducing communication overhead and processing delays.

Reliability and Durability: The system must ensure that it operates reliably, and that data is durable, by retaining key state information and maintaining data consistency across distributed nodes.

Scalability and Load balancing: The system's load balancing techniques should be efficient enough to distribute traffic and workload evenly across distributed nodes, ensuring optimal resource utilization and performance.

Monitoring and Management: The system should provide complete monitoring and management capabilities that allow administrators to monitor system health, diagnose problems, and perform maintenance tasks across distributed components.

1. **Design and Implementation**

This section describes the actual actions done to put the previously described theoretical ideas into practice.

* 1. **Design Overview**

The project's design overview involves using Mininet, a popular network emulator known for its adaptability and flexibility, to create a network architecture. An array of switches, hosts, and controllers is deployed in this architecture to resemble a condensed software-defined networking (SDN) environment. Controllers are positioned in key locations to supervise network traffic management and instruct switches on packet forwarding practices. Hosts facilitate the interchange of data packets by acting as endpoints for communication inside the network. The main goal of this design is to create a reliable testing environment that can be used to assess different load balancing methods. These algorithms include neural network-based, hybrid, dynamic, and static approaches; each has unique benefits and difficulties**.**

* 1. **Network Topology**

To simulate a distributed network environment, the network topology consists of numerous switches, hosts, and controllers connected to one another. The network is composed of three Open vSwitch (OVS) switches, namely s4, s6, and s7. Data exchange inside the network is facilitated by each switch's connections to a group of hosts and other switches. In order to regulate network traffic and guarantee effective communication between switches and hosts, three more controllers—designated as c0, c1, and c2—are installed. Twelve hosts (h1–h12) make up the topology, and each one has a distinct IP address within the 10.0.0.0/8 subnet. Hosts are connected to switches via direct links, enabling direct communication and data transfer.

In addition to the main network components, three hosts, h1, h2, and h3, are designated as servers dedicated to carrying out traffic load balancing algorithms. These servers are set up to perform computational activities such as analyzing network traffic patterns, determining appropriate load distribution techniques, and applying the load balancing algorithms of choice. By limiting these hosts to the load balancing process, the algorithms can run efficiently without interference from other network operations. By doing so, I have evaluated and compared the performance of various load balancing approaches under controlled conditions, allowing them to determine their usefulness in optimizing network resource utilization and enhancing overall system performance.

For building the topology, I first imported Mininet libraries.

A screenshot of a computer program

Description automatically generated

Figure 2: Importing Mininet libraries.

Then, I defined my network topology by creating a Mininet object with specified IP address range.

A black background with white text

Description automatically generated

Figure 3: my Network function

After creating Mininet object, all that was left was adding controllers, switches and hosts and linked them together.

A screen shot of a computer program

Description automatically generated

Figure 4: Devies being added to network

A screen shot of a computer

Description automatically generated

Figure 5: Linking the devices

Once links were built, the network was run by starting the controllers and activating the interfaces. Finally, pingall() command was used to post configure switches and hosts.

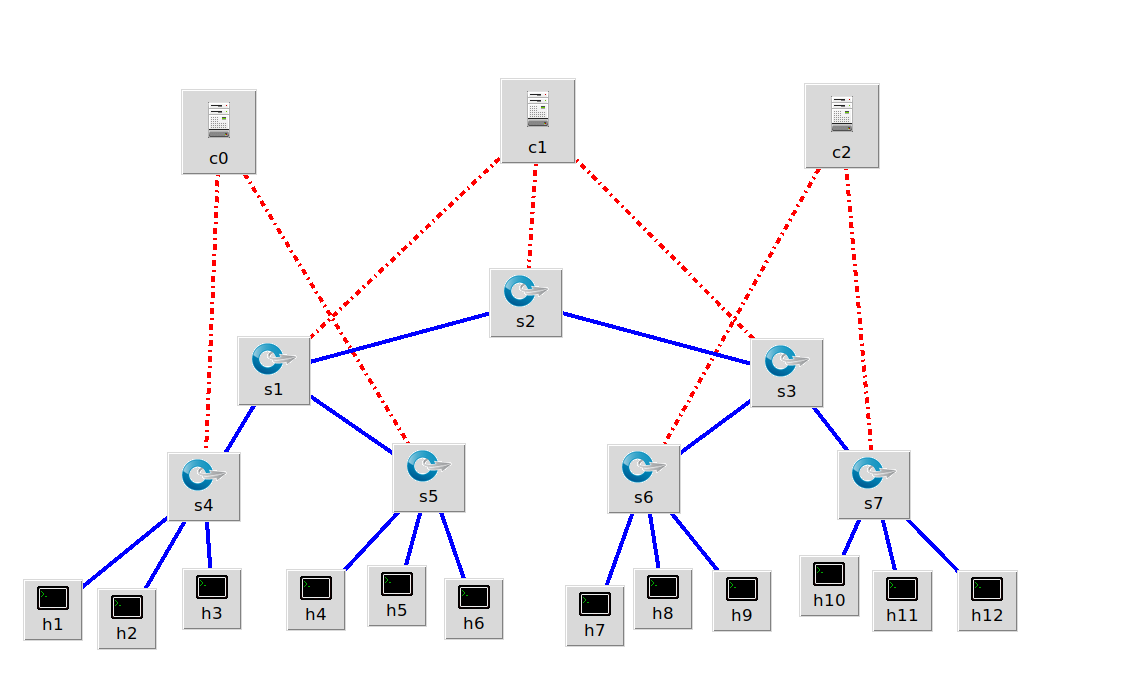


Figure 6: Network topology implementation.

Before running “topology.py”, I must start the Opendaylight controller for the network to locate the remote controllers running at port 6633 and 6653 using the command: karaf0.8.4/bin/karaf.

