UNIVERSITY OF PIRAEUS
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGIES
DEPARTMENT OF DIGITAL SYSTEMS

Consensus algorithms in Software Defined Networks (SDN) for multiple controllers

Doctoral Dissertation

Lalou Stavroula

Piraeus, 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ
ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αλγόριθμοι Συναίνεσης σε Δίκτυα Ορισμένων Λογισμικών (SDN) για πολλαπλούς ελεγκτές

Διδακτορική Διατριβή

Λάλου Σταυρούλα

Πειραιάς, 2025

Advisory Committee

Sokratis Katsikas,
Professor (Supervisor)
Norwegian University of Science and Technology

Costas Lambrinoudakis,
Professor (Supervisor)
University of Piraeus

Christos Xenakis,
Professor
University of Piraeus

Piraeus, 2025

Approval Sheet

UNIVERSITY OF PIRAEUS
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGIES
DEPARTMENT OF DIGITAL SYSTEMS

This is to certify that the Dissertation presented by Lalou Stavroula, entitled " ***Consensus algorithms in Software Defined Networks (SDN) for multiple controllers***", submitted in fulfillment of the requirement for the degree of Doctor of Philosophy, complies with the regulation of the University of Piraeus and meets the accepted standards with respect to originality.

Piraeus, 2025

Dissertation Committee

Sokratis Katsikas,
Professor
Norwegian University of Science and Technology

Stefanos Gritzallis
Professor
University of Piraeus

Costas Lambrinoudakis,
Professor
University of Piraeus

Christos Xenakis,
Professor
University of Piraeus

Georgios Kavallieratos,
Associate Professor
Norwegian University of Science and Technology

Georgios Spathoulas,
Professor
Norwegian University of Science and Technology

Georgios Kampourakis,
Professor
University of Aegean

Piraeus, 2025

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. I additionally declare that the opinions expressed in this document are my sole responsibility and do not necessarily represent the official position of the University of Piraeus.

Lalou Stavroula

Piraeus, July, 2025

Dedicated to my daughter Maria
and my husband Alexios

Ευχαριστίες (Acknowledgments)

Ολοκληρώνοντας την εκπόνησης της διδακτορικής μου διατριβής, θα ήθελα να ευχαριστήσω προσωπικά εκείνους που με υποστήριξαν κατά τη διάρκεια αυτής της προσπάθειας και να εκφράσω την ειλικρινή μου ευγνωμοσύνη για την ανεκτίμητη συνεισφορά τους.

Πρώτα απ' όλα, θα ήθελα να ευχαριστήσω από τα βάθη της καρδιάς μου τον επιβλέποντα καθηγητή μου, κ. Σωκράτη Κάστικα, στο Πανεπιστήμιο «Norwegian University of Science and Technology», ο οποίος πίστεψε σε εμένα από την πρώτη στιγμή και με υποστήριξε καθ' όλη τη διάρκεια του ερευνητικού αυτού ταξιδιού. Μου προσέφερε μεταξύ άλλων όλους τους πόρους και τα μέσα που χρειάστηκαν, τις πολύτιμες γνώσεις και τον σημαντικό χρόνο του. Οι καίριες επισημάνσεις του σε συνδυασμό με τα προαναφερθέντα, συνέδραμαν καθοριστικά στην επιτυχημένη ολοκλήρωση της παρούσας διδακτορικής διατριβής. Τον εκτιμώ βαθύτατα ως άνθρωπο και ως επιστήμονα και θεωρώ ηθική μου ανταμοιβή τη συνέχιση της συνεργασίας μας σε νέα ερευνητικά μονοπάτια.

Επιπλέον, θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες στον κ. Γιώργο Σπαθούλα, Εργαστηριακό Διδακτικό Προσωπικό του Τμήματος Πληροφορικής και Βιοϊατρικής Πληροφορικής του Πανεπιστημίου Θεσσαλίας από το 2014 και μεταδιδακτορικό ερευνητή στην ομάδα Ασφάλειας και Ανθεκτικότητας Κρίσιμων Υποδομών του NTNU CCIS, στο Norwegian University of Science and Technology, για την εξαιρετική καθοδήγηση και τις πολύτιμες συμβουλές του στην εκπόνηση αυτής της διατριβής. Η πολύτιμη γνώση, ο χρόνος και η αμέριστη υποστήριξή του υπήρξαν καθοριστικοί παράγοντες για την επιτυχή ολοκλήρωσή της.

Έπειτα, θα ήθελα να πω ένα μεγάλο ευχαριστώ στον κ. Χρήστο Ξενάκη, Καθηγητή στο Πανεπιστήμιο Πειραιά, και τον κ. Κώστα Λαμπρινουδάκη, Καθηγητή στο Πανεπιστήμιο Πειραιά, για τη συμμετοχή του στην επιτροπή και τη συνεχιζόμενη καθοδήγησή του στην πορεία αυτής της διατριβής.

Επιπλέον, θα ήθελα να ευχαριστήσω ιδιαίτερα και να εκφράσω την εκτίμησή μου σε όλα τα μέλη της επταμελούς εξεταστικής επιτροπής, για την πρόθυμη συμμετοχή και ενεργή συνεισφορά τους στην κρίση της διδακτορικής μου διατριβής, όσο και για την εποικοδομητική κριτική και τις χρήσιμες συμβουλές που μοιράστηκαν μαζί μου στα πλαίσια αυτής.

Τέλος, το μεγαλύτερο ευχαριστώ μου το οφείλω στην οικογένειά μου, στον αγαπημένο μου σύζυγο Αλέξη και την κόρη μας Μαρία, οι οποίοι στάθηκαν δίπλα μου σε κάθε βήμα αυτής της διαδρομής. Ήταν εκεί για να αφουγκραστούν τις ανησυχίες μου, να με στηρίξουν σε κάθε αποτυχία και να μοιραστούν μαζί μου κάθε επιτυχία στην εκπόνηση αυτής της διατριβής. Η παρουσία τους, η υπομονή τους και η αγάπη τους αποτέλεσαν την κινητήριο δύναμη που με ενέπνευσε να συνεχίσω και να ολοκληρώσω αυτό το έργο.

Με τιμή,
Σταυρούλα Λάλου

Abstract

This doctoral dissertation investigates the application of consensus algorithms to enhance collaboration among multiple controllers in Software Defined Networks (SDN). The research focuses on addressing critical challenges in SDN, such as scalability, fault tolerance, single point of failure, and network state synchronization, by leveraging consensus protocols. Traditional single-controller architectures face limitations in managing large-scale networks, including bottlenecks, overhead, and vulnerability to failures. In contrast, multi-controller architectures offer improved flexibility, scalability, and reliability. However, effective coordination and synchronization among distributed controllers remain essential for optimal network performance.

The study conducts a comparative analysis of existing consensus protocols and mechanisms, evaluating their performance under identical simulation environments. A novel consensus mechanism based on the Raft algorithm is proposed, designed to ensure stable, consistent, and efficient network state synchronization across controllers. The mechanism supports high throughput, dynamic view changes, fault tolerance, and controller synchronization, demonstrating superior performance compared to existing alternatives. Additionally, the research explores load balancing techniques to optimize resource utilization in the control plane, addressing challenges such as latency and packet loss during load migration.

Furthermore, the dissertation addresses the Controller Placement Problem (CPP), proposing a heuristic greedy algorithm to minimize end-to-end latency and reduce maximum latency between controllers and switches. This approach aims to enhance network performance by strategically positioning controllers and mitigating queuing delays. The study emphasizes the practical implementation of these solutions, providing a comprehensive framework for improving scalability, reliability, and efficiency in SDN architectures.

By integrating consensus algorithms, load balancing techniques, and optimized controller placement strategies, this research contributes to advancing SDN management, offering robust solutions for modern network challenges. The findings and proposed mechanism are experimentally validated, demonstrating their effectiveness in achieving stable and high-performing SDN environments.

Field of Science: Security

Keywords: Software Defined Networking, Multiple Controllers; Consensus Algorithm, Fault-Tolerance, Coordination, Load Balancing, Controller Placement, Latency.

Περίληψη

Αυτή η διδακτορική διατριβή ερευνά την εφαρμογή αλγορίθμων συναίνεσης για την ενίσχυση της συνεργασίας μεταξύ πολλαπλών ελεγκτών σε Δίκτυα Ορισμένα από Λογισμικό (SDN). Η έρευνα επικεντρώνεται στην αντιμετώπιση κρίσιμων προκλήσεων στα SDN, όπως η κλιμακωσιμότητα, η ανοχή σε σφάλματα, το σημείο μοναδικής αποτυχίας και ο συγχρονισμός της κατάστασης του δικτύου, μέσω της χρήσης πρωτοκόλλων συναίνεσης. Οι παραδοσιακές αρχιτεκτονικές με έναν μόνο ελεγκτή αντιμετωπίζουν περιορισμούς στη διαχείριση μεγάλων δικτύων, συμπεριλαμβανομένων των σημείων συμφόρησης, της υπερβολικής φόρτωσης και της ευπάθειας σε αποτυχίες. Αντίθετα, οι αρχιτεκτονικές με πολλαπλούς ελεγκτές προσφέρουν βελτιωμένη ευελιξία, κλιμακωσιμότητα και αξιοπιστία. Ωστόσο, η αποτελεσματική συντονισμός και συγχρονισμός μεταξύ των κατανεμημένων ελεγκτών παραμένει απαραίτητος για τη βέλτιστη απόδοση του δικτύου.

Η μελέτη πραγματοποιεί μια συγκριτική ανάλυση των υφιστάμενων πρωτοκόλλων και μηχανισμών συναίνεσης, αξιολογώντας την απόδοσή τους υπό τις ίδιες συνθήκες προσομοίωσης. Προτείνεται ένας νέος μηχανισμός συναίνεσης βασισμένος στον αλγόριθμο Raft, ο οποίος έχει σχεδιαστεί για να εξασφαλίζει σταθερό, συνεπή και αποτελεσματικό συγχρονισμό της κατάστασης του δικτύου μεταξύ των ελεγκτών. Ο μηχανισμός υποστηρίζει υψηλή απόδοση, δυναμικές αλλαγές προβολής, ανοχή σε σφάλματα και συγχρονισμό ελεγκτών, παρουσιάζοντας ανώτερη απόδοση σε σύγκριση με τις υπάρχουσες εναλλακτικές λύσεις. Επιπλέον, η έρευνα εξετάζει τεχνικές εξισορρόπησης φόρτου για τη βέλτιστη αξιοποίηση των πόρων του επιπέδου ελέγχου, αντιμετωπίζοντας προκλήσεις όπως η αύξηση της καθυστέρησης και η απώλεια πακέτων κατά τη μετανάστευση φόρτου.

Επιπλέον, η διατριβή ασχολείται με το Πρόβλημα Τοποθέτησης Ελεγκτών (CPP), προτείνοντας έναν ευρετικό αλγόριθμο για την ελαχιστοποίηση της καθυστέρησης από άκρο σε άκρο και τη μείωση της μέγιστης καθυστέρησης μεταξύ ελεγκτών και διακομιστών, με στόχο την εξάλειψη της καθυστέρησης στην ουρά των ελεγκτών. Η μελέτη τονίζει την πρακτική εφαρμογή αυτών των λύσεων, παρέχοντας ένα ολοκληρωμένο πλαίσιο για τη βελτίωση της κλιμακωσιμότητας, της αξιοπιστίας και της αποτελεσματικότητας στις αρχιτεκτονικές SDN.

Με την ενοποίηση αλγορίθμων συναίνεσης, τεχνικών εξισορρόπησης φόρτου και βελτιστοποιημένων στρατηγικών τοποθέτησης ελεγκτών, αυτή η έρευνα συμβάλλει στην πρόοδο της διαχείρισης των SDN, προσφέροντας ισχυρές λύσεις για τις σύγχρονες προκλήσεις των δικτύων. Τα ευρήματα και οι προτεινόμενοι μηχανισμοί επαληθεύονται πειραματικά, αποδεικνύοντας την αποτελεσματικότητά τους στην επίτευξη σταθερών και υψηλής απόδοσης SDN περιβαλλόντων.

Θεματική Περιοχή: Ασφάλεια Δικτύου

Λέξεις Κλειδιά: Δίκτυα Ορισμένα από Λογισμικό (SDN), Πολλαπλοί Ελεγκτές; Αλγόριθμος Συναίνεσης, Ανοχή σε Σφάλματα, Συντονισμός, Εξισορρόπηση Φόρτου, Τοποθέτηση Ελεγκτών, Καθυστέρηση.

Table of Contents

Table of Figures

Table of Tables

Abbreviations and Acronyms

CPP	Controller Placement Problem
SDN	Software Define Networks

Chapter 1: Introduction

1.1 Introduction

Software Defined Networking (SDN) is an emerging technology that separates the network control plane from the data forwarding plane, enabling centralized management and programmability. This decoupling promises significant improvements in network resource utilization, simplified management, and reduced operational costs. SDN is increasingly being adopted in various network environments, including campus networks, data center networks, Wide Area Networks (WANs), enterprise networks, and Internet exchange points. Its flexibility and scalability make it particularly well-suited for modern, dynamic network infrastructures, where traditional architectures struggle to keep pace with evolving demands. [1]

The SDN architecture introduces a novel approach to network management by decoupling the Control Plane and Data Plane, enabling centralized control and programmability. This architecture can include multiple controllers, especially in large-scale or wide-area networks, where the control layer manages network states globally through network policies, either centrally or in a distributed fashion. SDN abstracts the network by separating the Control Plane and Data Plane, with the forwarding state within the Data Plane governed by the SDN controller. Forwarding decisions are flow-based, with a flow defined as a sequence of packets between a source and a destination. Each flow is characterized by packet field values as match criteria and a set of actions or instructions, ensuring consistent service policies at forwarding devices. The OpenFlow Protocol serves as a standard protocol for communication between the Control Plane and Data Plane, enabling the SDN controller to configure and manage forwarding devices efficiently. [1].

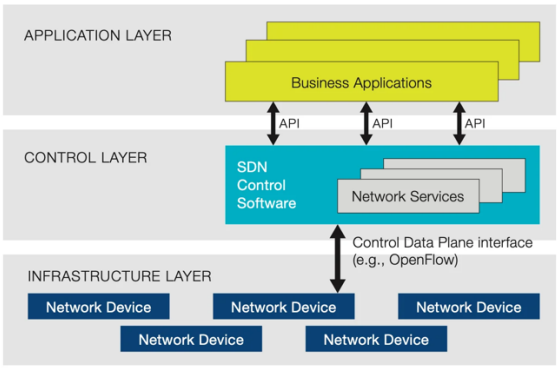

Figure 1:SDN Architecture

This dissertation aims to tackle key challenges in Software Defined Networking (SDN) by exploring multiple-controller architectures and developing a consensus algorithm to enhance scalability, fault tolerance, and synchronization. The study focuses on mitigating single points of failure while ensuring high-throughput operations and dynamic

adaptability. A critical aspect of the research is optimizing the coordination of workload among distributed SDN controllers, employing load-balancing techniques, and addressing the Controller Placement Problem (CPP) to improve network efficiency. The proposed consensus mechanism, built on the Raft algorithm, ensures stable and synchronized network states across controllers while enhancing reliability and performance. Additionally, a heuristic greedy algorithm is introduced to minimize end-to-end latency and reduce queuing delays between controllers and switches. By leveraging SDN's benefits, such as centralized control and programmability, this research contributes to improving network management, enhancing scalability, and ensuring the efficient deployment of SDN controllers. The findings aim to provide practical insights into SDN-enabled networks, refining controller placement strategies and advancing the implementation of multi-controller SDN architectures.

1.2 Problem Statement

The research focuses on addressing the challenges associated with multiple SDN controller architectures, particularly in large-scale networks such as data centers, cloud environments, and wide-area networks (WANs). Single controller architectures, while simpler to manage, are prone to single points of failure and scalability limitations, which can severely impact network performance and reliability. As networks grow in size and complexity, the need for distributed controller models becomes increasingly critical. These models not only enhance fault tolerance but also improve scalability by distributing the control plane workload across multiple controllers. However, this shift introduces new challenges that must be addressed to ensure efficient and consistent network operations.

One of the most significant challenges in multi-controller architectures is achieving controller synchronization, where network state information must be consistently shared across all controllers. Inconsistent or outdated network states can lead to incorrect forwarding decisions, increased latency, and even network failures. Additionally, the coordination of workload among distributed controllers is a critical issue, as uneven load distribution can result in performance bottlenecks and increased latency. Effective load balancing techniques are essential to optimize resource utilization and ensure smooth network operations. Another major challenge is the optimal placement of controllers within the network, known as the Controller Placement Problem (CPP). The strategic placement of controllers directly impacts network performance, particularly in terms of latency, reliability, and resource usage. Placing too few controllers can lead to increased communication delays between controllers and switches, while too many controllers can result in excessive inter-controller traffic.

These challenges highlight the need for innovative solutions to enhance the scalability, reliability, and overall performance of SDN-enabled networks. The research aims to address these issues by developing a consensus mechanism to ensure synchronized network states, exploring effective load balancing techniques to optimize workload distribution, and introducing a heuristic algorithm to solve the Controller Placement Problem. By tackling these

challenges, the study seeks to provide practical insights and solutions for the efficient management of large-scale SDN networks.

1.3 Research Topic

Software Defined Networking (SDN) has emerged as a transformative paradigm for managing large-scale networks, including data centers, cloud environments, and wide-area networks (WANs). By decoupling the control plane from the data plane, SDN enables centralized network management, programmability, and flexibility, which are essential for modern, dynamic network environments. However, traditional SDN architectures often rely on a single controller, which, while simplifying management, introduces significant limitations such as single points of failure and scalability challenges. To overcome these limitations, multiple SDN controller architectures have been proposed, offering improved fault tolerance and scalability. However, this shift introduces new challenges that must be addressed to ensure efficient and consistent network operations.

One of the most significant challenges in multi-controller architectures is achieving controller synchronization, where network state information must be consistently shared across all controllers. Inconsistent or outdated network states can lead to incorrect forwarding decisions, increased latency, and even network failures. To address this, two consistency models strong consistency and eventual consistency are commonly used. Strong consistency ensures that all controllers have the same view of the network at all times, while eventual consistency allows for temporary inconsistencies that are resolved over time. Choosing the appropriate consistency model is critical for balancing performance and reliability in multi-controller architectures. Additionally, the coordination of workload among distributed controllers is a critical issue, as uneven load distribution can result in performance bottlenecks and increased latency. Effective load balancing techniques are essential to optimize resource utilization and ensure smooth network operations.

Another critical aspect of multi-controller architectures is the optimal placement of controllers within the network, known as the Controller Placement Problem (CPP). The strategic placement of controllers directly impacts key performance metrics such as latency, reliability, and resource usage. Placing too few controllers can lead to increased communication delays between controllers and switches, while placing too many controllers can result in excessive inter-controller traffic. Determining the optimal number and location of controllers is a complex task that involves balancing conflicting requirements. For instance, controllers should be placed close to switches to minimize latency, but they must also be distributed to ensure fault tolerance and scalability. Addressing the CPP requires advanced algorithms and optimization techniques to find the best trade-offs for a given network topology and workload.

1.4 Contribution

The research aims to contribute to the field of SDN by addressing key challenges in multi-controller architectures and proposing innovative solutions. The primary contribution is the development of a consensus mechanism based on the Raft algorithm, which ensures stable, consistent, and synchronized network states across distributed controllers. This mechanism supports high throughput, dynamic view changes, and fault tolerance, addressing the limitations of existing approaches. Additionally, the study introduces a heuristic greedy algorithm to optimize the placement of controllers within the network. This algorithm minimizes end-to-end latency and reduces queuing delays between controllers and switches, improving overall network performance.

Another significant contribution is the exploration of the Controller Placement Problem (CPP), with a focus on minimizing latency and optimizing resource usage. The proposed solutions are designed to enhance the scalability, reliability, and performance of SDN-enabled networks, offering practical insights for network administrators and researchers. By addressing these challenges, the study advances the implementation of multi-controller SDN architectures and contributes to the broader understanding of SDN management. The research also provides a dynamic coordination system for workload distribution among controllers, ensuring efficient load balancing and reducing the risk of performance degradation.

The study draws on the benefits of SDN, such as centralized control, programmability, and flexibility, to address the challenges associated with single controller architectures and to enhance network management and performance. By exploring the coordination of workload among distributed SDN controllers, load balancing techniques, and the Controller Placement Problem, the research aims to contribute to the advancement of SDN-enabled networks and their efficient management. The proposed research provides valuable insights into the practical implementation of multiple SDN network architectures, consensus mechanisms, load balancing techniques, and controller placement strategies, with the goal of enhancing the scalability, reliability, and performance of SDN-enabled networks.

Finally, the research introduces a heuristic greedy algorithm designed to address the Controller Placement Problem. It aims to minimize end-to-end latency and reduce maximum latency between controllers and switches. By proposing a specific algorithmic approach, the study contributes to the practical implementation of controller placement strategies within SDN environments. The findings of this research have the potential to significantly improve the performance, reliability, and flexibility of modern networks, making them better suited to meet the demands of emerging applications and technologies.

The contributions of this thesis are:

I. An analysis of the existing mechanisms and protocols for SDN networks

II. A definition of the consensus problem for distributed SDN controllers.

III. An introduction to a novel mechanism leveraging the Raft consensus algorithm to ensure high throughput, dynamic view changes, and fault tolerance in multi-controller SDN setups. The mechanism supports seamless controller synchronization, enabling

efficient network operations even during dynamic changes or failures. Offering dynamic coordination and synchronization system among SDN controllers and switches, focusing on the impact of synchronization rates on network performance. The system ensures consistent and efficient communication between controllers and switches.

IV. A mechanism for controller fail-over. To address the single point of failure issue, a failover mechanism is introduced, enabling controllers to fail gracefully without causing disconnection of switches. This approach ensures continuous network operation and significantly improves fault tolerance.

V. A dynamic coordination and synchronization system among SDN controllers and switches that focus particularly on the impact of the rate of synchronization on the performance of network. Additionally, proposes a dynamic load balancing system that shifts the load across multiple controllers. This prevents any single controller from becoming overwhelmed and ensures efficient resource utilization.

VI. A system that can dynamically shift the load across multiple controllers through switches.

VII. An effective solution that addresses the challenges associated with placing controllers optimally in SDN environments with multiple controllers.

VIII. A heuristic greedy algorithm designed to address the Controller Placement Problem. It aims to minimize end- to-end latency and reduce maximum latency between controllers and switches. By proposing a specific algorithmic approach, the study contributes to the practical implementation of controller placement strategies within SDN environments.

Finally, the presentations and publications that were contributed to the literature as part of this doctoral dissertation are listed below:

Conference Proceedings

1	"Efficient Consensus Between Multiple Controllers in Software Defined Networks (SDN)," in SECURWARE 2022: The Sixteenth International Conference on Emerging Security Information, Systems and Technologies, Lisbon, Portugal, October 2022, pp. 35-40, doi: https://www.thinkmind.org/index.php?view=instance&instance=SECURWARE+2022	[22]
2	S. Lalou, S. Katsikas and G. Spathoulas, "Coordination of Controllers and Switches in Software Defined Networks (SDN) for Multiple Controllers," in SECURWARE 2023: SECURWARE 2023: The Seventeenth International Conference on Emerging Security Information, Systems and Technologies, Porto, Portugal, September 2023, pp. 76-81,	[40]

	doi:https://www.thinkmind.org/index.php?view=instance&instance=SECURWARE+2023	
3	S. Lalou, S. Katsikas and G. Spathoulas, "A Greedy Approach for Controller Placement in Software-Defined Networks for Multiple Controllers", in *ARIA Congress 2024: The 2024 IARIA Annual Congress on Frontiers in Science, Technology, Services, and Applications, Porto, Portugal, July 2024, pp.49-24, doi: https://www.thinkmind.org/library/IARIA_CONGRESS/IARIA_Congress_2024*	[18]

1.5 Dissertation Structure

The rest of this dissertation is organized as follows:

- **Chapter 1**: This chapter introduces the research topic, outlining the problem statement and the significance of the study. It provides an overview of Software Defined Networking (SDN) and the challenges associated with multiple controllers. The chapter also details the contributions of this research and describes the structure of the dissertation.

- **Chapter 2**: This chapter provides a comprehensive background and literature review essential for understanding the foundational concepts and challenges in Software Defined Networking (SDN) and multi-controller architectures. It begins by contrasting traditional networks with SDN, highlighting the transformative role of SDN in decoupling the control plane from the data plane. The chapter delves into the architecture of SDN, detailing its building blocks, including SDN switches and controllers, and explores the communication and coordination mechanisms among multiple controllers. It discusses the benefits of multi-controller setups, various architectures, and the role of the OpenFlow protocol in enabling coordination. The chapter also introduces the consensus problem in distributed systems, focusing on the Raft algorithm, its leader election and log replication processes, and its application in SDNs.

- **Chapter 3**: A thorough literature review examines multi-controller architectures, performance, scalability, fault tolerance, and coordination challenges, as well as the Controller Placement Problem (CPP) and solutions proposed in current research, including dynamic placement and heuristic algorithms. This chapter sets the stage for addressing the complexities of SDN environments and lays the groundwork for the proposed solutions in subsequent chapters.

- **Chapter 4**: This section presents a comprehensive mechanism designed to enhance the performance and reliability of multi-controller SDN setups by addressing key challenges such as high throughput, dynamic view changes, fault tolerance, and controller synchronization. The proposed solution leverages the Raft consensus algorithm to ensure seamless coordination and synchronization among controllers and

switches, enabling efficient network operations even during dynamic changes or failures. A novel fail-over mechanism is introduced to eliminate the single point of failure, allowing controllers to fail without disconnecting switches, thereby ensuring uninterrupted network operations. The chapter includes a detailed performance evaluation and comparative analysis, demonstrating the proposed mechanism's superior efficiency, scalability, and fault tolerance over existing alternatives, such as Paxos-based systems. Additionally, robustness is validated through simulations of various failure scenarios, including multiple Master node failures, showcasing the system's ability to maintain data integrity and operational continuity under challenging conditions. This chapter contributes a robust, scalable, and fault-tolerant framework for optimizing multi-controller SDN environments.

- **Chapter 5**: It explores the design of a system capable of dynamically shifting the load across multiple controllers through switches while ensuring effective coordination among them. The proposed solution introduces a novel coordination mechanism that integrates the RAFT consensus algorithm with OpenFlow's Master/Slave connection model, providing stability, scalability, and consistency in multi-controller SDN environments. A dynamic load balancing system is implemented to redistribute the load across controllers based on real-time metrics such as CPU utilization, memory usage, and active flows, preventing any single controller from being overwhelmed and optimizing resource utilization. The system incorporates a fail-over mechanism to detect controller failures and seamlessly redistribute their responsibilities, ensuring high availability and uninterrupted network operations. Extensive performance evaluations and scalability tests demonstrate the system's ability to maintain low response times, high throughput, and minimal packet loss even under heavy traffic conditions. Additionally, OpenFlow's flow table management is leveraged to dynamically assign switches to the least loaded controllers, optimizing traffic routing and reducing latency. This chapter presents a comprehensive framework for achieving efficient load distribution and robust coordination in multi-controller SDN setups.

- **Chapter 6**: This chapter further delves into the design of a heuristic greedy algorithm aimed at minimizing end-to-end latency and reducing maximum latency between controllers and switches in SDN environments. The proposed solution introduces a novel greedy algorithm that strategically places controllers in proximity to switches, minimizing latency by reducing the physical distance between controllers and their assigned nodes. The algorithm leverages a combination of Euclidean distance and transmission delay to calculate latency, ensuring that controllers are placed in locations that minimize both physical and network-based delays. A dynamic node assignment process is implemented to iteratively assign switches to the nearest available controller, optimizing load distribution and reducing overall network latency. Extensive performance evaluations are conducted using Mininet and Ryu in simulated SDN environments, demonstrating the algorithm's ability to maintain low latency and ensure efficient communication between controllers and switches.

