Failure Resilience and Traffic Engineering for Multi-Controller Software Defined Networking

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Nicu Florin Zaicu

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2022

Abstract

This thesis explores and proposes solutions to address the challenges faced by Multi-Controller SDN (MCSDN) systems when deploying TE optimisation on WANs. Despite the interest from the research community, existing MCSDN systems present limitations. For example, TE optimisation systems are computationally complex, have high consistency requirements, and need network-wide state to operate. Because of such requirements, MCSDN systems can encounter performance overheads and state consistency problems when implementing TE. Moreover, performance and consistency problems are more prominent when deploying the system on WANs as these network types have higher inter-device latency, delaying state propagation.

Unlike existing literature, this thesis presents several design choices that address all four challenges affecting MCSDN systems (scalability, consistency, resilience, and coordination). We use the presented design choices to build Helix, a hierarchical MCSDN system. Helix provides better scalability, performance and failure resilience compared to existing MCSDN systems by sharing minimal state between controllers, offloading operations closer to the data plane and deploying lightweight tasks.

A challenge that we faced when building Helix was that existing TE algorithms did not meet Helix's design choices. This thesis presents a new CSPF-based TE algorithm that needs minimal state to operate and supports offloading inter-area TE to local controllers, fulfilling Helix's requirements. Helix's TE algorithm provides better performance and forwarding stability, addressing 1.6x more congestion while performing up to 29x fewer path modifications than the other algorithms evaluated in our experiments.

While MCSDN literature has explored evaluating different aspects of system performance, there is a lack of readily available tools and concrete testing methodologies. To this end, this thesis provides concrete testing methodologies and tools readily available to the MCSDN community to evaluate the data plane failure resilience, control plane failure resilience, and TE optimisation performance of MCSDN systems.

Acknowledgements

First, I would like to acknowledge and thank my supervisors, Richard Nelson, Matthew Luckie, and Marinho Barcellos, for guiding me through my PhD and for the countless hours they spent proofreading my work and providing me with feedback. This work would not have been possible without their help.

I would like to thank my family and friends for all the support and time spent over the years. I would particularly like to thank my parents, Nicu and Carolina, and my brother Alex, for always supporting and encouraging me.

I want to thank all the members of the WAND research group who made my time at the University very enjoyable. I would particularly like to thank Brad Cowie and Richard Sanger for their help during my PhD. Finally, I would like to wish Chris and Dimeji all the best with their PhDs.

This work was funded, in part, by a University of Waikato Doctoral scholarship and department funding.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

SDN Software-Defined Networking

SCSDN Single-Controller SDN

MCSDN Multi-Controller SDN

LC Local Controller

RC Root Controller

TE Traffic Engineering

MPLS Multiprotocol Label Switching

OSPF Open-Shortest Path First

MPLS-TE Multiprotocol Label Switch with TE

BGP-TE Border Gateway Protocol with TE

OSPF-TE Open-Shortest Path First with TE

RSVP-TE Resource Reservation Protocol with TE

BFD Bidirectional Forwarding Detection

LLDP Link Layer Discovery Protocol

LoS Loss of Signal

QoS Quality of Service

QoE Quality of Experience

RTT Round-Trip Time

DC Data Centre

WAN Wide Area Network

ECMP Equal Cost Multi Path

MCF Multi Commodity Flow

VLB Valiant Load Balancing

CSPF Constrained Shortest Path First

AMQP Advanced Message Queuing Protocol

CPP Controller Placement Problem

CPLB Control Plane Load Balancing

UM Unidirectional Management

BM Bidirectional Management

SR Segment Routing

PCE Path Computation Element

Chapter 1

Introduction

1.1 Problem Statement

Multi-Controller SDN (MCSDN) systems address the scalability and performance concerns raised when using a single controller to manage a network. MCSDN systems divide the network into areas and deploy a controller in each area, reducing the load on controllers and decreasing switch-controller latency. Despite this, MCSDN systems are far more complex to implement and require solutions to deal with four challenges: scalability, resilience, consistency, and coordination. While MCSDN has received significant research attention, existing systems present performance, scalability and resilience issues when deploying critical network applications such as Traffic Engineering (TE).

For example, TE optimisation systems (e.g. SWAN [36], Espresso [99], and SMORE [53]) are computationally complex, have high state consistency requirements, and need network-wide state to operate. An MCSDN system can share state between controllers using strong or eventual consistency mechanisms. Strong consistency guarantees that state is always up to date (a requirement of TE algorithms) at the expense of introducing performance overheads [9, 70]. Strong consistency requires all controllers (quorum) to agree on state changes before applying them, delaying operations and decreasing performance. Eventual consistency sacrifices correctness for performance by

applying updates immediately without quorum agreement. Despite removing the performance overheads, eventual consistency can cause a system to make TE decisions based on inconsistent state, causing policy violation, packet loss, and decreased forwarding stability.

Performance overheads and state consistency problems are more prominent when deploying the system on a Wide-Area Network (WAN). WANs contain high inter-device latency, which delays quorum agreement and update propagation. Delaying quorum agreement makes strong consistency mechanisms slower to apply TE decisions, decreasing performance. Higher latency delays update propagation of eventual consistency mechanisms, increasing the chances that a controller makes decisions based on inconsistent state. Because of these problems, deploying TE on WANs is challenging as these networks strain MCSDN systems.

This thesis explores and proposes solutions to address the challenges faced by MCSDN systems when deploying TE on WANs. Existing MCSDN systems raise concerns as they considered the four challenges affecting MCSDN systems in isolation, making design choices that boast good performance addressing one challenge at the expense of another. Furthermore, most MCSDN literature has either not considered TE optimisation or analysed how their system affects TE performance. Systems often make design choices that degrade performance or make deploying TE impossible due to architectural constraints. In contrast, this thesis considered all four challenges, presenting solutions to improve failure resilience and scalability without sacrificing TE performance.

1.2 Contributions

This thesis provides four main contributions. First, this thesis presents Helix, a hierarchical MCSDN system that combines various techniques to deploy TE on WANs. Towards achieving this contribution, this thesis presents a literature review that contrasts different systems and approaches that address the four

challenges faced by MCSDN systems. Based on the identified issues with existing systems, this thesis proposes several design choices to build Helix. Helix provides better scalability, performance and failure resilience compared to existing systems by sharing minimal state between controllers, offloading operations closer to the data plane, and deploying computationally lightweight tasks. Helix demonstrates how to apply the design choices proposed in this thesis when building an MCSDN system, proving their feasibility. Moreover, Helix offers a concrete implementation enabling this thesis to evaluate the proposed design choices.

Second, this thesis presents a CSPF-based TE optimisation algorithm that is suitable for use with Helix. A challenge that we faced when building Helix was that existing TE algorithms require network-wide state to operate, going against Helix's design choices. The presented TE algorithm requires less state and supports offloading inter-area TE optimisation closer to the data plane. While we tailored the TE algorithm to Helix's requirements and deployment conditions, it is easily usable in other systems.

Third, this thesis provides concrete testing methodologies and tools available to the MCSDN community to evaluate data plane failure resilience, control plane failure resilience, TE optimisation performance, and forwarding stability of systems. While MCSDN literature has explored assessing various aspects of MCSDN systems, there is a lack of tools and concrete testing methodology available to the research community. Moreover, most presented evaluation approaches are specifically tailored to a system or do not consider the effects of the MCSDN system on critical network applications. We used the proposed evaluation tools to evaluate Helix's performance.

Finally, this thesis answers two research questions. (1) "What are the effects of loss of centralised scope on TE optimisation performance?". (2) "What are the effects of using multiple versus a single controller on system performance?". Literature has identified that SDN's centralised control-plane benefits network applications such as TE [61]. Distributed or non-logically

centralised MCSDN systems such as Helix lose their centralised scope. These systems allow controllers deployed in an area to perform decisions locally without needing network-wide state. While offloaded or local operations benefit resilience and remove the effects of latency on operation completion time, questions regarding the impact of losing global visibility on performance remain. This thesis explores the effects of using multiple controllers on performance by considering the two inter-area operations Helix offloads to local controllers, causing them to lose global visibility.

We make Helix's source code, and evaluation frameworks available to the research community [1, 2]. While Helix is an OpenFlow based controller built using Ryu, our assessment and proposed design choices are not limited to OpenFlow systems. The identified challenges, solutions, and presented tools have broader application to other non-OpenFlow SDN based protocols and approaches. We will discuss this further in chapter 9.

1.3 Thesis Structure

Chapter 2 introduces the relevant background for this thesis, presenting the main concepts and themes covered by this work. The chapter introduces data plane resilience by presenting several failure detections and recovery techniques. Next, it introduces TE optimisation methods and approaches, covering different trends in literature. Finally, the chapter introduces the four challenges that affect MCSDN systems and discusses the benefits of using single versus multiple controllers.

Chapter 3 contains a literature review of related work that has designed and implemented MCSDN systems. The literature review proposes several design choices that avoid the limitations of existing MCSDN systems.

Chapter 4 presents the design and architecture of Helix, an MCSDN system that uses the proposed design choice to address the issues faced when deploying TE on WANs. This chapter contrasts Helix's design choices against existing

systems, highlighting and discussing the benefits and issues of our approach.

Chapter 5 discuses Helix's implementation in depth.

Chapters 6 evaluates Helix's data plane resilience, comparing Helix's protection-based failure recovery against restoration-based recovery. While comparing protection versus restoration is not novel [84, 85], existing work has not investigated how latency affects a system's failure recovery performance.

Chapter 7 evaluates Helix's control plane resilience. While existing MCSDN literature has evaluated various aspects of MCSDN system performance, control plane failure resilience has received little attention. To this end, this chapter provides a concrete testing methodology and tools to evaluate a system's response to control plane failures. The presented emulation framework helps profile the speed with which an MCSDN system can recover from failures. The framework also provides insight into the system's capabilities, calculating metrics that enable planning for administrative events like restarting instances to apply upgrades.

Chapter 8 evaluates Helix's TE optimisation performance and forwarding stability. The final evaluation experiment of this chapter provides the fourth contribution of this thesis by collecting results using both a single and multi-controller deployment. We used the collected results to answer the two research questions from a TE optimisation perspective.

The three evaluation chapters provide the third contribution of this thesis by presenting concrete testing methodologies and tools that the research community can use to assess the performance of MCSDN systems.

Chapter 9 discusses the applicability of this work to other SDN management approaches.

Finally, chapter 10 concludes this thesis by summarising findings and presenting future research directions for this work. This chapter revisits the two proposed research questions, discussing how using multiple controllers affects the performance of data plane failure recovery and TE optimisation.

Chapter 2

Background

This chapter introduces the core topics covered in this thesis: data plane failure resilience, Traffic Engineering (TE), and Multi-Controller SDN (MCSDN). §2.1 provides a brief introduction to data plane failure resilience, discussing common failure detection and recovery mechanisms. §2.2 serves as an introduction to TE, showcasing approaches and trends in TE literature. Finally, §2.3 contains a brief introduction to MCSDN systems, discussing the motivation and advantages of using either multiple or a single controller.

2.1 Data Plane Failure Resilience

Data plane failures interrupt the flow of packets in the network, causing packet loss. Data plane failure resilience refers to a system's ability to *tolerate failures of forwarding devices*. We divide failure resilience into two main components: detection (§2.1.1) and recovery (§2.1.2).

2.1.1 Failure Detection

A system can detect failures using either active or passive mechanisms.

Active Failure Detection: Active detection mechanisms require a system to flood probe packets through the topology. A system uses the flooded probes to validate the network's forwarding behaviour and locate failures by

checking that a link or path forwards traffic. Active detection mechanisms also enable network operators to identify misconfigured devices that cause erratic forwarding behaviour or complex failures such as Byzantine faults [55]. Researchers have used two protocols to perform active detection in traditional and Software-Defined Networking (SDN) contexts.

First, Bidirectional Forwarding Detection (BFD) [48] is a protocol that tests forwarding on links or paths. The protocol requires two endpoints to establish a connection and exchange heartbeat packets at predetermined intervals. An endpoint that receives a heartbeat packet will echo the heartbeat back to the sender, testing bidirectional connectivity. An endpoint considers a link or path as failed if it does not receive several consecutive heartbeats.