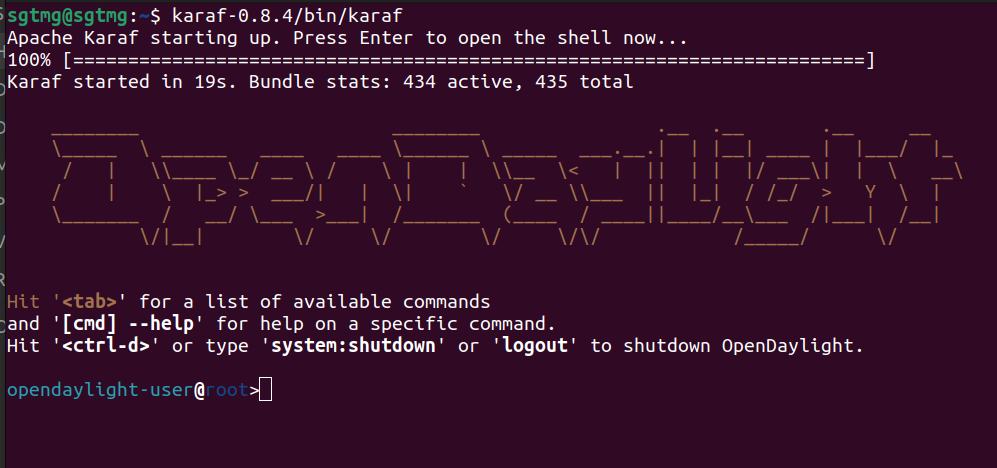


Figure 7: Running Opendaylight Controller (version 0.8.4)

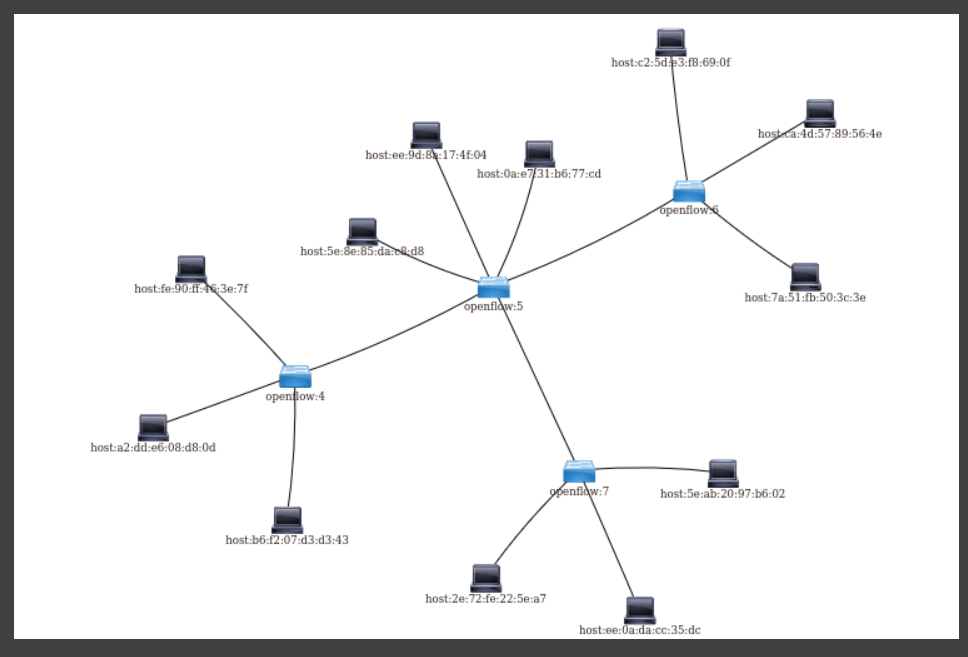
To check if the controller is integrating with Mininet custom topology, I checked <http://localhost:8181/index.html#/topology>.   
  


Figure 8: Topology detected by Opendaylight controller.

* 1. **Technological Stack**

The project's technology stack is made up of a wide range of frameworks and tools that have been carefully chosen to make software-defined networking (SDN) solution design, implementation, and evaluation easier. The technologies used for the project are as follows:

* **Mininet**: a powerful tool for creating virtual network environments. Leveraging lightweight virtualization techniques, Mininet enables the rapid deployment of complex network topologies, allowing users to simulate various network configurations and scenarios with ease.
* **Python2(Mininet Python API)**: Python 2, coupled with the Mininet Python API, is the primary programming language and interface for building network topologies, configuring network components, and programmatically performing network emulation activities. Its flexibility and ease of use make it ideal for coordinating complicated network behaviors and automating repetitive operations.
* **OpenFlow (OF) Protocol**: The OpenFlow protocol is the backbone of SDN designs, allowing for centralized control and management of network switches. By dynamically updating forwarding tables within switches, OpenFlow enables SDN controllers to choreograph network traffic flows in real time, improving performance and resource utilization.
* **OpenDaylight (ODL) Controller (version 0.8.4):** OpenDaylight (0.8.4) is a powerful and adaptable framework for controlling SDN-enabled networks. ODL makes it easier to create a wide range of SDN applications and services, from traffic engineering to network security, by supporting the OpenFlow protocol and providing a broad set of APIs and plugins.
* **DLUX graphical interface:** DLUX provides an intuitive graphical interface for monitoring and visualizing network information and performance indicators. By providing real-time insights into network activity, DLUX enables users to obtain a better knowledge of network behavior and more effectively diagnose any faults.
* **Wireshark**: Wireshark, a robust network protocol analyzer, is an essential tool for capturing and analyzing data in the emulated network environment. Its broad feature set allows users to study packet payloads, discover network irregularities, and precisely solve connectivity difficulties.
* **Ubuntu Operating System**: Ubuntu (24.04 LTS) serves as the host operating system for hosting the software components of the stack.
* **CLI (Command Line Interface):** The Command Line Interface is a versatile tool for interfacing with the emulated network environment, allowing users to easily execute commands, configure network components, and monitor system status.
  1. **Load Balancing Algorithms**

In this section, we conduct a comprehensive analysis of load balancing algorithms within the context of our network setup, leveraging insights gained from a thorough literature review and considering various alternative options. Our goal is to evaluate these algorithms based on key performance metrics, including latency, active connections, resource utilization, and throughput, to identify the most effective solution for our specific requirements.