Additionally, a comprehensive latency analysis is performed, examining both controller-node and controller-controller latency, providing valuable insights into the algorithm's effectiveness in optimizing network performance under varying conditions. This chapter presents a robust and efficient approach to latency minimization in SDN environments, enhancing overall network responsiveness and reliability.

- **Chapter 7**: The final chapter summarizes the key findings of the research and discusses their implications for future work. It highlights contributions to SDN coordination and controller placement, offering final thoughts on potential advancements in the field.

Chapter 2: Background

This chapter provides a comprehensive foundation for understanding the key concepts and technologies related to Software Defined Networking (SDN), as well as a critical review of existing research on controller coordination and placement in SDN environments. The chapter is divided into two main sections: Background and Literature Review.

The Background section introduces foundational concepts in SDN, including its architecture, the role of the OpenFlow protocol, and the Raft consensus algorithm. These concepts are essential for understanding the challenges and solutions discussed in the subsequent sections. By combining foundational knowledge with a critical analysis of existing research, this chapter sets the stage for the proposed solutions and methodologies presented in the subsequent chapters of this dissertation.

2.1 Introduction

Software Defined Networking (SDN) is a transformative network architecture that separates the control plane from the data plane, enabling dynamic, manageable, cost-effective, and adaptable network solutions. This separation allows for centralized control and programmability, making SDN particularly suited for high-bandwidth, dynamic applications. The SDN framework is structured into three layers: the Application Plane, Control Plane, and Data Plane, which collectively aim to deliver end-to-end Quality of Service (QoS) by enhancing network flexibility, abstraction, control, management, and programmability [3].

The concept of SDN originated in 2007 at Stanford University, led by Martin Casado, a PhD student, in collaboration with Professors Nick McKeown and Scott Shenker from the University of California, Berkeley. Casado's PhD thesis introduced a centrally controlled, flow-based Ethernet switch known as "Ethane," which laid the groundwork for the OpenFlow Protocol and various control applications that form the core of SDN [6].

Before the advent of SDN, the idea of programmable networks had been explored for years, with researchers advocating for high-speed programmable packet processing. The need for a new protocol and compatible communication nodes arose from the distributed nature of the internet, where devices communicate via specific protocols. SDN addressed these

challenges by allowing external control of communication nodes through flow table abstractions, thus introducing a necessary level of abstraction in network control. This approach also promoted software-based control over entire networks, clarifying the abstraction layers for external software control.

The term SDN was first introduced in an article about the OpenFlow project at Stanford University, published by MIT Technology Review. The pursuit of programmable networks has been ongoing, aiming to simplify the implementation of new network services and facilitate service creation and deployment. Despite being a relatively new concept, many IT experts predict rapid deployment of SDN in the coming years, even as it faces various challenges.

2.1.1 Traditional Networks

Traditional Networking is characterized by two primary aspects: (1) most network functions are implemented on dedicated appliances, and (2) these appliances are based on dedicated hardware. Examples of dedicated appliances include switches, routers, and application delivery controllers. In traditional networking, each switch operates with its own integrated Control and Data planes, forming what is known as a closed system [3].

This architecture is vertically integrated, complicating network management due to the tight coupling of the Control Plane and Data Plane within network devices. Changes to the network often require individual device configurations, making it difficult and error-prone to translate high-level policies into low-level commands, especially in large networks. Studies have shown that network misconfigurations are common, leading to undesirable behaviors. Current architectures struggle to meet future needs due to their lack of global resource visibility and real-time adaptability [8, 9].

2.1.2 Software Defined Networking

SDN is defined as a network architecture where the control plane is decoupled from the forwarding plane, allowing the control plane to manage multiple devices. The Open Networking Foundation (ONF) [4,5], a non-profit organization, developed the OpenFlow standard as the first communication interface between the control and data planes. The central idea of SDN is to centralize control in a software component known as the SDN controller, which manages the forwarding information of the OpenFlow switch while the hardware components handle traffic forwarding according to rules set by the software.

The architecture of SDN devices features the SDN controller as the control intelligence, implemented as a software program. The data plane remains in hardware components such as routers and switches, which forward packets based on instructions from the controller. This separation allows for a logically centralized control plane that provides programmable application programming interfaces (APIs) for managing the physical layer, while the data plane specializes in forwarding packets according to the controller's instructions [4,5].

SDN [2] has gained popularity for managing large-scale networks, including data centers and cloud environments. However, as these networks consist of numerous servers

interconnected with thousands of switches, a single centralized controller may not suffice. Therefore, the next logical step is to develop a logically centralized but physically distributed control plane, benefiting from the scalability and reliability of a distributed architecture while preserving the simplicity of centralized control.

Having multiple controllers that collaborate enhances scalability, persistency, workload sharing, and availability, ultimately leading to more efficient network management.

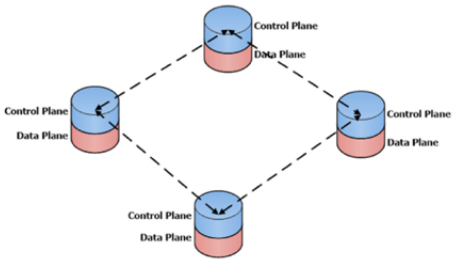

Figure 2:The architecture of traditional network devices.

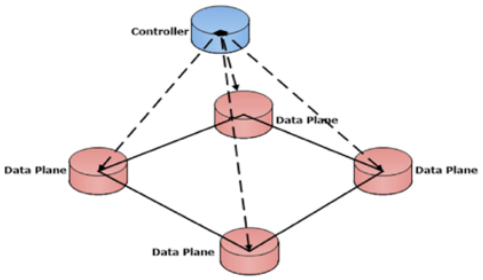

Figure 3:The architecture of Software Defined Networking devices.

2.2 SDN Architecture

Software Defined Networking (SDN) is a revolutionary approach to network management that decouples the control plane from the data plane. This separation allows for centralized control and programmability of the network, enabling more flexible and efficient network management. SDN aims to improve network control by enabling enterprises and service providers to respond quickly to changing business requirements.

SDN architecture consists of three main layers: the application layer, the control layer, and the infrastructure layer. Each layer has distinct functions and interacts with the other layers through well-defined interfaces [1, 4, 8].

Application Layer contains network applications and services that define the behaviour of the network. Examples include intrusion detection systems, load balancers, and firewalls. These applications communicate their requirements to the control layer using northbound APIs [1].

Control Layer houses the SDN controller, which is the brain of the SDN architecture. The controller has a global view of the network and makes decisions about how traffic should be handled. It communicates with the application layer through northbound APIs and with the infrastructure layer through southbound APIs. The controller translates the high-level requirements from the application layer into low-level instructions for the network devices [1].

Infrastructure Layer is also known as the data plane, this layer consists of network devices such as switches and routers that forward traffic based on the instructions received from the control layer. These devices are responsible for the actual data forwarding and do not make any control decisions[1].

SDN provides centralized control of the network, allowing for more efficient management and the ability to implement complex policies and services. Allowing network administrators to write software applications that automate network management tasks. SDN can scale to manage large and complex networks by distributing the control load across multiple controllers.

Additionally, allows for dynamic adjustment of network behaviour to meet changing business requirements. Network administrators can quickly deploy new services and applications without the need for manual configuration of network devices. Enhances vendor interoperability by using open standards and APIs, allowing network devices from different vendors to work together seamlessly.

2.2.1 Building Blocks of SDN

The core components of SDN architecture include the SDN switch, SDN controller, southbound and northbound interfaces, and SDN applications. Below is a detailed description of each.

2.2.2 SDN Switch

In Software Defined Networking (SDN), switches play a critical role as forwarding devices that execute instructions from the SDN controller. Unlike traditional switches, which combine both the control and data planes, SDN switches offload control logic to the SDN controller, enabling dynamic and programmable network management. This separation of control and data planes is a fundamental principle of SDN, allowing for centralized control and flexibility. SDN switches focus on forwarding packets based on flow rules installed by the controller, eliminating the need for independent routing decisions. They interact with the controller via open interfaces, such as the OpenFlow Protocol, which standardizes communication between the control and data planes. Forwarding decisions in SDN switches are flow-based, meaning packets are matched against flow table entries that specify actions (e.g., forward, drop, modify). Each flow is defined by a set of packet field values, such as source/destination IP addresses and port numbers [38, 39].

SDN switches come in various forms, including hardware-optimized switches for high-performance environments, virtual switches for cloud and NFV applications, and hybrid switches for transitioning legacy networks. They maintain flow tables that store forwarding rules, enabling efficient packet processing. When a packet arrives, it is matched against the flow table entries, and if no match is found, the packet is sent to the controller for further processing. This programmability allows network administrators to define custom forwarding rules and policies, supporting dynamic and adaptive network behavior. However, challenges such as flow table size limitations, latency in controller-switch interactions, and compatibility with legacy devices must be addressed to fully leverage the benefits of SDN switches [38, 39].

2.2.3 SDN Controller

The SDN controller serves as the central intelligence of the Software Defined Networking architecture, acting as an intermediary between network devices, such as switches and routers, and the applications that manage network policies and services. It communicates with network devices using southbound APIs (e.g., OpenFlow) and with applications through northbound APIs, enabling centralized control and programmability. This architecture facilitates efficient network management, dynamic configuration, and the implementation of complex policies. The controller performs critical functions, including network discovery to map the topology, traffic optimization to compute optimal paths for data flows, and policy enforcement by installing flow rules on network devices. Additionally, it collects and analyzes network statistics to monitor performance and health [10].

The internal design of an SDN controller is flexible and can be implemented as a single monolithic process, a set of identical processes for load sharing, or a collaborative arrangement of functional components. Controllers can operate on various platforms, including local computing resources or distributed environments like virtual machines in data centers. In multi-controller setups, synchronization is essential to maintain consistent network states and avoid conflicting commands. However, while SDN controllers offer significant benefits, such as improved flexibility and centralized control, they also introduce challenges. Security is a primary concern, as the centralized control plane can be a target for cyberattacks. Scalability is another challenge, as the controller may become a bottleneck as the network grows. Implementing distributed or hierarchical architectures can help address these issues. Additionally, the SDN controller represents a single point of failure, making redundancy mechanisms, such as active-standby or active-active configurations, crucial for ensuring high availability and minimizing the impact of controller failures [10].

2.3 Communication and Coordination among multiple controllers

Effective communication and coordination among multiple controllers are crucial for the success of a multi-controller architecture. Controllers need to share network state information and coordinate their actions to ensure consistent and efficient network management. Various

communication protocols and mechanisms have been proposed to facilitate this coordination, including:

- East-West communication refers to the communication between controllers at the same level in a flat architecture. This communication is essential for sharing network state information and coordinating actions [38].
- North-South communication refers to the communication between controllers at different levels in a hierarchical architecture. This communication allows the central controller to manage the overall network and coordinate the actions of regional controllers [38].
- Distributed databases can be used to store and share network state information among controllers. These databases ensure that all controllers have access to the latest network state information and can make informed decisions [38].

SDN architecture represents a significant shift from traditional network management approaches by decoupling the control plane from the data plane. This separation allows for centralized control, network programmability, and improved scalability, flexibility, and agility. The SDN controller plays a crucial role in managing the network, enforcing policies, and ensuring efficient traffic flow.

While SDN offers numerous benefits, it also introduces new challenges, including security, scalability, reliability, and complexity. Multiple controller architectures and effective communication and coordination mechanisms are essential to address these challenges and ensure the success of SDN deployments. As SDN continues to evolve, it will play a critical role in enabling more efficient and flexible network management, allowing enterprises and service providers to respond quickly to changing business requirements and unlock new opportunities for innovation and growth.

2.3.1 Types of Controllers

The controller is an important feature of the SDN, essential for the network's stability and functionality. It provides a comprehensive view of the SDN to the Application layer or Management Plane. The Control Plane can be managed either by a central controller or multiple controllers. The controller determines the routing path for each new data flow, and its architecture greatly influences the SDN's performance.

There have been proposed a few controller architectures such as Multi-Core, Logically Centralized, and Completely Distributed Controllers. A single controller may struggle to handle a high volume of data flows, leading to potential bottlenecks [11]. Multi-Core Controllers mitigate this issue by distributing the workload across multiple cores, although they can still become single points of failure and have limited scalability, which is a concern for network operators [12].

Logically Centralized Controllers operate with physically distributed controllers that share information to maintain a consistent network view. Examples include ONOS [13, 14]

and OpenContrail [13,15]. These controllers synchronize their states to optimize solutions and ensure a global network perspective. This architecture reduces response times for flow requests and increases the handling capacity, although it requires significant network resources for information exchange.

Completely Distributed Controllers also maintain a consistent global network view by sharing information through synchronization mechanisms. Enhancing scalability involves reducing the synchronization load while keeping information consistent across controllers [11].

The SDN controller, sometimes called the OpenFlow controller, is the central control logic of the network, managing all network activities and resources. It provides a comprehensive view of the network to the application layer and can be managed by either a single or multiple controllers. The routing paths for new flows are calculated by the controller. Researchers highlight the importance of three main interfaces of the controller North-Bound Interface, South-Bound Interface, and East-West Interface for enabling SDN components [11].

2.3.1.1 North-Bound Interface

The North-Bound Interface links the SDN applications in the Application Layer to the controller in the Data Plane Layer. This interface is crucial for supporting various networking applications and developing new ones. It abstracts the low-level commands used by South-Bound Interfaces to program forwarding devices [16].

2.3.1.2 South-Bound Interface

The South-Bound Interface facilitates communication between the SDN controller and the forwarding devices (elements in the Data Plane). It establishes the connection between controllers and switches, handles the setup of switches, and optimizes network management. The most well-known protocol for communication between forwarding elements and controllers is the OpenFlow Protocol [16].

According to researchers, the primary way to classify controllers is based on whether the SDN utilizes a centralized or distributed controller. A Centralized Controller is a single entity that manages all forwarding devices, whilst the Distributed Controller has multiple control elements that are distributed throughout the system.

2.3.2 Single Controller

In a single controller architecture, one SDN controller manages the entire network. This approach has several advantages, including simplicity, ease of management, and a unified view of the network. However, it also has some significant drawbacks, particularly in terms of scalability, reliability, and performance. With a single controller, the entire network is managed from a central point, providing a unified view of the network. This centralized control allows

for more efficient traffic management and the implementation of complex policies [2, 6, 25, 27].

One of the main drawbacks of a single controller architecture is scalability. As the network grows, the single controller may become a bottleneck, unable to handle the increased load. This limitation can lead to performance degradation and increased latency.

Another significant drawback is the issue of reliability. The single controller represents a single point of failure. If the controller fails, the entire network can be disrupted. This vulnerability necessitates robust backup and failover mechanisms to ensure network continuity.

The performance of a single controller can be a concern, especially in large networks. The controller must process a large volume of control messages, which can lead to increased latency and reduced network performance.

2.3.3 Multiple Controllers in SDN

In multi-controller SDN architectures, the control plane is distributed across multiple controllers, each responsible for managing a subset of the network. This approach offers several key benefits, including scalability, as distributing the control plane workload allows multiple controllers to handle larger networks and higher traffic volumes more efficiently. It also enhances fault tolerance, as the failure of one controller does not disrupt the network other controllers can take over its responsibilities, ensuring continuity and minimizing downtime. Additionally, load balancing is achieved by evenly distributing the workload among controllers, preventing any single controller from becoming overwhelmed and optimizing resource utilization. However, multi-controller architectures introduce challenges such as controller synchronization and coordination. Ensuring consistent network states across controllers requires robust consensus algorithms (e.g., Raft, Paxos) and efficient communication protocols (e.g., East-West communication). Furthermore, the Controller Placement Problem (CPP) becomes critical, as the optimal number and location of controllers must be determined to minimize latency and maximize reliability [2, 6, 25, 27, 64].

Multi-controller architectures can be implemented in two main types: flat architecture, where all controllers operate at the same level and manage different parts of the network independently, and hierarchical architecture, where controllers are organized in a tiered structure with a central controller overseeing the overall network and regional controllers handling specific areas. These architectures are particularly beneficial in data centers, where they enable efficient traffic management and resource allocation; in wide area networks (WANs), where they facilitate centralized control and optimization of network resources; and in cloud environments, where they support dynamic and scalable network management, ensuring high availability and reliability [2, 6, 25, 27].

2.3.4 Benefits of Multiple Controllers

The implementation of multiple controllers in Software Defined Networking (SDN) architectures offers significant benefits, enhancing performance, reliability, and scalability in network management. By distributing the control plane workload across multiple controllers, the system can handle increased traffic and control message volumes more efficiently, preventing bottlenecks that occur in single-controller setups. This distribution also improves scalability, as new controllers can be added seamlessly to accommodate network growth, whether through additional devices, increased user demand, or expanded services. Furthermore, the redundancy provided by multiple controllers enhances fault tolerance if one controller fails, others can take over its responsibilities, ensuring uninterrupted network operations. This failover capability is particularly critical in mission-critical environments where uptime is essential [2, 6, 64].

Strategically placing multiple controllers within the network reduces latency and improves overall performance. By positioning controllers closer to the data plane devices they manage (e.g., switches and routers), decision-making processes occur more rapidly, minimizing the time required for control messages to traverse the network. This is especially beneficial for real-time applications like video streaming or online gaming, where reduced latency enhances user experience [7, 54]. Additionally, multiple controllers enable dynamic load balancing, ensuring that no single controller becomes overwhelmed with traffic or control tasks. Load balancing mechanisms can redistribute tasks based on real-time metrics, such as CPU utilization, optimizing resource utilization and network responsiveness [75]. The architecture also supports fault recovery mechanisms, allowing the network to quickly adapt to controller failures and maintain operational stability. Finally, multiple controllers provide flexibility in network management, enabling specialized roles (e.g., handling specific traffic types or enforcing security policies) and tailored solutions for diverse network demands [2, 6].

2.4 Types of Multiple Controller Architectures

Multiple controller architectures can be broadly classified into two types, flat and hierarchical.

1. Flat Architecture: In a flat architecture, all controllers are at the same level and manage different parts of the network independently. They communicate with each other to share network state information and coordinate their actions. This architecture is simple to implement and provides good scalability and reliability.

2. Hierarchical Architecture: In a hierarchical architecture, controllers are organized in a hierarchical structure with a central controller at the top and regional controllers at lower levels. The central controller manages the overall network, while regional controllers manage specific regions. This architecture provides better scalability and allows for more efficient management of large networks.

2.4.1 Communication and Coordination

Effective communication and coordination among multiple controllers are crucial for the success of a multi-controller architecture. Controllers need to share network state information and coordinate their actions to ensure consistent and efficient network management. Various communication protocols and mechanisms have been proposed to facilitate this coordination, including:

1. East-West Communication: East-West communication refers to the communication between controllers at the same level in a flat architecture. This communication is essential for sharing network state information and coordinating actions.

2. North-South Communication: North-South communication refers to the communication between controllers at different levels in a hierarchical architecture. This communication allows the central controller to manage the overall network and coordinate the actions of regional controllers.

3. Distributed Databases: Distributed databases can be used to store and share network state information among controllers. These databases ensure that all controllers have access to the latest network state information and can make informed decisions.

While multiple controller architectures offer significant benefits, they also introduce new challenges, including increased complexity, coordination overhead, and potential inconsistencies. Various solutions have been proposed to address these challenges.

Consistency models ensure that all controllers have a consistent view of the network state. Strong consistency models provide strict guarantees but can introduce latency, while eventual consistency models offer better performance but may allow temporary inconsistencies.

Load balancing mechanisms distribute the control load evenly among controllers, preventing any single controller from becoming overwhelmed. These mechanisms can be based on various factors, including network topology, traffic patterns, and controller capacity.

Fault tolerance mechanisms ensure that the network can continue to operate even if one or more controllers fail. These mechanisms include backup controllers, failover protocols, and redundancy techniques.

2.4.2 Use Cases for Multiple Controllers

Multiple controller architectures are particularly suitable for large-scale networks and scenarios that require high reliability and performance. Large data centers with thousands of servers and network devices can benefit from multiple controllers to manage the network efficiently and ensure high availability. Wide Area Networks (WANs) that span multiple geographical locations can utilize multiple controllers to manage different regions and ensure low-latency communication. Additionally, Network Function Virtualization (NFV)-based services often require orchestration of heterogeneous resources across multiple administrative domains. In such cases, multiple controllers can manage these resources more effectively, ensuring seamless service delivery.

These architectures offer significant benefits in terms of scalability, reliability, and performance. By distributing the control load and providing redundancy, multiple controller architectures address the limitations of single controller systems and enable efficient management of large-scale networks. However, they also introduce new challenges, including increased complexity and coordination overhead. Various solutions, such as consistency models, load balancing mechanisms, and fault tolerance techniques, have been proposed to tackle these challenges and ensure the success of multiple

controller architectures in SDN. As SDN continues to evolve, the development of more sophisticated multiple controller strategies and architectures will be crucial to unlocking the full potential of this transformative networking paradigm.

2.5 OpenFlow Protocol

The OpenFlow protocol is the primary interface connecting the Data-Plane and Control-Plane layers in Software Defined Networking (SDN). Developed by Stanford University, the protocol supports implementation in both hardware and software. By utilizing the OpenFlow protocol, existing hardware can be repurposed to design and test new protocols, facilitating performance analysis.

In SDN, the controller, which is a software component, manages a network of switches to control traffic. Communication between the controller and the OpenFlow switch occurs via the OpenFlow protocol, which also enables the controller to manage the switch. SDN architecture allows for abstract and centralized management of low-level network functions by separating the network logic from data forwarding devices, consolidating control into centralized distributed controllers. However, this separation can lead to scalability and performance issues in large networks with dynamic traffic and topology changes. Research indicates that maintaining a centralized global network view in distributed SDN controllers poses scalability challenges.

OpenFlow is a critical component of the Software Defined Networking paradigm, providing a standardized protocol for communication between the control and data planes of a network. This protocol allows network administrators to configure and manage network devices directly from a centralized controller, [17, 18].

The OpenFlow architecture [17] separates the control and data planes of network devices, with the control plane intelligence centralized in the OpenFlow controller. This separation allows for more fine-grained control over network behaviour, as the controller can dynamically install flow rules on OpenFlow-enabled switches to dictate how traffic should be forwarded.

The OpenFlow protocol itself defines a set of messages and procedures for the controller to communicate with OpenFlow switches. This includes messages for querying switch state, installing flow rules, and modifying switch configurations.

Implementing the OpenFlow protocol provides several key benefits for network management. First, it enables network programmability, allowing administrators to create customized network applications and services on top of the OpenFlow controller. Second, it promotes centralized network control, simplifying the management of complex, heterogeneous network environments [18, 19]. Furthermore, the OpenFlow architecture has been shown to improve network scalability and resilience, with the controller providing optimal routing and failover capabilities [17,18,19].

The adoption of OpenFlow has been a key driver in the growing acceptance of Software Defined Networking in industry [17,18]. This protocol has facilitated a paradigm shift in how networks are designed, managed, and operated, leading to increased flexibility, scalability, and cost-effectiveness.

OpenFlow switches comprise three types of tables:

1. Flow Table: Determines actions on packets and matches incoming packets to specific flows.

2. Group Table: Executes various actions affecting one or more flows.

3. Meter Table: Initiates performance-related actions on a flow.

The OpenFlow protocol provides the SDN South-Bound interface, facilitating communication between the SDN controller (Control-Plane) and SDN switches (Data-Plane). This protocol allows the SDN controller to configure and manage the switches. Compatible with the GENI standard, OpenFlow enables users to create network slices without needing to understand the underlying physical infrastructure. The protocol specifies traffic flow management by allowing the controller to access and manipulate the flow tables of SDN switches, matching each incoming packet against rules and executing corresponding actions. Supported actions include forwarding or dropping packets, rewriting packet headers, and managing VLANs and MPLS tags.

2.5.1 OpenFlow Architecture

The fundamental elements of OpenFlow architecture include:

1. Data-Plane: Constructed with OpenFlow-compliant switches.

2. Control-Plane: Consists of one or more OpenFlow controllers.

3. Secure Control Channel: Connects switches to the Control-Plane.

In the SDN/OpenFlow architecture, forwarding devices handle packet forwarding, while controllers make decisions. OpenFlow-enabled forwarding devices operate based on flow table pipelines, with each flow table entry containing a matching rule, actions for matching packets, and counters for packet statistics [32].

The architecture encompasses three main components:

Switches: Comprising flow tables, communication channels, and the OpenFlow protocol. Each flow table entry includes action fields, and switches can have multiple flow tables, group tables, and OpenFlow channels [32].

Controllers: Update flow entries in flow tables by revising, adding, or deleting them.

Flow Entries: Configured within flow tables and managed by the group table.

Switches communicate with each other through OpenFlow ports over the network. Various dynamic routing schemes, such as DIFF and Software-defined Adaptive Routing (STAR), have been proposed to address issues like flow-table overflow and inefficient bandwidth allocation, enhancing network performance [32].

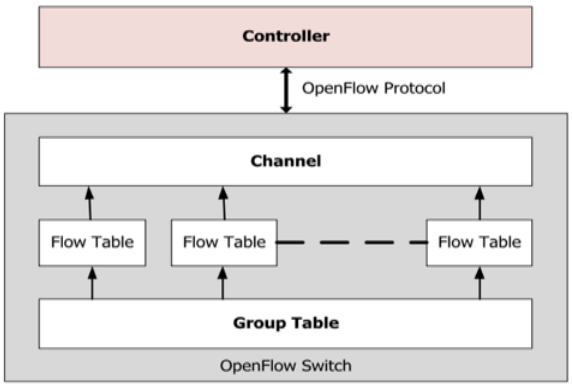

Figure 4:OpenFlow model

2.5.2 OpenFlow Configuration

When a new packet arrives, the lookup process begins immediately. This process starts with the first table and continues until either a match is found in one of the pipeline tables or there is no match. Rules are followed in the natural sequence of table numbers and row order within a flow table. Upon receiving a packet, an OpenFlow switch compares the header fields to the entries in its flow table. If a match is found, the packet is processed according to the corresponding actions in the flow table. If no match is found, the packet is encapsulated in an OpenFlow Packet-In message and sent to the controller. The controller can then install new forwarding rules to handle the packet flow. The controller informs the switch either to create a new entry in the flow table to manage the new flow or to drop the packet [5].

The SDN controller requests configuration details, such as active switch ports and their associated MAC addresses, from the SDN switch by initiating a protocol handshake using an OpenFlow OFPT_FEATURES_REQUEST message. This message informs the controller about the presence of switches in the network. An OpenFlow Packet-In (OFT_PACKET_IN) message is used to forward received data packets to the controller. Conversely, an OpenFlow Packet-Out (OFT_PACKET_OUT) message allows the controller to send a data packet to a switch with specific instructions in the form of an action list [5].

An OpenFlow switch can be configured to operate as a router, switch, firewall, load balancer, or traffic shaper, depending on the rules set by a controller application. There are four possible actions in a flow table: forward the packet to port(s), encapsulate and forward it to the controller, drop the packet, or send it to the normal processing pipeline. The OpenFlow protocol has multiple versions, each introducing new improvements and features [5].

Switches in the SDN context are programmable devices responsible for data forwarding. Unlike traditional switches that include both the control and data planes, SDN switches primarily execute instructions from the controller, acting as simple forwarding devices [5]. The synergy between controllers and switches is critical for ensuring the network behaves as intended. Controllers must provide switches with timely and accurate instructions, while switches must efficiently execute these instructions and provide real-time feedback to controllers.

Coordination between controllers and switches is primarily facilitated through the Southbound Interface (SBI), with OpenFlow being the most commonly used protocol. This section highlights the main elements of coordination. Controllers define flow tables on switches by installing flow rules that dictate how packets are handled. These rules specify matching criteria (e.g., source/destination IP) and actions (e.g., forward, drop, or modify) [5].