Second, Link Layer Discovery Protocol (LLDP) [42] is another popular active detection mechanism used in literature. While LLDP allows detecting any link failures in a topology, its primary purpose is to perform topology discovery. LLDP uses heartbeats to detect topology changes.

Because active detection mechanisms rely on probes, they suffer from three main problems. First, suppose the network is congested and dropping packets; in this situation, the active detection mechanism can incorrectly declare a link as failed, despite it still functioning. Second, active probing increases the load on links, though, in practice, the extra traffic generated by probe packets is negligible [95]. Third, active detection uses timeouts, delaying failure detection and decreasing recovery performance. Slow failure recovery increases packet loss, negatively affecting the systems forwarding performance.

Passive Failure Detection: Unlike active detection, passive failure detection mechanisms do not use heartbeat packets to detect failures. Instead, a system monitors regular data plane traffic, inferring and locating device failures from packet loss. In 2009, Friedl et al. [29] proposed a passive failure detection mechanism for SDN (OpenFlow). The controller installs forwarding rules for flows with expiration timers. The switch removes the rule and notifies the controller when a rule does not match packets for a specified timeframe.

The controller uses the expiry notification to infer and locate data plane failures. In 2010, Fioccola et al. [26] describe a passive failure detection technique that uses packet colouring. The method flags packets with different colours by applying an MPLS or VLAN tag to data plane traffic. A system checks the count of packets observed on switches at predetermined intervals to determine where loss has occurred in the topology.

Both passive and active mechanisms can suffer from similar problems. For example, passive and active failure detection mechanisms rely on packets travelling through the network to detect failures. If a network is congested and dropping packets, both will report false positive link failures. Like active detection, passive detection mechanisms can suffer from slow recovery performance depending on the configured polling interval. A critical difference between the two approaches is that passive detection generates no extra traffic and thus does not introduce extra load on links. Some SDN passive detection mechanisms use Loss of Signal (LoS) to allow a data plane device to respond to failures locally. For example, OpenFlow provides fast-failover groups. The fast-failover group typically uses LoS to decide if a particular link has failed. Fast-failover groups are affected by neither packet loss nor slow failure detection time because the detection event occurs locally.

2.1.2 Failure Recovery

The second part of data plane failure resilience is recovery. After the system has located a failure in the topology, it needs to modify the installed paths to restore traffic forwarding. Researchers have considered a fast or carrier-grade recovery [64] where a system restores traffic forwarding within 50ms as desirable. The 50ms requirement stems from video conferencing traffic, where a higher delay is considered detrimental to the service. For traditional networks, Atlas et al. propose fast IP reroute [6], a mechanism that achieves a carrier-grade recovery by pre-installing multiple paths onto the data plane and enabling devices to independently reroute traffic in response to failures.

We divide failure recovery mechanisms into two main categories: restoration and protection recovery.

Restoration Recovery: In restoration-based recovery, the system intervenes in the recovery decisions by modifying the installed forwarding paths to divert traffic away from a failed element. We refer to restoration as reactive recovery for SDN systems as the process involves the controller in all recovery decisions. In 2011, Staessens et al. [88] implemented a restoration-based SDN controller that targets a sub 50ms failure recovery time. After testing their implementation, the authors found that restoration recovery failed to meet carrier-grade requirements. In 2013, Sharma et al. implemented and compared protection and restoration-based recovery in an SDN context using in-band [84] and out-of-band [85] control plane deployments. Similar to the findings of Staessens et al., the authors found that restoration did not meet the requirements for a carrier-grade recovery. Moreover, restoration with an in-band deployment delays recovery time because the controller needs to restore control plane connectivity with isolated switches before dealing with the failure. Based on the literature, restoration recovery presents three flaws:

1. Recovery performance is affected by inter-device latency as the controller is involved in the recovery decision. A higher control-channel latency delays recovery, increasing the amount of packet loss and thus decreasing forwarding performance.

2. Recovery is slower when using in-band deployment because the controller first needs to restore connectivity with the data plane.

3. Restoration is a reactive task. Reactive tasks introduce dependencies between the controller and switches, affecting control plane failure resilience. If a data plane device loses controller connectivity, the network will no longer recover from data plane failures.

Moreover, problems are more prominent when considering deploying the system on Wide Area Networks (WANs). WANs have high inter-device latency

which contributes to poor recovery performance.

Protection Recovery: Protection-based failure recovery uses precomputed paths to restore forwarding. We refer to protection as proactive recovery for SDN because switches recover from failures without controller intervention. Researchers have demonstrated that protection can achieve a carrier-grade recovery when using an in-band or out-of-band SDN deployment [84, 85].

In 2015 and 2016, Capone et al. [11] and Cascone et al. [12], proposed and explored implementing a protection-based failure recovery system using OpenState. OpenState is an extension to OpenFlow that allows switches to make stateful decisions. An OpenState controller can push flow rules and finite state machines onto the data plane, enabling switches to make decisions independent of the controller. The switches perform local failure detection and recovery based on observed data plane packets. When a switch detects a link failure, it uses *crank-back routing* to send packets back to the previous nodes on the path. The backtracked packets notify nodes of the failure and allow the system to find an appropriate reroute point. When the backtracked packet reaches a node with a backup route, that switch will modify the installed forwarding rules to make packets use the new path. Despite the presented systems achieving a carrier-grade recovery, it raises two problems. First, *crank-back routing* increases forwarding latency by temporarily increasing path lengths. Second, the systems can have limited deployability as OpenState requires switch hardware and software modifications.

Conclusion: Protection provides three main benefits:

1. Latency does not affect proactive operations, making the method suitable for use in WANs.

2. The system does not need to restore control plane connectivity before recovering from data plane failures for in-band deployments.

3. Protection allows switches to independently deal with data plane failures even if they lose control plane connectivity, improving the system's

control plane failure resilience.

Based on these observations, we conclude that protection recovery deals with failures faster, minimising packet loss. A concern with protection recovery is the optimality of paths. Protection mechanisms use precomputed information to deal with failures. Due to topology changes, preinstalled recovery paths can lose optimality. Implementing protection recovery in conjunction with TE will alleviate some optimality concerns.

2.2 Traffic Engineering (TE)

TE optimises packet forwarding through the modelling characterisation and control of traffic to achieve a specific performance objective [7]. A TE system modifies forwarding paths based on gathered metrics to meet current demands and achieve concrete performance goals. Depending on the selected goal, the TE system collects different metrics. For example, a TE system that aims to minimise congestion will collect link usage metrics, while one that aims to reduce power consumption gathers the power usage of devices.

Multiple longitudinal studies of internet traffic found that bandwidth demand is steadily increasing, putting more strain on TE systems [27, 14, 93]. For example, in their study conducted over five years (2013-2017) at an ISP, Trevisan et al. found that customers doubled their bandwidth consumption in 2017 compared to 2013 [93]. Researchers have proposed various approaches to improve TE optimisation performance to deal with the steady growth of traffic both in distributed (§2.2.1) and SDN (§2.2.2) contexts.

2.2.1 Distributed TE Systems

We identify three trends in the distributed traffic engineering literature.

Bandwidth Reservation: A common TE optimisation approach is to reserve bandwidth. Users of a network negotiate and advise how much traffic they will send to a destination, allowing operators to distribute traffic in the

topology. Traditional decentralised protocols such as MPLS-TE [87], BGP-TE [69], and OSPF-TE [47], implement reservation-based TE optimisations.

Predicted Demands: A second common approach to perform TE is through predicted traffic demands. A network operator generates a prediction TE matrix based on historical traffic patterns observed in the network. A TE system uses the predicted demands with a capacity-aware path computation algorithm to generate offline paths that distribute traffic onto links, targeting a specific goal. For example, the Constrained Shortest Path First (CSPF) algorithm can compute paths that do not exceed the network's link capacity based on predicted demand. Traditional distributed TE systems such as Multiprotocol Label Switching (MPLS) use CSPF to perform TE. An alternative is to use the predicted demands as a constraint for the Multi Commodity Flow (MCF) problem to generate a routing scheme. Systems use linear programming (e.g. [53]) or heuristic-based approximation (e.g. [99, 41]) to solve the formulated MCF problem.

Load Balancing: Load balancing algorithms split traffic across multiple paths to increase the network's forwarding capacity. Load balancing algorithms do not necessarily perform TE because they do not adapt to metric changes and are not capacity aware. Despite this, load balancing systems provide an alternative to TE systems because they aim to improve forwarding performance by distributing traffic on multiple paths. Equal Cost Multi-Path (ECMP), Valiant Load Balancing (VLB) [94], and Raeke [73] are examples of load balancing algorithms. ECMP splits traffic equally across multiple paths, while VLB and Raeke use random node indirection to split traffic randomly. There has been significant research effort invested in making load balancing systems TE-cable by adjusting how much traffic a system sends on each path to optimise forwarding (e.g. B4 [43], SWAN [36], and SMORE [53]).

Summary: Despite the overall maturity of load balancing and TE methods, problems that hinder the performance or accuracy of TE systems still exist today. For example, while a large number of TE systems use predictions

to make decisions, prediction matrices often contain errors and are difficult to compute [4]. Moreover, we can identify three issues that still affect traditional decentralised protocols, degrading TE performance [61]. First, interoperability problems are commonplace with traditional systems because they often use multiple complex protocols to function. Second, traditional systems suffer from delayed or slow TE optimisation because they must propagate state changes (metrics) to every device. Third, load balancing algorithms that split traffic often require strict split ratios that are hard to achieve in practice.

2.2.2 SDN TE Systems

The research community has explored deploying TE using SDN systems to address issues raised with traditional protocols. Literature in the area has demonstrated that the characteristics offered by SDN are beneficial for TE. Mendiola et al. [61] presented a taxonomy of current state-of-the-art TE systems, discussing SDN characteristics that benefit TE. An example in the paper is that using Resource Reservation Protocol with TE (RSVP-TE) can delay traffic forwarding due to slow reservation time. Notably, SDN can help address this limitation by offering centralised network control and by not requiring path reservation state to be sent across multiple switches.

A significant body of work in the area of SDN has explored deploying TE on a Data Center (DC) network. Tristain et al. present MiceTrap [92], a scalable TE system that optimises mice flows in DC networks by grouping multiple mice flows into a single forwarding aggregate and distributing them on multiple paths. Other notable works in this area are in-band load-balancing systems that use P4 [10] to perform TE optimisation. In-band load balancing systems such as Conga [3], Hula [45], Contra [38], and Dash [39], operate similar to non-SDN distributed protocols. These P4 systems perform load balancing in DC networks by splitting traffic on multiple paths based on propagated metrics.

Researchers have also considered deploying TE-capable SDN systems on other network types, with several notable examples. In 2014, Hong et al. pro-

posed SWAN, an SDN system that targets resource optimisation on a private DC WAN. SWAN schedules traffic-heavy tasks during off-peak hours based on available residual bandwidth. In the same year, Jain et al. presented B4, a WAN SDN controller that interconnects Google's DC networks. Like SWAN, B4 operates in a topology with elastic traffic where the system can defer heavy flows to off-peak hours. In 2017, Yap et al. presented Espresso, Google's SDN peering edge that enables global per-application TE optimisation. Espresso targets the TE goal of Quality of Experience (QoE) maximisation, performing decisions based on metrics collected from end-user devices. Espresso operates in a hybrid SDN network, interacting with traditional routing protocols.

In an attempt to address issues of performance and reliability with existing systems, in 2018, Praveen et al. presented SMORE [53]. SMORE uses Raeke's algorithm to compute multiple paths and load balance traffic. The system performs TE optimisation by dynamically adjusting traffic split ratios to reduce congestion. A traffic split ratio represents the percentage of the total traffic that the system will send on each path. SMORE generates traffic split ratios by using a constrained version of the MCF problem to minimise link usage.

Despite the interest from the research community, TE systems still suffer from poor performance, scalability, and failure resilience. Furthermore, some systems pose significant challenges when deploying the system on WANs. For example, while in-band load balancing systems operate well in a DC network, they perform poorly on WANs. A WAN often contains high inter-device latency, delaying sharing of metrics and severely constraining performance. Moreover, in-band load balancers require hierarchical node structures to propagate state, which may not exist in WANs.

A notable limitation of existing systems that perform TE in the literature is their use of Single-Controller SDN (SCSDN) architectures. Fully centralised solutions (e.g. SWAN [36] and SMORE [53]) suffer from poor scalability and raise performance or failure resilience concerns. A solution to scalability, performance and resilience problems is using an MCSDN architecture. However,

MCSDN systems are more complex and require dealing with challenges such as sharing state across multiple controllers.