* + 1. **Round Robin**

Round Robin (Wensong Zhang, 1998) is one of the simplest and most used load balancing algorithms. It operates on the notion of cyclically spreading incoming requests among a network of servers. Each new request is routed to the subsequent server in the rotation. This sequential assignment distributes the burden evenly among all servers in the pool. Round Robin is simple to build and has low computational overhead, making it ideal for basic load balancing purposes.

To run this algorithm, I have stored information about servers, the current server index for Round Robin, a counter for generating unique flow IDs, and input ports in variables.

A screen shot of a computer

Description automatically generated

Figure 9: Setting global variables for round robin algorithm.

I created add\_flow\_rule() function that adds a flow rule to a server in the OpenFlow switch. It constructs a unique flow ID, defines the flow rule in JSON format, and sends a PUT request to the OpenDaylight controller's REST API to add the flow rule.

A screen shot of a computer program

Description automatically generated

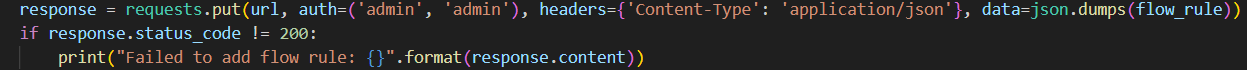


Figure 10: Add to flow function.

The load balancing is performed using the balance\_traffic() method where round robin method is implemented and flow request is made. For every in port, an out port is selected on circular basis and added to the flow.

A screen shot of a computer program

Description automatically generated

Figure 11: Balance traffic function.

A screenshot of a computer

Description automatically generated

Figure 12: Load balancer function

Finally, load balancing algorithm is run every 10 seconds using load\_balancer() function.

* + 1. **Weighted Round Robin**

An addition to the Round Robin algorithm, Weighted Round Robin (Weikun Wang, Giuliano Casale, 2014) adds the idea of weights allotted to each server. The capacity or performance of the servers is represented by these weights. In comparison to servers with lower weights, those with greater weights receive more requests. This makes it possible to use server resources more effectively, which is especially helpful in scenarios with a heterogeneous server configuration where servers have varying capacities.

I have implemented this algorithm using two main functions geared towards server monitoring and load balancing within a network infrastructure. The monitor\_metrics(net) function orchestrates the real-time tracking of various performance metrics across designated servers, encompassing CPU and memory utilization, active connections, latency, and throughput. By leveraging subprocesses, it executes commands such as ping and iperf to measure latency and throughput between hosts, while utilizing the psutil library to gather CPU and memory usage data. Additionally, it employs netstat to count active connections on each server. These metrics are collated into a dictionary named host\_metrics and concurrently logged into a CSV file at regular intervals of 10 seconds.

A screen shot of a computer program

Description automatically generated

Figure 13: Monitor metrics function

On the other hand, the weighted\_round\_robin() function implements the weighted round-robin load balancing strategy. This algorithm selects the next server to handle incoming requests based on predefined weights assigned to each server. By adjusting the weights dynamically, it aims to distribute traffic more effectively, ensuring optimal resource utilization across the network.

A computer screen shot of a program

Description automatically generated

Figure 14: Weighted round robin function

Through these functions, the code encapsulates both monitoring and load balancing functionalities crucial for maintaining the performance and stability of the network environment.

* + 1. **Least Connection**

As per the Least Connection algorithm (Liangshuai Zhu et al, 2018), incoming requests are routed to the server having the fewest active connections of the request. By delivering new requests to the server with the fewest active connections, this method seeks to equally share the burden among servers. The Least Connection algorithm assists in preventing any one server in the pool from becoming overloaded by dynamically modifying the distribution of requests based on server load.

To integrate this algorithm in my topology, first I have declared a global dictionary active\_connections to store the number of active connections on each server. Then, I have created three major functions:

* Monitor metrics function that is responsible for monitoring various metrics such as CPU usage, memory usage, latency, and throughput on each host. It iterates over all hosts in the network and retrieves the relevant metrics using commands executed on the hosts. The metrics are then stored in a dictionary host\_metrics with the host name as the key.

A computer screen shot of a computer program

Description automatically generated

Figure 15: Monitor metrics function that monitors performance metrics on each host.

* Least connection function that selects the server with the least number of active connections. It sorts the servers based on their active connections count and returns the server with the least connections.

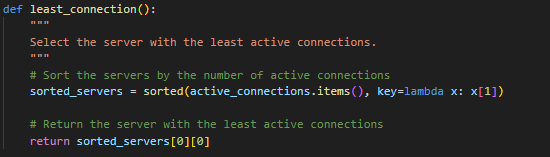


Figure 16: Least connection function

* Generate traffic and monitor metrics function that generates traffic to the server with the least active connections and continuously monitors metrics. It selects a random source host from the list of hosts excluding the servers and initiates an iperf session to the server with the least connections. After generating traffic, it monitors metrics using the monitor metrics function and writes them to a CSV file and sleeps for interval of 10 seconds.

A screen shot of a computer program

Description automatically generated

Figure 17: Generate traffic and monitor metrics function.

Then, myNetwork() function sets up the Mininet network with switches, controllers, and hosts as per the specified topology. Additionally, it starts a separate thread for generating traffic and monitoring metrics using the generate traffic and monitor metrics function.