Switches periodically send statistics (e.g., packet counts, port status) to controllers. This feedback enables controllers to assess network performance and make informed decisions about traffic engineering and resource allocation. Also, notify controllers about events like link failures, congestion, or unknown packet headers. Controllers then update flow tables or reroute traffic to mitigate issues. The coordination relies on secure channels to prevent malicious actors from intercepting or injecting commands. Encryption and authentication mechanisms are essential to maintaining the integrity of this communication [5].

2.5.3 Coordination between controllers with OpenFlow Architecture

The OpenFlow architecture consists of numerous pieces of OpenFlow-enabled switching equipment which are managed by one or more OpenFlow controllers. It depicts the fundamental concept of the SDN architecture [2]. Network traffic can be partitioned into flows, where a flow could be a Transmission Control Protocol (TCP) connection, packets with the same MAC address or IP address, packets with the same Virtual Local Area Network (VLAN) tag, or packets arriving from the same switch port [6].

An OpenFlow switch contains multiple flow and group tables. Each flow table consists of many flow entries. These are specific to a particular flow and are used to perform packet look-up and forwarding. The flow entries can be manipulated as desired through OpenFlow messages exchanged between the switch and the controller on a secure channel. By maintaining a flow table, the switch can make forwarding decisions for incoming packets by a simple look-up on its flow- table entries. Open-Flow switches perform an exact match check on specific fields of the incoming packets. For every incoming packet, the switch goes through its flow table to find a matching entry. The flow tables are sequentially numbered. The packet- processing pipeline always starts at the first flow table. The packet is first matched against the entries of a flow table. If the packet matches a flow entry in a flow table, the corresponding instruction set is executed. Instructions associated with each flow entry describe packet forwarding, packet modification, group table processing, and pipeline processing [6].

Pipeline-processing instructions enable packets to be sent to subsequent tables for further processing and enable aggregated information (metadata) to be communicated between tables.

2.5.4 Connection Strategy

The first issue is to address the switch-to-controller connection strategy and how switches are connected to SDN controllers. In early OpenFlow version, switches can only attach to one controller. Furthermore, that link is static, meaning that operators have to configure the switch manually when it needs to attach to a new controller. A distributed SDN controllers setup, on the other hand, requires a dynamic connection between switches to controllers. The dynamic connection enables to move a switch from one controller to another controller during a fail-over or load balancing process. Fortunately, there are two options to deploy such flexible switch to controller connection, using the IP alias connection or OpenFlow Master/Slave connection.

In the Master state, the controller has full access to the switch as in the Equal role. When the controller changes its role to Master, the switch changes the other controller in the Master role to have the Slave role. The role change does not affect controllers with the Equal role. The controller receives from switch asynchronous Port- status messages. The controller can send Asynchronous- Configuration messages to set the asynchronous message types it wants to receive. An OpenFlow instance can connect to one or more controllers, depending on the controller connection mode the OpenFlow instance uses ether Single instance in which the OpenFlow instance connects to only one controller at a time. When communication with the current controller fails, the OpenFlow instance uses another controller, or the Multiple instances so it can simultaneously connect to multiple controllers. When communication with any controller fails, the OpenFlow instance attempts to reconnect to the controller after a reconnection interval [6].

2.6 Consensus

Consensus is the central protocol behind services replicated for fault tolerance. Consensus protocols are the foundation for building many fault-tolerant distributed systems and services. Software Defined Networking commonly utilizes a logically centralized control plane to manage the network. However, as networks grow in scale and complexity, relying on a single controller can introduce bottlenecks and single points of failure. To address this, SDN architectures often employ multiple distributed controllers [20, 21].

The introduction of multiple controllers in SDN gives rise to the need for a mechanism to ensure consistency and synchronization of the network state across all controllers. This challenge is what consensus algorithms aim to solve. All controllers must have a unified and up-to-date understanding of the network topology, flow rules, and policies. The system should be resilient to controller failures. In the event of a failure, the remaining controllers should be able to reach a consensus and continue operating without disrupting the network. The control plane should be highly available to avoid network downtime. Consensus algorithms should facilitate seamless failover and recovery. As the network size and the number of controllers increase, the consensus algorithm should maintain performance and avoid becoming a bottleneck [21].

Several challenges complicate the design and implementation of effective consensus algorithms in SDN. Communication delays between controllers can impact the speed and efficiency of consensus. The algorithm must be robust enough to handle various types of controller failures (crash failures, Byzantine failures). The consensus process should not significantly impact the performance of the SDN control plane.

The Raft consensus algorithm is a fault-tolerant protocol designed for maintaining consistency in distributed systems. It achieves this by enabling multiple network nodes to agree on shared states, even when some of the nodes may fail. Raft's architecture is built to be simpler than traditional consensus algorithms like Paxos, offering a more comprehensible framework while preserving robustness, making it widely adopted in distributed systems like cloud storage, databases, and Software Defined Networks (SDNs) [21].

2.6.1 The Consensus Problem in Distributed Systems

In this chapter the Consensus Problem in Distributed Systems is presented. Consensus algorithms solve the challenge of maintaining consistency across a distributed system, where multiple nodes must agree on the same value or state to ensure correct system behaviour. In large-scale distributed systems, individual nodes often operate concurrently and can fail at any time, introducing complexities that make achieving consensus difficult. A reliable consensus algorithm ensures that the system continues functioning correctly even if some nodes experience crashes, network partitions, or delayed communications.

In a distributed system, every node may have different knowledge or data about the network at any given time, and these discrepancies must be reconciled. The goal of consensus is to ensure that every node eventually agrees on the same view of the system. For instance, a consensus algorithm is critical in systems that manage sensitive data (such as financial transactions or SDNs) because any inconsistencies in data can lead to major errors. The consensus algorithm must guarantee safety (no two nodes make conflicting decisions) and liveness (all nodes eventually reach a decision), even in adverse conditions.

2.7 Raft Algorithm

The Raft consensus algorithm was introduced by Diego Ongaro and John Ousterhout in their 2014 paper, "In Search of an Understandable Consensus Algorithm." Raft was designed to be more comprehensible than Paxos, the dominant consensus algorithm at the time, without sacrificing correctness or performance. This simplicity has made Raft highly popular and an essential building block for many distributed systems [20].

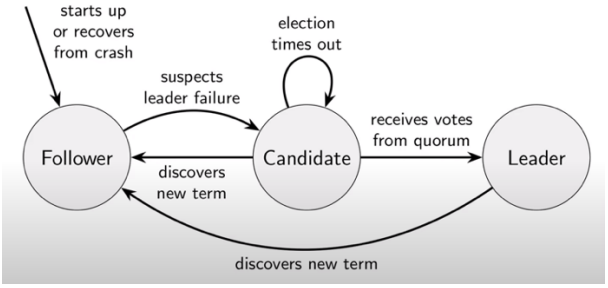

Figure 5: Raft Algorithm

Raft is divided into three primary components:

1. Leader Election: selecting a node to act as the leader.

2. Log Replication: ensuring that all follower nodes replicate the same log entries that the leader appends.

3. Safety: ensuring that committed log entries are not lost, even if the leader crashes.

Raft is based on the concept of a leader node that manages the network's operations. The leader receives commands from clients, appends them to its log, and then replicates these commands to the follower nodes. Follower nodes store a copy of the leader's log and periodically send heartbeat messages to ensure the leader is alive and functioning. If the leader fails, the follower nodes elect a new leader. This clear division of roles makes Raft easier to implement and reason about compared to Paxos, which lacks this structure [20, 21, 24].

2.7.1 Leader Election in Raft

Leader election is one of the most critical aspects of the Raft algorithm. Raft assumes that only one node, the leader, is allowed to propose new log entries at any given time. The other nodes, known as followers, simply replicate the leader’s log. If the leader crashes or becomes unavailable, the system must quickly select a new leader to maintain progress [20, 21, 24].

Raft [20, 21, 24] handles leader election by dividing time into discrete terms. Each term starts with an election. If a follower node does not receive heartbeat messages from the leader within a certain time (called the election timeout), it assumes that the leader has failed and starts a new election. In this election, the follower becomes a candidate and asks the other nodes to vote for it. If the candidate receives votes from the majority of the nodes, it becomes the new leader. Once a leader is elected, it serves until the end of the term unless it fails.

Raft's leader election process is designed to be both simple and efficient. By using a randomized election timeout, Raft minimizes the chances of split votes, where two candidates receive an equal number of votes. This randomization ensures that one candidate is more likely to win quickly, reducing the time spent without a leader.

2.7.2 Log Replication

Once a leader is elected, it starts receiving commands from clients and replicates them to the follower nodes. These commands are appended to the leader's log, which is a sequence of records representing changes to the system's state. The log plays a crucial role in maintaining consistency across the distributed system. Each time a leader receives a new command from a client, it appends the command to its log and sends an AppendEntries message to all followers. The followers append the entry to their logs if it is consistent with the previous entries [20, 21, 24].

A log entry is considered committed when it has been replicated on a majority of the nodes (a quorum). Once a log entry is committed, it is applied to the system's state machine, and the leader informs the client that the operation has been completed. This ensures that even if the leader crashes, the system will not lose committed log entries because they have already been replicated across a majority of nodes.

Raft ensures that logs remain consistent by only allowing leaders to append new log entries. If a follower receives an inconsistent log entry, it will reject it, and the leader will adjust its log to match the follower's state. This mechanism guarantees that all nodes eventually converge to the same log, maintaining system-wide consistency [20, 21, 24].

One of Raft's key guarantees is safety, meaning that no two leaders can ever commit conflicting logs. Raft achieves this by ensuring that only the leader with the most up-to-date log can be elected. During the election process, each candidate must have the latest version of the log, or it will not receive votes from the majority of nodes.

Raft's approach to safety also includes mechanisms for handling network partitions and node failures. For instance, if a network partition causes the leader to lose contact with a majority of the nodes, it will step down and stop processing client requests, preventing it from making decisions without consensus. Similarly, if a node crashes, it can recover by fetching the latest log entries from the leader once it rejoins the network.

Raft's fault tolerance is based on the principle that as long as a majority of nodes are operational, the system can continue functioning. This property is known as quorum-based consensus, which means that a majority of nodes (the quorum) must agree on each decision. Even if minority nodes fail or become partitioned, the quorum can still make progress, ensuring the availability and consistency of the system [20, 21, 24].

2.7.3 Comparison to Paxos

Raft was designed to address some of the complexities of Paxos, a consensus algorithm developed by Leslie Lamport [23]. While Paxos is widely regarded as the gold standard for distributed consensus, it is notoriously difficult to understand and implement correctly. Paxos has a more abstract

and less structured approach to consensus, with no clear separation of phases like leader election and log replication.

In contrast, Raft provides a more modular and understandable framework, with clear roles for leaders and followers and well-defined phases for leader election and log replication. This simplicity does not come at the cost of performance; Raft is comparable to Paxos in terms of latency and throughput while being significantly easier to implement.

Raft's main advantage over Paxos is its readability. By breaking down the consensus problem into leader election, log replication, and safety, Raft makes it easier for developers to reason about the algorithm's behaviour. This clarity has made Raft more popular in industry implementations [23, 25, 26].

2.7.4 Raft in Software Defined Networks (SDNs)

In the context of SDNs, Raft plays a critical role in maintaining a consistent network state across multiple controllers. SDNs decouple the control plane from the data plane, allowing network administrators to manage resources programmatically. By implementing a multi-controller architecture with Raft, SDNs can ensure that all controllers agree on the overall network state, even in the event of failures. This reduces the risk of network outages and improves fault tolerance.

Raft ensures that network changes, such as topology updates or policy modifications, are consistently applied across all controllers, fostering a unified and resilient network. This consistency is crucial in SDNs, where different parts of the network must operate under the same rules to avoid conflicts that could lead to packet loss, misrouting, or instability.

In a multi-controller SDN, Raft ensures that only one controller acts as the leader at any given time. The leader is responsible for handling all updates to the network state, while the other controllers, known as followers, synchronize their state with the leader. This leader-follower model eliminates the possibility of multiple controllers issuing conflicting decisions for the same part of the network.

If the leader fails, Raft's leader election mechanism activates. A follower that times out while waiting for the leader's heartbeat promotes itself to a candidate and solicits votes from other controllers to become the new leader. This process ensures that there is always a functioning leader, allowing the network to remain operational with minimal disruption. Raft's randomized election timeouts help prevent split votes, ensuring that one controller quickly wins the election and reducing the time spent without a leader, which could otherwise delay network updates and traffic management [25].

Raft's log replication feature is vital for maintaining a consistent state across all SDN controllers. Whenever the leader receives a network update such as a change in routing policies or configurations it appends this update to its log and sends the log entry to its followers. Log entries are committed only after a majority (quorum) of controllers acknowledge receipt, ensuring that even if some controllers fail or become temporarily disconnected, the system can continue to operate, and network consistency is maintained [26]. Once a log entry is committed, the leader applies the change to the system's state, and the followers do the same, ensuring that all controllers have an identical view of the network.

Raft's focus on safety ensures that even in the presence of node failures, no conflicting network configurations are applied. This is particularly important in SDNs, where an incorrect routing rule or policy could lead to significant network disruptions. Raft achieves safety through mechanisms such as

leader completeness, which ensures that only a leader with the most up-to-date log can be elected, preventing outdated or inconsistent states from being propagated to the followers.

In SDNs, network outages or controller failures are unavoidable. However, Raft ensures that these failures do not lead to inconsistent states across the network. For example, if a controller crashes and then rejoins the network, it will automatically synchronize its state with the leader by fetching the latest log entries. This self-healing capability allows SDNs to recover quickly from failures, minimizing downtime and maintaining operational continuity.

Chapter3: Literature Review

The Literature Review section analyzes relevant publications on multi-controller architectures, coordination mechanisms, and the Controller Placement Problem (CPP). It highlights the current state of research, identifies gaps in the literature, and discusses the implications of various approaches for improving the scalability, reliability, and performance of SDN-enabled networks.

3.1 Multi-Controller Architectures in SDNs

The use of multiple controllers in SDNs resolves the limitations of single-controller architectures by distributing the control plane across several nodes. This reduces the burden on any single controller and provides redundancy. If one controller fails, others can take over without compromising the functionality of the network. However, the distributed nature of these controllers creates a challenge: how do they ensure that each controller maintains the same view of the network state [40].

In a distributed system like SDN with multiple controllers, consensus algorithms ensure that all controllers can agree on a common state, even in the presence of failures or network partitions. Without consensus, inconsistent states across controllers could lead to conflicts in network configurations, packet forwarding rules, and policies, ultimately causing network instability. Thus, a robust consensus algorithm is critical for ensuring that multiple controllers function as a cohesive unit.

3.2 Performance, Scalability, and Fault Tolerance in SDN

While Raft is highly effective in maintaining consistency across multiple Software Defined Networking (SDN) controllers, it presents certain challenges regarding performance, scalability, and fault tolerance. SDNs require low-latency communication to effectively manage real-time network operations, such as dynamic routing, traffic engineering, and load balancing. The leader-centric model of Raft introduces limitations in terms of latency and throughput, as all changes must be processed through the leader [25]. This can create bottlenecks in high-performance environments, particularly in data centers or cloud computing infrastructures where rapid changes to the network state are frequent.

To address these concerns, various optimizations have been proposed, including Fast Paxos Consensus (FPC) [30] and other enhancements to the Raft algorithm. These optimizations aim to reduce consensus time and improve throughput, making Raft more suitable for high-performance SDN environments. For instance, FPC introduces a multi-phase process comprising Propose, Accept, Update,

and Adjust that handles consensus more efficiently, thereby reducing the overhead associated with traditional Raft implementations [27]. Additionally, hierarchical controller architectures can be employed, where regional controllers serve as leaders for specific portions of the network, while higher-level controllers coordinate between these regions. This multi-tiered approach alleviates the burden on any single controller and allows the system to scale more effectively as the network expands [27].

By localizing consensus decisions, operators can significantly reduce latency and enhance the overall responsiveness of the network. However, it is crucial to carefully consider the trade-offs between consistency, performance, and scalability in high-performance SDN environments. Techniques such as batching, pipelining, and caching mechanisms can further enhance performance by alleviating the load on controllers and minimizing communication overhead [27].

One of Raft's most important contributions to SDN architectures is its ability to handle dynamic network topologies. SDN environments are inherently dynamic, with frequent changes in network configuration, such as adding or removing switches, updating routes, or modifying security policies. Raft's consensus model ensures that these changes are consistently applied, even as the topology evolves [26]. This consistency is critical for maintaining network stability and performance in environments where rapid changes are common, such as data centers, cloud computing infrastructures, and telecommunications networks.

Raft also enhances the fault tolerance of SDNs by ensuring that the control plane remains operational even when some controllers fail or become unreachable due to network partitions. For example, if a network partition isolates a set of controllers, the controllers in the larger partition can continue to make decisions and apply changes, ensuring that the majority of the network remains operational. Once the partition is resolved, the isolated controllers rejoin the network and synchronize their state with the leader, ensuring that no conflicting states arise [26]. This fault tolerance is essential in large-scale SDNs, where network availability and reliability are paramount. By using Raft, these systems can achieve high availability and ensure that network management operations continue even in the presence of failures.

Several alternative consensus algorithms and systems can be integrated into SDN environments to improve performance, scalability, and fault tolerance. EPaxos (Egalitarian Paxos) [88] is a variant of the Paxos consensus algorithm that improves performance by eliminating the need for a single leader in the consensus process. In traditional Paxos, a leader is responsible for proposing values, which can create bottlenecks and increase latency. EPaxos allows any node to propose values, significantly enhancing the system's responsiveness and throughput. This egalitarian approach helps distribute the workload more evenly across nodes, reducing the likelihood of bottlenecks. Key features of EPaxos include leaderless consensus, which enables multiple nodes to propose values simultaneously, and reduced latency, which is crucial for real-time applications. EPaxos is particularly beneficial in SDN environments where rapid decision-making is required. The ability to achieve consensus without a single point of failure enhances the resilience and performance of the control plane, making it suitable for dynamic and large-scale networks.

Flexible Paxos [90] is an extension of the original Paxos consensus algorithm that enhances performance and scalability by relaxing some of the constraints associated with quorum intersections. In traditional Paxos, a strict quorum requirement must be met for any decision to be made, which can lead to inefficiencies, especially in large distributed systems. Flexible Paxos allows for more adaptable quorum configurations, enabling different sets of nodes to participate in the consensus process based on current network conditions. This flexibility can lead to reduced latency and increased throughput, making it particularly suitable for high-performance applications. Key features of Flexible Paxos

include quorum intersection, which allows overlapping quorums to minimize the number of messages required for consensus, and adaptability, which enables the algorithm to adjust to changing network conditions. These features make Flexible Paxos especially well-suited for high-performance SDN environments, where low latency and high throughput are critical. In such environments, rapid changes to network state and configuration are common, and the ability to achieve consensus quickly and efficiently is essential for maintaining optimal network performance [90].

SWIM (Scalable Weakly-consistent Infection-style Process Group Membership Protocol) [87] is a protocol for maintaining membership and detecting failures in distributed systems. It ensures that controllers are aware of the status of other controllers, improving fault tolerance and dynamic topology management. SWIM's lightweight and scalable design makes it suitable for large-scale SDN environments.

Zookeeper [83] provides a distributed coordination service that can be used for leader election and state synchronization. It ensures consistent network states across controllers, enhancing fault tolerance in SDN environments.

Etcd [84] is a distributed key-value store that uses the Raft consensus algorithm. It can be integrated into SDN for distributed state management, providing a reliable and consistent way to store and retrieve network state information [84].

Consul is a service mesh solution that offers distributed consensus and can be used for SDN controller coordination. It provides features such as service discovery, health checking, and key-value storage, making it a versatile tool for managing distributed SDN architectures. Consul's ability to handle dynamic network changes and its fault-tolerant design make it particularly useful in large-scale SDN environments [85].

Chord is a distributed hash table (DHT) protocol that can be employed for fault-tolerant and scalable SDN controller placement. It provides a decentralized approach to resource location within a network, ensuring that controllers can efficiently locate and manage network resources. Chord's scalability and fault tolerance make it a valuable addition to SDN architectures, especially in environments with a large number of nodes [86].

LogCabin is a distributed key-value store that implements the Raft consensus algorithm. It provides a simple and efficient way to achieve consensus in distributed systems, ensuring strong consistency and fault tolerance. LogCabin is particularly useful for applications that require reliable state management across distributed nodes, making it a suitable choice for SDN environments where maintaining a consistent network state is critical [91].

Optimizations such as Fast Paxos Consensus (FPC) [30] and hierarchical controller architectures can significantly enhance the effectiveness of Raft in managing the control plane of large-scale, dynamic networks [26, 27]. Fast Paxos is an advanced variant of the traditional Paxos consensus algorithm designed to improve performance and reduce latency in distributed systems. Unlike the original Paxos, which relies on a leader to propose values and requires multiple communication rounds to reach consensus, Fast Paxos introduces a more efficient multi-phase process that allows for faster decision-making. In FPC, any node can propose a value, enabling a more egalitarian approach that mitigates the bottlenecks associated with a single leader. This is achieved through a mechanism that allows nodes to propose values directly to a quorum of acceptors, thereby reducing the number of message exchanges required to achieve consensus. The protocol operates in two main phases: the Propose phase, where a proposer sends a value to a quorum of acceptors, and the Accept phase, where

the acceptors respond with their decisions. By allowing proposals to be accepted in a single round under certain conditions.

3.3 SDN Consensus multiple controllers platforms

The use of a single SDN controller offers flexibility and efficiency in network management but leads to problems such as single points of failure and scalability issues. Existing studies propose multiple SDN controller architectures to address the above issues. The synchronization of the network state information among controllers is a critical problem, known as the controller consensus problem. To synchronize the network information between controllers, a proper consistency model should be chosen. Strong consistency and eventual consistency are two consistency models commonly used in distributed systems.

Many papers have proposed systems for SDN. The authors of [30] introduce a multi-controller SDN architecture, which employs a Fast Paxos based Consensus (FPC) algorithm to handle the consensus between multiple SDN controllers. The concept of leader election is also supported, as three roles, namely Listener, Proposer, and Chairman are applied to different controllers. The proposal was tested on a small-scale multi-controller architecture. In this research the evaluation of the performance was conducted in an ODL (OpenDaylight) Clustering on 3 node clustering. The FPC is composed of four phases, namely Propose, Accept, Update, and Adjust. A controller maintains a table that records its current controller state.

Hyperflow [31], [32] is described as a flat design. It is a distributed event based control plane for OpenFlow. HyperFlow is logically centralized but physically distributed: it provides scalability while keeping the benefits of network control centralization. By passively synchronizing network wide views of OpenFlow controllers, HyperFlow localizes decision making to individual controllers, thus minimizing the control plane's response time to data plane requests. HyperFlow is resilient to network partitioning and component failures. It also enables interconnecting independently managed OpenFlow networks, an essential feature missing in current OpenFlow deployments.

The network is structured into several domains, where each domain is controlled by a controller situated within its own local network view. Controllers communicate with others through their East-westbound interfaces to get the global view of the network. Each controller only processes flow requests sent from the switches that belong to its local domain. Network events (e.g., flow information, routing information) are transmitted based on specific publish/subscribe mode among controllers [31].

Onix uses Paxos Consensus [33]. It is a multiple SDN controller architecture that provides a control application with a set of general APIs to facilitate access to the network state. It adopts a distributed architecture approach to offer the programmatic interface for the upper control logic and uses Network Information Base (NIB) to maintain the global network state. ONIX was created to address the limitations of traditional network control planes, which were typically tightly coupled with specific hardware and lacked flexibility in terms of programmability. As networks grew larger and more complex, traditional control planes struggled to keep pace, leading to difficulties in managing resources, balancing loads, and implementing policies consistently. ONIX aimed to overcome these limitations by providing a centralized, programmable control plane with a focus on scalability, fault tolerance, and resilience. This approach allowed network operators to programmatically control the network from a central point, making it easier to manage policies, configurations, and network states.

In Onix, the controller stores network information in key value pairs by utilizing the NIB, which is the core element of the model. It synchronizes the network state by reading and writing to the NIB, thus it provides scalability and resilience by replicating and distributing the NIB across multiple NIB instances. Once a change of a NIB on one Onix node occurs, the change will be propagated to other NIBs to maintain the consistency of the network.

ONOS (Open Network Operating System) [29] is an open-source Software Defined Networking (SDN) operating system developed by the Open Networking Foundation (ONF) to facilitate the creation of scalable, high-performance network infrastructure for telecommunication and data center environments. It provides the control plane for SDN, managing network components such as switches and links, and running software programs or modules to deliver communication services to end hosts and neighboring networks. ONOS applications and use cases often consist of customized communication routing, management, or monitoring services for SDNs [29]. Designed with a focus on high availability, scalability, and performance, ONOS enables centralized control of network devices while abstracting lower-level networking functions to provide a simplified and programmable interface for network operators. Its architecture is distributed and fault-tolerant, emphasizing high availability and reliability; multiple ONOS instances can work together in a cluster, ensuring the network continues to function smoothly even if some controllers fail. This fault tolerance is achieved through ONOS's use of the Raft consensus algorithm for leader election and state replication, ensuring consistency and reliability across controller instances [28, 29]. The design of ONOS emphasizes scalability to accommodate the increasing number of devices and network connections in large networks, allowing it to manage thousands of devices and connections efficiently. This makes ONOS ideal for Internet service providers (ISPs), telecommunications companies, and data centers that demand high-performance networking solutions, and it is capable of handling large-scale applications, including network slicing, traffic engineering, and security services.

Kandoo [34] Kandoo is a hierarchical SDN controller framework focused on scalability. The root controller communicates with multiple domain controllers to get the domain information, but the domain controllers do not contact each other. It doesn't directly use a traditional consensus algorithm like Raft or Paxos; instead, it creates a two-layer hierarchy. The lower layer of local controllers operates independently for local events, minimizing the need for global coordination. For global events, Kandoo relies on a centralized root controller, which reduces the need for distributed consensus but introduces single-point dependency.

DISCO (DIStributed multi-domain SDN COntroller) [35] is a distributed SDN controller scheme, implemented on top of Floodlight. It was introduced to partition a wide area network (WAN) into constrained overlay networks. A DISCO controller manages its own domain and communicates with other controllers via a lightweight and manageable control channel to provide end-to-end network services.

Akka [36] is a toolkit used in ODL Clustering and is responsible for communication and notification among controllers. In the default clustering scheme, switches connect to all controllers, and these controllers coordinate among themselves to choose a master controller. The ODL uses the Raft algorithm to reach controller consistency. The Raft consensus algorithm periodically elects a controller as a leader controller, and all data changes will be sent to the leader controller to handle the update. Akka is a distributed systems toolkit rather than a standalone SDN controller. It uses an actor model to simplify building distributed applications and can implement consensus algorithms. Akka provides tools for clustering and state replication (e.g., through Akka Distributed Data) and supports consensus

indirectly by allowing developers to implement Raft or Paxos if needed, although it doesn't enforce consensus on its own.

LogCabin is a distributed key-value store that implements the Raft consensus algorithm. It is designed to provide a simple and efficient way to achieve consensus in distributed systems. LogCabin focuses on providing strong consistency and fault tolerance while being easy to use and deploy. It is particularly useful for applications that require reliable state management across distributed nodes. Key features of LogCabin include its Raft implementation, which ensures that all nodes maintain a consistent state, and its key-value store interface, which makes it accessible for various applications. LogCabin is suitable for SDN environments where maintaining a consistent view of the network state is critical. Its use of the Raft algorithm ensures that network controllers can achieve consensus efficiently, which is essential for managing dynamic network configurations [89].

OpenDaylight (ODL) is a collaborative project under the Linux Foundation that provides a modular open-source platform for SDN and Network Functions Virtualization (NFV). It supports multiple controller instances and uses the Akka toolkit for clustering and communication among controllers. ODL employs the Raft consensus algorithm to ensure consistency across its distributed architecture, allowing for scalable and resilient network management [74].