2.3 Multi-Controller SDN

While using an SCSDN system to manage a small network is feasible [33], deploying SCSDN systems on large topologies that contain high inter-device latency (e.g. WANs) raises three concerns. First, SCSDN systems move the network's "intelligence" to a single central controller. The controller needs to process and respond to requests generated by every switch in the topology, raising scalability concerns. Second, a large request load on a controller or high control-channel latency delays reactive operations, decreasing performance. Finally, SCSDN systems have poor failure resilience because if the central controller fails, the system loses all traffic steering capabilities.

Due to these problems, SCSDN systems are not suitable for use in WANs, and as such, researchers have proposed using multiple controllers to address these limitations. MCSDN systems separate the network into areas and deploy a cluster of controllers to manage each site, providing three main benefits:

1. MCSDN systems have better scalability because controllers interact with a subset of the network's data plane devices.

2. MCSDN systems improve performance by deploying controllers closer to the data plane devices, reducing the switch to controller latency.

3. MCSDN systems are better at tolerating control plane failures because if an area loses its controllers, the system still maintains partial management capabilities over the network.

MCSDN systems perform inter-area operations by coordinating between multiple clusters. We can identify two control plane architectures used by MCSDN systems from the available literature. First, in a flat control plane

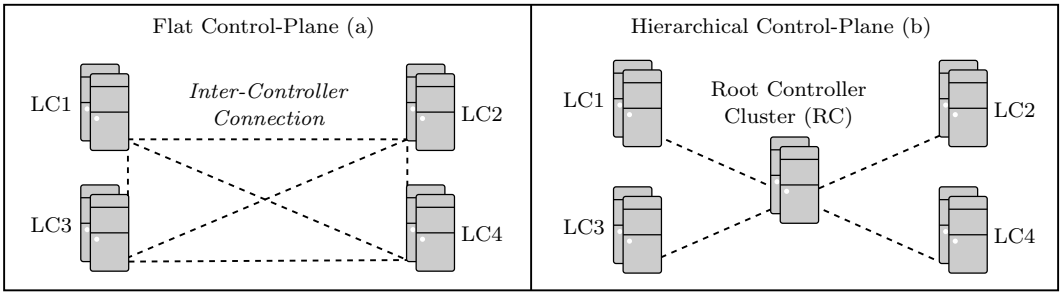


Figure 2.1: Example of the two MCSDN control plane architectures. Flat (a): Multiple Local Controller Clusters (LCs) are interconnected and coordinate with each other to perform inter-area operations. Hierarchical (b): LCs are connected to a Root Controller Cluster (RC) which manages inter-area operations. Controller clusters contain multiple controller instances to remove single-point of failures and improve the system's control plane failure resilience.

architecture (e.g. ONOS [9]), all local controller clusters connect and communicate with one another to perform inter-area operations such as routing and TE. Second, in a hierarchical control plane architecture (e.g. Espresso [99]), the local controllers are no longer inter-connected and instead connect to a cluster of root controllers. The root controllers communicate with the local controllers to coordinate and perform inter-area operations.

MCSDN systems improve their control plane failure resilience through replication (e.g. [50, 99, 25]). Network operators deploy multiple redundant copies of controllers, thus removing any single-point of failure concerns in the distributed control plane. For example, when using a hierarchical control plane architecture, network operators deploy multiple redundant instances of the root controller. When the primary root controller instance fails, a backup device takes over the management of the root controller cluster, restoring control plane connectivity and the system's ability to perform inter-area operations.

Figure 2.1 presents an example of a flat and hierarchical control plane architecture. A benefit of using a hierarchical architecture is that the system can scale better to larger networks by deploying multiple layers. As such, Helix implements a hierarchical control plane architecture.

2.3.1 MCSDN System Challenges

MCSDN systems are far more complex to implement than their SCSDN counterparts, requiring solutions to deal with four challenges:

(1) Scalability: The scalability of MCSDN systems primarily relates to controller workload. As the number of devices in a topology increases, so does the load on controllers. The scalability of an MCSDN system strongly correlates to how the switches and controllers interact. For example, SDN systems can implement two operation types, reactive or proactive. Reactive operations involve the controller in all decisions, increasing the load placed on the controller and limiting the system's scalability. In contrast, the controller pre-installs instructions onto the switches for proactive operations, enabling switches to perform forwarding changes without contacting the controller. Because proactive operations do not involve the controller in decisions, they decrease the load placed on controllers, improving the system's scalability.

In an MCSDN system, each controller interacts with a subset of the network's devices, inherently reducing the load on controllers. Despite this, MCSDN systems still need strategies to maintain good scalability. For example, suppose an MCSDN system statically assigns switches to controllers and deploys computationally heavy tasks that require numerous data plane interactions. Researchers have demonstrated that due to the dynamic nature of network traffic, load on controllers can change over time [19]. As such, when statically assigning switches to controllers, the controllers are susceptible to overloading and increased flow setup time (decreased performance) due to the system's inability to adapt to variation in controller load [19, 51]. Changes in the number of requests generated by switches will lead to load imbalances on controllers that cause parts of the network to become overused while others remain idle, slowing down operation completion time and decreasing the system's scalability [20]. MCSDN systems need to address workload imbalances, ensuring an even spread of tasks to prevent such problems.

(2) Resilience: Resilience refers to the system's ability to tolerate fail-

ures. Of particular interest to SDN-based systems is control plane failure resilience. Control plane failures leave the network in an unmanaged state, prevent the system from performing TE, leading to poor forwarding performance and packet loss. Similar to scalability, MCSDN systems improve control plane failure resilience; because MCSDN divides the network into areas, only part of the network becomes unmanaged if a controller fails. Despite this improvement, partial control plane failures will still disrupt forwarding and affect performance in some areas of the network. MCSDN systems need mechanisms to recover quickly from failures to minimise the time the network is unmanaged.

(3) Consistency: Because MCSDN systems use a distributed control plane, multiple devices need to propagate state changes to allow the system to make coherent routing decisions. If controllers have an inconsistent state, they can make forwarding decisions that significantly degrade performance. For example, Levin et al. [57] found that inconsistencies in link utilisation metrics between controllers when performing load balancing causes MCSDN systems to make increasingly poorer decisions, introducing imbalances in link usage (poor performance).

A challenge with ensuring state consistency between controllers is that existing mechanisms raise several concerns. MCSDN systems can use two mechanisms to share state, strong or eventual consistency. Strong consistency guarantees that state is always up to date, a requirement for some network applications (e.g. TE), at the expense of introducing significant performance overheads [9]. While eventual consistency removes the performance overheads, it may cause a system to make decisions based on inconsistent state, leading to policy violations, congestion and packet loss [57]. Moreover, the used consistency mechanism affects the system's ability to address the other challenges. For example, strong consistency mechanisms can impact a system's scalability because they require controllers to negotiate state changes, increasing controller workload.

(4) Coordination: Coordination characterises how controllers interact. Because MCSDN systems implement distributed control planes, multiple controllers must coordinate to perform inter-area operations and ensure coherent policy and forwarding rule application. A challenge in coordinating multiple devices is sharing state. An MCSDN system's coordination strategy will define the system's consistency requirements. For example, MCSDN systems can use two coordination strategies. First, logically centralised MCSDN systems (e.g. [50, 71]) share the complete network-wide state between all devices, allowing controllers to make end-to-end decisions. To ensure coherent policy application and prevent poor performance, logically centralised systems need to provide all controllers with up to date state, requiring strong consistency. As a consequence of this requirement, logically centralised systems can have poor performance as strong consistency mechanisms impose overheads by delaying operations [57, 9]. In contrast, distributed MCSDN systems (e.g. [78, 30]) sacrifice some of the benefits of making decisions using a full centralised scope to improve performance.

The outlined four challenges are not independent of one another. A particular design choice can excel in addressing one challenge but offer poor performance for another [66]. For example, logically centralised systems improve failure resilience because all controllers can make global traffic steering decisions. If a controller fails, a neighbouring controller can take over the management of its area. Despite this benefit, logically centralised systems need to provide strong consistency to ensure coherent policy application. Strong consistency mechanisms introduce overheads by delaying state changes, causing logically centralised systems to perform poorly [9, 70].

2.4 Summary

Data plane failures can be detected using either *active* or *passive* detection mechanisms. Both failure detection mechanisms suffer from similar problems:

(a) If the topology is congested, the mechanisms can encounter false positive failure detection. (b) Using a large poll interval or timer timeout values will delay failure detection leading to decreased performance and increased packet loss. A difference between the two mechanisms is that passive mechanisms do not introduce traffic into the network. Moreover, some passive SDN-based failure detection mechanisms use LoS to locally detect link failures and are not affected by poor performance or false positive detection issues.

After locating the failure in the topology, a system can use either *restoration* or *protection* recovery to deal with the data plane failure. Restoration recovery is a reactive operation involving the controller in the recovery process. Restoration recovery is less suitable for use in a WAN as higher latency decreases recovery performance. Furthermore, reactive operations decrease the system's control plane failure resilience by introducing a dependency between the data plane and the control plane. In contrast, protection recovery does not involve the controller in the recovery operation, making the method suitable for use in a WAN and removing any dependency between the data plane and the control plane. A concern with protection recovery is the loss of path optimality which occurs due to topology changes. Deploying protection in conjunction with TE will alleviate some of these concerns.

Due to its importance, *TE* has received significant research attention. Despite the overall maturity of existing TE methods, traditional TE systems and algorithms suffer from problems that affect their performance and accuracy. A notable limitation of existing SDN-based TE systems is their use of SCSDN architectures, which can lead to poor scalability, performance, and decreased control plane failure resilience. Moreover, the requirements of existing TE algorithms make deploying TE on WANs difficult due to the amount of state the system disseminates between devices and the level of consistency needed to ensure coherent TE policy application. A solution to address these limitations is to implement an MCSDN system.

MCSDN systems address the scalability, performance, and failure resilience

concerns raised by using a single controller to manage the entire network. MCSDN systems divide a network into areas and deploy a controller to manage each part. In a flat MCSDN system, all area controllers are interconnected with each other to share state and coordinate inter-area operations. In contrast, in a hierarchical MCSDN system, the local area controllers are no longer interconnected and instead connect to a cluster of root controllers which perform inter-area operations. A hierarchical MCSDN system offers better scalability because: (a) the system can use abstraction to reduce the amount of state shared between controllers; (b) network operators can define multiple layers in the control plane hierarchy to better scale the system to larger topologies.

Despite the added benefits, MCSDN systems are far more complex to implement requiring solutions to deal with four challenges: scalability, resilience, consistency, and coordination. A key factor contributing to the limitation of existing work in literature is that these four challenges are not independent of one another. A particular design choice can excel in addressing one challenge but offer poor performance for another [66].

Chapter 3

Literature Review

This chapter presents a literature review of related work that has designed and implemented Multi-Controller SDN (MCSDN) systems. Previously, §2.3.1 identified the four challenges MCSDN systems face (scalability, resilience, consistency, coordination). The literature review divides related work into four sections based on the targeted MCSDN challenge. Later in this thesis, §4.1 discusses how Helix avoids the identified limitations of existing systems. Helix combines various techniques to reduce the amount of shared state and controller workload, offering improved scalability, performance, and resilience.

3.1 Scalability: Controller Workload

The separation of a network into areas and the location of controllers affect the scalability and performance of an MCSDN system. Wide Area Networks (WANs) impose link and topological constraints on inter-device communications, affecting state propagation and controller reaction time. High latency delays inter-device coordination messages, thus decreasing the performance of reactive tasks [33]. Uneven distribution of switches to controllers can overload controller instances, affecting performance and causing failures. Researchers have identified this as the Controller Placement Problem (CPP).

CPP solvers [33, 62, 44, 60, 97] consider topological constraints to improve the scalability, performance, and failure resilience of MCSDN systems.