* + 1. **Dynamic Threshold based Load Balancing**

Incoming request routing is dynamically adjusted by dynamic threshold-based load balancing (Songzhou Li et al, 2023), which is based on thresholds that are dynamically determined. These thresholds consider variables like response times, server load, and network conditions. This method optimizes the workload distribution among servers in real-time, guaranteeing effective resource utilization and upholding high system performance, by continuously monitoring these metrics and modifying routing decisions accordingly.

To implement dynamic thresholding in my network topology, I have set global variables i.e. active\_connections, servers, server\_index. Also, I initialized threshold for each server. Moving on, I have set parameter that would determine dynamic adjustment of threshold and given path for csv file to store metrics i.e. timestamp, server name, connections, and adjusted threshold. Finally, I have created three major functions:

* Monitor connection function that monitors active connections on servers h1, h2, and h3 and dynamically adjusts thresholds based on load. Firstly, it iterates through the servers and checks if the number of active connections is below the threshold. It then calculates the adjusted threshold based on the current number of active connections and a load factor and then sleeps for a specified update interval before repeating the monitoring process.

A screen shot of a computer program

Description automatically generated

Figure 18: Monitor connections function.

* Dynamic threshold-based load balancing function that implements a dynamic threshold-based load balancing algorithm to select the next server for incoming connections. It first loops through the servers and checks if the number of active connections is below the threshold. It returns the server with the least number of active connections below its threshold. Lastly, it updates the index to keep track of the next server selection.

A computer screen shot of text

Description automatically generated

Figure 19: Dynamic threshold-based load balancing function

* Generate traffic function that generates high traffic from multiple hosts to servers h1, h2, and h3 to test the load balancing algorithm. It randomly selects a source host and destination server for each connection and uses the iperf command to generate traffic from the source host to the destination server. To simulate real-world traffic patterns, I added small delays between connections and larger delays between iterations.

A computer screen shot of text

Description automatically generated

Figure 20: Generate traffic function.

* + 1. **Neural Network based Load Balancing**

(S WilsonPraksh et al, 2019) This method determines the best course for incoming requests by making use of machine learning techniques, particularly neural networks. These neural networks are trained using historical data, which includes network traffic patterns, server performance metrics, and other relevant parameters. The neural network can determine in real time how to divide traffic across servers to maximize system performance and guarantee effective resource utilization by evaluating this data.

Implementation of this algorithm to my topology was challenging. I tried using neural network model using TensorFlow and Keras. The model architecture consists of an input layer with four features (representing network traffic data) and two hidden layers with 64 neurons each, followed by an output layer with three neurons (representing the three servers). The output layer uses a softmax activation function to predict the probabilities of routing traffic to each server.

A diagram of a machine

Description automatically generated

Figure 21: Neural network representation

Firstly, I defined neural network architecture namely number of input features and number of servers.

A computer code on a black background

Description automatically generated

Figure 22: Defining neutral network architecture.

Compilation of the model is done using Adam optimization algorithm, which is an adaptive learning rate optimization algorithm. Adam is well-suited for training neural networks and is widely used in practice due to its efficiency and effectiveness. The loss function I have used is 'sparse\_categorical\_crossentropy' which is a specific form of the categorical cross-entropy loss function designed for classification problems where the target labels are integers (sparse). In this case, the output layer of the neural network has multiple neurons, each representing a different class (in this case, the three servers). The loss function computes the difference between the predicted probabilities and the actual class labels, aiming to minimize this difference during training. The third parameter the model uses is ‘accuracy’ which is a commonly used metric for classification tasks and provides a straightforward measure of the model's performance.

A computer screen shot of a program code

Description automatically generated

Figure 23: Monitor connections function.

The monitor\_connections() function continuously monitors the active connections on servers and dynamically adjusts thresholds. It updates the CSV file with timestamp, server, active connections, and adjusted thresholds.

A screen shot of a computer program

Description automatically generated

­Figure 24: Neural network load balancing function

The neural\_network\_load\_balancing() function uses the trained neural network model to predict the next server for load balancing based on input features. The traffic have been periodically generated in matter of 10 second among hosts using generate\_traffic() method and real-time traffic data from network interfaces using collect\_traffic\_data() function.

A computer screen shot of a program

Description automatically generated

Figure 25: Traffic generation and data collection for neural networks

* 1. **Implementation Details**

Firstly, I’ve set the environment by installing Ubuntu operating system on my computer. I could have used virtual machine software like VMware or VirtualBox, but I’ve rather installed the operating system. Installation and configuration of Mininet, Python2, Opendaylight Controller (version 0.8.4) and Wireshark was ensured.

All the required dependencies for the Opendaylight controller such as odl-dlux-core, odl-restconf, odl-l2switches were installed. Using the Mininet Python API, I have programmatically defined the network topology, specifying switches, hosts, links, and other essential components as shown in figure 6. Network topology consists of three controllers, three servers h1, h2 and h3 and 9 hosts and 7 switches. Python scripting has been employed for automation, ensuring consistency and reproducibility in defining the network topology. Moreover, libraries such as tensorflow, psutil, numpy and threading have been installed as part of implementation of load balancing algorithms.

**A diagram of a system

Description automatically generated**

Figure 26: Overview of implementation of the project

Once the topology is run using Python script, traffic is generated randomly to these servers. These requests are then analyzed and processed using the assigned load balancing algorithm. OpenFlow (OF) protocol, which is a standardized communication protocol used in Software-Defined Networking (SDN) environments to manage the behavior of network switches, has been utilized. This integration ensures seamless communication between Mininet, OpenFlow-enabled switches, and the OpenDaylight Controller. The load balancing algorithm provides flow direction assigning port number of the dedicated server.