CORD (Central Office Re-architected as a Data Center) is an open-source project that aims to transform traditional telecommunications networks into cloud-based architectures. It utilizes a distributed control plane that can manage multiple controllers, enabling efficient resource allocation and service delivery across various network domains. CORD emphasizes scalability and flexibility, making it suitable for large-scale deployments.

Ryu [66] is an open-source SDN framework that provides a component-based architecture for building network applications. It supports multiple controllers and offers a RESTful API for communication between controllers and network devices. Ryu can be integrated with various consensus algorithms to ensure consistency and reliability in distributed environments.

Floodlight [92] is an open-source SDN controller that supports multiple controller instances and provides a REST API for application development. It can be extended with various consensus mechanisms to manage state synchronization among controllers, making it suitable for large-scale network deployments.

Pox is a Python-based SDN controller that allows for the development of custom network applications. While it is primarily designed for educational purposes, it can be adapted to support multiple controllers and integrate with consensus algorithms for state management [92].

As observed from the existing distributed controller architectures, the problem of single point of failure of the SDN controller was solved using multiple distributed controllers. The Copycat [37] project is an advanced, feature-complete implementation of the Raft consensus algorithm that diverges from recommendations. For instance, Raft dictates that all reads and writes are executed through the Master Controller (node), but Copycat's Raft implementation supports per-request consistency levels that allow clients to sacrifice linearizability and read from followers. Similarly, the Raft literature recommends snapshots as the simplest approach to log compaction, but Copycat prefers an incremental log compaction approach to promote more consistent performance throughout the lifetime of a cluster. Copycat's Raft implementation extends the concept of sessions to allow server state machines to publish events to clients.

3.4 Coordination in Multi-Controller Software Defined Networking

SDN aims to offer easier management and faster through dynamically customizing the operation of a network with the combination of a centralized controller and programmable network devices. When SDN is deployed in large-scale networks, they may consist of multiple controllers or different administrative vendors. Hence, how multiple controllers and switches coordinate is a critical issue. OpenFlow [63], is one of the mostly representative protocol for SDN, carries the message between an SDN controller and the underlying network infrastructure. One of the most fundamental features of the OpenFlow protocol is the packet-in message.

If a packet arrived at a OpenFlow based switch does not match any forwarding rule, the packet can be configured as the packet-in message to be forwarded to the controller for corresponding policing processing. The use of multiple controllers is an approach that offers greater availability but it also introduces a new problem regarding the coordination between multiple controllers and switches. Insufficient coordination may result in sub optimal network performance. In large data centres, the traffic patterns are usually unpredictable due to elastic resources and flexible services. For example, if the switch to controller map-ping configuration is static, some switches can generate a larger number of packet-in messages than other switches of the network.

This condition implies that the corresponding controller, which these switches are mapped to will become overloaded, while other controllers will remain under-utilized. On the other hand, in multi-controller SDN [7] each controller is responsible for a set of switches (domain). A controller can be master, equal, or slave, where the first two types can process the flow requests from the switches and install the forwarding rules in the switches. A slave controller can only read the switch flow table, but cannot update it. Each switch can have multiple equal and slave controllers, but only one master controller. Furthermore, a master controller for a specific switch can be slave controller for another one, and whenever a master controller fails, a slave or local controller can request (via OpenFlow role-request message) to become the new master of the affected switches. The multi-controller paradigm is shown to improve many aspects of SDN, but it presents many challenges, especially for controllers' utilization when switch-controller assignments are static. The load of a controller is mainly caused by the processing of the packet-in messages sent from the switches, and due to network dynamics, the number of these messages vary both regionally and temporally. As a result of these variations, some controllers will be over committed, while some others will be underutilized. This leads to domain failure (and multi-domain failure), or network under utilization.

3.5 Coordination Mechanisms for Distributed SDN Controllers

The coordination among controllers is a major issue with several protocols proposed thus far [2]. OpenDaylight [74] and ONOS [29], two state-of-the- art controller implementations, rely on RAFT and Anti-entropy protocols for disseminating coordination messages among controllers. Each controller is responsible fora part of the network only, commonly referred to as the controller's domain. The messages disseminated by a controller to the other controllers convey its view on the state of its domain (e.g., available links and installed flows). The composition of these messages allows the controllers to synchronize and agree on the state of the entire network. Also the authors of [75] proposed a load re balancing method based on switch migration mechanism for clustered controllers. They also using the OpenFlow 1. 3.

The multiple controllers use JGroups [75] to coordinate actions for switch migration. The whole network is divided into several groups and each group has a controller cluster set up. In this paper we

will discuss the available mechanisms that offer multiple controller coordination and network synchronization.

Distributed SDN controller deployments require a coordination protocol among controllers. To address the coordination, synchronization and performance challenge different systems and approaches have been introduced. ONOS [29] stands for Open Network Operating System, uses RAFT, and provides the control plane for a software-defined network (SDN). It manages network components, such as switches and links, and runs software programs or modules to provide communication services to end hosts and neighboring networks. The most important benefit of an operating system is that it provides a useful and usable platform for software programs designed for a particular application or use case. ONOS applications and use cases often consist of customized communication routing, management, or monitoring services for Software Defined Networks .

Devoflow [76] actually reduce the overhead of the control plane by offloading the controller by delegating some work to the forwarding devices and enable a cluster of controller nodes to achieve distributed control plane.

Onix [33], Kandoo [34], and HyperFlow [31, 32] use this approach to achieve a control plane with high scalability and reliability. ElastiCon [77] supports for elastic behaviour which in-creases or decreases the number of controllers based on load estimates of control plane.

The “Doing It Fast And Easy” (DIFANE) [78] approach examines the scalability issues that arise with OpenFlow in large networks and with many fine-grained flow entries. Scalability concerns can be classified by (1) the number of flow entries inside the switches and (2) the load on the controller that is caused when many flows have to be installed simultaneously. DIFANE installs all forwarding information in the fast path, i.e., in TCAM, in a few selected switches, called authority switches. This is achieved by wildcard match fields and the intelligent distribution of flow table entries on the authority switches. The other switches forward traffic that cannot be resolved by their own flow table towards authority switches. The authors show the applicability of DIFANE in large networks by evaluating their proposal on various metrics, such as the number of required TCAM entries, packet loss caused by failures, etc.

Some recent works propose to reduce the overhead of control traffic by strategically placing the controllers in the network [79] or by finding the appropriate forwarding paths for load balancing on control traffic. Eventual consistency, where the controllers coordinate periodically rather than on demand basis, is another way to reduce control overheads.

Levin et al. [80] showed that certain network applications, like load-balancers, can work around eventual consistency and still deliver acceptable performance. This would require some additional effort to be made to ensure that conflicts such as for-warding loops, black holes and reachability violation are avoided. The authors [79] studied the problem of finding the optimal synchronization rates among controllers in a distributed eventually-consistent SDN system. They considered two different objectives, namely, (i) the maximization of the number of controller pairs that are consistent, and (ii) the maximization of the performance of applications which may be affected by the synchronization decisions, as highlighted by emulations on a commercial SDN controller.

3.6 Challenges in Coordination

Despite its benefits, coordination between controllers and switches presents several challenges, including latency, scalability, fault tolerance, reliability, and compatibility with legacy systems. The

centralized nature of SDN controllers can lead to bottlenecks, especially in large-scale deployments where numerous switches communicate simultaneously. This can degrade overall network performance, necessitating techniques like controller clustering and distributed control planes, which introduce additional complexity [41].

The reliance on a centralized controller means that its failure can render the entire network inoperable, highlighting the need for robust fault tolerance mechanisms. Redundancy through backup controllers can mitigate this risk, but it adds complexity to the network architecture. Reliable communication links between controllers and switches are crucial, as disruptions can lead to significant outages [42]. Techniques such as heartbeat signals and state synchronization can help maintain reliability [43].

Many organizations operate hybrid networks that include both SDN and legacy systems, complicating integration efforts. Legacy devices may not support OpenFlow or other SDN standards, leading to inefficiencies and increased operational costs [44]. To facilitate integration, organizations may need to implement translation layers or hybrid controllers, which can introduce additional overhead and complexity [45].

3.6.1 OpenFlow Protocol in Coordination

The OpenFlow architecture consists of various OpenFlow-enabled switching devices managed by one or more OpenFlow controllers. Network traffic is partitioned into flows, which can be defined by criteria such as Transmission Control Protocol (TCP) connections, packets with the same MAC or IP address, or packets with the same Virtual Local Area Network (VLAN) tag [6].

An OpenFlow switch contains multiple flow and group tables, with each flow table consisting of numerous flow entries specific to a particular flow. These entries are used for packet lookup and forwarding and can be manipulated through OpenFlow messages exchanged between the switch and the controller over a secure channel. By maintaining a flow table, the switch can make forwarding decisions for incoming packets through a simple lookup process.

When processing incoming packets, OpenFlow switches perform exact match checks on specific fields. The packet-processing pipeline begins at the first flow table, where the packet is matched against flow entries. If a match is found, the corresponding instruction set is executed, which may include packet forwarding, modification, or further processing through subsequent tables [6]. This structured approach allows for efficient packet handling and enables the communication of aggregated information (metadata) between tables.

3.6.2 Connection Strategy

A key issue in SDN is the switch-to-controller connection strategy, particularly how switches connect to SDN controllers. In earlier versions of OpenFlow, switches could only attach to one controller, and this connection was static, requiring manual configuration for any changes. However, a distributed SDN controller setup necessitates a dynamic connection that allows switches to move between controllers during failover or load balancing processes.

Two options facilitate this flexible switch-to-controller connection: the IP alias connection and the OpenFlow Master/Slave connection. In the Master state, the controller has full access to the switch,

while other controllers may hold the Slave role. The role change does not affect controllers with an Equal role. The controller receives asynchronous port-status messages from the switch and can send Asynchronous-Configuration messages to specify the types of messages it wishes to receive [46].

An OpenFlow instance can connect to one or more controllers, depending on the connection mode. In the Single instance mode, the OpenFlow instance connects to only one controller at a time. If communication with the current controller fails, it can switch to another controller. In the Multiple instances mode, the OpenFlow instance can connect to multiple controllers simultaneously, attempting to reconnect to any controller after a specified reconnection interval [46].

This dynamic connection strategy enhances the flexibility and resilience of SDN architectures, allowing for better load balancing and fault tolerance. By enabling switches to seamlessly transition between controllers, the network can maintain operational continuity even during controller failures or high traffic loads. This adaptability is crucial for modern SDN deployments, where network conditions can change rapidly and require immediate responses.

3.7 Controller Placement Problem (CPP)

The Controller Placement Problem (CPP) addresses the strategic positioning of Software Defined Networking (SDN) controllers within a network, crucial for efficient network man- agement. It impacts network performance in latency, reliability, scalability, and resource usage. SDN architecture, separating control and data planes, enhances scalability and programmability compared to traditional architectures. Determining the optimal number and placement of controllers is a key challenge, with network latency being a primary performance factor.

The rise of Software Defined Networking (SDN) has been prompted by the escalating demands for new networks and the expansion of Internet coverage. As contemporary requirements surpass the limitations of traditional networks, SDN emerges as a promising paradigm. It addresses these challenges by separating the control and data planes, enabling the programmability of network configuration. The pivotal factor influencing SDN deployment and applications is the strategic placement of controllers. Within the SDN architecture, the use of single or multiple controllers is instrumental in achieving programmable, flexible, and scalable configurations.

SDN is a network architecture that separates the control plane from the data plane. The control plane is responsible for making decisions and configuring the network, while the data plane is responsible for forwarding network traffic. Multiple controllers are often used in SDN networks to distribute the control plane workload and improve network scalability. However, in a multi-controller SDN network, additional delays can be introduced due to communication between controllers.

Transmission delay between controllers can cause delays in network decision making, leading to reduced network performance and efficiency. In a multi-controller SDN network, each controller must have up-to-date information about the network topology and state, which is communicated through inter-controller messaging. Excessive transmission delay between controllers and switches can lead to increased packet delay, jitter, and even packet loss, leading to reduced network performance and efficiency.

The CPP [47] in Software Defined Networking (SDN) involves strategically deploying controllers within the network, impacting various performance metrics like availability, fault tolerance, and convergence time. SDN, with its separated control and data planes, offers solutions to network challenges, making CPP a critical factor for optimizing network performance. Early research by Heller

et al. [47] framed CPP as a facility placement problem, highlighting its complexity. Subsequent studies focused on minimizing propagation delay by determining optimal controller locations and quantities.

Techniques like K-center and Multiobjective Optimization Controller Placement (MOCP) [53] address different aspects such as network reliability, load balance, and latency. CPP requires careful planning considering factors like network topology, latency, and controller placement to ensure optimal performance and resilience. Our approach offers a heuristic implementation of CPP, aiming to minimize the number of controllers while addressing latency concerns, ultimately enhancing network responsiveness and robustness in the face of failures.

3.7.1 General formulation of CPP

The primary network components for an SDN-enabled network are switches, controllers, and the links that join them. Therefore, the network is often modeled as an undirected graph:

$$G = (SEC) \quad (1)$$

where S represents the set of switches and E is the set of physical links among those switches or controllers, and C be the controllers to be placed in the network. The switches and the controllers are represented as follows: S = s1, s2, s3, ... sn, where n denotes the number of switches in the network, and C = c1, c2, c3, ... ck, where k denotes the number of controllers located in the network. Here, Pi = p1, p2, p3, ... pk is one possible placement of the k controllers. The relationship between switches and controllers is represented by the condition that the set of controllers C is a subset of the set of switches S, denoted by $C \subseteq S$. This indicates that controllers are located within the network's switches. The shortest path between a switch $s \in S$ and a controller $c \in C$ indicates the minimum number of links (or hops) required to reach the controller from the switch [48].

3.7.2 Multiple controller placement problem

The challenge of deploying multiple controllers arises from the limitations of a single centralized controller, which struggles to keep pace with the growing demands of expanding networks and applications. Relying on a solitary controller creates a potential single-point bottleneck and failure risk, as it shoulders the entirety of control activities. Consequently, any network failure could severely impact overall network performance. Thus, the adoption of a multi-controller approach emerges as a viable solution for large-scale Software-Defined.

Networking (SDN), particularly concerning the scalability of the control plane. The inadequacy of deploying a single controller for managing extensive networks, advocating instead for the deployment of multiple controllers. However, effectively placing multiple controllers remains a complex task. To optimize network scalability and minimize latency, particularly in larger networks, leveraging multiple controllers to manage control plane traffic is deemed most effective. Two prevalent architectures for multi-controller setups are flat architecture and hierarchical architecture [48]. Deploying multiple controllers in a large-scale SDN environment aims to reduce latency, distribute controller workload, and optimize various network performance indicators related to controller placement. Consequently, an SDN controller can oversee multiple Network Operating Systems (NOS) [64]. While a single controller suffices for small networks like those in data centers, the adoption of multi-controller deployment is increasingly favored to bolster the scalability and stability of Wide Area

Networks (WANs). Dhar et al. [66] underscore the necessity of deploying multiple controllers to maintain scalability and reliability in large SDN setups.

3.7.3 Greedy Heuristic Algorithm in CCP

A greedy heuristic algorithm is a specific type of greedy algorithm that employs a heuristic approach to make decisions based on the best available information at each step. Unlike traditional algorithms that may explore all possible solutions to find the optimal one, greedy heuristic algorithms focus on making the most advantageous choice at each stage without considering the broader implications of that choice. This approach is particularly useful in complex problems where finding an exact solution is computationally infeasible. Greedy heuristics are often employed in optimization problems, where the goal is to find a satisfactory solution quickly, even if it may not be the absolute best [69].

In the context of the controller placement problem in Software Defined Networking (SDN), greedy heuristic algorithms can be particularly effective due to their ability to make rapid, locally optimal decisions. The process typically begins with the selection of the first controller's location. A common heuristic might involve choosing the location with the highest number of connected switches, as this would maximize the immediate control over the network. Alternatively, the algorithm could prioritize locations based on the highest traffic load, ensuring that the most heavily used parts of the network are managed effectively [70, 72].

Once the first controller is placed, the algorithm evaluates potential locations for the next controller. This evaluation is based on immediate benefits, such as minimizing the maximum distance to the switches or balancing the load across the network. For example, if the first controller is placed in a central location, the next controller might be placed in a more peripheral area to ensure that all switches are within an acceptable distance, thereby reducing latency. The algorithm may also consider factors such as redundancy and fault tolerance, ensuring that if one controller fails, others can still manage the network effectively. This iterative process continues, with each subsequent controller being placed based on the current state of the network and the previously made decisions. The algorithm may stop once all controllers are placed or when a predefined number of controllers is reached, ensuring that the network remains manageable and efficient [71].

One of the primary advantages of using greedy heuristic algorithms for controller placement in SDN is their computational efficiency. Greedy algorithms typically exhibit lower time complexity compared to other optimization techniques, such as dynamic programming or exhaustive search methods [68]. This efficiency is particularly crucial in large-scale networks, where the number of potential controller placements can be vast and the computational resources may be limited. By focusing on local optimization, greedy algorithms can quickly converge on a solution without the need for extensive calculations. This rapid convergence is essential in environments where network conditions can change dynamically, requiring quick adaptations.

Additionally, greedy algorithms are relatively simple to implement and understand, making them accessible for network engineers and researchers who may not have extensive backgrounds in algorithm design. The straightforward nature of greedy heuristics allows practitioners to quickly adapt and modify the algorithm to suit specific network requirements or constraints. This adaptability is particularly valuable in dynamic environments, where network conditions may change frequently, necessitating rapid adjustments to controller placements. Furthermore, the ease of implementation allows for quick

prototyping and testing, enabling network operators to evaluate different strategies without significant overhead [70].

3.8 Controller Placement: A Review of Solutions from Current Research

Latency holds significant importance in Software Defined Networking (SDN) due to the frequent interaction between switches and controllers. Existing solutions for the CPP typically prioritize minimizing both propagation delay and controller processing delay. Regarding propagation delay, CPP resembles the facility location problem.

Heller et al. [47] were pioneers in CPP research, aiming to minimize worst-case delay while proving its NP-Hard complexity. Since then, numerous researchers have proposed diverse CPP solutions, extending optimization objectives from original switch-controller delay to inter-controller delay, controller capacity, and cost considerations.

Zhu et al. [49] focus on minimizing propagation delay between switches and controllers, formulating CPP as a control plane delay minimization problem and introducing a new algorithm based on clustering and Dijkstra's algorithm. For controllers with limited capacities, Yao et al. [51] define a capacitated CPP and develop an efficient algorithm for optimizing propagation delay.

In another study [46], the authors address controller placement under dynamic network traffic by integrating the controller module placement algorithm with a dynamic flow management algorithm, albeit suitable for small-scale networks only. Tanha et al. [53] concentrate on CPP within Software Defined Wide Area Network (SDWAN), introducing a clique-based approach from graph theory for polynomial-time solution derivation. In subsequent works [48, 54] CPP is formulated considering switch-to-controller delay, controller-to-controller delay, load balancing, and link utilization rate as objectives.

Wang et al. [55] identify that a critical difficulty in SDN is selecting suitable locations for controllers to reduce the latency between controllers and switches. The CPP described a few of the performance factors that were taken into consideration, including control plane overhead, latency, load imbalance, cost, and connection. They use the controller-to-node latency (propagation, queuing, and processing delay) as a crucial performance parameter.

Mamushiane et al. [56] extended and used a facility location approach known as Partition Around Medoids (PAM) with propagation latency to determine the optimum places to put SDN controllers. They suggested using the Silhouette and Gap Statistics algorithms to decide how many controllers to deploy in a wide-area network.

Rasol et al. [57] assess the Joint Latency and Reliability-aware Controller Placement (LRCP) optimization model. With the help of alternate backup channels, LRCP gives network administrators a variety of options for balancing the reliability and latency trade-offs between controllers and switches. This study suggests the Control Plan Latency (CPL) metric, the sum of average switch-to-controller latency and the average inter-controller latency, in order to evaluate the controller placements offered by LRCP and determine how effective they are in an actual controller deployment.

In each link failure state, Fan et al. [58] further take into account the number of control path reroutings and the worst-case latency between the controller and the switch. To solve the problem, they offer a heuristic approach based on particle swarm optimization. The suggested algorithm's usefulness

is demonstrated by the numerical results. Additionally, it demonstrates that in the majority of link failure conditions, the suggested technique may ensure the latency and reliability of the control layer.

The authors in [59] present the Resilient Controller Placement (RCP) algorithm. The objective of this approach is to to minimize the average latency between all switches and the appropriate controllers in the event of a single broken link.

The latency of each path is made up of the latency of the primary path plus the average of any potential backup paths that might be available in the event of a single link failure. Chen et al. [59] present that the network is separated into many sub-networks, and the essential performance metric is the latency between the controller and switch. Liao et al. [62] can be used to determine the latency model in this study or a reliable CPP, Singh et al [67]. suggest a Varna Based Optimization (VBO) to guarantee that it reduces the overall average latency. Their results demonstrate that the proposed VBO algorithm outperforms other effective heuristic algorithms for the Reliability-aware CPP (RCPP), such the Particle Swarm Optimization PSO. PSO and Teachning Learning-Based Optimization (TLBO) and their experimental results show that TLBO performs better than PSO for publicly accessible topologies.

In [54], reducing network latency between controllers and switches is crucial for SDN performance. The study introduces a Controller Node Partitioning Algorithm (CNPA) to minimize end-to-end latency. By partitioning the network and deploying controllers strategically, the aim is to decrease latency between controllers and switches in SDN-enabled wide-area networks.

3.9 Dynamic Controller Placement

Dynamic controller placement in Software Defined Networking (SDN) is essential for effective management of network resources, particularly in response to fluctuating traffic patterns and network conditions. Greedy heuristic algorithms can be integrated with monitoring tools that continuously assess metrics such as link utilization, latency, packet loss, and node availability. By leveraging real-time data, these algorithms can make informed decisions about controller locations and traffic routing [71, 68].

For instance, if a specific area of the network experiences a surge in traffic, the algorithm may deploy an additional controller in that region to reduce latency and enhance responsiveness [70]. Additionally, greedy algorithms can adapt routing paths when network topology changes occur, such as when a link goes down or becomes congested, quickly identifying alternative paths to minimize performance impact. This capability is particularly valuable in SDN, where centralized control allows for rapid reconfiguration of routing tables [71].

Moreover, by continuously evaluating the load on different paths, greedy algorithms help distribute traffic evenly across the network, preventing any single path from becoming a bottleneck. This dynamic placement of controllers not only maintains high performance in unpredictable environments but also allows networks to scale efficiently, deploying additional controllers as demand increases without necessitating a complete redesign of the network architecture.

Chapter 4: Mechanism Design for High Throughput and Fault Tolerance in SDN Controllers

This chapter addresses how can a mechanism be designed to support high throughput, dynamic view changes, fault tolerance, and controller synchronization in multiple SDN controller setups. The proposed mechanism, built on the RAFT consensus algorithm, aims to provide a scalable and efficient solution for managing multiple SDN controllers.

4.1 Implementation of Proposed mechanism

The proposed mechanism introduces an innovative framework that utilizes a sophisticated network of multiple controllers, all operating in conjunction with the Raft consensus algorithm. This design is specifically engineered to facilitate the seamless connection and coordination of several distributed Software Defined Networking (SDN) controllers, which serve a critical role as backup controllers in the event of a failure. The implementation of multiple controllers not only enhances the system's resilience but also enables efficient data load sharing. This is particularly beneficial when a single controller becomes overwhelmed by a high volume of flow requests, as the workload can be distributed across the network of controllers. Overall, our approach is designed to significantly reduce latency, enhance scalability, and improve fault tolerance, thereby providing a higher level of availability in SDN deployments.

As illustrated in Figure 7, the proposed mechanism comprises a collection of independent controllers, referred to as nodes, each of which is responsible for storing the necessary data within its own memory. Each controller is assigned a unique identifier, ensuring that it can be distinctly recognized within the network. In our implementation and testing scenarios, we utilized a variety of nodes, each operating in different states to demonstrate the flexibility and robustness of the system.

Each controller can exist in one of three distinct states: Master, Candidate, or Worker. When a node is in the Master state, it takes on the responsibility of managing and controlling the network. In this capacity, the Master node is also capable of processing data and disseminating update information to the other controllers within the network.

In contrast, a node in the Candidate state has the ability to send and receive data to and from other nodes. A Candidate node that possesses updated data has the potential to transition into the Master state if the current Master node experiences a failure. This dynamic allows for a smooth transition of control and ensures that the network remains operational even in the face of unexpected disruptions.

Finally, a node in the Worker state operates in a more passive role, primarily receiving data from either Master or Candidate nodes without actively participating in the decision-making processes. This division of roles among the nodes contributes to the overall efficiency and reliability of the network.

In the event that a Master node fails, the Raft election process is promptly initiated to elect a new Master node, thereby mitigating the risks associated with a single point of failure. During this election process, a node in the Candidate state is selected to assume the role of the new Master node, while the previous Master node transitions to the Candidate state. This systematic approach ensures that the system maintains its stability and fault tolerance, allowing it to continue functioning effectively even when faced with challenges. Through these mechanisms, our proposed solution enhances the resilience and performance of SDN deployments, ultimately leading to a more robust networking environment.

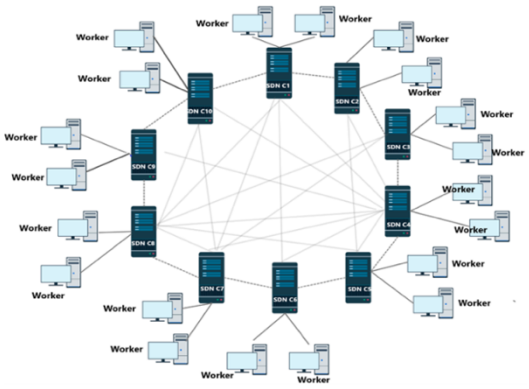

Figure 6: Proposed System

In the proposed framework, inputs are introduced into the network of multiple Software Defined Networking (SDN) controllers by client nodes. When a new piece of data is generated by one of these nodes, it is forwarded to the Master Controller in accordance with the Raft consensus protocol. Upon receiving a series of information packets, the Master Controller logs this data and replicates it across all the controllers within the mechanism, ensuring that each controller stores the information in its memory.

Once the Master Controller has gathered a sufficient amount of data, it initiates a broadcast process to disseminate this information to all nodes in the network. This broadcast ensures that all controllers are updated with the latest information. Initially, the data stored in the Master Controller's memory is marked as "not read," indicating that it has yet to be acknowledged by the other nodes. The Master node then sends the data to all Candidate nodes, which in turn forward the information to their neighboring Worker nodes. Throughout this process, the Master node actively monitors whether all Candidate nodes have successfully received and stored the new data, thereby ensuring that each of them maintains an updated memory.

Each Candidate node is responsible for recording the newly received data and also for monitoring its own data records to prevent any duplicates from being stored. Following this, each Candidate node sends the data to its attached Worker nodes. The same systematic approach is employed to guarantee the successful delivery of data to all nodes within the network.

Once all nodes have successfully forwarded the data, the mechanism transitions into an "OK" state, signifying that all controllers have identical data stored in their memory. However, if a controller receives new data, it exits the "OK" state, prompting the need to send this new information to the other controllers and Worker nodes in the network, following the previously described procedure.

At the core of the proposed mechanism is the Raft node, which is deployed alongside each SDN controller within the overall SDN architecture. Each node maintains a set of records in its memory that adhere to a specific structure, consisting of two variables: "data" and "send." The "data" variable holds the information that is to be exchanged among the nodes, while the "send" variable is a Boolean flag that indicates whether a particular record has been successfully forwarded to the network. Each node is uniquely identified by an ID, ensuring clear differentiation among them.