A CPP solver provides insight into controller locations, the number of areas, and the assignment of switches to controllers. CPP solvers are complementary to MCSDN systems because they tell network operators where to deploy controllers to minimise inter-device latency (improve performance) [33]. In essence, a CPP solver is a planning tool that answers two questions: "How many controllers to use?"; "Where to deploy the controllers in the topology?";

While CPP solvers offer insight into how to divide a network, they generate a static switch to controller mapping. Because network traffic is dynamic, the load on controllers may change over time [19]. Variation in controller load can lead to imbalances where some controllers become overloaded while others are underutilised, slowing down operation completion time and affecting the system's scalability [20, 19]. Researchers have proposed implementing Control Plane Load Balancing (CPLB) systems in response to these problems. CPLB systems such as ElastiCon [20], BalCon [13], BalConPlus [96] and others [51, 101, 80, 63, 31, 21, 8] actively monitor controller load and resolve imbalances by migrating switches to different controller instances. Despite improving scalability, CPLB systems raise three new problems. We will consider the example scenario illustrated in figure 3.1 to explain these issues.

The example scenario divides a network into two areas and deploys two controllers. We assume that within an area, a controller has a communication latency to a switch of 4ms. The two controllers need to communicate via the inter-area links. As such, the inter-controller communication latency is 20ms (average latency of a WAN calculated in §6.2). In this scenario, a CPLB system detects that $C1$ is overloaded (100% usage) while $C2$ is underutilised (20% usage). In response to the imbalance in controller workload, the CPLB system will migrate a (switch generating highest load) to $C2$. Based on this example, we can identify the following three problems:

(1) Migrating switches to remote controllers in different network regions introduces higher switch-to-controller latency, delaying operations and thus decreasing the system's performance. In the example scenario, after the CPLB

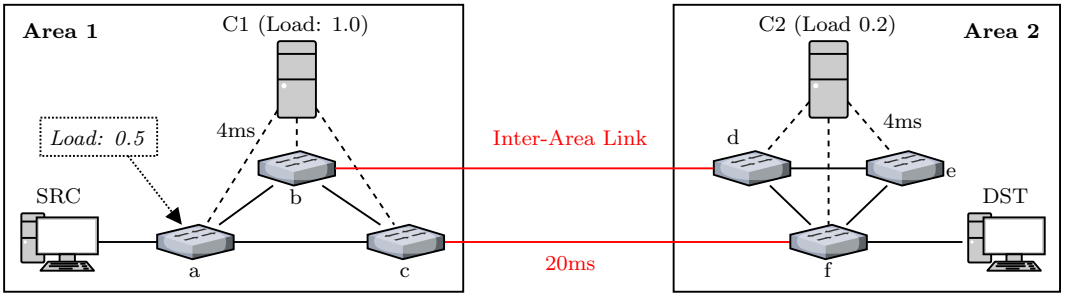


Figure 3.1: Control Plane Load Balancing (CPLB) system example scenario. The scenario divides the topology into two areas and deploys two controllers. Controllers have a local latency to switches of 4ms and an inter-controller latency of 20ms. $C1$ is overloaded (100% usage) while $C2$ is underutilised (20% usage). Switch a generates the most load (50%) in Area 1.

system migrates a to $C2$, a's control channel latency increases to 20ms. While the CPLB system has balanced the controller workload, the increase in latency between a and its controller will delay reactive operations affecting performance. For example, chapter 6 presents experiments that characterise the effects of latency on restoration-based failure recovery performance. In the conducted evaluation, when we increased the control-channel latency from 4ms to 20ms, restoration-based recovery took 3.5x longer to deal with data plane failures, leading to significantly more packet loss.

(2) Switch blackouts occur while migrating devices because switches enter a temporary unmanaged state. In the example scenario, while the system migrates a to $C2$, it cannot modify a's forwarding rules, delaying reactive operations and thus decreasing their performance. BalConPlus [96] proposes a partial solution to switch blackouts for path installation, where the system temporarily installs paths for new flow arrivals using alternative nodes that avoid the non-responsive switch. BalConPlus does not provide solutions for other reactive operations (e.g. TE) and cannot deal with blackouts for edge switches. For example, BalConPlus cannot divert traffic arriving at a as it connects to a host node.

(3) CPBL systems must share metrics and coordinate switch migration, increasing inter-controller communications. Furthermore, to ensure correct

policy application and prevent overwritten or lost updates, load balancers need to provide strong consistency between controllers, slowing down state changes and decreasing performance [9].

Despite addressing scalability concerns, implementing a CPLB system will over-complicate controller design and increase controller workload as load balancing operations are computationally intensive. For example, Cello et al. [13] prove that identifying an optimal SDN switch migration is impracticable due to its computational complexity as the controler load balancing problem is NP-Complete. As a result, both BalCon [13] and BalConPlus [96] use heuristic approximations to decide where to migrate switches. Moreover, CPLB systems only benefit reactive operations. Proactive operations do not introduce load on controllers and will not affect the system's scalability because they do not involve the controller in traffic forwarding decisions.

A second factor influencing the scalability of MCSDN systems is the amount of state (information) maintained by the controllers. As the amount of state shared between controllers increases, so does the number of inter-controller messages, placing more load on controllers, thus decreasing the system's scalability. Abstraction reduces the amount of shared state by hiding parts of the network from remote controllers. Systems that use abstraction (e.g. [78, 30, 41]) decrease the amount of information shared between controllers, improving scalability by reducing inter-controller messages and controller workload.

Since all load balancing approaches suffer from problems, we propose to address scalability concerns for MCSDN systems by decreasing the computational intensity of tasks executed by the controllers and the amount of state maintained by the system. We can achieve these two design choices by not implementing computationally heavy operations (e.g. load balancing) and using abstraction. Abstraction enables an MCSDN system to hide parts of the network's complexity, decreasing inter-controller messages.

3.2 Failure Resilience: Control Plane

Failure resilience refers to the MCSDN system's ability to respond to and tolerate control plane failures. The majority of MCSDN systems provide control plane failure resilience through replication [50, 9, 99, 25, 43]. These MCSDN systems will deploy several copies (instances) of a controller in a cluster, replicating the controller's state across multiple instances. We can categorise the controller's response to failures based on the inter-instance coordination strategy. First, in the *single device strategy*, a primary instance is elected by the MCSDN system. Only the primary device can interact with the data plane and make state changes. When the primary device fails, the MCSDN system elects a new primary device from the available backup instances, recovering from the control plane failure.

A second inter-instance coordination strategy is the *multi-device strategy*. In this strategy, multiple devices can modify the cluster's state and perform traffic steering decisions in parallel. The multi-device cluster strategy will aid the scalability of the MCSDN system as it effectively increases the computational capacity of a controller. Despite this, the multi-device strategy is more prone to state consistency problems; because multiple devices can perform changes, all instances must have an up-to-date copy of the cluster's state to ensure atomic policy application. State inconsistencies can cause the system to install competing forwarding rules that degrade performance and cause loops [57], or lead to lost updates if multiple instances overwrite the same forwarding rules. The multi-device strategy needs strong consistency, as eventual consistency mechanisms do not guarantee the required consistency level (i.e. atomic updates). Using strong consistency mechanisms decreases performance by delaying operations [9, 70] (discussed in §3.3).

The second aspect of control plane failure resilience is for the MCSDN system to ensure that updates are correctly applied even under failure conditions. To achieve this requirement, researchers have explored using transaction semantics [46, 79, 16].

In 2015, Katta et al. proposed Ravana [46] a fault-tolerant SDN control platform that uses transaction semantics to guarantee that OpenFlow messages are processed once and in order. Ravana improves failure resilience by providing correct and consistent state changes when faced with controller and switch failures. Transaction-based systems behave similarly to relational databases, preventing partial state changes. If part of the state modification fails, the state is reverted to the initial version, preventing unexpected partial modifications and providing atomic operations.

In 2016, Schiff et al. [79] presented an in-band centralised synchronisation system for distributed control planes. Similar to Ravana, the system uses transaction semantics to ensure atomic state changes without the need for complex out-of-band consensus protocols (i.e. strong consistency mechanisms). The synchronisation system provides API calls that convert transaction primitives into a sequence of OpenFlow commands. The system allows controller instances to lock forwarding states by reserving TCAM space for metadata. A controller can check the status of a particular forwarding rule by retrieving its metadata field. Compared to Ravana, the system provides better deployability because it does not require switch or protocol modifications.

In 2019, Curic et al. presented FitSDN [16], a system similar to the proposal of Schiff et al. [79]. FitSDN applies transaction semantics to the data plane forwarding state (forwarding rules) by using switches as distributed data stores. FitSDN requires switches to deploy a module to make them transaction aware and guarantee atomic operations.

Transaction-based systems such as Ravana [46] and FitSDN [16] ensure consistency when applying state changes, even under failure conditions. Moreover, these systems allow deploying the multi-device instance coordination strategy without complex consensus and consistency mechanisms (i.e. strong consistency). By enabling multiple instances to interact with the data plane, transaction-based systems increase the computational capacity of the control plane, improving the MCSDN system's scalability. Despite these benefits, such

systems introduce two new critical problems:

1. Transaction-based systems delay operations and increase inter-controller communications by requiring consensus between devices before applying changes. For example, controllers will need to frequently communicate to exchange acknowledgement messages and poll lock status of forwarding state. Moreover, systems that use switches as distributed data stores require frequent inter-device communications to retrieve the network's state. Accessing information from remote locations increases reaction time, delaying operations and decreasing performance. Problems related to performance are more severe when considering deploying these systems on WANs, where inter-device latency tends to be higher. Chapter 6 conducted experiments that show that higher latency inflates operation completion time when interacting with remote devices in a WAN.

2. Transaction-based systems generally require data plane support (e.g. [46, 16]), affecting the deployability of the system in production networks. While researchers have proposed solutions to improve the system's deployability [79], these systems use TCAM memory to store metadata to implement transaction semantics. TCAM memory is a limited and expensive commodity of SDN devices, and as such, using more flow-table space severely constrains the maximum network size the system tolerates.

Since all transaction-based systems affect performance by delaying operations and increasing the load on controllers, a far better approach to address failure resilience concerns of MCSDN systems is through replication. Applying the single-device coordination strategy to restrict state change to a primary device will simplify inter-instance interaction and remove strict consistency requirements of controller state. The single-device coordination strategy enables an MCSDN system to avoiding needing to implement strong consistency and transaction-based semantics.

3.3 Consistency: State Synchronisation

Because MCSDN systems distribute the control plane across multiple devices and replicated instances, controllers and instances need to synchronise their state and coordinate to ensure coherent policy and forwarding decisions. Existing work in the literature proposes using either strong or eventual consistency to share state between controllers [50, 9, 70, 57]. Strong consistency guarantees that state is always up-to-date by requiring devices to obtain complete consensus before applying any modification, ensuring that controllers always perform forwarding decisions based on consistent state. Despite preventing inconsistencies, strong consistency mechanisms introduce performance overheads by delaying operations [9, 70].

For example, Sakic et al. [77] compared the effects of different consistency models on the performance of an MCSDN load balancer. In the paper, the authors found that when serving 1000 subsequent load balancing requests, strong consistency took 4.5x longer to respond to the requests than eventual consistency. Similar to this trend, Berde et al. [9] found that using strong consistency severely affected ONOS's performance. After evaluating the first strong consistency ONOS prototype, the authors found that the system took up to 20s to recover from link failures compared to 116ms for the second ONOS prototype that used eventual consistency. In response to the introduced overheads, researchers have endeavoured to improve the performance of strong consistency mechanisms.

In 2016, Ho et al. proposed FPC (Fast Paxos-based Consensus) [35] a model that provides strong consistency. FPC provides an average consensus time that is 35.3% lower than Raft [68]. Compared with other strong consistency algorithms, FPC is 26% faster at retrieving data and 59.7% faster at storing it. Despite the improved performance, FPC still introduces performance overheads by delaying updates. Strong consistency algorithms require consensus before applying state changes, resulting in slower operations.

In 2018, Aslan et al. [5] presented an adaptive tunable consistency model

where controllers can specify the required consistency level. The system is built on Apache Casandra [54] and allows configuring the number of instances (replicas) required to agree on a modification before the system considers the change as valid (committed). A controller uses functions to decide on the required consistency level of a specific datum. While tunable consistency models offer better flexibility and performance compared to their strong consistency counterparts, they still increase inter-device communications and delay updates by requiring consensus before applying state changes [77].