The controller can add, edit, and delete flow entries in flow and group tables (as well as meters in the meter table) both proactively and reactively (in reaction to the properties of incoming packets) using the OF protocol. The controller then adds a flow rule to the appropriate flow table in the network switches, instructing them on how to handle incoming packets. DLUX graphical interface provided by OpenDaylight is configured to visualize and monitor network statistics in real-time. Wireshark is set up for traffic analysis, capturing and analyzing network traffic to facilitate detailed packet-level analysis for troubleshooting and performance optimization. Data were gathered after the implementation of every algorithm and stored in CSV file and later used for analysis. For implementation of neural network-based load balancing algorithm, data gathered using every other algorithm was used to estimate the cpu usage, memory usage, network traffic and response time. The collected data were then preprocessed for input to neural network. The train model was integrated into load balancing system and the load predictions was used to dynamically adjust request routing to optimize load distribution across servers.

1. **Discussion and Evaluation**

This section provides a critical analysis of the project's outcomes and lays the groundwork for future work and improvements. By comparing these outcomes against initial objectives and benchmarks, a deeper understanding of how effectively our solutions addressed the identified problem or question can be achieved. Moreover, this section offers valuable insights into the effectiveness of the implemented solutions and informs decision-making for optimizing network performance and scalability.

* 1. **Performance Analysis and Comparison**

This part of analysis focuses on determining which algorithm best meets the specific requirements. Among many performance metrics, three of the chosen metrics are throughput, resource utilization and latency. Throughput measures the rate of processing, or the number of packets transferred per unit of time. It is one of the most used performance metrics to assess the efficiency of systems, networks, or algorithms. Resource utilization is a measure of how efficiently system resources (such as CPU, memory, disc I/O, and network bandwidth) are used. It is critical to ensure that resources are used appropriately to avoid bottlenecks and maximize efficiency. High resource utilization can indicate that a system is working hard, but it can also be a sign of inefficiency or overload. Latency is the time between the start of a process and the start of the response. In computing, it is frequently used to describe the time it takes for a packet of data to move from one point to another on a network, or the time it takes for a system to execute a request and produce a response. Low latency is desirable in many systems, particularly real-time or interactive applications, because it provides a more responsive user experience. Moving on, data collection in the csv file has been used for comprehensive performance analysis of each load balancing algorithm.

Figure 27: Round robin analysis

Initially, with a throughput of 35 and 50 active connections, the system has a moderate workload. However, as throughput progressively rises to 50.6 and active connections reach 250, it shows a significant increase in demand. Accompanying this surge is an increase in CPU utilization from 12% to 75%, indicating increased computing requirements to manage the expanding workload. Nonetheless, the boost in demand is accompanied by a considerable increase in latency, with values rising from 0.58 to 3.58. Such latency spikes indicate potential performance bottlenecks, implying that the system is straining to cope with the increased workload, resulting in delayed responses.

Figure 28: Weighted round robin analysis

Moving on, the system has a modest workload, with a throughput of 44.5 and 50 active connections, as well as a reasonably low CPU utilization of 16% and a latency of 0.28 when implemented with weighted round robin algorithm. However, as the demand from hosts increases (number of requests from hosts), with throughput progressively increasing to 68.2 and active connections peaking at 250, the system sees a significant increase in CPU utilization, reaching 55%. This significant increase in CPU utilization indicates greater computational need to handle the expanding workload efficiently. Latency also increases from 0.28 to 3.18, indicating lengthier reaction times as the system strains to handle the increased demand. While the system displays strong scalability, as indicated by its capacity to handle increased workload, the large increase in latency indicates possible performance limitations.

Figure 29: Performance analysis of least connection

The network topology with least connection load balancing algorithm operates under a modest workload, with a throughput of 44.5 and 50 active connections, and a CPU utilization of only 16%. This phase demonstrates effective resource management and low latency, with a latency rating of 0.28 suggesting rapid response times. However, as demand increases, with throughput continuously climbing to 68.2 and active connections peaking at 250, so does CPU utilization, which reaches 55%. This increase in CPU consumption indicates a higher computational demand to efficiently manage the rising workload. As a result, latency increases significantly, reaching 3.18, indicating delays in request processing or response times as the system deals with the increased traffic. While the system's scalability is commendable, the significant rise in latency indicates future performance constraints.

Figure 30: Performance analysis of dynamic threshold-based load balancing

Initially, with a throughput of 47 and 50 active connections, the network with dynamic threshold-based load balancing operates within manageable limits. Despite a modest CPU utilization of 26%, it efficiently handles the workload, resulting in a minimal latency of 0.26. As the workload steadily increases, the threshold for each server has increased by the load factor of 1.2. Hence, servers were able to cope with more loads. Consequently, with throughput peaking at 79 and 250 active connections, the system's CPU utilization climbs to 64%. This surge in CPU demand indicates the system's struggle to manage the heightened workload effectively. Consequently, latency experiences a substantial increase, reaching 2.14, suggesting delays in request processing or response times. While the system demonstrates commendable scalability, evidenced by its ability to accommodate the escalating workload, the notable increase in latency implies potential performance bottlenecks. To sustain optimal performance and responsiveness, proactive measures such as resource optimization, load balancing, or infrastructure scaling are crucial.

Figure 31: Performance analysis of neural network-based load balancing

In the beginning, with a throughput of 44.5 and 50 active connections, the system after integrating neural network-based load balancing algorithm is within reasonable limitations, with a modest CPU utilization of 18%. This demonstrates efficient resource utilization, resulting in a low latency of 0.25 and quick reaction times. As the workload increases, with throughput reaching 62.2 and 250 active connections, CPU utilization leaps dramatically to 59%, showing higher computing demands to manage the greater traffic. As a result of the increased demand, latency rises significantly to 3.18, indicating delays in request processing or response times. The ability of the neural network-based load balancing solution to adapt to changing workloads demonstrates its response in dynamically allocating resources and optimizing request distribution. Moreover, the trained model is later evaluated on the validation set to assess its performance and adjust hyperparameters if needed.