The Raft consensus algorithm plays a crucial role in coordinating the sharing of information between nodes. It establishes the framework for creating a group of controllers, referred to as candidates, and outlines the necessary processes for electing one of these controllers as the Master node.

The Master node is responsible for managing the data flow within the mechanism and leading the group when it is active.

In the event that the Master node fails, a new Master node must be elected through the established mechanism. A specific time frame is defined during which the Master node sends a message to the other controllers. If this message does not arrive within the allotted time, a Controller node may send a message requesting to assume the role of the Master node. In response, the other controllers will evaluate this request, and the designated node will be appointed as the new Master node.

To maintain consistency, stability, and availability within the network, the nodes operate based on several key processes:

- The Read Process: This process involves reading from the mechanism's memory and checking the records to verify whether they have been successfully distributed to other nodes.
- The Send Process: This process is responsible for sending data to other controllers within the network.
- The Send-to-All Process: This process iteratively sends data to all neighboring controllers, ensuring that information is disseminated throughout the network effectively.

Through these processes, the proposed mechanism ensures that the network of SDN controllers operates efficiently, maintaining a high level of consistency and availability even in the face of potential failures.

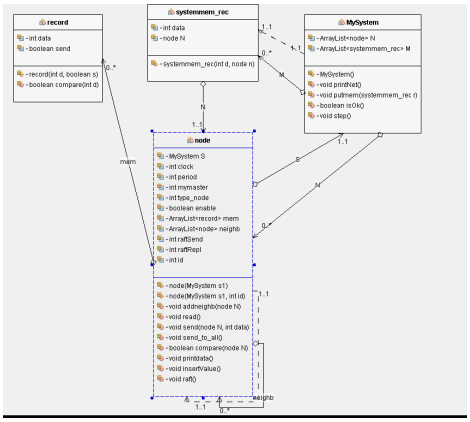

Figure 7:Proposed System UML

4.2 Performance Evaluation of Proposed System

To thoroughly assess the effectiveness of the proposed mechanism, a comprehensive network simulation was undertaken, specifically designed to evaluate its performance under a variety of conditions. This evaluation was not merely a superficial analysis; it involved a detailed comparison of the proposed mechanism against existing alternatives, focusing on several critical performance metrics. These metrics included consensus time, distribution time, data access time, and the presentation of test results. By examining these factors, the research aimed to provide a clear understanding of how well the proposed design performs in real-world scenarios and to highlight its advantages over other

mechanisms currently in use. The insights gained from this evaluation are crucial for understanding the practical implications of the proposed mechanism in operational environments.

The performance evaluation also sought to identify potential areas for improvement and optimization within the proposed mechanism. By systematically analyzing the results of the simulations, the research aimed to uncover any weaknesses or limitations that could hinder the mechanism's effectiveness. This thorough approach ensures that the proposed solution is not only theoretically sound but also practically viable, capable of meeting the demands of modern network management. The findings from this evaluation will serve as a foundation for future enhancements and refinements, ultimately contributing to the development of more robust and efficient network management solutions.

4.3 Experimental Setup

The experimental setup is meticulously detailed in Table 1, which outlines the primary simulation parameters that guided the testing process. The simulations were executed on a robust system equipped with an AMD Ryzen 5, 4500 U CPU, which provided the necessary computational power to run complex simulations efficiently. A Java program has been created [Appendix A1] that simulates a distributed system with a leader election and data propagation mechanism.

The primary objective of this experimental setup was to measure the time required for the core functionalities of the proposed design, ensuring that a comprehensive picture of its performance was captured. By establishing a controlled environment for the simulations, the research aimed to minimize external variables that could skew the results, thereby enhancing the reliability of the findings.

To achieve this, an experimental simulation was initiated that focused on the time response of the algorithm under specific conditions. This approach allowed for the collection of valuable data on the operational efficiency of the proposed mechanism. The careful selection of parameters and the systematic execution of the simulations were critical in ensuring that the results accurately reflected the performance of the proposed design. The insights gained from this experimental setup are essential for understanding the practical implications of the proposed mechanism and for guiding future research efforts in the field of network management.

Parameters	**Value**
Consensus algorithm	Raft
Data	In all node types
Number of controllers	10
Number of Workers per Controller	2
Test Enviroment	Windows
Hardware	AMD Ryzen 5, 4500 U
Compiler	NetBeans 8.2
Code	Java

Table 1:Simulation Parameters

4.4 Failure Scenario Simulation

In the initial testing phase, a failure scenario was simulated where the proposed algorithm was executed for a duration of 100 seconds. This scenario involved a mechanism with 30 controllers, comprising 10 Master nodes and 20 Worker nodes. During the simulation, data transfer between nodes was closely monitored to ensure accuracy and reliability. A critical aspect of this test was the intentional failure of a Master node approximately 20 seconds into the simulation. This failure was strategically chosen to evaluate the mechanism's ability to maintain stability and mitigate the impact of a single point of failure. By simulating this failure, the research aimed to observe how quickly and effectively the system could adapt to changes and continue functioning without significant disruption.

The results of this failure scenario simulation provided valuable insights into the resilience of the proposed mechanism. Observing the system's response to the failure allowed for an assessment of its robustness and adaptability in real-time situations. The ability to maintain operational continuity in the face of failures is a crucial aspect of modern network management, and the findings from this simulation underscore the importance of designing mechanisms that can withstand unexpected disruptions. This research contributes to the broader understanding of fault tolerance in network systems and highlights the significance of developing solutions that prioritize stability and reliability.

4.4.1 Extended Testing with Node Failures

To evaluate the proposed approach, we have run an ex- periment to assess the time response of the algorithm. We first run an experimental simulation of a failure scenario in which the proposed algorithm is executed for 100 sec for a mechanism with 30 controllers, 10 of which run as Master nodes and 20 run as Worker nodes. We check that data is being transferred correctly between nodes. In this scenario we assume that at a specific time point, around 20 sec after the start, the master node fails. The main objective is to maintain mechanism stability at all times and avoid the effects of a single point of failure.

```
Create a system with 9 controllers , 1 master and 2 workers per controller
The system is the following:
Node 0 is  master  and has Neigboors
            1 2 3 4 5 6 7 8 9 10 11
Node 1 is  controller  and has Neigboors
            0 2 3 4 5 6 7 8 9 12 13
Node 2 is  controller  and has Neigboors
            0 1 3 4 5 6 7 8 9 14 15
Node 3 is  controller  and has Neigboors
            0 1 2 4 5 6 7 8 9 16 17
Node 4 is  controller  and has Neigboors
            0 1 2 3 5 6 7 8 9 18 19
Node 5 is  controller  and has Neigboors
            0 1 2 3 4 6 7 8 9 20 21
Node 6 is  controller  and has Neigboors
            0 1 2 3 4 5 7 8 9 22 23
Node 7 is  controller  and has Neigboors
            0 1 2 3 4 5 6 8 9 24 25
Node 8 is  controller  and has Neigboors
            0 1 2 3 4 5 6 7 9 26 27
Node 9 is  controller  and has Neigboors
            0 1 2 3 4 5 6 7 8 28 29
Node 10 is  worker  and has Neigboors
            0
Node 11 is  worker  and has Neigboors
```

```
            0
Node 12 is  worker  and has Neigboors
            1
Node 13 is  worker  and has Neigboors
            1
Node 14 is  worker  and has Neigboors
            2
Node 15 is  worker  and has Neigboors
            2
Node 16 is  worker  and has Neigboors
            3
Node 17 is  worker  and has Neigboors
            3
Node 18 is  worker  and has Neigboors
            4
Node 19 is  worker  and has Neigboors
            4
Node 20 is  worker  and has Neigboors
            5
Node 21 is  worker  and has Neigboors
            5
Node 22 is  worker  and has Neigboors
            6
Node 23 is  worker  and has Neigboors
            6
Node 24 is  worker  and has Neigboors
            7
Node 25 is  worker  and has Neigboors
            7
Node 26 is  worker  and has Neigboors
            8
Node 27 is  worker  and has Neigboors
            8
Node 28 is  worker  and has Neigboors
            9
Node 29 is  worker  and has Neigboors
            9
Node 30 is  worker  and has Neigboors

time=0
Node 0 send data:204 to 1
Node 0 send data:204 to 2
Node 0 send data:204 to 3
Node 0 send data:204 to 4
Node 0 send data:204 to 5
Node 0 send data:204 to 6
Node 0 send data:204 to 7
Node 0 send data:204 to 8
Node 0 send data:204 to 9
Node 0 send data:204 to 10
Node 0 send data:204 to 11
Node 1 receive data:204 from 1
Node 1 send data:204 to 0
Node 1 send data:204 to 2
Node 1 send data:204 to 3
Node 1 send data:204 to 4
Node 1 send data:204 to 5
Node 1 send data:204 to 6
Node 1 send data:204 to 7
Node 1 send data:204 to 8
Node 1 send data:204 to 9
Node 1 send data:204 to 12
Node 1 send data:204 to 13
Node 2 receive data:204 from 2
Node 2 send data:204 to 0
```

Node 2 send data:204 to 1
Node 2 send data:204 to 3
Node 2 send data:204 to 4
Node 2 send data:204 to 5
Node 2 send data:204 to 6
Node 2 send data:204 to 7
Node 2 send data:204 to 8
Node 2 send data:204 to 9
Node 2 send data:204 to 14
Node 2 send data:204 to 15
Node 3 receive data:204 from 3
Node 3 send data:204 to 0
Node 3 send data:204 to 1
Node 3 send data:204 to 2
Node 3 send data:204 to 4
Node 3 send data:204 to 5
Node 3 send data:204 to 6
Node 3 send data:204 to 7
Node 3 send data:204 to 8
Node 3 send data:204 to 9
Node 3 send data:204 to 16
Node 3 send data:204 to 17
Node 4 receive data:204 from 4
Node 4 send data:204 to 0
Node 4 send data:204 to 1
Node 4 send data:204 to 2
Node 4 send data:204 to 3
Node 4 send data:204 to 5
Node 4 send data:204 to 6
Node 4 send data:204 to 7
Node 4 send data:204 to 8
Node 4 send data:204 to 9
Node 4 send data:204 to 18
Node 4 send data:204 to 19
Node 5 receive data:204 from 5
Node 5 send data:204 to 0
Node 5 send data:204 to 1
Node 5 send data:204 to 2
Node 5 send data:204 to 3
Node 5 send data:204 to 4
Node 5 send data:204 to 6
Node 5 send data:204 to 7
Node 5 send data:204 to 8
Node 5 send data:204 to 9
Node 5 send data:204 to 20
Node 5 send data:204 to 21
Node 6 receive data:204 from 6
Node 6 send data:204 to 0
Node 6 send data:204 to 1
Node 6 send data:204 to 2
Node 6 send data:204 to 3

…..

Node 0 is controller had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895

true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true)
Node 1 is master is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (112 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (459 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (224 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)
Node 2 is controller is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)
Node 3 is controller is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)
Node 4 is controller is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)
Node 5 is controller is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)

Node 6 is controller is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)
Node 7 is controller is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)
Node 8 is controller is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)
Node 9 is controller is enable had data:(204 true) (125 true) (735 true) (254 true) (557 true) (791 true) (605 true) (270 true) (706 true) (341 true) (379 true) (30 true) (155 true) (738 true) (181 true) (402 true) (504 true) (751 true) (818 true) (171 true) (595 true) (133 true) (224 true) (711 true) (677 true) (56 true) (112 true) (239 true) (863 true) (187 true) (343 true) (99 true) (259 true) (592 true) (893 true) (365 true) (301 true) (42 true) (459 true) (859 true) (746 true) (15 true) (262 true) (465 true) (600 true) (941 true) (841 true) (423 true) (620 true) (910 true) (278 true) (51 true) (506 true) (178 true) (257 true) (615 true) (938 true) (659 true) (755 true) (828 true) (450 true) (412 true) (723 true) (524 true) (820 true) (67 true) (303 true) (119 true) (469 true) (680 true) (838 true) (949 true) (606 true) (922 true) (734 true) (307 true) (388 true) (221 true) (803 true) (707 true) (778 true) (396 true) (414 true) (87 true) (895 true) (663 true) (431 true) (915 true) (785 true) (241 true) (78 true) (59 true) (107 true) (407 true) (868 true) (916 true) (959 true)

Figure 8: System components data

To further investigate the robustness of the proposed mechanism, additional tests were conducted where newly elected Master nodes experienced failures at various time interval specifically at 20 seconds, 30 seconds, and 60 seconds. This extended testing was crucial for understanding how the system would respond to multiple failures in quick succession. Throughout these tests, the read and write performance across all nodes was closely monitored, regardless of their state as Master, Controller, or Worker. This comprehensive monitoring was essential to assess the overall resilience of

the network under failure conditions, ensuring that any weaknesses or areas for improvement in the proposed mechanism could be identified.

The system's failover mechanism, allows the election of a new master if the current one fails. The final output indicates the status of each node, showing which are enabled and have data, suggesting they are operational and ready for tasks. The system is designed with redundancy and resilience, allowing multiple controllers to take over the master role, which is crucial for maintaining availability. Additionally, the presence of data across nodes implies effective information management, while the architecture supports scalability by enabling the addition of more controllers and workers to handle increased workloads. Overall, the results demonstrate a robust distributed system with a clear hierarchy and failover mechanisms, ensuring continued functionality in the event of node failures and facilitating efficient data processing and management for applications requiring high availability and reliability.

The findings from these extended tests provided a deeper understanding of the proposed mechanism's capabilities in handling dynamic and challenging environments. By simulating multiple failures, the research was able to evaluate the system's ability to recover and maintain data consistency across all nodes. This aspect is particularly important in distributed systems, where the integrity of data is paramount. The insights gained from this extended testing phase will inform future iterations of the proposed mechanism, guiding enhancements that can further improve its resilience and performance in real-world applications.

4.4.2 Data Consistency and Node Election

In the conducted tests, it was ensured that all nodes, except for the original Master node and its connected Worker nodes, maintained consistent data after the initial Master node failure. This consistency is vital for the integrity of the network, as it ensures that all controllers have access to the same information, thereby reducing the risk of discrepancies that could lead to further failures. Following the failure of the Master node, a new Master node was elected through a well-defined process, and this election process was repeated multiple times throughout the experiment. The ability to elect a new Master node efficiently is crucial for maintaining network stability and performance, as it allows the system to recover quickly from failures and continue operating smoothly.

To further investigate the robustness of the proposed mechanism, additional tests were conducted where newly elected Master nodes experienced failures at various time intervals specifically at 20 seconds, 30 seconds, and 60 seconds. This extended testing was crucial for understanding how the system would respond to multiple failures in quick succession. The timeline of time events, ranging from time 0 to time100, illustrates a sequence of operations and state changes within the system. Key events include node failures, such as Node 0 being disabled at time 20, prompting Node 1 to assume the master role, followed by Node 1's failure at time 31, leading to Node 2 becoming the new master at time 32. This extended testing was crucial for understanding how the system would respond to multiple failures in quick succession. Throughout these tests, the read and write performance across all nodes was closely monitored, regardless of their state as Master, Controller, or Worker. This comprehensive monitoring was essential to assess the overall resilience of the network under failure conditions, ensuring that any weaknesses or areas for improvement in the proposed mechanism could be identified.

As demonstrated in the experiments, there are two changes in the master state. Initially, node 0 'fails,' and using the Raft algorithm, node 1 becomes the master controller. Subsequently, node 1 'fails,' and node 2 takes over as the master. Additionally, all nodes, except for those of the initial master and its workers, have the same data since the master 'failed' and node 2 assumed the master role.

```
Create a system with 9 controllers , 1 master and 2 workers per controller
The system is the following:
Node 0 is  master  and has Neigboors
        1 2 3 4 5 6 7 8 9 10 11
Node 1 is  controller  and has Neigboors
        0 2 3 4 5 6 7 8 9 12 13
Node 2 is  controller  and has Neigboors
        0 1 3 4 5 6 7 8 9 14 15
Node 3 is  controller  and has Neigboors
        0 1 2 4 5 6 7 8 9 16 17
Node 4 is  controller  and has Neigboors
        0 1 2 3 5 6 7 8 9 18 19
Node 5 is  controller  and has Neigboors
        0 1 2 3 4 6 7 8 9 20 21
Node 6 is  controller  and has Neigboors
        0 1 2 3 4 5 7 8 9 22 23
Node 7 is  controller  and has Neigboors
        0 1 2 3 4 5 6 8 9 24 25
Node 8 is  controller  and has Neigboors
        0 1 2 3 4 5 6 7 9 26 27
Node 9 is  controller  and has Neigboors
        0 1 2 3 4 5 6 7 8 28 29
Node 10 is  worker  and has Neigboors
        0
Node 11 is  worker  and has Neigboors
        0
Node 12 is  worker  and has Neigboors
        1
Node 13 is  worker  and has Neigboors
        1
Node 14 is  worker  and has Neigboors
        2
Node 15 is  worker  and has Neigboors
        2
Node 16 is  worker  and has Neigboors
        3
Node 17 is  worker  and has Neigboors
        3
Node 18 is  worker  and has Neigboors
        4
Node 19 is  worker  and has Neigboors
        4
Node 20 is  worker  and has Neigboors
        5
Node 21 is  worker  and has Neigboors
        5
Node 22 is  worker  and has Neigboors
        6
Node 23 is  worker  and has Neigboors
        6
Node 24 is  worker  and has Neigboors
        7
Node 25 is  worker  and has Neigboors
        7
Node 26 is  worker  and has Neigboors
        8
Node 27 is  worker  and has Neigboors
        8
Node 28 is  worker  and has Neigboors
        9
Node 29 is  worker  and has Neigboors
        9
Node 30 is  worker  and has Neigboors
```

```
time=0
time=1
time=2
time=3
time=4
time=5
time=6
time=7
time=8
time=9
time=10
time=11
time=12
time=13
time=14
time=15
time=16
time=17
time=18
time=19
time=20
Node 0 is disable
Node 1 change to master
time=21
time=22
time=23
time=24
time=25
time=26
time=27
time=28
time=29
time=30
time=31
Node 0 is disable
Node 1 is disable
Node 2 change to master
time=32
time=33
time=34
time=35
time=36
time=37
time=38
time=39
time=40
time=41
time=42
time=43
time=44
time=45
time=46
time=47
time=48
time=49
time=50
time=51
time=52
time=53
time=54
time=55
time=56
time=57
time=58
time=59
```

```
time=60
Node 0 is disable
Node 1 is disable
time=61
time=62
time=63
time=64
time=65
time=66
time=67
time=68
time=69
time=70
time=71
time=72
time=73
time=74
time=75
time=76
time=77
time=78
time=79
time=80
time=81
time=82
time=83
time=84
time=85
time=86
time=87
time=88
time=89
time=90
time=91
time=92
time=93
time=94
time=95
time=96
time=97
time=98
time=99
Node 0 is controller   had data:
Node 1 is controller   had data:
Node 2 is master  is enable   had data:
Node 3 is controller  is enable   had data:
Node 4 is controller  is enable   had data:
Node 5 is controller  is enable   had data:
Node 6 is controller  is enable   had data:
Node 7 is controller  is enable   had data:
Node 8 is controller  is enable   had data:
Node 9 is controller  is enable   had data:
Node 10 is worker  is enable   had data:
Node 11 is worker  is enable   had data:
Node 12 is worker  is enable   had data:
Node 13 is worker  is enable   had data:
Node 14 is worker  is enable   had data:
Node 15 is worker  is enable   had data:
Node 16 is worker  is enable   had data:
Node 17 is worker  is enable   had data:
Node 18 is worker  is enable   had data:
Node 19 is worker  is enable   had data:
Node 20 is worker  is enable   had data:
Node 21 is worker  is enable   had data:
Node 22 is worker  is enable   had data:
```

```
Node 23 is worker  is enable   had data:
Node 24 is worker  is enable   had data:
Node 25 is worker  is enable   had data:
Node 26 is worker  is enable   had data:
Node 27 is worker  is enable   had data:
Node 28 is worker  is enable   had data:
Node 29 is worker  is enable   had data:
Node 30 is worker  is enable   had data:
```

Figure 9:Master Controller Failure

The research highlights the importance of data consistency in distributed systems, particularly in the context of network management. Ensuring that all nodes have access to the same information not only enhances the reliability of the system but also facilitates effective decision-making processes. The election of a new Master node is a critical component of this process, as it directly impacts the network's ability to respond to failures and maintain operational continuity. The findings from this aspect of the research underscore the need for robust mechanisms that can efficiently manage node elections and ensure data consistency, ultimately contributing to the overall resilience of the network.

4.4.3 Network Overhead and Performance Metrics

The proposed mechanism demonstrated a significant reduction in network overhead while ensuring proper network operation, even in the event of a controller failure. This reduction in overhead is particularly important in large-scale networks, where excessive overhead can lead to performance bottlenecks and decreased efficiency. Additionally, the mechanism provided synchronization among controllers, ensuring that all network controllers had access to the same data. This synchronization is essential for maintaining a cohesive network environment, as it allows for seamless communication and coordination among nodes. The mechanism's reliability, scalability, fault tolerance, and interoperability were also highlighted, particularly in scenarios involving Master node failures, where a new controller was promptly selected through the Raft consensus algorithm.

The implications of reduced network overhead are significant, as they contribute to improved overall performance and efficiency in network operations. By minimizing the resources required for communication and coordination among nodes, the proposed mechanism allows for more effective utilization of available bandwidth and processing power. This efficiency is particularly beneficial in environments where resources are constrained, enabling organizations to achieve optimal performance without incurring excessive costs. The research findings emphasize the importance of designing mechanisms that prioritize both performance and resource efficiency, ultimately leading to more sustainable and effective network management solutions.

4.5 Comparative Analysis with Existing Mechanisms

During the tests we compare the proposed mechanism and the Paxos-based mechanism [31], in terms of consensus time, distribution of normalized consensus time, and data access time. The consensus time is the time that elapses from when a transaction is created to when the transaction is committed.According to the research these are closest to our approach and are implemented according to distributed SDN multiple controller architecture and consensus algorithms.

Feature	Proposed	CopyCat	Raft Paxos
Consensus algorithm	Raft	Raft	Paxos
Controllers	10	1	3
Workers	2/controller	1 client	6 End Users
Time to read data	10 ms	0.20 s	6.425 ms
Time to write/send data	10.4 ms	0.40 s	17.814 ms
Time to elect a Master	10.06 ms	0.060 s	28 ms

Table 2:Comparison of Protocols/ Algorithms

Table 2 presents a comparative analysis of the proposed mechanism against the CopyCat project and a Paxos-based system. All models utilized consensus algorithms within a distributed Software Defined Networking (SDN) architecture. The comparison revealed that the CopyCat mechanism required more computational resources, making it less suitable for large-scale networks. The average time that the Copycat system needs to start and elect a Master Controller is 23.23 seconds.

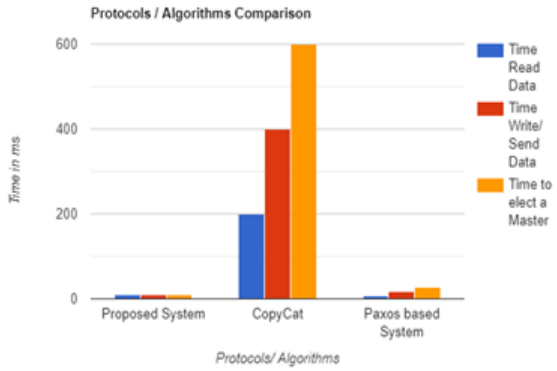

Figure 10:Comparison of Protocols/ Algorithms

This increased resource demand can lead to higher operational costs and reduced efficiency, particularly in environments where resources are limited. Furthermore, the time taken for data reading, writing, and transmission was notably higher in the CopyCat system compared to the proposed mechanism, indicating that the design is more efficient in handling data operations.

The comparative analysis serves to highlight the advantages of the proposed mechanism in terms of resource utilization and operational efficiency. By demonstrating that the proposed design outperforms existing alternatives, the research provides compelling evidence for its adoption in real-world applications. The findings underscore the importance of developing mechanisms that not only meet the functional requirements of network management but also optimize resource usage to enhance overall performance. This analysis contributes to the ongoing discourse in the field of network management, emphasizing the need for innovative solutions that can effectively address the challenges posed by modern network environments.

4.6 Consensus and Data Handling Efficiency

The average time required for the CopyCat system to initiate and elect a Master Controller was recorded at 23.23 seconds, whereas the proposed mechanism exhibited a stable consensus time that was significantly lower.

The Write/Send Data time for our protocol was impressively low at 10.4 milliseconds, showcasing its efficiency in handling data operations. These low and stable times for data handling significantly enhance the overall performance of the mechanism, contributing to a robust and functional architectural design. The ability to maintain low latency in data operations is crucial for applications that require real-time processing and quick decision-making, making our mechanism particularly well-suited for such scenarios.

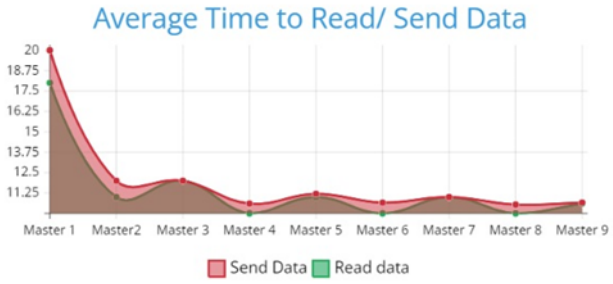

Figure 11:Read and Write Average Time.

4.7 Stability and Performance Results

Our simulation results indicated that even after a controller failure, all controllers maintained consistent data throughout the 100 second simulation period, underscoring the stability of the network. This stability is a key indicator of the mechanism's effectiveness, as it demonstrates the system's ability to recover from failures without compromising data integrity. The average time required for Master node election in our proposed mechanism was recorded at a mere 10.06 milliseconds, demonstrating both stability and efficiency in the election process. This rapid election time is critical for maintaining operational continuity, as it minimizes downtime and ensures that the network can quickly adapt to changes.

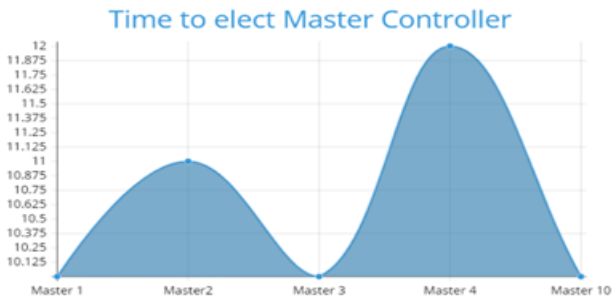

Figure 12:Controller election Time with Raft.

The proposed mechanism represents a promising advancement in the realm of network management, particularly within the context of Software Defined Networking. By leveraging the Raft consensus algorithm, our implementation effectively synchronizes network state information across multiple nodes, ensuring high throughput, fault tolerance, and controller synchronization. The simulation results affirm that our mechanism can support multiple controllers while maintaining a consistent global view of the network, thereby paving the way for future developments in reliable and scalable network architectures. As the demand for efficient and resilient network solutions continues to grow, our proposed mechanism stands out as a viable option for organizations seeking to enhance their network management capabilities and ensure seamless operations in the face of challenges.

4.8 Conclusion of Chapter

This chapter addressed the research question: How can a mechanism be designed to support high throughput, dynamic view changes, fault tolerance, and controller synchronization in multiple SDN controller setups? The proposed mechanism, built on the Raft consensus algorithm, successfully achieves these objectives by introducing a robust framework for managing multiple SDN controllers.

The mechanism ensures high throughput by efficiently distributing data and workload across controllers, reducing latency and improving network performance. It supports dynamic view changes by enabling seamless transitions between Master, Candidate, and Worker nodes, ensuring that the network adapts quickly to changes in controller states. Fault tolerance is achieved through a fail-over mechanism that allows controllers to fail without disconnecting switches, eliminating single points of failure and maintaining network stability. Finally, controller synchronization is maintained through the Raft algorithm, which ensures that all controllers have a consistent global view of the network, even during failures or dynamic changes.