As an alternative, eventual consistency removes the overheads imposed by strong consistency mechanisms, improving the system's performance and decreasing the load placed on controllers. In their evaluation, Sakic et al. [77] found that eventual consistency generated less load on controllers (messages) and had better performance compared to both strong and adaptive consistency mechanisms (such as [5]). Eventual consistency relaxes the guarantees of the model by removing the quorum agreement phase and applying state updates immediately. After an update is applied, devices use a chatter protocol to disseminate changes to other instances, eventually reaching consensus.

Despite removing the main performance overheads, eventual consistency still presents two limitations. First, the chatter protocol may increase inter-device communications, which can lead to scalability problems and delay operations. Second, using eventual consistency may cause a system to make routing decisions based on inconsistent state, leading to policy violation, decreased forwarding stability, and packet loss [57].

Eventual consistency can still be a promising avenue to reduce performance overheads when synchronising state between devices. The amount of shared state and how often the system uses it to make decisions influence the load introduced by the chatter protocol and the likelihood that the system makes decisions based on inconsistent network state. As a result, MCSDN systems can mitigate the issues introduced when using eventual consistency to share state by applying the following two design principles:

1. Using abstraction to hide parts of the network from remote devices will reduce the amount of state shared between controllers. Using abstraction will also improve the MCSDN system's scalability by decreasing inter-controller messages and controller workload.

2. Implementing proactive or offloading operations closer to the data plane reduces the system's usage of shared state. Offloaded or proactive operations use the local state to make decisions and are not subject to state inconsistency problems.

3.4 Coordination: Control Plane Design

This section discusses different MCSDN coordination strategies that define how controllers interact. We divide this section into three categories. First, §3.4.1 discusses logically centralised control plane systems which use strong consistency to allow multiple controllers to make end-to-end decisions. Second, §3.4.2 covers distributed single controller frameworks which takes a single controller implementation and automatically deploys it across multiple controllers. Finally, §3.4.3 discusses distributed control plane systems which introduce the notion of local and global decisions/state. Distributed systems use abstraction to hide parts of the network from neighbouring controllers, decreasing the amount of state shared between controllers. Table 3.1 presents a taxonomy of the features/mechanisms used by existing MCSDN systems presented in this section. The table divides work into two groups based on their control plane coordination strategy.

3.4.1 Logically Centralised Systems

Onix [50] allows network operators to implement controllers as logically centralised distributed systems. The system uses a distributed transaction aware data store to provide strong consistency, allowing controllers to operate with centralised scope. Onix is a framework that provides general-purpose APIs and

Logically Centralised Systems					
	Architecture	*Consistency*	*Failure Resilience*	*TE*	*Comments*
Onix [50]	Hierarchical	Strong	Replication	x	
HyperFlow [91]	Flat	Strong*	Reassignment	x	(*) Relaxed Guarantee
Kandoo [32]	Hierarchical	x	x	x	No Inter-Controller Coordination
ONOS [9]	Flat	Strong; Eventual	Reassignment*	x	(*) Also Supports Replication
DISCO [71]	Flat	Eventual	x	x	
Google Orion [25]	Hierarchical	Strong	Replication	x	
Distributed Single Controller Frameworks (Logically Centralised)					
Beehive [100]	Flat	Strong	Reassignment	x	
SCL [70]	Flat	Eventual	Reassignment	x	
Distributed Systems					
D-SDN [78]	Hierarchical	x	x	x	
Orion [30]	Hierarchical	Eventual	Replication	x	
Hua et al. [41]	Hierarchical	Strong	Replication	✓*	(*) Bottom-UP TE

Table 3.1: Table summarising systems that implemented a logically centralised or distributed control plane. The failure resilience column indicates the system's controller failure recovery mechanism (two approaches): Replication = deploys multiple controller copies; Reassignment = migrates switches to a new controller on failures. Most systems did not consider TE, or the proposed method delays TE optimisation.

does not implement network applications such as routing or TE. The system improves its control plane resilience by allowing the use of replication (§3.2) coupled with a leader election protocol that enforces a single device instance coordination strategy. Due to its architecture and design, Onix has poor scalability. For example, Google found that scaling Onix to meet their WAN's (B4 [43]) requirements was increasingly difficult due to Onix's tightly-coupled architecture in which control apps share a common threading pool [25].

HyperFlow [91] is a flat logically-centralised MCSDN system that enables controllers to perform end-to-end decisions based on a centralised scope. HyperFlow provides a framework that allows multiple controllers to communicate and coordinate inter-area operations. The system automatically synchronises the network state between controllers and proxies OpenFlow requests to appropriate switches if the current controller does not manage them. Controllers share state and events through a publish-and-subscribe system built using WheelFS [89]. When a state change occurs, the system propagates relevant local events to all controllers, allowing them to update their state and become

globally consistent. HyperFlow deals with controller failures by reassigning the switches managed by the failed device to an adjacent controller.

Kando is [32] a logically-centralised hierarchical MCSDN system that aims to improve scalability by offloading independent applications that do not require global-state closer to switches. The system deploys two controller types and distinguishes between local and non-local applications. Kandoo offloads local applications that can operate based on constrained visibility (local state) to the local controllers (switch proxies). Local controllers do not communicate or coordinate with other local controllers. The system can offload applications such as link-failure detection or elephant flow detection to local controllers; however, it cannot offload operations that require network-wide state such as routing or TE. Moreover, Kando does not abstract or separate the network into areas, essentially implementing an SCSDN system because Kandoo performs non-local applications on a single controller (root controller).

ONOS [9] is a flat logically-centralised MCSDN system. The first presented prototype of ONOS used strong consistency mechanisms to share state and coordinate modifications. The authors found that strong consistency imposed high overheads, affecting performance. The second ONOS prototype moved from strong to eventual consistency by caching frequently read information on controllers. Being a flat architecture, ONOS deals with controller failures by allowing switches to connect to multiple ONOS clusters (reassignment). ONOS can also use replication to improve its control plane failure resilience.

ONOS distributes network-wide state to all controllers, allowing them to make end-to-end routing decisions. Because ONOS shares network-wide state with all controllers, the system is more prone to state inconsistency problems.

DISCO [71] is a flat multi-controller MCSDN system that uses an Advanced Message Queuing Protocol (AMQP)-based communication channel for inter-area communication. A controller manages its area and communicates with other controllers to coordinate inter-area operations. DISCO deploys agents on controllers to monitor and share aggregated network-wide state allowing

the system to perform end-to-end operations. The system computes paths using a modified version of Dijkstra's algorithm that uses Quality of Service (QoS) metrics as link weights.

Google's Orion [25] is a framework that implements a logically-centralised system, allowing multiple micro-services to coordinate and run in parallel. The system uses a hierarchical control plane coupled with the idea of micro-services and intent-based networking. Google developed Orion as a replacement for Onix, which suffered scaling issues. Orion provides atomic state changes across areas using a centralised replicated data store.

Using a logically centralised architecture improves the control plane's failure resilience of an MCSDN system by allowing multiple controllers to interact with the same data plane device (i.e. take over when failures occur). Despite allowing controllers to make end-to-end decisions using a full centralised scope, logically centralised systems require strong consistency to share network-wide state. Using strong consistency raises scalability and performance concerns (as described by the authors of ONOS).

While using eventual consistency to share state between controllers mitigates some performance overheads, it raises consistency problems, namely, performing traffic forwarding changes based on inconsistent state. Because a logically centralised system shares the complete network-wide state with all controllers (cannot use abstraction), a controller's local cached state is more likely to become inconsistent with the global state. This inconsistency is more prominent in logically centralised systems because more state needs to be synchronised, resulting in slower propagating time [57]. Furthermore, a logically centralised system is more likely to perform path changes based on inconsistent information because controllers frequently use the locally-cached global state. Making path changes based on inconsistent network state can lead to policy violation, decrease forwarding stability and cause packet loss [57].

Another issue encountered by logically centralised systems is poor scalability. Hu et al. [40] modelled and compared the scalability of different MCSDN

controller architectures. The authors found that while a logically centralised system scaled better than an SCSDN system, its scalability was significantly poorer than a distributed system which does not share complete network-wide state with all controllers. Logically centralised systems share more state with controllers than distributed systems, resulting in decreased scalability. Moreover, because logically centralised systems need to use strong consistency mechanisms, these systems suffer from poor performance. For example, strong consistency delays path changes when implementing TE, implying that the system is slower to perform TE optimisation. As a consequence of the slower TE optimisation, the network will experience more congestion and packet loss.

3.4.2 Distributed Single Controller Frameworks

Beehive [100] offers a framework to simplify the process of building MCSDN systems. The framework converts an SCSDN image (application) into an MCSDN system by distributing it across multiple controllers. To use Beehive, network operators need to develop applications that run as asynchronous message handlers. The framework partitions application logic and automatically distributes state between sites. Beehive uses RAFT [68] to implement a transactional dictionary datastore that the system distributes across multiple controller instances. Beehive's datastore provides strong consistency and ensures that a key (from the state dictionary) belongs to a single thread.

Beehive implements a CPLB mechanism that monitors application performance (bees) and migrates bees to different clusters (hives) once they become overloaded. Beehive's CPLB mechanism improves the system's scalability by evenly distributing controller workload. Despite this, CPLB systems delay operations and require strong consistency, which can introduce significant overheads when implementing critical network operations such as failure recovery and TE optimisation (discussed in §3.1).

SCL [70] distributes a single controller image on multiple controller instances, acting as a coordination layer for distributed systems. Different from

Beehive, SCL uses eventual consistency to reduce state synchronisation overheads. SCL's primary focus is to provide control plane failure resilience. The controllers interact with specialised agents that maintain network event history. To ensure eventual awareness of the current topology and data plane state, SCL controllers exchange gossip messages with other controllers every second and poll the network agents (switches) every 20s. When a network agent receives a poll request, it will respond with the switch's current active links and installed forwarding rules (data plane state).

A key benefit of using a distributed single controller framework is that they simplify the development of MCSDN systems because SCSDN images are far easier to implement. While both Beehive and SCL streamline and automate the development of MCSDN systems, they introduce five new problems:

1. Distributed single controller frameworks are logically centralised, suffering from poor scalability and performance. Beehive and SCL use a flat control plane architecture that requires replication of the global network state across multiple controller instances. Beehive's strong consistency introduces delays when performing reactive operations, decreasing the system's performance.

2. Neither SCL nor Beehive supports abstracting the network into areas to improve scalability and mitigate consistency issues. SCL requires all controllers to connect to every data plane device for resilience. The authors of Beehive argue that the system supports hierarchical structures by formatting datastore keys. While this may be the case, Beehive's CPLB mechanism prevents network operators from restricting where the system deploys keys and applications. Beehive will automatically migrate applications to different controllers to improve their performance. A consequence of the CPLB mechanisms is that it is difficult for network operators to hide parts of the topology from a particular Beehive controller.

3. A Beehive datastore key belongs to a single thread at any given point. Implementing other applications, such as TE, would require coordination between multiple bees. Because multiple remote devices need to interact, this can delay TE optimisation, increasing congestion and potentially packet loss.

4. Beehive implements a control plane load-balancing mechanism which introduces performance overheads when considering deploying the system on a WAN (discussed in §3.1).

5. SCL suffers from deployability issues as it requires switch modification to deploy the proxy agent portion of the architecture. Furthermore, SCL's consistent probing of the proxy agent logs can increase the control-channel load, which affects the system's scalability.

3.4.3 Distributed Multi-Controller Systems

D-SDN [78] is a framework that logically and physically distributes the control plane, offering an example of a distributed MCSDN system. D-SDN targets security, allowing deployment in environments where different organisations manage different network parts. The framework uses a hierarchical control plane and defines two controller types, secondary controllers (SC) and main controllers (MC). D-SDN does not provide solutions to coordinate or share state between applications/controllers. Instead, D-SDN uses the MC as an authentication layer where SCs contact it to receive authorization to interact with a set of switches. SCs are independent entities with constrained visibility that do not coordinate with other SCs to perform inter-area operations.

Orion [30] is a hierarchical MCSDN system that minimises the inter-area path stretch by sharing node count information with the root controller. The system does not provide solutions to perform TE optimisation and uses a restoration-based failure recovery. As discussed in §2.1, using restoration-based recovery may lead to increased packet loss due to slower recovery time.

Orion abstracts local controller topology information to reduce the load on the root controller.