Moving on to comparative analysis, I have provided line charts to show and explain the performance based on active connections on the server.

Figure 32: Latency Comparison of load balancing algorithms

The dataset obtained provides a thorough understanding of the performance of various load balancing strategies as measured by their latency. Latency, which represents the time delay between initiating a request and receiving a response, is an important parameter for assessing the responsiveness and efficiency of load balanced systems.

To begin, Round Robin displays latency values ranging from 0.58 to 3.58, indicating constant but moderate performance across various demand levels. While straightforward to build, this algorithm's static nature may make it difficult to optimize latency under varying traffic conditions, perhaps resulting in inferior response times. Weighted Round Robin, on the other hand, outperforms Round Robin, with latency ranging from 0.28 to 3.18. Weighted Round Robin seeks to distribute requests more wisely by assigning weights to servers based on their capacities or performance, reducing latency variances, and potentially attaining higher overall performance.

In contrast, the Least Connection technique produces competitive latency values ranging from 0.28 to 2.2. Least Connection minimizes latency and assures optimal resource utilization by routing requests to servers with the fewest active connections, making it an ideal solution for latency-sensitive applications.

Dynamic Threshold has latency values ranging from 0.26 to 2.14, demonstrating its capacity to adjust to shifting workload situations. As the load rises, Dynamic Threshold reduces latency and optimizes resource allocation by dynamically altering load distribution depending on server load thresholds, making it an appealing solution for dynamic environments.

Finally, the Neural Network-based load balancing method has latency values ranging from 0.25 to 3.18, exhibiting competitive performance across a variety of workloads. Using machine learning techniques, this solution dynamically optimizes request distribution based on real-time server measurements, resulting in effective latency management and potentially higher performance in complex and dynamic contexts.

In summary, given the presented latency data, Least Connection appears to be the best algorithm for minimizing delay across a range of workload levels. It continuously provides low latency values, making it ideal for applications or services that require fast reaction times. However, Dynamic Threshold and Neural Network-based load balancing provide competitive performance and may provide additional benefits such as adaptability and scalability, depending on the system's specific requirements.

Figure 33: CPU usage comparison of algorithms

Round Robin, a simple but extensively used algorithm, displays a wide range of CPU utilization, from 12% to 75%. However, its static structure may impede optimum resource allocation, particularly in contexts with varying server capacities or changing workloads. Round Robin might be suitable for small-scale environments with homogeneous server capacities and stable workloads but may struggle to adapt to dynamic or heterogeneous environments. Weighted Round Robin is an upgrade to Round Robin that includes server weights based on their capacity or performance, resulting in more balanced CPU utilization ranging from 16% to 55%. In contrast, Least Connection dynamically sends requests to servers with the fewest active connections, resulting in a more equal workload distribution and competitive CPU utilization rates ranging from 16% to 54%. This algorithm is well-suited for environments where maintaining balanced resource usage is critical, as it helps prevent server overload and ensures optimal performance without requiring detailed knowledge of server capacities or performance.

Dynamic Threshold, with its dynamic load adjustment based on server load thresholds, effectively manages CPU utilization in the range of 26% to 64% while adjusting well to changing workload conditions. On the other side, Neural Network-based load balancing uses machine learning approaches to optimize request distribution, resulting in CPU utilization rates ranging from 18% to 59%. This technique is scalable and adaptable, making it appropriate for complex and dynamic settings. While each load balancing method has distinct advantages and trade-offs in terms of CPU utilization management, the choice is determined by individual requirements, workload characteristics, and the desired balance of simplicity and efficiency in resource management.

To recap, Least Connection may be selected in circumstances where maintaining a balanced resource use is critical. Dynamic Threshold is appropriate for contexts with changing workloads, whereas Neural Network-based load balancing is adaptable and scalable, making it suitable for complex and dynamic environments. Thorough testing and assessment in a representative setting is critical for determining the best algorithm for the specific situation.

Figure 34: Throughput comparison of algorithms.

Lastly, analysis of throughput metrics was performed. Throughput statistics range from 35 to 50.6, indicating steady but modest performance at various workload levels. Round Robin distributes requests uniformly between servers in a cyclical way, without regard for server capacity or workload changes. While Round Robin is simple to build, it may not successfully optimize performance, particularly in contexts with diverse server capacity or changing workloads.

Weighted Round Robin improves Round Robin by allocating server weights based on capacity or performance. The throughput statistics range from 44.5 to 68.2, demonstrating a more evenly distributed demand across servers than Round Robin. This method has the potential to improve throughput by intelligently distributing queries to servers with higher capacity or performance. Moreover, when I utilized the Least Connection algorithm, which sends requests to servers with the fewest active connections, attempting to balance workload among servers, throughput ratings range from 40.2 to 69, indicating competitive performance in efficiently processing requests at different workload levels as this approach prioritizes load balancing depending on server capacity, which could increase throughput in contexts with changing traffic patterns.

Dynamic Threshold stands out as the top performer in terms of throughput, consistently demonstrating the highest values ranging from 47 to 79. This algorithm dynamically adjusts load distribution based on server load thresholds, effectively optimizing throughput, and preventing server overload. Its adaptability to changing workload conditions makes it well-suited for environments with fluctuating traffic patterns or unpredictable workloads. By efficiently processing requests, Dynamic Threshold ensures optimal system performance and resource utilization.