The performance evaluation and comparative analysis demonstrated the mechanism's superiority over existing systems, such as Paxos-based mechanisms, in terms of consensus time, data access time, and network overhead. Extensive failure scenario simulations further validated the mechanism's resilience, showing its ability to maintain data consistency and operational continuity under challenging conditions.

In conclusion, the proposed mechanism effectively answers the research question by providing a scalable, efficient, and fault-tolerant solution for managing multiple SDN controllers. Its ability to ensure high throughput, dynamic adaptability, fault tolerance, and synchronization makes it a significant contribution to the field of SDN architectures. Future work could explore its application in larger-scale networks and its integration with emerging SDN technologies to further enhance its capabilities.

Chapter 5: Designing a System for Dynamic Load Balancing and Coordination of Multiple Controllers in SDN Environments

This chapter addresses the research question: How can a system be designed to dynamically shift the load across multiple controllers through switches and ensure effective coordination among them? In Software Defined Networking (SDN), the use of multiple controllers has become essential to meet the

demands of scalability, fault tolerance, and performance. However, managing the coordination and load distribution among these controllers remains a significant challenge. This chapter proposes a novel system that leverages the RAFT consensus algorithm and OpenFlow's Master/Slave connection model to address these challenges.

5.1 Implementation

The proposed mechanism is designed around a network of multiple Software Defined Networking (SDN) controllers that work collaboratively to effectively manage the overall network infrastructure. Each individual controller is assigned the responsibility of overseeing a specific subset of switches within the network, allowing for a distributed approach to network management. This division of responsibilities not only enhances efficiency but also ensures that no single controller becomes a bottleneck in the system.

To facilitate effective communication and coordination among the controllers, the system employs a robust coordination and synchronization mechanism. This mechanism enables controllers to exchange critical information regarding their current load and operational status with one another. By utilizing a consensus-based approach to coordination and synchronization, we ensure that all controllers maintain a consistent view of the network's state.

Each controller registers with a dedicated coordination service, which acts as a central point for managing the distributed coordination protocol. This coordination service is responsible for maintaining a shared state that includes essential information such as the network topology, controller assignments, and the current load distribution across the controllers. This shared state is crucial for enabling informed decision-making and ensuring that all controllers are aligned in their operations.

To keep the system synchronized, controllers periodically update their local state by aligning it with the shared state maintained by the coordination service. This synchronization process is achieved through a combination of data replication techniques, particularly utilizing flow tables that store information about active flows and their associated switches. This ensures that all controllers have access to the most up-to-date information regarding the network's operational status.

In scenarios where a controller joins or leaves the system, the coordination service plays a vital role by notifying the remaining controllers. This notification prompts them to update their understanding of the network and, if necessary, redistribute the load among themselves to maintain optimal performance.

Load balancing across the controllers is a key feature of our implementation, achieved through the dynamic redistribution of switches and their associated flows. Controllers employ sophisticated load balancing algorithms that take into account various factors, including the current workload of each controller, the capacity of the switches they manage, and prevailing network traffic patterns. When decisions regarding load balancing are made, the controllers engage in negotiations to transfer the ownership of switches and their corresponding flows based on the newly determined load distribution. This dynamic approach ensures that no single controller is overwhelmed, thereby enhancing the overall responsiveness and efficiency of the network.

To address potential controller failures, a comprehensive fail-over mechanism is integrated into the system. In the event that a controller fails, the coordination service is designed to detect this failure promptly and initiate a fail-over process. This process involves selecting a new controller to assume the responsibilities of the failed controller. The newly designated controller then establishes connections

with the switches that were previously managed by the failed controller, ensuring a seamless transition that minimizes disruption to network operations.

Communication between the controllers and the switches is facilitated through a standardized protocol known as OpenFlow. This protocol allows for the exchange of network control messages, enabling the controllers to effectively manage the switches and their associated flows. The coordination and synchronization mechanism described above further enhances this communication by allowing controllers to exchange coordination messages, thereby maintaining consistency across the network and distributing control responsibilities in an efficient manner. Overall, this implementation provides a robust framework for managing distributed SDN environments, ensuring high availability, scalability, and fault tolerance.

5.2 Load Balancing

In the proposed system, effective load balancing is achieved through the exchange of load information among the various controllers within the network. This process is facilitated by the Controller Election Process, which is implemented using the OpenFlow protocol. During this election process, the controllers engage in negotiations to determine which controller will assume the role of the master and which will serve as the backup. This decision-making is accomplished through the use of OpenFlow's Role Request message, ensuring a clear hierarchy and distribution of responsibilities among the controllers.

Each controller continuously monitors its own load by utilizing various performance metrics, including CPU utilization, memory usage, and the number of active flows it is managing. This load information is periodically updated and maintained in the controller's records, allowing for real-time assessment of its operational status. Each controller then compares its own load metrics with those received from other controllers in the network. This comparative analysis is crucial for identifying the least loaded controller among the available options.

If a controller determines that it is the least loaded based on this comparison, it continues to handle incoming traffic as usual. However, if it identifies another controller with a lower load, it takes appropriate actions to facilitate load balancing. The decisions made regarding load balancing can be implemented by modifying the flow table entries in the switches, effectively redirecting traffic to the appropriate controllers based on the established load balancing algorithm.

The SDN controllers leverage the OpenFlow protocol to dynamically install, update, or remove flow rules, which are essential for achieving effective load balancing. When a new request arrives at a switch, the switch forwards this request to its designated controller. The OpenFlow protocol enables switches to direct incoming packets to a specific controller based on the rules defined in their flow tables. By configuring these flow tables, switches can match incoming requests and forward them to the appropriate controller according to the load balancing policies in place.

Each switch maintains a record of the load information received from the controllers it is connected to. By comparing the load information of these connected controllers, the switch can select the least loaded controller as the destination for incoming requests. This capability allows the switch to make informed decisions about which controller to forward requests to, thereby ensuring effective load balancing across the SDN system.

To implement load balancing effectively, specific packet fields such as source IP address, destination IP address, and transport protocols are recorded in the flow table entries. These fields are

crucial for directing packets to the desired controller. The OpenFlow protocol is utilized to set the flow actions in the flow rules of the switches, which include:

- Output Action: This action specifies the output port of the switch to forward the traffic to the desired controller, ensuring that packets are sent to the correct destination.

- Controller Action: This action directs the traffic to the controller by specifying the action to send the packet to the controller's designated port, facilitating seamless communication between the switch and the controller.

Load information from the controllers is periodically collected and analyzed to determine which controller is currently the least loaded. Based on this dynamic load information, the flow tables are updated to reflect the current load balancing requirements. Given the dynamic nature of network traffic, the flow rules may need to be updated periodically or in response to significant load changes.

The SDN controller plays a pivotal role in monitoring the load, collecting load information, and making necessary updates to the flow rules as required. This can be accomplished by sending OpenFlow messages to the switches to modify or add flow rules, ensuring that the load balancing mechanism remains responsive and effective in adapting to changing network conditions. By leveraging the capabilities of the OpenFlow protocol, this research ensures that load balancing is not only efficient but also adaptable, ultimately enhancing the overall performance and reliability of the SDN infrastructure.

5.3 Performance evaluation

5.3.1 Experimental Setup

A simulation has been conducted to assess the performance of the proposed scheme. The system on which the simulation was executed was based on a VM with Ubuntu 22.04 OS, 16 GB of memory, and OpenFlow switches. We evaluated the performance of our system in terms of load balancing, response time, throughput, packet loss, delay, and the time overhead imposed by the controllers and switches to coordinate. The coordination performance and scalability between controllers, switches, and hosts were also depicted according to the scenario of routing several packets that are successfully routed (without traversing any failed link) to their destinations. We emulated the performance using Mininet [73] and Ryu [66], a component-based Software-Defined Networking framework. Ryu [66] provides software components with well-defined APIs that make it easy for developers to create new network management and control applications. We created a topology of 10 SDN controllers, consisting of one master controller and nine SDN controllers, along with 20 switches.

- Master Controller: Controller M

- SDN Controllers: Controller C1, Controller C2, Controller C3, Controller C4, Controller C5, Controller C6, Controller C7, Controller C8, Controller C9

- Switches 1–20: S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20

Initially, switches were distributed evenly among the client controllers. Master Controller M did not handle any switches directly. Each controller could handle up to three switches. We assigned an initial load distribution of switches to controllers: Controller C1: S1, S2, S3; Controller C2: S4, S5, S6; and so on. We set initial metric values for each controller (CPU utilization, memory usage, number of

active flows) based on the simulation scenarios. Periodically, metrics were collected from each controller and switch, and the metric values were updated based on the simulated workload and network conditions.

5.3.1.2 Performance Metrics Collection

To evaluate the system's efficiency, we monitored CPU and memory usage for each controller using a Python script leveraging the psutil library [Appendix A2]. Metrics were collected every 10 seconds during the test duration, providing insights into the resource utilization of the controllers under varying workloads. The results are summarized in Table 3.

Time (s)	CPU Usage (%)	Memory Usage (%)
10	25.0	60.0
20	30.0	62.5
30	28.0	61.0
40	35.0	63.0
50	32.0	64.5
60	29.0	62.0
	Average CPU Usage:	29.83%
	Average Memory Usage:	62.00%

Table 3:CPU and Memory Usage

5.3.1.3 Workload Scenarios

We emulated the performance for three different scenarios of workload to test the controllers' coordination and the management of the switch. Our system shifts dynamically the load across the switches and the controllers. We simulated three different workloads to stress controllers through adjusting the flow rate:

- Workload A: 1000 packets sent in 100 ms.
- Workload B: 2000 packets per second (pps).
- Workload C: 5000 packets (flood mode).

The simulation results are shown in Table 4.

Packets	Average Time in ms	Packet loss
Workload A- 1000	0.041	1.2 %
Workload B- 2000	0.049	3 %
Workload C-5000	0.060	3.5 %

Table 4:Network performance packets send

5.3.1.4 Network Communication and Response Time

The communication between all nodes within the network was tested over a duration of 100 seconds. In an OpenFlow network, the response time of the controller has a direct impact on the flow completion times. The average response time was evaluated at 0.041 ms for packet transmission throughout the network. The performance of the routing application is determined by the number of packets that are successfully routed to their destinations without traversing any failed links. To analyze the load balancing algorithm, various network scenarios and workload conditions were simulated. Different weights were assigned to various metrics, and the resulting load distribution among the controllers was observed. The performance was emulated across three distinct scenarios, where all controllers synchronized at the following rates: (i) 0.041 ms, (ii) 0.049 ms, and (iii) 0.060 ms (messages per second). The results of these simulations are illustrated in Figure 14.

5.3.1.5 Throughput Evaluation

To evaluate and plot the mean throughput under varying workloads, iperf [82] was utilized. Additional emulations were conducted to assess the performance of a load balancing application. In this setup, the switches generated flows uniformly at random, which could be routed and queued to any of the ten controllers. Each controller maintained awareness of the load it managed, allowing for the identification of the least loaded server. This synchronization among controllers ensured optimal load distribution at all times.

5.3.2 Results

In the course of the conducted tests, a comprehensive comparison was made between the proposed system and an alternative system that is also based on OpenFlow switch connections, as referenced in [75]. The primary focus of this comparison was on two critical performance metrics: response time and throughput. The results of this evaluation revealed that the proposed approach exhibited significantly shorter response times when transferring packets across the network, particularly when juxtaposed with the system previously introduced in [75]. This finding underscores the efficiency of the proposed system in managing network traffic.

To further evaluate and illustrate the mean throughput of the proposed system, a detailed comparison was conducted with the system described in [75] under a variety of workload conditions. This comparison is visually represented in Figure 13. The data indicates that, under conditions of heavy load, the proposed method maintains a remarkably steady performance, requiring an average of only 0.041 milliseconds to successfully send all packets. In stark contrast, the system referenced in [75] required an initial response time of 0.3 milliseconds for the first workload test, with response times progressively increasing as the number of packet requests escalated. This disparity highlights the robustness of the proposed system in handling high traffic volumes.

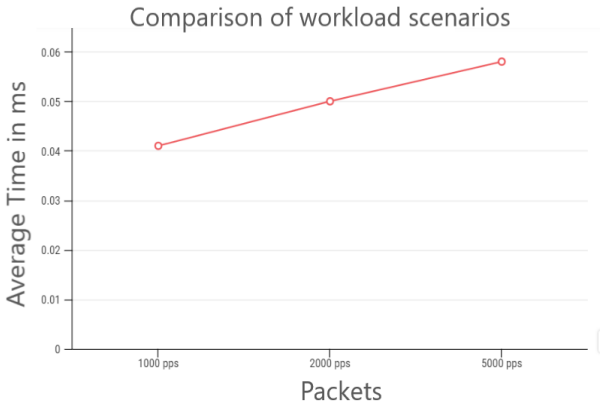

Figure 13:Packet forwarding with OpenFlow in a switch.

5.3.2.1 Network Stress Testing

To rigorously evaluate controller response times under varying workloads, the hping3 tool was employed to generate TCP SYN traffic. The command used for this purpose was: hping3 -c n packets -d 120 -S -p 80 10.0.0.2.

This command sends a specified number of TCP SYN packets to the IP address 10.0.0.2 on port 80, with each packet having a payload size of 120 bytes. hping3 is a versatile tool often used for network testing, port scanning, or stress testing a server. In this study, it was utilized to measure response times and packet loss under different workloads, ranging from 1000 to 5000 packets. It is important to note that sending a large number of packets (e.g., 5000) can overwhelm the target server if it is not configured to handle such traffic. However, in this controlled emulation environment, the tests were conducted without unintended network disruption.

```
hping3 -c 1000 -d 120 -S -p 80 10.0.0.2

64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.035 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.040 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.042 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.038 ms
```

Figure 14:hping command

5.3.2.2 Packet Loss Analysis

The packet loss rates were meticulously measured as a percentage of lost packets in relation to the total number of packets sent. The results of this analysis indicated a small and stable packet loss rate, ranging from 1.2% to 3.5%. Notably, nearly all packets were successfully received, demonstrating the effectiveness of the proposed system in maintaining reliable communication even under varying network conditions. The packet loss distribution for different workloads is illustrated in Figures 15–17.

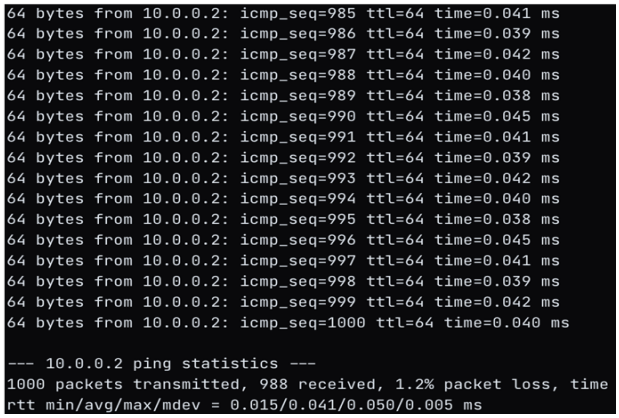

Figure 15: 1000 packets data loss

Figure 16: 2000 packets data loss

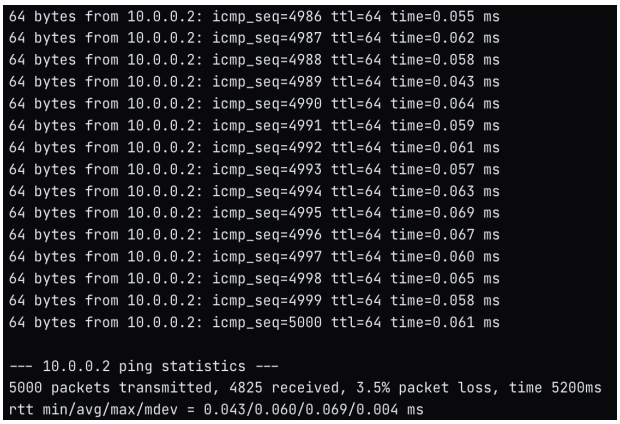

Figure 17: 5000 packets data loss

5.3.2.3 Workload and Response Time Analysis

As illustrated in Figure 18, it became evident that the workload exerted a substantial influence on the response times observed in both systems. Specifically, in the system described in [75], the response time exhibited a marginal increase under Workload B, but this increase became more pronounced under Workload C. This phenomenon can be attributed to the queuing effects that arise when the packet interval rate exceeds the processing capacity of the controller, resulting in a substantial

escalation of response times. In contrast, the proposed system demonstrated consistent performance, with response times remaining below 0.060 milliseconds even under Workload C.

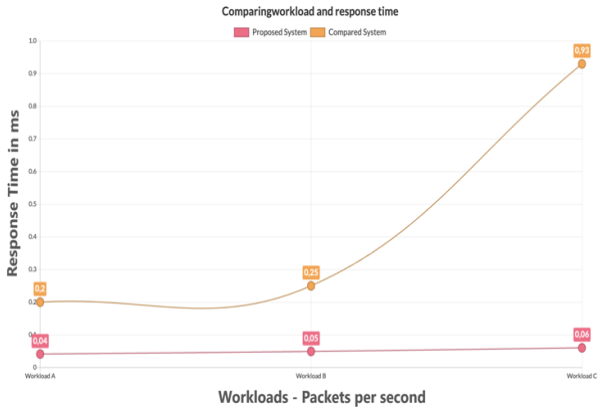

Figure 18:Comparing workload and response time

5.3.2.4 Switch Migration and Failover Analysis

In addition to response time analysis, the study also measured the time overhead associated with assigning roles to the switches, as well as the costs associated with the switch migration process in the compared system [75]. The migration process was observed to take approximately 2 milliseconds under Workload A, with the time required for migration increasing as the workload intensified. Furthermore, the failover process was found to average around 20 milliseconds, a duration that is primarily influenced by the failure detection mechanism based on heartbeat messages provided by JGroups. In contrast, the proposed system demonstrated a more efficient performance, requiring an average of 10.06 milliseconds to assign the Master role to a controller node.

5.3.2.5 Packet Loss and Delay Emulation

To simulate realistic network conditions, a normal distribution was defined for the delay experienced by packets. As a result, all packets exiting controller C1 through its interface C1-eth0 experienced a delay that was normally distributed within the range of 10 milliseconds to 20 milliseconds. This delay was taken into account to reflect the additional time incurred due to the master election process. The NETEM tool was utilized to specify a distribution that characterizes how delays vary within the network, allowing for a more accurate emulation of real-world scenarios.

The following configurations were applied:

• Delay: A delay of 10 ms with a normal distribution (±5 ms) was added to simulate network latency.

```
sudo tc qdisc add dev eth0 root netem delay 10ms 5ms distribution normal
```

• Packet Loss: Packet loss rates of 1.2%, 3.0%, and 3.5% were introduced for Workloads A, B, and C, respectively, to emulate congestion.

```
sudo tc qdisc add dev eth0 root netem loss 1.2%
```

• Reordering: To simulate packet reordering, 10% of packets were reordered with a delay of 10 ms.

```
sudo tc qdisc add dev eth0 root netem delay 10ms reorder 10%
```

These configurations were applied to the virtual switches in the Mininet topology to replicate real-world network behavior. The NETEM settings were validated using the tc command, which confirmed the applied delay, packet loss, and reordering parameters. The packet loss rates, as shown in Figure 19, remained stable (1.2–3.5%), demonstrating the system's resilience to congestion. Figure 19 depictes the packet loss.

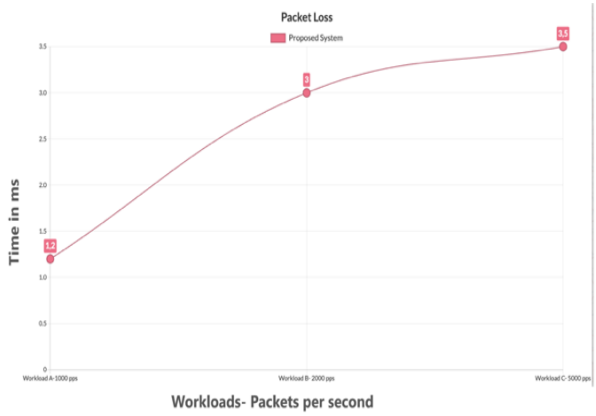

Figure 19:Packet loss

5.4. Chapter Conclusion

This chapter addressed the research question: How can a system be designed to dynamically shift the load across multiple controllers through switches and ensure effective coordination among them? The proposed system successfully achieves this by integrating the RAFT consensus algorithm with OpenFlow's Master/Slave connection model, creating a robust framework for managing multiple SDN controllers. The system ensures dynamic load balancing by evaluating controller metrics such as CPU utilization, memory usage, and active flows, and redistributing switches to controllers with lower loads. This process is supported by a weighted scoring mechanism that identifies overloaded and underloaded controllers, enabling efficient load redistribution and maintaining optimal network performance.

The fail-over mechanism enhances fault tolerance by detecting controller failures and seamlessly transferring their responsibilities to other controllers, ensuring uninterrupted network operations. The performance evaluation demonstrated the system's ability to maintain low response times (e.g., 0.041 ms for 1000 packets) and high throughput even under heavy workloads. The system also exhibited minimal packet loss (ranging from 1.2% to 3.5%), highlighting its reliability and efficiency.

Comparative analysis with existing systems, such as the one described in [75], showed that the proposed system outperforms alternatives in terms of response time, scalability, and fault tolerance.

In conclusion, the proposed system effectively answers the research question by providing a scalable, efficient, and fault-tolerant solution for coordinating multiple SDN controllers. Its ability to dynamically balance load, ensure seamless fail-over, and maintain high performance under varying workloads makes it a significant contribution to the field of SDN architectures.

Chapter 6: Controller Placement for SDN, a Greedy Approach

This chapter presents a novel greedy approach for the Controller Placement Problem (CPP) within the context of Software Defined Networking (SDN). As the SDN landscape evolves, the use of multiple controllers has become essential to meet the demands of scalability and performance. Recent advancements have led to the development of various solutions aimed at optimizing controller placement. This study explores the CPP by employing objective optimization techniques alongside the proposed algorithms.

Interaction time between the controller and the switch is an important parameter in locating the controllers. The proposed approach is a greedy algorithm for controller placement in a network. The algorithm aims to minimize the total cost of connecting nodes to controllers.

This thesis focuses on latency, which is one of the most frequently utilized performance indicators in network management and optimization. Latency is a critical factor that can significantly impact the overall performance of a Software Defined Networking (SDN) environment. It encompasses several components, including transmission delay, propagation delay, queuing delay, and processing delay, all of which contribute to the total latency experienced in the network.

In this research, two specific types of latency are evaluated: a) switch-to-controller latency (also referred to as controller-node latency) and b)controller-to-controller latency. These latency types is essential for optimizing the performance of SDN architectures, as they directly influence the responsiveness of the network to control messages and data flows.

6.1 Proposed Algorithm for Controller Placement

In the realm of Software Defined Networking (SDN), effective controller placement is crucial for optimizing network performance and minimizing latency. This section introduces a novel greedy algorithm specifically designed to strategically position controllers in close proximity to the switches they manage. By focusing on reducing the physical distance between controllers and their associated nodes, the algorithm aims to enhance the overall efficiency of the network. The approach leverages a systematic evaluation of latency, incorporating both Euclidean distance and transmission delay to inform placement decisions. Through a structured process of cost matrix initialization and node assignment, the algorithm not only seeks to minimize latency but also promotes a balanced load distribution across the network, ultimately contributing to a more responsive and reliable SDN infrastructure.

The proposed algorithm is a greedy algorithm designed to strategically place controllers in proximity to the switches they manage, with the primary objective of minimizing latency between the controllers and the nodes. This algorithm operates under the premise that reducing the physical distance

between controllers and switches will lead to lower latency, thereby enhancing the overall efficiency of the network.

6.1.1 Latency Calculation

The algorithm calculates the latency between each controller and its assigned nodes using a combination of the Euclidean distance and the transmission delay. The Euclidean distance provides a straightforward metric for assessing the physical distance between the controller and the node, while the transmission delay accounts for the time taken to send data packets over the network. The total latency for each controller-node pair is computed as follows:

{Total Latency} = {Euclidean Distance} + {Transmission Delay}

This calculation allows the algorithm to evaluate the effectiveness of different controller placements based on their proximity to the nodes they serve.

6.1.2 Initialization of Cost Matrix

The initialization of the cost matrix is a crucial step in the algorithm, serving as a foundation for connecting switches to potential controllers. This matrix, represented as (costs[i][j]), quantifies the connection cost from switch i to controller j, with costs calculated primarily through Euclidean distance, which acts as a proxy for connection expense. By establishing these costs, the algorithm efficiently identifies optimal connections, thereby enhancing resource utilization within the network.

6.1.3 Assignment of Nodes to Controllers

The algorithm commences with the initialization phase, where an array known as assigned is created. This array contains lists, with each list corresponding to the nodes allocated to a specific controller. Additionally, an unassigned list is established to monitor nodes that have not yet been allocated to any controller. This sets the stage for the node assignment process.

In the succeeding phase, the algorithm enters a loop to assign nodes from the unassigned list to the nearest available controller. The closest controller is determined based on the lowest cost associated with the switch of the node being evaluated. Should a node fail to be assigned during a given iteration, it is simply skipped. This iterative assignment continues until all nodes are successfully allocated, resulting in an output comprising lists that map each node to its corresponding controller.

The greedy algorithm employed in this framework focuses on minimizing latency and enhancing the performance of the Software Defined Networking (SDN) infrastructure. By strategically positioning controllers in relation to the switches they oversee, the algorithm promotes optimal placement. This systematic evaluation of latency fosters an efficient load distribution, which significantly boosts the network's responsiveness. Consequently, such improvements contribute to a more efficient and reliable SDN environment, ultimately enhancing service delivery and user experience.

6.2 Implementation of the Greedy Algorithm

This thesis utilizes controller-node latency and controller-controller latency, which encompasses propagation, queuing, and processing delays, as crucial performance parameters in the context of Software Defined Networking (SDN). A heuristic approach based on a greedy algorithm has been implemented to address the controller placement problem (CPP) [Appendix A3]. The fundamental objective of this approach is to minimize the number of controllers while simultaneously reducing the inter-node distances, thereby achieving acceptable latency from nodes to their assigned controllers and between controllers.

The greedy algorithm operates by employing the Euclidean distance between nodes and controllers as the cost function to determine the best immediate solution. In the context of the controller placement problem, the algorithm calculates the Euclidean distance between each unassigned node and each available controller, subsequently assigning the node to the controller with the smallest distance. This process is critical for ensuring that nodes are connected to the most proximate controller, thereby minimizing latency.

6.2.1 Initialization of the Cost Function

The algorithm begins by initializing a cost matrix, where each element represents the Euclidean distance between a switch and a controller. This matrix serves as the basis for determining the optimal assignments. The algorithm also initializes an assigned array, which keeps track of which nodes have been allocated to which controllers, and a list of unassigned nodes that have yet to be connected.

6.2.2 Assignment Process

The algorithm enters a loop that continues until all nodes have been assigned to a controller. In each iteration, the algorithm selects an unassigned node and identifies the closest available controller based on the previously calculated distances. If the selected controller does not have the capacity to handle the node, the algorithm searches for the next nearest controller and repeats the assignment process. This iterative approach continues until a suitable controller is found for each unassigned node.

6.3 Characteristics of the Greedy Heuristic Algorithm

A greedy heuristic algorithm is characterized by its strategy of making locally optimal choices at each stage with the hope of finding a global optimum. This approach is both simple and fast, making it suitable for real-time applications. In the context of the CPP in multiple SDN environments, the greedy heuristic algorithm can be employed to find a solution that minimizes latency between controllers and switches.

The algorithm iteratively places controllers in locations that minimize the maximum distance to any switch, without considering the potential impact on future decisions. While this method is efficient, it may not always yield the optimal solution and could result in higher latency between certain controllers and switches.