Hua et al. [41] describe a hierarchical MCSDN system that performs routing and optimisation between different organisations. The system uses a bottom-up TE optimisation mechanism. The root controller notifies and waits for local controllers to reassign flows on different paths before performing inter-area TE. While using a bottom-up TE optimisation mechanism ensures consistency, it introduces three new problems:

1. The root controller optimises the network after the local controllers complete their TE optimisation. This approach delays inter-area TE, increasing congestion and potentially leading to packet loss if the network's usage exceeds its capacity.

2. The TE optimisation mechanism requires controllers to coordinate between layers, introducing a dependency between devices. If a controller in the chain fails, the TE optimisation request will not propagate to all relevant controllers, preventing the system from optimising paths.

3. Performing operations at remote locations raise performance concerns as latency influences the completion time of TE optimisation.

A second problem with the system presented by Hua et al. [41] is its use of strong consistency mechanisms. Strong consistency mechanisms delay operations, introducing performance overheads. In comparison, Helix (described in chapter 5) uses eventual consistency to avoid such overheads and provide better performance. Helix also does not delay inter-area TE optimisation, allowing the system to detect and resolve congestion faster. Compared to Hua et al. [41], Helix offers better control plane failure resilience and scalability because it offloads inter-area TE optimisation to the local controllers. Offloaded operations provide better performance as they are not affected by communication latency and better control plane failure resilience as they remove the

dependency between controllers. If the root controller fails, the Helix local controllers can still perform inter-area TE optimisation, improving the system's resilience to controller failures. Furthermore, offloaded operations decrease the load placed on central remote controllers, improving the system's ability to scale to larger topologies.

Implementing a distributed MCSDN system is a promising avenue for deploying TE on WANs. Despite this, existing work has either not considered TE optimisation (e.g. [78, 30]) or proposed approaches that are computationally intensive and delay TE optimisation (e.g. [41]).

Comparing the two coordination strategies (logically centralised and distributed), implementing distributed MCSDN systems provides the most benefits. Distributed systems should use eventual consistency to reduce the overheads imposed by strong consistency mechanisms, especially when deploying the system on a WAN. Applying the previously proposed design choices of reducing shared state through abstraction and offloading operations closer to the data plane will mitigate the problems raised by eventual consistency mechanisms (discussed in §3.3).

3.5 Conclusion

Despite the interest from the research community, existing MCSDN systems and approaches present performance, scalability, and resilience concerns. A critical factor contributing to these limitations is that existing work has considered the four challenges affecting MCSDN systems in isolation. By not considering all four challenges, current systems have made design choices that degrade performance or make deploying TE difficult due to architectural constraints (e.g. [70, 100]). Problems are further complicated when considering deploying the system on WANs, which have high latency between devices requiring extra care to provide adequate forwarding performance and stability. Moreover, the majority of work in the literature has either: (a) not considered

TE (e.g. [78, 30, 51, 9]); (b) explored deploying TE on SCSDN architectures (e.g. [36, 53]); (c) proposed TE approaches that delay optimisation, decreasing performance (e.g. [41]); (d) proposed TE methods that will only work for specific networks as they target specific metrics/resources that are unavailable in other topologies (e.g. [23, 43, 99]).

MCSDN systems can overcome the limitations affecting existing work by applying the following three design choices (discussed further in §4.1)

- Minimises the state shared between controllers and inter-controller messages by using abstraction.
- Use proactive or offload operations closer to the data plane.
- Avoid using computationally heavy operations.

Chapter 4

Helix: Design & Architecture

This chapter presents the design and architecture of Helix, which, unlike previous work, offers a complete system for deploying TE on WANs by providing solutions for all four challenges faced by MCSDN systems (scalability, failure resilience, consistency, and coordination). Considering all four challenges allows Helix to make design choices that provide good scalability and failure resilience without sacrificing TE performance.

This chapter is structured as follows. §4.1 presents Helix's design and discusses how Helix addresses the four challenges faced by MCSDN system. §4.2 presents an overview of Helix's architecture and discusses how the system implements its design choices. Finally, §4.3 concludes this chapter. Further on in the thesis, chapter 5 discusses Helix's implementation in depth.

4.1 Helix Design

Helix applies three design choices to address the limitations identified with existing MCSDN systems (chapter 3):

1. Helix minimises the state shared between controllers and inter-controller messages by using abstraction.

2. Helix uses proactive operations and offloads critical reactive tasks to controllers closer to the data plane devices (switches).

3. Helix deploys computationally lightweight operations and aims to keep its design simple.

Scalability: Unlike other MCSDN systems that have implemented CPLB mechanisms (e.g. [20, 13, 96]), Helix improves its scalability by reducing the number of messages sent between controllers and decreasing controller workload. MCSDN systems encounter scalability problems because of two factors:

(a) Controllers exchange a large number of messages.

(b) The system runs computationally intensive tasks on controllers.

Helix's design addresses scalability problems by targeting both factors:

First, Helix *reduces the number of messages exchanged between controllers* by limiting inter-controller interactions. The Helix architecture defines two controller types (discussed further in §4.2), local controllers (LCs) and root controllers (RC). Helix LCs do not communicate with other LCs, reducing inter-controller messages. In the Helix architecture, the LCs will only exchange inter-controller messages with the RC, which uses the received information to compute inter-area paths and perform inter-area TE (if the LCs failed to resolve inter-area congestion locally). Helix further reduces the number of inter-controller messages by abstracting areas as single nodes to reduce the size of the RC's topology and the number of metrics the LCs will send to the RC. §5.6 provides an example that quantifies the reduction in the size of the RC's topology when applying Helix's topology abstraction mechanism. When comparing the Helix RC against a centralised TE system such as SMORE [53] and SWAN [36] on the example network, the Helix topology abstraction mechanism reduces the number of metrics the RC collects (receives from the LCs) by 94%. Centralised TE systems collect metrics for every link in the topology, implying that the controller maintains a complete view of the network. In the case of the Helix RC, the RC will only collect and use metrics related to inter-area links, reducing the amount of state the RC maintains and the number of inter-controller messages it receives from other controllers.

Helix also *reduces the number of inter-controller messages* by implementing proactive or offloading operations closer to the data plane devices. Because proactive/offloaded operations make decisions based on local state, the system does not need to propagate state changes to remote controllers to execute them, reducing the number of inter-controller messages and improving the system's scalability.

Second, Helix *reduces the CPU load on controllers* by not using computationally intensive tasks. For example, related work has proposed dealing with congestion by formulating and solving the MCF problem (e.g. [36, 99, 41]), which is computationally intensive. In contrast, Helix's TE algorithm minimises its search space to improve performance and reduces the CPU load generated on controllers. Moreover, because Helix implements proactive/offloads operations, the system reduces the load placed on remote controllers [32].

Failure Resilience: Helix uses replication with a single-device coordination strategy to deal with controller failures. The single-device coordination strategy ensures consistency within a cluster by restricting state changes and data plane interaction to a single primary instance. Unlike other MCSDN systems that use replication (e.g. Onix [50] and Google Orion [25]), Helix instances do not use a role assignment phase to negotiate roles. Instead, Helix instances assign themselves a role based on ID order such that, in response to a controller failure, the backup device with the lowest ID will take over the cluster (discussed in §5.5). By not negotiating roles, Helix can recover from controller failures faster. In contrast, systems that use a role assignment phase require instances to exchange multiple messages to elect a new primary device. In essence, the role assignment phase delays the control plane failure recovery operation and increases the number of inter-instance messages.

Because Helix uses eventual consistency to share network state between instances, the system must prevent lost updates during controller failures. For example, suppose that the primary instance of a controller cluster modified the forwarding rules of the data plane and subsequently failed. If the failure oc-

curred prior to the chatter protocol disseminating the performed state change to the backup instances, the primary instance's forwarding rule updates are lost. Once a new primary instance is selected, this device will contain inconsistent path information that does not reflect the actually installed data plane forwarding rules. While this problem seems inherent with eventual consistency mechanisms, Helix avoids this situation altogether. Unlike other MCSDN systems [9, 70], Helix instances do not use a chatter protocol to synchronise state changes. Instead, a Helix instance rebuilds its state from the data plane (or LCs in the case of the RC) when it becomes the primary instance of a cluster. Recovering state from the data plane implies that the new primary device is aware of all state changes, solving the lost update problem.

Consistency: Helix uses eventual consistency mechanisms to share state between devices. Unlike other MCSDN systems that use eventual consistency (e.g. ONOS [9] and SCL [70]), Helix provides solutions to mitigate the problems raised by these mechanisms (state inconsistencies and scalability). The extent of the problems encountered when using an eventual consistency mechanism depends on two factors:

(a) How much state the system shares between devices. Suppose a system needs to maintain a lot of state on multiple controllers. Because there is more state to keep in sync, the system will generate more update messages and thus increase the time it takes for all instances to become consistent.

(b) How often the system uses the shared state to make decisions. If operations frequently use state synchronised from remote controllers, the system is more likely to perform decisions based on inconsistent state. The amount of shared state and its volatility also influence the load generated by the chatter protocol (increase in inter-device communications). As the amount of state the system needs to maintain grows, so does the number of inter-device messages.

In the Helix architecture, an LC performs decisions based on local state (metrics and topology). Because Helix LCs are oblivious to other local controllers, the LC's state is not subject to eventual consistency problems. In the Helix architecture, only the RC is subject to eventual consistency problems as the RC performs decisions based on messages it receives from the LCs. Helix mitigates the problems faced when using eventual consistency for the RC's state by targeting the previous two factors.

First, Helix hides a large portion of the network from the RC, *reducing the amount of state* the LCs send to the RC (discussed in §4.2). By decreasing the amount of state maintained and used by the RC, Helix can propagate state changes quicker, reducing the likelihood that the RC makes decisions based on inconsistent information. Second, Helix offloads critical reactive operations to the LCs to *reduce how often the RC uses the state received from the LCs to make decisions*. The RC offloads inter-area failure recovery and TE optimisation to the LCs.

While systems such as DIFANE [102], Kandoo [32], TurboEPC [81], and others [17, 83, 82], have explored offloading path installation and authentication closer to the data plane to improve scalability and performance, previous work has not explored offloading inter-area TE. Helix's offloaded inter-area TE optimisation uses locally gathered metrics to make decisions, implying that the operation is not affected by eventual consistency issues. While Helix still implements an RC-based TE optimisation, this operation occurs less frequently than the offloaded LC operation. §8.3.3 presents an evaluation of Helix's TE optimisation, which shows that the LC can resolve the majority of encountered inter-area congestion without involving the RC.

A potential downside to offloading inter-area TE is the loss of global optimality, which can cause controllers to make suboptimal inter-area TE decisions. §8.3.3 discusses the effects of offloading inter-area TE to LCs on TE performance.

Coordination Strategy: One of Helix's design choices is to decrease

the number of messages exchanged between controllers to improve the system's performance and scalability. Helix meets this design choice by using a distributed control plane coordination strategy (§3.4.3) coupled with various abstraction mechanisms. A distributed MCSDN system can use abstraction to reduce the amount of state shared between controllers and decrease the number of inter-controller messages.

As Helix uses abstraction, a hierarchical system is a natural fit for the control plane architecture. A hierarchical control plane simplifies inter-instance coordination as all controllers need to coordinate via a primary controller. Moreover, a hierarchical control plane enables better scalability because the system can use multiple layers of controllers to further abstract and divide the network into smaller components, decreasing the load on controllers.

4.2 Architecture

Figure 4.1 presents an example of Helix's control plane architecture. Similar to other hierarchical MCSDN systems such as Espresso [99], TurboEPC [81], and Google's Orion [25], Helix separates the networks into multiple areas and defines two controller types. An area contains a subset of the topology (data plane) and a cluster of local controllers (control plane). A Helix local controller (LC) performs local operations and is the only controller type that interacts with the data plane. Local controllers do not directly communicate or connect to devices in other areas. For example, LC_a instances are not connected to LC_b instances or T_b switches (components of $Area_b$).

The root controller (RC) connects to all LC clusters (areas) and coordinates inter-area operations such as inter-area routing. The Helix RC is a specialised version of the LC that does not connect to any data plane device. Instead, the RC relies on the LCs to perform inter-area topology discovery, install paths and provide metrics for inter-area TE optimisation.