Load balancing using neural networks yields competitive throughput numbers (44.5-62.2). Using machine learning techniques, this solution optimizes request distribution based on real-time server characteristics, potentially increasing performance. While it may not have the best throughput numbers when compared to Dynamic Threshold, Neural Network-based load balancing is adaptable and scalable, making it ideal for complex and dynamic contexts where standard methods may fall short.

To sum up, based on the data collected, Dynamic Threshold appears to be the best-suited load balancing algorithm for optimizing throughput and ensuring efficient request processing across varying workload levels. However, the choice ultimately depends on specific requirements, workload characteristics, and the desired balance between performance, adaptability, and scalability.

* 1. **Challenges and Limitations**

As I investigate the complexities of load balancing algorithms in Software-Defined Networking (SDN) environments, I have encountered several obstacles. My research covers a wide range of load balancing strategies, including traditional methods like round-robin and weighted round-robin, as well as more advanced approaches like least connection, dynamic threshold-based load balancing, and neural network-based load balancing. As I move across this landscape, numerous major challenges emerge, each demanding careful analysis and imaginative solutions.

One of the most difficult challenges I have is developing an experimental framework that accurately represents real-world SDN setups. SDN represents an elementary advancement in network administration, with centralized controllers orchestrating network traffic flows dynamically. Ensuring that my experimental setup adequately depicts the complex aspects of SDN systems, such as controller scalability, network topology alterations, and dynamic traffic patterns, is critical to the validity of my study.

The impending end-of-life for Python 2 presents a challenge for projects relying on legacy codebases or libraries that have not been updated to Python 3. Future research could focus on migrating existing load balancing algorithms and tools to Python 3-compatible versions. Additionally, efforts to develop new load balancing solutions should prioritize Python 3 compatibility to ensure future-proofing and long-term sustainability.

Another key problem is the deployment and integration of various load balancing algorithms within SDN frameworks. Each algorithm has a unique collection of parameters, optimization strategies, and tradeoffs. To ensure flawless compatibility between these algorithms and SDN controllers, as well as to address any potential conflicts or bottlenecks, I must pay close attention to detail and do extensive testing.

The lack of comprehensive documentation for load balancing algorithms and SDN frameworks have hindered me adoption and understanding, and particularly for researchers and practitioners new to the field. Future efforts should prioritize the development of clear, well-documented resources, including code repositories, tutorials, and technical guides. Collaborative initiatives within the research community can help create and maintain centralized repositories of documentation, best practices, and use cases for load balancing in SDN.

Insufficient training data can limit the effectiveness and generalizability of neural network-based load balancing algorithms. Future research should explore techniques for collecting and curating large-scale datasets that capture diverse network conditions, traffic patterns, and performance metrics. The project has faced resource constraints in terms of time and access to specialized hardware or software tools. Conducting comprehensive experiments and simulations to evaluate multiple load balancing algorithms requires substantial computational resources and may exceed the project's limitations.

Moreover, while I strive to create realistic experimental environments, simulations inherently differ from real-world SDN deployments. Factors such as hardware variability, network congestion, and unforeseen events may not be fully captured in simulations, potentially limiting the generalizability of my findings.

* 1. **Future Recommendations**

As I work through the constraints and limitations of analyzing traffic load balancers in SDN, I anticipate several future recommendations that will broaden the scope, depth, and effect of my research.

As this project is limited to only five load balancing algorithms, more advanced algorithms could be used for analysis purposes. Model training for neural networks needs to be properly conducted in real world scenarios for more accurate predictions and efficient load balancing. Going on, if I were to redo this project, I would surely try collecting more performance metrics for solidifying my conclusions.

Additionally, monitoring and incorporating developing technologies such as machine learning, artificial intelligence, and edge computing into load balancing algorithms can help unleash new possibilities and increase performance in dynamic SDN systems. Exploring how new technologies might supplement conventional load balancing solutions is critical for being at the cutting edge of network management.

Promoting interoperability and standardization of load balancing protocols, APIs, and interfaces among SDN platforms and manufacturers can help to ensure seamless integration and compatibility. Collaboration with standards organizations and industry alliances can drive consensus on best practices and improve ecosystem maturity. Furthermore, holistic performance evaluation frameworks encompassing a wide array of metrics beyond traditional throughput and latency can offer a more nuanced understanding of load balancing effectiveness.

Field trials and deployment of load balancing solutions in real-world SDN contexts can reveal significant information about their effectiveness, scalability, and practical issues. Collaboration with industry partners, as well as the deployment of experimental setups in operational networks, can help validate research findings and promote technology transfer.

Soliciting comments from network operators, administrators, and end users can shed light on the practical issues and requirements of load balancing in real-world installations. Adopting a user-centric design approach and incorporating user feedback into algorithm development can help load balancing systems meet the needs of a variety of stakeholders.

1. **Conclusion**

In conclusion, this project’s aim was to create a comprehensive understanding of traffic load balancer in software defined networks. Throughout this research, I explored a variety of load balancing strategies, including traditional approaches like round-robin and weighted round-robin, as well as more sophisticated methods like least connection, dynamic threshold-based load balancing, and neural network-based load balancing. By addressing the issues, constraints, and future recommendations, I have got significant knowledge of the complex landscape of load balancing in SDN systems. Moreover, this journey presented me with a journey of exploration, understanding and innovation.

According to my report, by analyzing latency, CPU usage, and throughput data for various load balancing strategies such as Round Robin, Weighted Round Robin, Least Connection, Dynamic Threshold-based, and Neural Network-based load balancing, valuable insights into their effectiveness in optimizing resource utilization and system performance were discovered.

The investigation found that Dynamic Threshold-based algorithm consistently had the highest throughput numbers, followed by Neural Network-based load balancing and Least Connection. These algorithms performed competitively in processing requests at varied workload levels, demonstrating their adaptability and effectiveness in dynamic contexts. In contrast, Round Robin and Weighted Round Robin, while simple to build, produced lower throughput figures and may be less suitable for systems with varied server capacity or changing workloads.