6.4 Methodology for Controller Placement in SDN

The following steps outline the proposed approach for the controller placement problem in SDN:

1. Initialize a List of Available Locations for Controllers: A list of potential controller locations is created based on the network topology.

2. Iterative Assignment of Controllers:

While there are still switches without assigned controllers:

a. Identify the location that is closest to the highest number of unassigned switches.

b. Place a controller at that location.

c. Assign each switch within the controller's coverage area to that controller.

3. Calculate Latency: After assigning switches to controllers, the latency between each switch and its assigned controller is calculated.

6.5 Data Structures and Algorithm Functionality

In this implementation, switches are represented as a list of (x, y) coordinates indicating their physical locations, while controllers are similarly represented. Each node has an attribute indicating the switch to which it is connected.

The greedy controller placement function first calculates the Euclidean distance between each switch and controller, storing these distances for further processing. It initializes a list to keep track of which nodes have been assigned to which controllers.

The function then permutes the list of unassigned nodes and assigns each node to the controller with the lowest cost based on the previously calculated distances. The output of the function is an assigned list that indicates which nodes have been allocated to which controllers.

The algorithm employs a vector of Location structs to represent the available locations for controllers. Each Locationstruct contains the index of the location, the distance to the nearest unassigned switch, and a vector of unassigned switches that fall within the controller's coverage area.

The algorithm sorts the locations based on their distance to unassigned switches and the number of switches within their coverage. It iterates over the sorted locations, assigning a controller to each unassigned location.

The final output consists of a vector of Controller structs, where each struct contains the index of the controller and the number of switches assigned to it. Additionally, the switch distances list represents the distances from each switch to a central location, while the controller coverage variable indicates the coverage radius of each controller.

The function iteratively places controllers in locations that are closest to the most unassigned switches, assigning all switches within the controller's coverage to that controller. This process continues until all switches have been assigned to a controller. At this point, the algorithm has effectively minimized the latency between each switch and its corresponding controller, ensuring that the network operates efficiently. The final output of the algorithm is a comprehensive mapping of

switches to controllers, which can be represented as a list or a data structure that indicates which switches are managed by which controllers.

6.6 Performance evaluation

6.6.1 Experimental Setup

A simulation has been conducted to assess the performance of the proposed scheme. The system on which the simulation was executed was based on a Virtual Machine (VM) with Ubuntu 22.04 OS, 16 GB of memory and OpenFlow Switches. We emulate the performance using Mininet [73] and Ryu controller [75] component-based Software Defined Networking framework.

The performance of the proposed system was evaluated in terms of latency and transmission delay. These factors are crucial in Software Defined Networking (SDN) due to the frequent communication between switches and controllers. The Controller Placement Problem in SDN often focuses on minimizing both transmission delay and controller processing delay. Transmission delay in CPP is similar to the facility location problem, where the goal is to find the best location to place the controllers to reduce the distance between the switches and the controllers. This is an important factor in ensuring efficient communication and reducing latency in SDN networks.

6.6.2 Network Configuration and Latency Analysis

A network consisting of six controllers and eight nodes (switches) has been established to evaluate the performance of the proposed system. The analysis focuses on two key types of latency: a) controller-node latency and b) controller-controller latency.

6.6.2.1 Controller-Node Latency

Controller-node latency refers to the time required for a message to travel from a controller to a node within the network. This latency is influenced by several factors, including transmission delay, which is the time taken for a message to be transmitted over the physical link connecting the controller and the node. Controller-node latency is critical because it directly affects the responsiveness of the network. High latency can lead to delays in data processing and decision-making, which can hinder the performance of applications relying on real-time data transmission, such as video streaming, online gaming, and financial transactions.

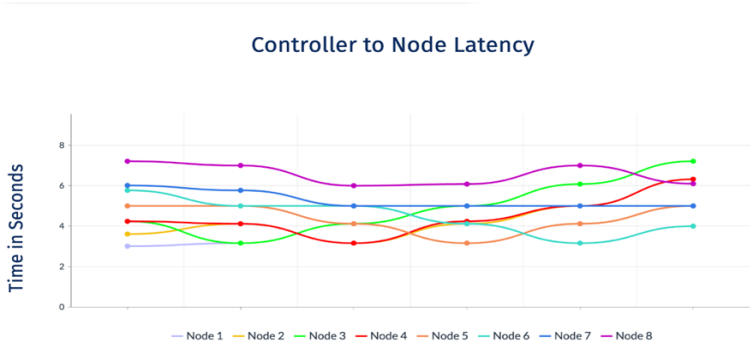

Figure 20: Controllers to Node Latency.

As illustrated in Figure 20, the latency between each controller and its assigned nodes is calculated in seconds. The total transmission latency from controllers to nodes amounts to 10.067 seconds, representing the cumulative sum of all latencies between each controller and its respective assigned nodes, as detailed in Table 5.

The total transmission delay is computed by aggregating the minimum latencies between each node and its assigned controller. Table 5 presents the latency values for each controller and the nodes assigned to it. For example, the first controller exhibits a latency of 0.21 seconds for the first node and 0.42 seconds for the second node.

Controllers	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Controller 1	0.21	0.42	0.44	0.42	0.43	0.44	0.46	0.43
Controller 2	0.48	0.21	0.33	0.33	0.30	0.33	0.36	0.34
Controller 3	0.59	0.43	0.21	0.22	0.22	0.21	0.22	0.22
Controller 4	0.57	0.46	0.32	0.21	0.21	0.22	0.22	0.21
Controller 5	0.53	0.41	0.32	0.33	0.21	0.22	0.22	0.21
Controller 6	0.63	0.48	0.36	0.35	0.34	0.21	0.22	0.21

Table 5: Controller- Node Transmission Delay in Seconds

6.6.2.2 Controller-Controller Latency

The latency between two controllers is determined by calculating the Euclidean distance between them, in addition to the transmission delay incurred as the signal travels from one controller to the other and back. This round-trip time is represented as the Euclidean distance between two controllers increases due to their physical separation the latency between them also escalates.

The nodes are strategically positioned at coordinates (3, 3), (4, 4), (3, 4), (3, 5), (4, 3), (4, 4), (5, 3), and (5, 4). For instance, the Euclidean distance between Controller 1 and Node 1 is calculated using the formula:

$$\sqrt{\{(0 - 3)^2 + (5 - 3)^2\}} = 3.61$$

Thus, the latency between Controller 1 and Node 1 is also 3.61 seconds. The controllers are positioned at (0, 5), (10, 5), (20, 5), (30, 5), (40, 5), and (50, 5). The Euclidean distance between adjacent controllers is 10 units, leading to a latency calculation of $10 + 2 \times transmission\ delay$ for communication between them.

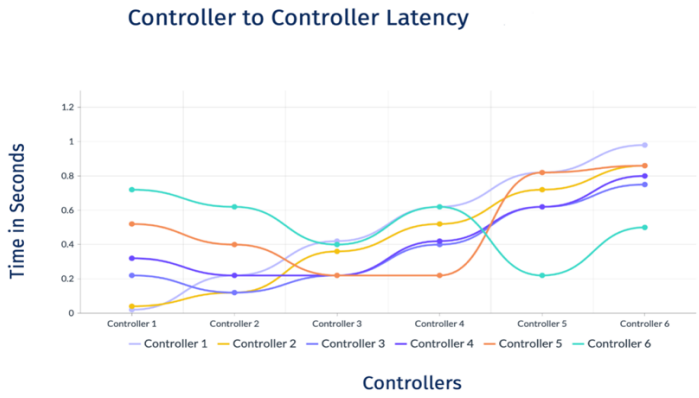

Figure 21:Controller to Controller Latency.

6.7 Algorithm Implementation

In the implementation, a nested loop is utilized to iterate over each node and controller, identifying the minimum latency for each node-controller pair. The algorithm sums these minimum latencies to derive the total transmission delay. The controller with the lowest latency is then assigned to each node.

Table 6 provides a calculation of the average controller latency, derived from the sum of the transmission delays. The results indicate that in an SDN network with multiple controllers, the latency values are relatively low. These latency figures represent the time taken for a controller to send a message to another controller, calculated as the average of the time taken for the sender to transmit the message and the time required for the receiver to acknowledge receipt.

Controllers	**Average Transmission Delay per Controller**
	Average Latency
Controller 1	0.122 Seconds
Controller 2	0.118 Seconds
Controller 3	0.133 Seconds
Controller 4	0.132 Seconds
Controller 5	0.124 Seconds
Controller 6	0.115 Seconds

Table 6: Average Transmission Delay per Controller

The transmission delay between controllers is measured at 1.414 seconds, which is notably lower than the latencies observed between the controllers themselves. The transmission latency, as depicted in Figure 20, encompasses the time required for the switch to receive data from the sender, process it, and subsequently forward it to the intended receiver.

6.7.1 Controller-Controller Latencies

The controller-controller latencies reflect the time taken for each pair of controllers to communicate. These latencies are relatively small, indicating that the switches can transmit data quickly between controllers. The latencies represent the time required for the switch to process the data. Notably, these latencies are lower than the total transmission latencies because the switch processes the data only once, rather than for each controller sending messages to one another.

The transmission delay per controller is illustrated in Figure 22. This metric is obtained by dividing the total transmission delay by the number of controllers. The transmission delay per controller is calculated by averaging the latencies for each controller, achieved by summing the latencies for all controllers and dividing by the number of nodes.

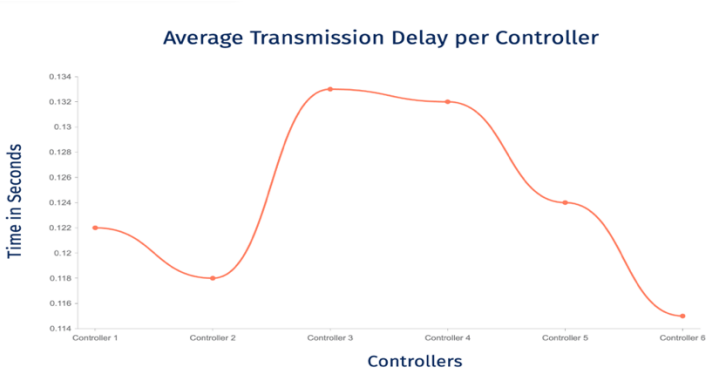

Figure 22:Transmission Delay per Controller.

The total transmission latency between controllers and nodes is higher than the controller-controller latencies due to the additional processing tasks performed by the switch. This includes the time required for the switch to receive data from the sender, process it, and forward it to the receiver. The analysis highlights the importance of optimizing both controller-node and controller-controller latencies to enhance the overall performance of the SDN architecture.

6.8. Chapter Conclusion

This chapter addressed the research question: How can a heuristic greedy algorithm be designed to minimize end-to-end latency and reduce maximum latency between controllers and switches in SDN environments? The proposed greedy algorithm successfully achieves this by strategically placing controllers in proximity to switches and dynamically assigning nodes to the nearest available controller.

The algorithm leverages Euclidean distance and transmission delay to calculate latency, ensuring that controllers are placed in locations that minimize both physical and network-based delays. The dynamic node assignment process ensures efficient load distribution, reducing overall network latency and improving responsiveness.

The performance evaluation, conducted using Mininet and Ryu, demonstrated the algorithm's ability to maintain low latency in both controller-node and controller-controller communication. The results showed that the total transmission latency between controllers and nodes was 10.067 seconds,

with individual latencies ranging from 0.21 to 0.63 seconds. Controller-controller latencies were even lower, averaging 1.414 seconds, highlighting the algorithm's efficiency in optimizing network performance.

In conclusion, the proposed greedy algorithm effectively answers the research question by providing a scalable and efficient solution for controller placement in SDN environments. Its ability to minimize latency, ensure efficient load distribution, and maintain low transmission delays makes it a significant contribution to the field of SDN architectures. Future work could explore the algorithm's application in larger-scale networks and its integration with emerging SDN technologies to further enhance its capabilities.

Chapter 7: Conclusion

This thesis has addressed critical challenges in Software Defined Networking (SDN), particularly in the context of centralized control architectures, by proposing innovative solutions to enhance reliability, scalability, fault tolerance, and interoperability. The existing literature has often fallen short in addressing high throughput, dynamic view changes, fault tolerance, and controller synchronization in multi-controller SDN setups. In response, this research introduced a novel mechanism leveraging the Raft consensus algorithm, which ensures efficient synchronization of network state information across multiple nodes, even under dynamic traffic conditions and failures. The findings of this thesis not only validate the proposed mechanisms but also provide a foundation for future research and practical applications in SDN.

7.1 Key Contributions and Research Findings

The first key contribution of this thesis is the implementation of a mechanism built on the Raft consensus algorithm, which successfully addresses the challenges of high throughput, dynamic view changes, and fault tolerance in multi-controller SDN setups. The proposed mechanism leverages the Raft consensus protocol to maintain a consistent global view of the network across multiple controllers, ensuring that all controllers have the same data even in the event of a Master node failure. The experimental results demonstrate that the mechanism maintains low and stable consensus times and ensures seamless controller synchronization during failures. Additionally, the system's ability to recover from multiple Master node failures without disrupting network operations highlights its robustness and reliability. The average times required for reading, writing in memory, and sending data to neighboring controllers are also low and stable, further contributing to the overall reliability and efficiency of the network. This implementation not only enhances fault tolerance but also ensures high throughput and dynamic adaptability, making it a significant advancement in the field of SDN architectures.

The second key contribution of this thesis is the creation of a system that ensures high throughput and fault tolerance. The experimental results indicate that the proposed system effectively supports these objectives by dynamically shifting the load across multiple controllers through switches, ensuring effective coordination and resource utilization. By leveraging OpenFlow's Master/Slave connection model and a weighted scoring mechanism, the system redistributes switches to underloaded controllers, maintaining optimal performance. This capability minimizes the time taken to elect a new controller,

which is critical for maintaining network stability during controller failures and ensuring continuous operation in dynamic environments where network conditions can change rapidly. The performance evaluation demonstrated low response times, and minimal packet loss ranging even under heavy workloads. This combination of high throughput and robust fault tolerance underscores the system's effectiveness in managing network operations efficiently.

Another key factor was to the proposed system to be able to coordinate efficiently and synchronized with multiple controllers. The evaluation findngs presented in this thesis confirm that the proposed system can efficiently coordinate and synchronize the controllers and switches within the network. This synchronization is achieved in a stable and timely manner, ensuring optimal performance under varying traffic conditions. The ability to maintain system coordination and network stability is a significant advancement in the field of SDN.

This thesis also addressed the Controller Placement Problem (CPP) by adapting concepts from facility location problems to identify optimal controller placements within the network. The greedy algorithm developed for this purpose aims to minimize latency between controllers and nodes, as well as between controllers themselves. The experimental results demonstrate the efficiency of this implementation, achieving near-optimal solutions within the CPP framework.

Additionally, a critical aspect of the proposed solution is its ability to minimize transmission delay between controllers and between controllers and switches. The results indicate that this reduction in transmission delay is vital for ensuring high network performance and efficiency. By optimizing controller placement and enhancing communication pathways, the proposed system significantly improves the throughput and response time of the control plane.

The proposed mechanisms were rigorously compared with existing systems, including Paxos-based solutions, to evaluate their performance across several critical metrics. The results consistently demonstrated that the proposed system outperformed these alternatives in terms of consensus time, data access time, and network overhead. Specifically, the consensus time for the proposed mechanism was significantly lower, averaging just 10.06 ms for Master node elections, compared to the longer times observed in Paxos implementations. This reduction in consensus time is crucial for maintaining the responsiveness of the network, particularly during periods of high demand or when failures occur.

In addition to faster consensus times, the proposed system exhibited superior data access times, allowing for quicker retrieval and processing of information across the network. This efficiency is vital for applications that require real-time data processing, such as video streaming, online gaming, and financial transactions, where delays can lead to degraded user experiences.

Furthermore, the proposed mechanism demonstrated lower network overhead, which is essential for optimizing resource utilization in large-scale networks. By minimizing the amount of control traffic generated during operations, the system ensures that more bandwidth is available for actual data transmission, thereby enhancing overall network performance.

The system’s ability to maintain high throughput, low latency, and robust fault tolerance under varying workloads and failure scenarios further underscores its superiority over existing alternatives. Even in the face of multiple controller failures, the proposed mechanism was able to quickly elect new controllers and redistribute workloads without significant disruption to network operations. This resilience is particularly important in dynamic environments where network conditions can change rapidly, ensuring continuous operation and reliability.

Overall, the comparative analysis highlights the effectiveness of the proposed mechanisms in addressing the limitations of traditional consensus protocols, making them a compelling choice for modern Software Defined Networking applications that demand high performance and reliability.

7.2 Implications for Future Research

The findings of this thesis have several implications for future research in the field of SDN. First, the successful implementation of the Raft consensus algorithm opens avenues for exploring other consensus protocols that may further enhance network reliability and performance. Additionally, the optimization strategies developed for the CPP can be extended to more complex network topologies and dynamic environments, where the placement of controllers may need to adapt in real-time to changing conditions.

While the Raft consensus algorithm has proven effective, future work could explore other consensus protocols, such as Paxos variants or Byzantine Fault Tolerance (BFT) algorithms, to further enhance fault tolerance and scalability in SDN environments. Comparative studies could evaluate the trade-offs between these protocols in terms of performance, complexity, and resource utilization.

Moreover, machine learning techniques could be integrated into the proposed mechanisms to predict traffic patterns, detect anomalies, and dynamically adjust controller placements and configurations. This would enhance the system's adaptability to changing network conditions and improve resource allocation efficiency.

Future research could focus on testing the proposed mechanisms in larger-scale networks with hundreds or thousands of controllers and switches. This would involve optimizing the algorithms for scalability and evaluating their performance in real-world, heterogeneous network environments.

As energy consumption becomes a critical concern in network infrastructure, future work could explore energy-efficient controller placement and load balancing strategies. This would involve minimizing the energy footprint of controllers and switches while maintaining high performance and reliability.

The proposed mechanisms could be extended to incorporate security features, such as encryption, authentication, and intrusion detection, to protect against cyber threats in SDN environments. This would ensure the integrity and confidentiality of network communications while maintaining high throughput and fault tolerance.

7.3 Final Thoughts

In conclusion, this thesis has made significant strides in addressing the challenges associated with centralized control architectures in Software Defined Networking. The proposed mechanisms leveraging the Raft consensus algorithm, dynamic load balancing, and a greedy controller placement algorithm offer scalable, efficient, and fault-tolerant solutions for managing multiple SDN controllers. The experimental results and comparative analyses validate the effectiveness of these mechanisms, demonstrating their superiority over existing alternatives in terms of performance, reliability, and adaptability.

The contributions of this research extend beyond the immediate context of SDN, providing valuable insights and frameworks for addressing similar challenges in other domains. The findings of

this thesis not only enhance our understanding of distributed systems and network management but also lay the groundwork for future advancements in these fields. As the demand for scalable, reliable, and efficient network solutions continues to grow, the proposed mechanisms will play a crucial role in shaping the future of network architectures and technologies.

By ensuring high throughput, fault tolerance, and efficient controller synchronization, this research contributes to the development of more agile, flexible, and efficient network infrastructures. The insights gained from this work will be invaluable for both academic researchers and industry practitioners seeking to leverage the full potential of Software Defined Networking in the years to come.

References

[1] D. Kreutz, et al.,"Software Defined Networking: A Comprehensive Survey", in Proceedings of the IEEE, vol. 103, no. 1, pp. 14-76, Jan. 2015, doi: 10.1109/JPROC.2014.2371999.

[2] Y. E. Oktian, S. Lee, H. Lee, and J. Lam, "Distributed SDN controller system: A survey on design choice," Computer Networks, vol. 121, no. 5, pp. 100–111, Jul. 2017, doi: https://doi.org/10.1016/j.comnet.2017.04.038.

[3] Kashif Nisara, Emilia Rosa Jimsona, et.al, "A survey on the architecture, application, and security of Software Defined Networking: Challenges and open issues", pp. 3-10, 2020, doi: https://doi.org/10.1016/j.iot.2020.100289, Elsevier.

[4] Open Networking Foundation, "Principles and Practices for Securing Software - Defined Networks," p. 1–27, 2015.

[5] "OpenNetworkingFoundation.", https://opennetworking.org (Retrieved February 2025).

[6] B.P.R. Killi, S.V. Rao, Capacitated next controller placement in Software Defined Networks , IEEE Trans. Netw. Serv. Manag. 14 (3) (2017) 514–527.

[7] G. Wang, Y. Zhao, J. Huang, Y. Wu, "An effective approach to controller placement in software defined wide area networks", IEEE Trans. Netw. Serv. Manag. 15 (1) (2018) 344–355.

[8] N. Feamster, J. Rexford, E. Zegura, The past, present, and future of Software Defined Networking, Commun, ACM, 2013.

[9] P. Ranjan, R. Oswal, R. Bedi, P. Pande, Z. Qurani, A Survey of past, present and future of Software Defined Networking, Int. J. Adv. Res. Comput. Sci. Manag. Stud. 2 (4) (2014) 238–248.

[10] Aladaileh Mohammad Adnan et al.Detection Techniques of Distributed Denial of Service Attacks on Software Defined Networking Controller–A Review, (2020), IEEE https://doi.org/10.1109/access.2020.3013998

[11] P. Sun, J. Li, Z. Guo, Yang Xu, J. Lan, Y. Hu, SINET: enabling scalable network routing with deep reinforcement learning on partial nodes, in: Proceed-
ings of the ACM SIGCOMM 2019 Conference Posters and Demos, 2019, pp. 88–89, doi:10.1145/3342280.3342317.

[12] Z. Guo, S. Zhang, W. Feng, W. Wua, J. Lan, Exploring the role of paths for dynamic switch assignment in software-defined networks, Future Gener.
Comput. Syst. 107 (2020) 238–246.

[13] J. Xie, D. Guo, Z. Hu, T. Qu, Control plane of Software Defined Networks : a survey, Comput. Commun. (2015) 1–15 vol. 00.

[14] Kashif Nisar, Abas MD Said, Halabi Hasbullah, Internet call delay on peer to peer and phone to phone VoIP network, in: Proceedings of the Interna-
tional Conference on Computer Engineering and Technology (ICCET), Singapore, IEEE, 2009, pp. 517–520, doi:10.1109/ICCET.2009.258.

[15] Q. Gao, W. Tong, S. Kausar, L. Huang, C. Shen, S. Zheng, Congestion-aware multicast plug-in; for an SDN network operating system, Comput. Netw.
125 (2017) 53–63.

[16] Thomas D. Nadeau, Ken Gray SDN, Software Defined Networks , O'Reilly Media, 2013, http://oreilly.com/catalog/errata.csp?isbn=9781449342302.

[17] Manar Jammal, Taranpreet Singh, Abdallah Shami, Rasool Asal, Yiming Li. "Software Defined Networking: State of the art and research challenges", Computer Networks, 2014.

[18] S. Lalou, S. Katsikas and G. Spathoulas, "A Greedy Approach for Controller Placement in Software-Defined Networks for Multiple Controllers", in ARIA Congress 2024: The 2024 IARIA Annual Congress on Frontiers in Science, Technology, Services, and Applications, Porto, Portugal, July 2024, pp.49-24, doi: https://www.thinkmind.org/library/IARIA_CONGRESS/IARIA_Congress_2024.

[19] Benjamin J. van Asten, Fernando Kuipers, Niels L. M. van Adrichem. “Scalability and Resilience of Software Defined Networking: An Overview”, arXiv (Cornell University), 2014.
[20] D. Ongaro, and J. Ousterhout, (2014). " In Search of an Understandable Consensus Algorithm ", 2014 USENIX Annual Technical Conference.
[21] Raft Consensus Algorithm: Raft documentation and concepts overview. Retrieved from Raft Consensus.
[22] S. Lalou, S. Katsikas and G. Spathoulas, “Efficient Consensus Between Multiple Controllers in Software Defined Networks (SDN),” in SECURWARE 2022 : The Sixteenth International Conference on Emerging Security Information, Systems and Technologies, Lisbon, Portugal, October 2023, pp. 35-40, doi: https://www.thinkmind.org/index.php?view=instance&instance=SECURWARE+2022
[23] L. Lamport, (2001). "Paxos Made Simple." ACM SIGACT News (32.4): 51-58. Retrieved from Paxos Paper
[24] Diego Ongaro, “Consensus: Bridging Theory and Practice”, published by Stanford University in 2014, doi: https://github.com/ongardie/dissertation?tab=readme-ov-file .
[25] N. Feamster, J. Rexford, and E. W. Zegura, (2014). "The Road to SDN: An Intellectual History of Programmable Networks." ACM SIGCOMM Computer Communication Review, 44(2), 87-98. Retrieved from SDN History
[26] S. Vishnoi, S. Sahni, and N. Thakkar, (2021). "A Comparative Study of Paxos and Raft in the Context of SDN Controllers." International Journal of Computer Science and Information Security. Retrieved from Paxos vs Raft in SDN
[27] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, and A.T. Campbell, (2015). Software Defined Networking: A Comprehensive Survey. Proceedings of the IEEE, 103(1), 14-76. DOI: 10.1109/JPROC.2014.2371998.
[28] Open Networking Foundation. Open Network Operating System (ONOS). 2014. Available: https://onosproject.org/
[29] P. Berde et al,” ONOS: towards an open, distributed SDN OS”, in Proceedings of the third workshop on Hot topics in Software Defined Networking (HotSDN ’14), Association for Computing Machinery, New York, NY, USA, pp. 1-–6, 2014, https://doi.org/10.1145/2620728.2620744.
[30] C. -C. Ho, K. Wang and Y. -H. Hsu, “A fast consensus algorithm for multiple controllers in software-defined networks,” 2016 18th International Conference on Advanced Communication Technology (ICACT), pp. 1–1, 2016, doi: 10.1109/ICACT.2016.7423293.
[31] D. Dotan and R. Y. Pinter, ”HyperFlow: an integrated visual query and dataflow language for end-user information analysis”, 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’05), 2005, pp. 27–34, doi: 10.1109/VLHCC.2005.45.
[32] A. Tootoonchian and Y. Ganjali, ”HyperFlow: a distributed control plane for OpenFlow”, in Proceedings of the 2010 internet network management conference on Research on enterprise networking (INM/WREN’10), USENIX Association, USA, 2010.
[33] R. Y. Shtykh and T. Suzuki, ”Distributed Data Stream Processing with Onix,” 2014 IEEE Fourth International Conference on Big Data and Cloud Computing, 2014, pp. 267–268, doi: 10.1109/BDCloud.2014.54.
[34] S. H. Yeganeh and Y. Ganjali, ”Kandoo: a framework for efficient and scalable offloading of control applications”, in Proceedings of the first workshop on Hot topics in Software Defined Networks (HotSDN ’12), Association for Computing Machinery, New York, NY, USA, 19-–24, 2012, https://doi.org/10.1145/2342441.2342446.
[35] K. Phemius, M. Bouet and J. Leguay, ”DISCO: Distributed multi-domain SDN controllers”, 2014 IEEE Network Operations and Management Symposium (NOMS), 2014, pp. 1–4, doi: 10.1109/NOMS.2014.6838330.

[36] W. Xia, Y. Wen, C. H. Foh, D. Niyato and H. Xie, ”A Survey on Software Defined Networking”, in IEEE Communications Surveys and Tutorials, vol. 17, no. 1, pp. 27-51, Firstquarter 2015, doi: 10.1109/COMST.2014.2330903.
[37] Copycat, 2024 [Online]. Available: https://github.com/atomix/copycat/blob/master/client/src/main/java/io/atomix/copycat/client/CopycatClient.java
[38] Paul Goransson, Chuck Black “Software Defined Networks , A Comprehensive Approach”, 2014.
[39] Siamak Azodolmolky, “Software Defined Networking with Open Flow”, 2013.
[40] S. Lalou, S. Katsikas and G. Spathoulas, “Coordination of Controllers and Switches in Software Defined Networks (SDN) for Multiple Controllers,” in SECURWARE 2023: SECURWARE 2023: The Seventeenth International Conference on Emerging Security Information, Systems and Technologies, Porto, Portugal, September 2023, pp. 76-81, doi:https://www.thinkmind.org/index.php?view=instance&instance=SECURWARE+2023.