Area Abstraction: Helix's first design choice states that the system aims

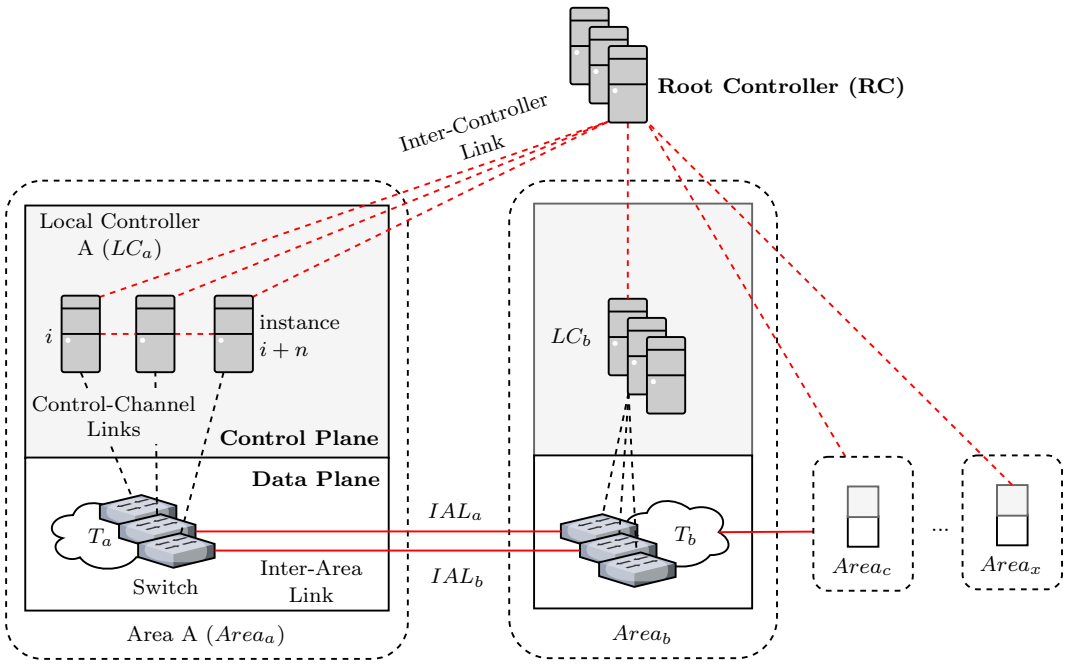


Figure 4.1: Example of the components found in Helix's control plane architecture. Helix divides the network into areas (e.g. $Area_a$). Each area contains a subset of the topology and a cluster of local controllers (LCs). LCs connect to a cluster of root controllers (RC). The RC coordinates inter-area operations. IAL_a and IAL_b are inter-area links.

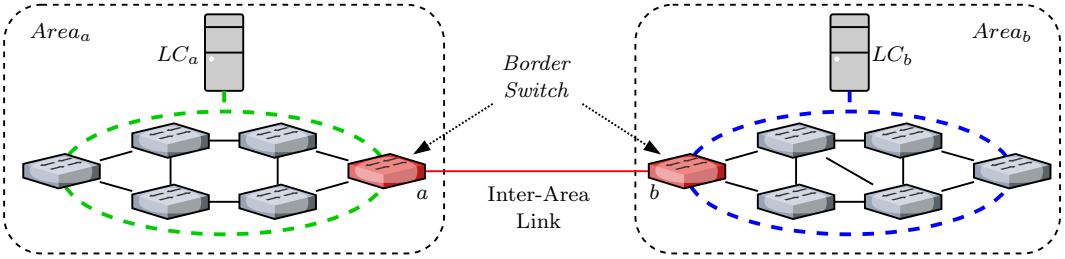
to reduce the amount of state shared between controllers and the number of inter-controller messages. Helix implements this design choice by hiding parts of the network from the RC through abstraction. Helix abstracts areas as single nodes, only exposing the inter-area links to the RC (discussed in §5.6). Helix's topology abstraction mechanism reduces the load on the root controller by shielding the RC from the majority of topology changes and data plane failure events. The RC is oblivious to any topology changes that occur within an area. Moreover, the topology abstraction improves the performance of reactive tasks because it minimises the frequency of LC-RC communications, freeing up RC processing time and allowing the root controller to respond quicker to other events such as inter-area congestion.

Cluster Abstraction: Helix uses abstraction to hide the underlying instances in an LC cluster from the RC, further decreasing inter-controller messages. For example, the RC is unaware that $Area_a$ contains $i+n$ instances. The RC does not communicate directly with an instance but instead communi-

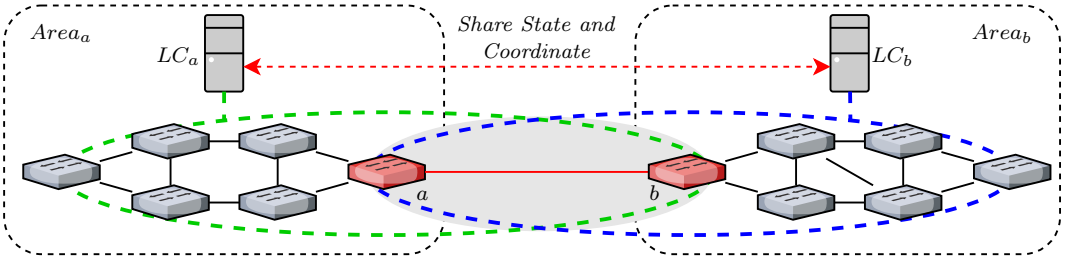
cates with an LC cluster. When the RC sends a request to an LC, all instances in the cluster receive the message, but only the primary instance processes and responds to the request. When a local cluster failure occurs, the LCs do not need to propagate role changes or involve the RC in the recovery process, decreasing the number of LC-RC messages. Because Helix deploys instances in the same cluster, inter-instance latency is low, allowing Helix to aggressively detect and recover from instance failures, improving failure resilience. In comparison, involving the root controller in the recovery operation would delay failure recovery because inter-controller latency in a WAN will be higher than the inter-instance latency, reducing Helix's failure recovery performance.

Offloaded Operations: Helix's second design choice states that the system should use proactive or offload operations closer to the data plane. Helix implements this design choice by offloading computationally intensive reactive operations from the RC to the LCs. By offloading operations, Helix improves its performance because LC-RC communication latency no longer delays completion time. In the Helix architecture, the RC offloads inter-area data plane recovery and TE optimisation to the LCs. Both operations involve inter-area links. An inter-area link joins the data planes of two areas via two border switches (e.g. IAL_a connects T_a to T_b).

Two management strategies are available for the LCs when interacting with an inter-area link. First, the Unidirectional Management (UM) strategy restricts an LC to interact with a single border switch, allowing the controller to influence traffic in one direction. In the UM strategy, two neighbouring LCs perform traffic steering decisions on both ends of the link (i.e. single-device coordination). Second, the Bidirectional Management (BM) strategy allows an LC to manage both ends of an inter-area link. The BM strategy effectively extends the diameter of an area to include both border switches of the link. By extending the diameter, the BM strategy introduces an overlap in management, where a border switch has two managing controller clusters (i.e. multi-device coordination). Figure 4.2a shows an example of the UM strategy,



(a) Unidirectional Management (UM) Strategy: Border switch connects to one LC



(b) Bidirectional Management (BM) Strategy: Border switch connects to two LCs

Figure 4.2: The two inter-area link (IAL) management strategies. We connect an LC to one border switch in the Unidirectional Management (UM) Strategy. We connect an LC to both border switches in the Bidirectional Management (BM) Strategy. Dashed circles represent the control channel connections (effective area) and shaded area shows overlap in border switch management.

while 4.2b shows an example of the BM strategy, highlighting the overlap in management.

The BM strategy allows a Helix LC to influence both inbound (ingress) and outbound (egress) inter-area traffic. Because an LC can change how traffic enters its area based on local conditions (congestion on upstream links), the LC can make better TE decisions, improving inter-area TE optimisation performance. Despite this benefit, the BM strategy raises two problems:

- It deploys a multi-device coordination strategy between LCs. Due to the overlap in management, the neighbouring LCs need to coordinate to ensure coherent traffic steering decisions and prevent overwriting or installing competing forwarding rules. As such, the BM strategy requires strong consistency, decreasing scalability and introducing performance overheads (discussed in §3.4).

- It does not fully resolve the management problems raised by the UM strategy. Instead, the BM strategy moves the issue further into the

neighbouring area's topology by effectively increasing an area's diameter. For example, LC_a cannot influence the traffic arriving at b, implying that the two switches adjacent to b are effectively border switches.

Because the BM strategy presents two new problems, Helix uses the UM strategy. Using the UM strategy implies that Helix cannot address all inter-area congestion locally. The LCs perform inter-area TE without upstream congestion information. A controller may be unaware that a particular path change would cause congestion in another area (upstream). To resolve this limitation, Helix implements an RC-based TE optimisation. The RC's TE optimisation is triggered when an LC fails to address congestion on an inter-area link. Once the RC receives an optimisation request from an LC, it will modify the installed inter-area paths to avoid using a congested area. Because the RC has visibility over all inter-area links, it can reroute traffic away from the congested links, avoiding the limitation of the offloaded TE operation.

4.3 Summary and Discussion

By using abstraction, offloading operations closer to the data plane, and not executing computationally intensive operations on controllers, Helix aims to improve its performance, scalability and failure resilience.

Helix's topology abstraction hides a significant portion of the network from the RC, decreasing the number of inter-controller messages and thus improving Helix's scalability. Moreover, the used topology abstraction mechanism also reduces the number of metrics and links considered when performing inter-area operations such as computing paths and performing TE optimisation, decreasing the computational load these operations place on the RC.

Using offloaded or proactive operations provide Helix with four benefits:

1. They have better performance as the system executes the operation locally, implying that inter-device latency no longer affects completion time. Because latency no longer affects the completion time, offloaded

operations are beneficial when deploying the system on WANs, as these network types have higher inter-device latency.

2. Offloaded operations benefit scalability because they reduce the load on remote controllers as a central entity does not make decisions.

3. Offloaded operations are not subject to state inconsistency problems as they make decisions using locally collected state. Moreover, by offloading operations from the RC to the LC, Helix reduces how often the RC uses the state received from the LCs to make decisions, decreasing the system's chances of making decisions based on inconsistent state.

4. Offloaded operations improve the system's failure resilience by removing dependencies between controllers. For example, if all RC instances have failed, the LCs can still forward traffic, respond to data plane failures and resolve most inter-area congestion locally.

A potential issue with offloading operations is loss of global visibility, which can lead to suboptimal decisions (discussed in §10.2).

Chapter 5

Helix: Implementation

This chapter discusses Helix's implementation in depth. Figure 5.1 presents a diagram outlining the modules of the Helix local (LC) and root (RC) controllers. The following sections will discuss each module of the controllers.

The Helix LC uses Ryu [75], a Python OpenFlow framework. The LC connects to and interacts with switches to perform topology discovery, install paths, collect metrics, and perform TE optimisation. The modules of the LC are as follows: The *topology discovery module* (§5.1) detects the local topology and inter-area links. The *path computation module* (§5.2) uses the discovered topology to compute and install paths. The *TE module* (§5.3) is used to detect and address local and inter-area congestion. The *leader election module* (§5.5) performs instance discovery, failure detection, and instance failure recovery within an LC or RC controller cluster. The *inter-controller communication module* (§5.4) allows instances and controllers to communicate.

The Helix RC is implemented as a standalone Python application. The RC modules are as follows: The RC *topology module* (§5.6) stores the topology information the RC receives from the LCs. The *path computation module* (§5.7) uses the received topology information to compute inter-area paths. The *TE module* (§5.8) performs inter-area TE optimisation if the LCs fail to resolve inter-area congestion locally.