In context of latency, the Least Connection algorithm appears to be the best fit for minimizing latency across a range of workload levels. It continuously provides low latency values, making it appropriate for applications or services that require fast reaction times. However, Dynamic Threshold and Neural Network-based load balancing demonstrate competitive performance and may provide additional benefits such as adaptability and scalability, depending on the system's specific requirements. Talking about resources usage, Least Connection may be preferred for environments where maintaining balanced resource usage is paramount. Dynamic Threshold is suitable for environments with fluctuating workloads, while Neural Network-based load balancing offers adaptability and scalability.

Looking into the future, various ideas emerge that can help advance the subject of traffic load balancing in SDN. Exploring emerging technologies like machine learning and edge computing shows promise for improving the adaptability and agility of load balancing algorithms, allowing them to adjust dynamically to changing network conditions. Real-world validation through field testing and industry collaborations will provide essential insights into the practical efficacy and scalability of load balancing technologies, bridging the theory-practice divide.

**10 References**

Lamiae Boukraa, Safaa Mahrach, Khalid El Makkaoui & Redouane Esbai SDN (2022). Southbound Protocols: A Comparative Study. International Conference on Networking, Intelligent Systems and Security. [online] Available at: <https://link.springer.com/chapter/10.1007/978-3-031-15191-0_39#SnippetTab>

Rukmini Bhat B \*, Sneha N S, Keerthana Bhat, Chaithra C Kamath, Chaitha Naik (2021). Improving the Efficiency of Software Defined Network through Load Balancing Algorithms. Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV). [online] Available at: <https://ieeexplore.ieee.org/document/9388512>

Fakhrun Jamal, Tamanna Siddiqui (2021). Comparative Analysis of Load Balancing Techniques in Cloud Computing, Based on LB Metrices. 5th International Conference on Information Systems and Computer Networks (ISCON). [online] Available at: <https://ieeexplore.ieee.org/document/9702508>

C. Fancy, M. Pushpalatha, Pushpa (2019). Experimentation of Traditional Load Balancing Algorithms in Software Defined Network. International Journal of Recent Technology and Engineering (IJRTE), ISSN: 2277-3878, Volume-8, Issue-1S4, June 2019. [online] Available at: ijrte.org/wp-content/uploads/papers/v8i1s4/A10970681S419.pdf

M.C. Nkosi, A.A. Lysko, S. Dlamini (2018). Multi-path Load Balancing for SDN Data Plane. 2018 International Conference on Intelligent and Innovative Computing Applications (ICONIC). [online] Available at: https://ieeexplore.ieee.org/document/8601241

P. Dharam and M. Dey (2021). A Mechanism for Controller Failover in Distributed Software-Defined Networks. 2021 8th International Conference on Computer and Communication Engineering (ICCCE). [online] Available at: <https://ieeexplore.ieee.org/document/9467174>

Kai-Yu Wang, Shang-Juh Kao and Ming-Tsung Kao (2018). An Efficient Load Adjustment for Balancing Multiple Controllers in Reliable SDN Systems. Proceedings of IEEE International Conference on Applied System Innovation 2018. [online] Available at: <https://ieeexplore.ieee.org/document/8394323>

Songzhou Li, Zhenghua Xin, Xu Xu & Zhiwei Zhang (2023). Load Balancing Algorithm of SDN Controller Based on Dynamic Threshold. 2023 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS). [online] Available at: <https://ieeexplore.ieee.org/document/10145235>

Zhihao Shang, Han Wu, Guang Peng and Katinka Wolter (2019). Dynamic Load Balancing in the Control Plane of Software-Defined Networks. 2019 IEEE 19th International Conference on Communication Technology. [online] Available at: <https://ieeexplore.ieee.org/document/8947243>

Poonam Dharam, Mithila Dey (2021). A Mechanism for Controller Failover in Distributed Software-Defined Networks. 2021 8th International Conference on Computer and Communication Engineering (ICCCE). [online] Available at: <https://ieeexplore.ieee.org/document/9467174>

Dmitry Perepelkin, Maria Ivanchikova, Tin Nguyen (2023). Neural Network Multipath Routing in Software Defined Networks Based on Artificial Bee Colony Algorithm. 2023 XVIII International Symposium Problems of Redundancy in Information and Control Systems (REDUNDANCY). [online] Available at: <https://ieeexplore.ieee.org/document/10330174>

Alexandra Eftimie, Eugen Brococi (2020). SDN controller implementation using OpenDaylight: experiments. 2020 13th International Conference on Communications (COMM). [online] Available at: SDN controller implementation using OpenDaylight: experiments

Linux Foundation, Documentation of Open vSwitch [online] Available at: <https://www.openvswitch.org/>

Wensong Zhang, Linux Virtual Server for Scalable Network Services (1998), National Laboratory for Parallel & Distributed Processing [online] Available at: <http://www.linuxvirtualserver.org/ols/lvs.pdf>

Weikun Wang, Giuliano Casale (2014), Evaluating Weighted Round Robin Load Balancing for Cloud Web Services, 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing [online] Available at: <https://ieeexplore.ieee.org/document/7034709>

Liangshuai Zhu, Jianming Cui, Gaofeng Xiong (2018), Improved dynamic load balancing algorithm based on Least-Connection Scheduling, IEEE 4th Information Technology and Mechatronics Engineering Conference (ITOEC) [online] Available at: <https://ieeexplore.ieee.org/document/8740642>

S. WilsonPrakash, P. Deepalakshmi (2019), Artificial Neural Network Based Load Balancing on Software Defined Networking, IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS) [online] Available at: <https://ieeexplore.ieee.org/document/8951365>