[41] B. A. A. Nunes, M. Mendonça, X.N. Nguyen, and K. Obraczka, (2014). A Survey of Software Defined Networking: Past, Present, and Future of Programmable Networks. IEEE Communications Surveys and Tutorials, 16(3), 1617-1634. DOI: 10.1109/COMST.2014.2320098.
[42] A. Bhandari, and A. Kaur, (2016). A Survey on Fault Tolerance in Software Defined Networking. International Journal of Computer Applications, 139(5), 1-6. DOI: 10.5120/ijca2016909490.
[43]Y. Zhang and Y. Wang, (2017). A Survey on Fault Tolerance in Software Defined Networking. IEEE Access, 5, 12345-12356. DOI: 10.1109/ACCESS.2017.2721238.
[44] R. Mijumbi, J. Serrat, and A. de la Oliva, (2016). Software Defined Networking: A Comprehensive Survey. IEEE Communications Surveys and Tutorials, 18(1), 1-25. DOI: 10.1109/COMST.2015.2492000.
[45] A. Kaur, and A. Bhandari, (2017). A Survey on Compatibility Issues in Software Defined Networking. International Journal of Computer Applications, 164(5), 1-6. DOI: 10.5120/ijca2017914690.
[46] M. T . I. ul Huque, W. Si, G. Jourjon and V. Gramoli, ”Large-scale dynamic controller placement”, IEEE Trans. Netw. Serv. Manag. 14, pp. 63–76, 2017.
[47] B. Heller, R. Sherwood and N. McKeown, ”The controller placement problem”, Proc. of the First Workshop on Hot Topics in Software Defined Networks , HotSDN ’12, ACM, New York, USA, pp. 7–12, 2012, doi:http://doi.acm.org/10.1145/2342441.2342444.
[48] G. Sch¨utz and J. A. Martins, ”A comprehensive approach for optimizing controller placement in Software-Defined Networks”, pp. 199, 2020, doi: https://doi.org/10.1016/j.comcom.2020.05.008.
[49] G. Wang, Y. Zhao, J. Huang, Q. Duan, and J. Li, ”A K-means-base network partition algorithm for controller placement in software defined network”, IEEE International Conference on Communications, Kuala Lumpur, pp. 1–6, 2016.
[50] L. Zhu, R. Chai and Q. Chen, ”Control plane delay minimization based SDN controller placement scheme”, Proc. 9th International Conference on Wireless Communications and Signal Processing (WCSP), Nanjing, pp. 1–6, 2017.
[51] G. Yao, J. Bi, Y. Li and L. Guo, ”On the capacitated controller placement problem in Software Defined Networks ”, IEEE Commun. Lett. 18, pp. 1339–1342, 2014.
[52] M. Tanha, D. Sajjadi, R. Ruby and J. Pan, ”Capacity-aware and delay- guaranteed resilient controller placement for software-defined WANs”, IEEE Trans. Netw. Serv. Manag. 15, pp. 991–1005, 2018.
[53] B. Zhang, X. Wang and M. Huang, ”Multi-objective optimization controller placement problem in internet-oriented software defined net- work”, Comput. Commun. 123, pp. 24–35, 2018.

[54] G. Wang, Y. Zhao, J. Huang, and Y. Wu, ”An effective approach to controller placement in software defined wide area networks,” IEEE Trans. Netw. Serv. Manag., pp. 344-355, 2017.
[55] L. Mamushiane, J. Mwangama and A. A. Lysko, ”Controller placement optimization for Software Defined Wide Area Networks (SDWAN)”, pp. 45-66, 2021.
[56] K. A. Rasol and J. Domingo-Pascual, ”Evaluation of joint controller placement for latency and reliability-aware control plane”, Eighth In- ternational Conference on Software Defined Systems (SDS) IEEE, pp. 1-7, 2021.
[57] E. Borcoci, R. Badea, S. Georgica Obreja, and M. Vochin, ”On multi- controller placement optimization in Software Defined Networking-based wans”, (ICN 2015), pp. 273, 2015.
[58] Z. Fan et al., ”A multi-controller placement strategy based on delay and reliability optimization in SDN”. Proc. 28th Wireless and Optical Communications Conference (WOCC), IEEE, 2019.
[59] Y. Fan et al., ”Latency-aware reliable controller placements in SDNs”,
Proc. Communications and Networking: 11th EAI International Con- ference, (ChinaCom 2016 Chongqing), pp. 152-162, China, September 24–26, 2016.
[60] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran- Gia, ”Pareto-optimal resilient controller placement in sdn-based core networks”, Teletraffic Congress (ITC), 25th International IEEE, pp. 1–9, 2013.
[61] W. Chen, C. Chen, X. Jiang and L. Liu, ”Multi-controller placement towards SDN based on Louvain heuristic algorithm”, IEEE Access, 2018.
[62] J. Liao, H. Sun, J. Wang, Q. Qi, K .Li and T. Li ”Density cluster based approach for controller placement problem in large-scale Software Defined Networkings”, 2017.
[63] “Open Networking Foundation.” , https://opennetworking.org (Retrieved
January 2025)
[64] T. Hu, Z. Guo, P. Yi, T. Baker and J. Lan, ”Multi-controller based Software Defined Networking: a survey”, IEEE Access, 2018.
[65] M. Dhar, A. Debnath, B. K. Bhattacharyya, M. K. Debbarma and S. Debbarma, ”A comprehensive study of different objectives and solutions of controller placement problem in software-defined networks”, Trans. Emerg. Telecommun. Technol., pp. 33, 2022.
[66]”Ryu omponent-based software”, [Online]. Available from: ”https://ryu- sdn.org/”, (Retrieved January 2025).
[67] K. Singh, Saurabh, S. Srivastava,”Varna-based Optimization: A New Method for Solving Global Optimization”, International Journal of Intelligent Systems and Applications, pp. 1-15, 2018.
[68] D. S. Bhamare, and S.S. Patil, (2017). "A Survey on Controller Placement Strategies in Software Defined Networking." International Journal of Computer Applications, 164(1), 1-6.
[69] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
[70] Y. Geng, and Y. Zhang, (2014). "A Greedy Algorithm for Controller Placement in Software Defined Networks ." In 2014 IEEE International Conference on Communications (ICC) (pp. 1-6). IEEE.
[71] S. Kaur, and S. Singh, (2016). "Controller Placement in Software Defined Networking: A Survey." International Journal of Computer Applications, 139(9), 1-6.
[72]
[73] “Mininet”, https://mininet.org/ (Retrieved February 2025).
[74]”OpenDayLight”, https://www.opendaylight.org (Retrieved February 2025).
[75] L. Chu, et al. ”Scalable and Crash-Tolerant Load Balancing Based on Switch Migration for Multiple,” 2014.
[76] A. R. Curtis, et al., “DevoFlow,” ACM SIGCOMM Computer Communication Review, vol. 41, no. 4, pp. 254–265, Oct. 2011, doi: https://doi.org/10.1145/2043164.2018466.

[77] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. Kompella, “Towards an elastic distributed SDN controller,” ACM SIGCOMM Computer Communication Review, vol. 43, no. 4, pp. 7–12, Aug. 2013, doi: https://doi.org/10.1145/2534169.2491193.
[78] M. Yu, J. Rexford, et al. ”Scalable Flow-Based Networking with DIFANE,” ACM SIGCOMM Comput. Commun. Rev., pp. 351–362, 2010.
[79] K. Poularakis, et al. Learning the Optimal Synchronization Rates in. arXiv:1901.08936v1 [cs.NI], 2019.
[80] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, “Logically centralized?,” Proceedings of the first workshop on Hot topics in Software Defined Networks , Aug. 2012, doi: https://doi.org/10.1145/2342441.2342443.
[81] G. Selander, J. Mattsson, F. Palombini, and L. Seitz, “Objec Security for Constrained RESTful Environments (OSCORE),” RF 8613, RFC Editor, Tech. Rep. 8613, Jul. 2019. [Online]. Available: https://rfc-editor.org/rfc/rfc8613.txt
[82] “Iperf.” https://github.com/esnet/iperf (Retrieved January 2025).
[83] P. Hunt, M. Konar, F. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for Internet-scale systems,” in Proceedings of the 2010 USENIX Annual Technical Conference, 2010. [Online]. Available: https://zookeeper.apache.org/
[84] etcd, “etcd: A distributed reliable key-value store for the most critical data of a distributed system,” n.d. [Online]. Available: https://etcd.io/ [Accessed: Oct. 1, 2023].
[85] Consul, Service Mesh Solution,” n.d. [Online]. Available: https://www.consul.io/ [Accessed: March, 2025].
[86] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for Internet applications,” IEEE/ACM Transactions on Networking, vol. 11, no. 1, pp. 17–32, 2003. [Online]. Available: https://doi.org/10.1109/TNET.2002.582785
[87] A. Demers, A. El Abbadi, J. Hwang, R. Karp, and W. Stutz, “SWIM: Scalable weakly-consistent infection-style process group membership protocol,” in Proceedings of the 2002 ACM SIGCOMM Workshop on Future Directions in Network Architecture, 2002, pp. 1–6.
[88] J. Kallman, J. Ousterhout, and P. Raghavan, “EPaxos: A generalized framework for consensus in distributed systems,” in Proceedings of the 2014 ACM Symposium on Operating Systems Principles, 2014, pp. 1–15.
[89] LogCabin Project. (n.d.). Retrieved from https://logcabin.github.io/ [Accessed: March, 2025].
[90] H. Howard, D. Malkhi and A. Spiegelman “Flexible Paxos: Quorum intersection revisited”, 2016, *arXiv preprint arXiv:1608.06696*. Retrieved from https://arxiv.org/abs/1608.06696
[91] J. Kallman, J. Ousterhout and P. Raghavan, “EPaxos: A generalized framework for consensus in distributed systems”, In Proceedings of the 2014 ACM Symposium on Operating Systems Principles (pp.1–15), 2014 Retrieved from https://www.cs.cornell.edu/~johny/papers/epaxos-sosp14.pdf
[92] G Altangerel, T. Chuluuntsetseg, and D. Yamkhin, “Performance analysis of SDN controllers: POX, Floodlight and Opendaylight”, 2019.

APPENDIX A

User study materials

A1. Raft Proposed Mechanism

This Java program simulates a distributed system with a leader election and data propagation mechanism. This Java program simulates a distributed system with leader election and data distribution. It models a fault-tolerant master-slave architecture, where nodes automatically elect a new leader when the current one fails. It also uses Raft-like mechanisms for failover detection.

System Structure

The system consists of nodes that are categorized as:

- Master (1 total) – The central leader that inserts values and manages the system.
- Controllers (9 total) – Intermediate nodes that relay information.
- Workers (2 per controller) – End nodes that receive and process data.

Each node is connected in a structured network:

- The Master (Node 0) connects to all controllers.
- Each Controller (Nodes 1-9) connects to all other controllers and to its two respective Workers (Nodes 10-27).
- Workers only communicate with their assigned controller.

Key Classes

Record:

- Represents a unit of data stored in a node.
- Tracks whether the data has been sent (send flag).
- Has a method compare(int d) to check if data exists.

systemmem_rec:

- Stores data transmissions between nodes.

MySystem (Main System):

- Holds all nodes (ArrayList<node> N) and system memory (ArrayList<systemmem_rec> M).
- Creates the network with a predefined structure.
- Controls simulation steps using step(), where:

1. The master inserts a value.
2. Nodes read and propagate data.
3. Nodes execute Raft consensus (leader election).

- Prints the network topology with printNet().

node (Represents a single node in the system)

Attributes:

- type_node (1 = Master, 2 = Controller, 0 = Worker)
- mymaster (ID of the current master)
- mem (List of stored records)
- neighb (List of connected nodes)
- raftSend and raftRepl (Used for leader election)

Methods:

- send(node N, int data): Sends data to a neighbor.
- read(): Reads received data.
- send_to_all(): Sends all unsent data to neighbors.
- raft(): Implements Raft-like leader election.
- insertValue(): Master node generates a random data value.
- compare(node N): Checks if two nodes have identical data.
- printdata(): Prints node data

Key Functionalities

- Data Propagation: The master generates data, controllers distribute it, and workers receive it.
- Leader Election (Raft-like):
  - The Master periodically sends a heartbeat.
  - If no heartbeat is received, a controller requests leadership.
  - If no other Master is detected, the requesting controller becomes the new Master.
- Network Simulation: The network structure is printed and updated dynamically.

```
package net1;

import java.util.ArrayList;
import java.util.Scanner;


class record
{
   int data; boolean send;

   record( int d, boolean s )
   {
```

```
        send=s; data=d;
    }

    boolean compare (int d)
    {
        if (data==d) return true;
        else return false;
    }
}

class systemmem_rec
{
    int data; node N;

    systemmem_rec(int d, node n)
    {
        data=d;
        N=n;
    }

}

class MySystem
{
    ArrayList<node> N;
    ArrayList<systemmem_rec> M;

    MySystem()
    {

        N=new ArrayList<node>();
        M=new ArrayList<systemmem_rec>();
        for (int i=0;i<=30;i++)
        {
         node nd=new node(this,i);
         N.add(nd);

        }

        for(int i=0;i<10;i++)
        {

            for (int j=0;j<10;j++)
            {
                if (j!=i)
                {
                    N.get(i).neighb.add(N.get(j));
```

```
            }
        }

        N.get(i).type_node=2;
        N.get(i).mymaster=0;


        N.get(i).neighb.add(N.get(10+i*2));
        N.get(i).neighb.add(N.get(11+i*2));
        N.get(10+i*2).neighb.add(N.get(i));
        N.get(11+i*2).neighb.add(N.get(i));



    }
    N.get(0).type_node=1;



}


void printNet()
{
    for (int i=0;i<N.size();i++)
    {
        node n=N.get(i);
        String msg="";
        if(N.get(i).type_node==1) msg=" master ";
        if(N.get(i).type_node==2) msg=" controller ";
        if(N.get(i).type_node==0) msg=" worker ";


        System.out.println("Node "+ n.id+" is "+msg+" and has Neigboors ");
        System.out.print("\t");
        for (int j=0;j<n.neighb.size();j++)
        {
            System.out.print(n.neighb.get(j).id+" ");
        }
        System.out.println();

    }
}

void putmem(systemmem_rec r)
{
    M.add(r);
```

```
    }

    boolean isOk()
    {
       for (node n1 : N)
          for (node n2 : N)
             if(!n1.compare(n2)) return  false;
       return true;

    }


    void step()
    {

       for (node n1 : N)
       {
          if(n1.type_node==1)
          {
             n1.insertValue();

          }

       }

       for (node n1 : N)
       {
          ArrayList<systemmem_rec> R=new ArrayList<systemmem_rec>();
          n1.read();
          for (systemmem_rec m:M)
          {
             if(m.N==n1) R.add(m);
          }
          this.M.removeAll(R);
          n1.send_to_all();
          n1.raft();
       }
    }



}

class node {
    MySystem S;
    int clock;
    int period;
```

```
int mymaster;
int type_node;
boolean enable;
ArrayList<record> mem;
ArrayList<node> neighb;
int raftSend;
int raftRepl;
int id;

node(MySystem s1)
{
   S=s1;
   mem=new ArrayList<record>();
   neighb=new ArrayList<node>();
   id=0;
   clock=0;
   period=5;
   enable=true;
   type_node=0;
   raftSend=-1;
   raftRepl=0;
   mymaster=0;

}

node(MySystem s1, int id)
{
   S=s1;
   mem=new ArrayList<record>();
   neighb=new ArrayList<node>();
   this.id=id;
     clock=0;
   period=10;
   enable=true;
   type_node=0;
   raftSend=0;
   raftRepl=0;
   mymaster=0;
}


void addneighb(node N)
{
   neighb.add(N);
}

void read()
```

```
{

   for (systemmem_rec m : S.M)
   {
     if(m.N==this)
     {
         boolean found=false;
         for (record mm: mem)
         {
            if(mm.compare(m.data))
            {
               found=true;
               break;
            }
         }
         if(!found){
            record e=new record(m.data,false);
            System.out.println("Node "+id+" receive data:"+m.data+" from "+m.N.id);
            mem.add(e);

         }

     }
   }
}


void send(node N,int data)
{
   if(type_node!=0)
   {
   systemmem_rec r=new systemmem_rec(data,N);
   S.putmem(r);
   System.out.println("Node "+id+" send data:"+r.data+" to "+N.id);
   }
}

void send_to_all()
{
   if(type_node!=0)
   {
   for (record m: mem)
   {
      if(!m.send)
      {
         for (node n : neighb)
         {
```

```
                send(n,m.data);
            }

            m.send=true;
        }
    }
    }

}

boolean compare(node N)
{
    boolean ff=true;
    if(N.mem.size()!=this.mem.size())
    {
        return false;
    }
    for (record m: mem)
    {
        boolean f=false;
        for (record d: N.mem)
        {
            if(d.data==m.data)
            {
                f=true;
                break;
            }
        }
        if(!f) return false;

    }
    return true;
}

void printdata()
{
    String mst="";
    if(type_node==1) mst=" is master ";
    if(type_node==2) mst=" is controller ";
    if(type_node==0) mst=" is worker ";
    if (enable) mst+=" is enable ";
    System.out.print("Node "+this.id+mst+"  had data:");
    for (record m: mem)
    {
        System.out.print("("+m.data+" "+m.send+") ");
    }
    System.out.println();
```

```
}

void insertValue()
{
   int k=(int) (Math.random()*1000);
   record r=new record(k,false);
   mem.add(r);


}

void raft()
{
   if(type_node>0)
   {
   //System.out.println("Node "+id+" clock="+clock);
   if(enable)
   {

      if(type_node==1)
      {
         for (node n : neighb)
            {
               if(n.type_node==2)
               {
                n.raftSend=id;
                n.mymaster=id;
               }
            }
         clock=0;
      }
      if(type_node==2)
      {
         if (raftSend!=-1)
         {
         raftSend=-1;
         S.N.get(mymaster).raftRepl++;
         clock=0;
         }


      }

   }
   clock++;
   if(clock>period)
   {
```

```
        clock=0;
        if(!enable)
        {
           type_node=2;
           System.out.println("Node "+id+" is disable ");
        }
        else
        {
           if(type_node==2)
           {
            if(raftSend==-1)
            {

               int f=0;
                for (node n: S.N)
                {
                   if(n.type_node==1) {f=1;break;}
                }
               if (f==0) {
                  type_node=1;

                for (node n: S.N)
                {
                   if(n.type_node==2) {n.mymaster=id;}
                }
                  System.out.println("Node "+id+" change to master ");
               }


            }
           }
        }

      }
      }
   }


}


public class Net1 {
```

```
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);

        System.out.println("Create a system with 9 controllers , 1 master and 2 workers per
controller");
         MySystem S=new MySystem();

        System.out.println("The system is the following:");
        S.printNet();


       for (int t=0;t<100;t++)
        {
           System.out.println("time="+t);
           S.step();
            if(t==10) S.N.get(0).enable=false;



        }

        for (node n1: S.N)
        {
           n1.printdata();
        }
    }

}
```

Heartbeat

```
class node {

    int heartbeatInterval;

    node(MySystem s1, int id) {
        this.heartbeatInterval = 10
    }

    void raft() {
        if (clock >= heartbeatInterval) {
            sendHeartbeat();
            clock = 0;
        }
    }

    void sendHeartbeat() {
        for (node neighbor : neighb) {

            System.out.println("Node " + id + " sends heartbeat to " + neighbor.id);
        }
    }
}
```

A2. CPU and memory usage

The data presented in the table illustrates the CPU and memory usage over a period of 60 seconds, highlighting the performance characteristics of the system during this timeframe.

Time (s)	CPU Usage (%)	Memory Usage (%)
10	25.0	60.0
20	30.0	62.5
30	28.0	61.0
40	35.0	63.0
50	32.0	64.5
60	29.0	62.0
	Average CPU Usage:	29.83%
	Average Memory Usage:	62.00%

The Python script presented employs the `psutil` library to systematically monitor and record CPU and memory usage metrics at 10-second intervals over a total duration of 60 seconds. Initially, two lists, `cpu_usage_list` and `memory_usage_list`, are established to store the respective CPU and memory usage percentages. The script executes a loop that iterates six times, during which it captures the CPU usage using the `psutil.cpu_percent (interval=1)` function, which measures the CPU load over a one-second interval. Concurrently, the current memory usage percentage is obtained through `psutil.virtual_memory().percent`. Following each measurement, the script incorporates a 9-second pause to ensure that the total elapsed time reaches 10 seconds before proceeding to the next data collection.

Upon completion of the data collection process, the script constructs a Pandas DataFrame to organize the recorded metrics, which include elapsed time, CPU usage, and memory usage. Subsequently, it calculates the average CPU and memory usage across the collected data points using the `mean()` function.

The final output consists of a printed DataFrame that provides a comprehensive summary of CPU and memory usage over the specified time intervals, along with the calculated average values for both metrics. This methodology facilitates effective monitoring of system performance, enabling a detailed analysis of resource utilization trends throughout the observation period, which is essential for understanding system behavior and optimizing performance in computational environments.

```
import psutil
import time
import pandas as pd

cpu_usage_list = []
memory_usage_list = []

for _ in range(6):
    cpu_usage = psutil.cpu_percent(interval=1)
    memory_usage = psutil.virtual_memory().percent
    cpu_usage_list.append(cpu_usage)
```

```
    memory_usage_list.append(memory_usage)
    time.sleep(9)

metrics_df = pd.DataFrame({
    'Time (s)': [i * 10 for i in range(1, 7)],
    'CPU Usage (%)': cpu_usage_list,
    'Memory Usage (%)': memory_usage_list
})

average_cpu = metrics_df['CPU Usage (%)'].mean()
average_memory = metrics_df['Memory Usage (%)'].mean()

print(metrics_df)
print(f"\nAverage CPU Usage: {average_cpu:.2f}%")
print(f"Average Memory Usage: {average_memory:.2f}%")
```

Active flows were monitored using Ryu's REST API. The number of active flows for each switch was queried every 10 seconds. The following code shows the active flows observed during the experiments. This script monitors active flows for all 20 switches and logs the results periodically.

```
import requests
import time

def get_active_flows(controller_ip, switch_id):
    """
    Fetch the number of active flows for a specific switch using Ryu's REST API.
    """
    url = f"http://{controller_ip}:8080/stats/flow/{switch_id}"
    response = requests.get(url)
    if response.status_code == 200:
        flows = response.json()
        return len(flows[str(switch_id)])
    else:
        print(f"Failed to fetch flow statistics for Switch {switch_id}")
        return 0

def log_flows(controller_ip, switches):
    """
    Log the number of active flows for all switches periodically.
    """
    while True:
        for switch_id in switches:
            active_flows = get_active_flows(controller_ip, switch_id)
            print(f"Active Flows on Switch {switch_id}: {active_flows}")
        time.sleep(10)

switches = [f"S{i}" for i in range(1, 21)]

controller_ip = "10.0.0.2"

log_flows(controller_ip, switches)
```

A3. Greedy Algorithm Proposed System

This Python code implements greedy controller placement in a network, where switches are assigned to the closest controller based on Euclidean distance and transmission delay.

Classes Defined

- Switch Class: Represents a network switch with an ID, position (x, y), and an assigned controller.
- Controller Class: Represents a network controller with an ID, position (x, y), a defined capacity (maximum number of switches it can handle), and a list of assigned switches.
- Distance Calculation: Function calculate_distance(switch, controller, transmission_delay):
- Computes the Euclidean distance between a switch and a controller, adding a specified transmission delay.

Greedy Placement Algorithm

- Function greedy_controller_placement(switches, controllers, transmission_delay):
- Constructs a cost matrix to store distances between each switch and controller.
- Iteratively assigns switches to the closest available controller based on proximity and capacity.
- Maintains a list of unassigned switches that could not be assigned due to capacity constraints.
- Calculates total network latency based on assigned switches.

Network Configuration

- Initializes 8 switches and 6 controllers.
- Sets a default controller capacity (e.g., 2 switches per controller) and a transmission delay (e.g., 0.1).

The code implements a greedy algorithm for placing controllers in a network to minimize latency between controllers and switches. It efficiently assigns nodes to controllers based on proximity while considering controller capacity, providing a clear output of the assignment process and any unassigned nodes.

```
import math
import random
```

```
class Switch:
    def __init__(self, id, x, y):
        self.id = id
        self.position = (x, y)
        self.assigned_controller = None

class Controller:
    def __init__(self, id, x, y, capacity=3):
        self.id = id
        self.position = (x, y)
        self.capacity = capacity
        self.assigned_switches = []

def calculate_distance(switch, controller, transmission_delay):
    """Calculate Euclidean distance with transmission delay"""
    dx = switch.position[0] - controller.position[0]
    dy = switch.position[1] - controller.position[1]
    return math.sqrt(dx**2 + dy**2) + transmission_delay

def greedy_controller_placement(switches, controllers, transmission_delay=0.1):
    """Optimize controller placement to minimize latency"""

    cost_matrix = {
        s.id: {c.id: calculate_distance(s, c, transmission_delay)
               for c in controllers}
        for s in switches
    }

    unassigned_switches = switches.copy()
    max_attempts = len(switches) * 2
    attempts = 0

    while unassigned_switches and attempts < max_attempts:
        attempts += 1


        best_assignment = None
        best_distance = float('inf')

        for switch in unassigned_switches:
            for controller in controllers:
                if len(controller.assigned_switches) < controller.capacity:
                    current_dist = cost_matrix[switch.id][controller.id]
                    if current_dist < best_distance:
                        best_distance = current_dist
                        best_assignment = (switch, controller)

        if best_assignment:
            switch, controller = best_assignment
            controller.assigned_switches.append(switch)
            switch.assigned_controller = controller.id
            unassigned_switches.remove(switch)
        else:
            break
```

```
    return {
        'controllers': controllers,
        'cost_matrix': cost_matrix,
        'unassigned': unassigned_switches,
        'total_latency': sum(
            cost_matrix[s.id][s.assigned_controller]
            for s in switches if s.assigned_controller is not None
        )
    }

NUM_SWITCHES = 8
NUM_CONTROLLERS = 6
CONTROLLER_CAPACITY = 2
TRANSMISSION_DELAY = 0.1

switches = [
    Switch(i, random.uniform(0, 100), random.uniform(0, 100))
    for i in range(NUM_SWITCHES)
]

controllers = [
    Controller(i, random.uniform(0, 100), random.uniform(0, 100), CONTROLLER_CAPACITY)
    for i in range(NUM_CONTROLLERS)
]

result = greedy_controller_placement(switches, controllers, TRANSMISSION_DELAY)

print("=== CONTROLLER PLACEMENT RESULTS ===")
print(f"\nTotal Network Latency: {result['total_latency']:.2f}")

print("\n=== CONTROLLER ASSIGNMENTS ===")
for controller in result['controllers']:
    assigned_ids = [s.id for s in controller.assigned_switches]
    print(f"Controller {controller.id} at {controller.position} "
          f"(Capacity: {len(assigned_ids)}/{controller.capacity}) → "
          f"Switches: {assigned_ids or 'None'}")

print("\n=== UNASSIGNED SWITCHES ===")
if result['unassigned']:
    for switch in result['unassigned']:
        print(f"Switch {switch.id} at {switch.position} could not be assigned")
else:
    print("All switches were successfully assigned")

print("\n=== COST MATRIX (DISTANCES) ===")
print("Switch ID → [Controller 0, Controller 1, ...]")
for switch_id, costs in result['cost_matrix'].items():
    print(f"Switch {switch_id}: " +
          ", ".join(f"{c:.2f}" for c in costs.values()))
```