The complete Helix system is implemented in Python and contains 7,000

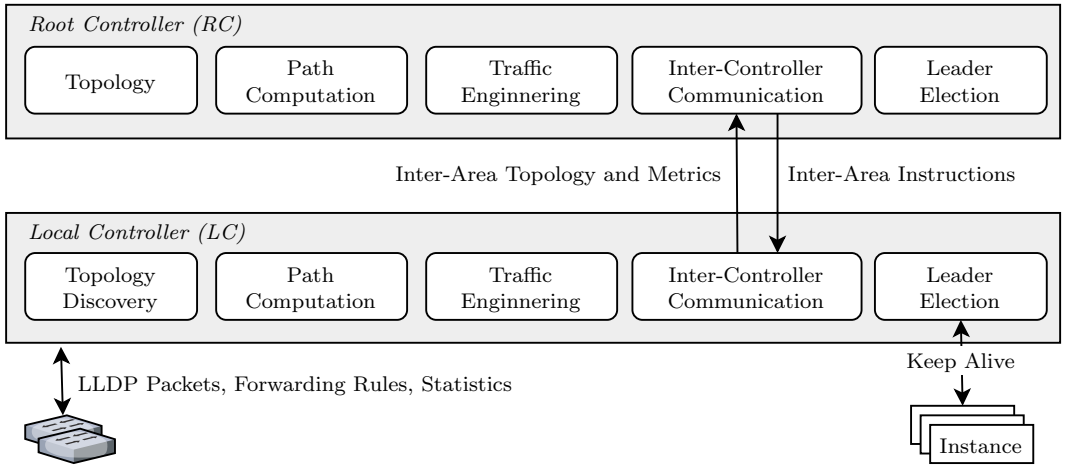


Figure 5.1: Diagram showing modules (rounded squares) of the Helix controllers (squares). The LC connects to and interacts with the data plane. The RC does not connect to the data plane, relying on the LCs to provide information. The LC and RC share several modules and have similar designs.

lines of code divided across 12 Python modules. Along with the Helix system, we developed a comprehensive testing suite that combines multiple unit tests, integration tests and emulation-based black-box tests that span 6,700 lines of Python code. We make all our code, testing suites, and emulation frameworks available at [1, 2].

5.1 Topology Discovery

The topology discovery module of an LC detects the local area topology by flooding Link Layer Discovery Protocol (LLDP) packets on the data plane. The topology discovery module installs special flow rules on switches connected to the LC. These flow rules instruct a switch to send any received LLDP packets to the controller using an OpenFlow *packet-in message*. The LC's topology discovery module will generate and flood LLDP packets on the data plane at predetermined intervals. The generated LLDP packets contain a switch's data path ID (e.g. A) and port number (A_1), identifying one end of a link. The module establishes the details of the other end (e.g. port A_1 of A connects to port B_1 of B) from the metadata of the packet-in message the LC received from a switch.

The module detects topology changes by utilising the LLDP packets as keep-alive or heartbeat messages. If a particular port does not generate an LLDP packet within a timeout interval, the module considers the port has failed. Because Helix recovers from data plane failures using protection-based recovery, Helix only uses the timeout mechanism to update its topology and perform path optimisation (§5.2).

Inter-Area Topology Discovery: The LLDP packet flooding mechanism does not distinguish between a local and an inter-area port. Inevitably, the flooding process will generate and send LLDP packets between areas. When an LC receives an LLDP packet from an unknown source switch, it identifies the corresponding link as an inter-area connection. The module leverages its flooding behaviour to perform inter-area topology discovery without requiring a new mechanism, maintaining Helix's design simplicity.

When the module detects a new inter-area link, it will send an unknown neighbour request to the RC. The RC effectively uses the unknown neighbour request messages to discover the inter-area topology. The LC ensures that the RC eventually receives any unknown neighbour request messages (deals with failures) by implementing a retransmission on timeout mechanism. If the RC does not respond to the LC's unknown neighbour request within a timeframe, the module will retransmit the message. The timeout mechanism ensures that the RC is eventually aware of the discovered inter-area topology.

The LC's topology discovery module detects inter-area link failures and topology changes using the standard LLDP timeout mechanism. When the module detects that an inter-area link has failed, it will notify the RC to update its topology and optimise the installed inter-area paths. Helix uses its standard protection-based mechanism to recover from inter-area link failures without waiting for the RC to intervene.

LLDP Mechanism Impact on Network: The LC topology discovery module's LLDP packet flood interval (send rate) is user configurable. By default, the module uses a packet send rate (τ_{lldp}) of 0.4s. This value of τ_{lldp}

implies that an LC will flood 2.5 LLDP packets per second on every port of every data plane device it manages. Helix's topology discovery module has a negligible impact on the network, using a very small amount of the data plane's capacity. The size of each flooded LLDP packet is roughly 568 bits which implies that the topology discovery mechanism will send 1,240 bits per second of traffic on every link in the network. Assuming that a particular link has a capacity of 1 gigabit, the topology discovery module will use 0.000142% of its capacity. Even if τ_{lldp} is decreased to 0.1s (4x decrease), the module will only use 0.000568% of the link's capacity.

5.2 Path Computation and Protection

The LC path computation module computes and installs paths onto the data plane. Helix uses protection recovery to deal with data plane failures.

The module deploys protection recovery by pre-installing multiple paths onto the data plane. Algorithm 5.1 describes how the module computes paths. For every source-destination pair in the area, the module generates two minimally overlapping paths (P_{prim} and P_{sec}) and a set of path splices ($P_{splices}$) using Dijkstra's algorithm [18]. The path computation algorithm generates minimally overlapping paths through link weight manipulation by setting the weights of links used in P_{prim} to large values (lines 5-7) and computing a new path (P_{sec}, line 8) between SRC and DST. Because the links of P_{prim} have large weights, the algorithm will avoid using these links when generating P_{sec}, producing a minimally overlapping path. Finally, the algorithm computes a path splice for every unique node in P_{prim} to every unique node in P_{sec} and vice-versa (lines 9-10). Path splices allow Helix to tolerate simultaneous failures by offering switches a detour between paths.

An essential aspect that network operators need to consider when using protection recovery is protection coverage. Protection coverage specifies the degree of availability or number of alternative ports that a switch can use to

Algorithm 5.1: Local controller path computation algorithm

```
   Input  : Local controller topology T
   Input  : Path source-destination pair (SRC, DST)
   Result: Set of paths to use to forward traffic from node SRC to DST
 1 Function computePath(T, SRC, DST) begin
 2     P_splices ← list()
 3     T' ← T.copy() // Make a copy of the topology
 4     P_prim ← T'.dijkstra(SRC, DST) // Compute the shortest path
 5     foreach L ∈ P_prim do
 6         L.cost ← int_max // Set link cost to a large value
 7     end foreach
 8     P_sec ← T'.dijkstra(SRC, DST)
       // Compute path splices between unique nodes in P_prim and P_sec
 9     P_splices.extend(computeSplice(T, P_prim, P_sec))
10     P_splices.extend(computeSplice(T, P_sec, P_prim))
11     return (P_prim, P_sec, P_splices)
12 end function

13 Function computeSplice(T, P_a, P_b) begin
14     P_splices ← list()
15     foreach a ∈ unique(P_a) do
16         P_tmp ← list()
17         foreach b ∈ unique(P_b) do
18             P_sp ← T'.dijkstra(a, b)
19             P_tmp.append(P_sp)
20         end foreach
21         P_sp ← best(P_tmp) // Select shortest overall path
22         P_splices.append(P_sp)
23     end foreach
24     return P_splices
25 end function
```

divert traffic away from a failure. Improving protection coverage benefits the system's failure resilience, as a switch is more likely to resolve failures without involving the controller. A naïve approach to increase protection coverage is to compute backup paths for every possible link failure in the topology (e.g. [59]). While this approach improves protection coverage, it increases path computation time, TCAM memory requirements, and load on controllers as the system computes more paths.

In contrast, computing minimally overlapping paths and a set of path splices cover more links in the topology, thus tolerating more link failures. Unlike the naïve approach, using path splices allow Helix to tolerate simultaneous link failures. Moreover, because Helix computes a path splice only for unique nodes in P_{prim} or P_{sec}, it is more likely that Helix will install fewer paths, requiring less TCAM memory compared to the naïve approach.

Figure 5.2 presents an example of the paths computed by the algorithm. In the example, P_{prim} is denoted by blue arrows and P_{sec} by red arrows. The algorithm will generate and install two splices (black arrows). The algorithm

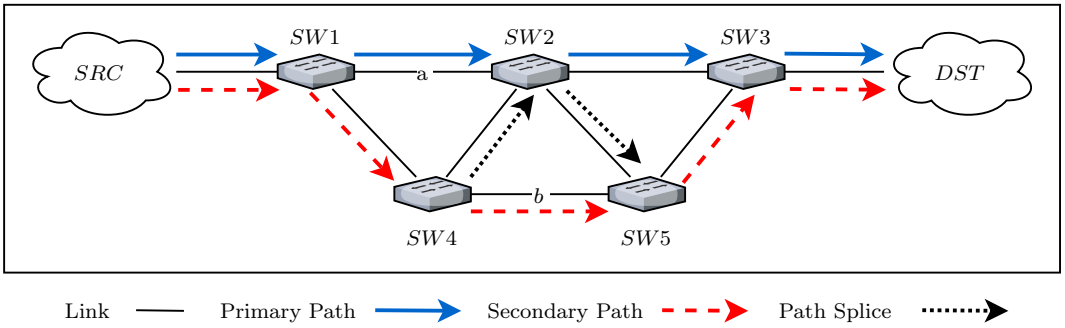


Figure 5.2: Example of paths computed by Helix local controller between nodes SRC and DST. Blue arrows denote the primary path, red dashed arrows the secondary, and dotted black arrows the path splices.

also computed $\{SW5, SW3\}$ as a potential path splice but did not install it because the path splice overlaps with (P_{sec}).

When a failure occurs, the switch adjacent to the failed link will reroute the traffic via an alternative port. For example, suppose that links a and b fail. In this situation, $SW1$ detects that link a has failed and redirects traffic towards $SW4$ (P_{sec}). Likewise, $SW4$ detects that link b has failed and redirects traffic using a path splice towards $SW2$ (P_{prim}). In essence, traffic from SRC to DST will use path $\{SW1, SW4, SW2, SW3\}$.

A limitation of protection recovery and Helix's path splice mechanism is optimality. If the network topology changes, the computed protection paths may no longer provide the shortest route to the destination. Moreover, Helix may inadvertently reroute packets on longer than necessary paths when switches use a path splice to avoid a failed link. Helix addresses the first issue through a reactive path re-optimisation process. When the LC detects a topology change, the path computation module ensures the installed protection paths still provide the shortest route to the destination. Helix also optimises path splices by generating multiple potential splices for every unique node in the first path (Pa) to every node in the secondary path (P_b, lines 15-20) and applying selection criteria. The algorithm installs the shortest splice (from the potential splices) that intersects with P_b closest to DST (line 21). In other words, the selection criteria aim to minimise the overall length of paths and

ensure that a splice always moves traffic closer to the destination.

The module deploys the protection mechanism by translating the paths into a list of ports for each switch and installing them into the data plane using fast-failover groups. A fast-failover group provides an ordered list of ports, where a switch forwards traffic using the first active port from the list.

5.3 Local Controller TE Optimisation

The LC TE optimisation module uses active measurement to target the TE optimisation goal of congestion minimisation. The TE optimisation module monitors bidirectional link usage on the topology by querying switches within an area for port and flow statistics. The module uses port statistics (averaged over the poll interval length) to determine when a particular link is congested and flow statistics from the ingress switches to determine how much traffic each source-destination pair (candidate) is sending.

The module considers a link as congested when its transmission rate exceeds a user-configurable threshold ($TE_{threshold}$). The module reduces the load on a congested link by modifying the installed candidate paths to divert traffic away from the overutilised link. Helix's TE optimisation algorithm modifies candidate paths by performing a Constrained Shortest Path First (CSPF) prune of the topology and recomputing candidate paths. The algorithm uses the candidate send rates (flow statistics) to determine how much traffic the algorithm needs to move away from the overutilised link.

Every τ_{stats} (poll) interval, the LC sends statistics requests to every connected switch, which causes the switches to send flow and port statistics to the LC. When a response is received, the module calculates a port's send rate (tx_{rate}) based on the amount of traffic forwarded by the port during the poll interval. A port is congested if its tx_{rate} exceeds the $TE_{threshold}$. If the port is congested, the module saves its details and schedules a TE optimisation consolidation period to group multiple optimisation tasks into a single update

Algorithm 5.2: Local controller TE optimisation algorithm

```
1  Function CSPFPrune(T, con, c_path, c_usage) begin
2      T.remove(con) // Prune topology of congested link
3      foreach p ∈ T do
4          usage ← p.tx_rate // Get current usage on port
5          if p ∉ c_path then usage ← (usage + c_usage) ;
6          if usage > TE_threshold then T.remove(p) ;
7      end foreach
8  end function

9  Function optimisePort(T, con) begin
10     fix ← list()
11     candidates ← getCandidates(con)
12     candidates.sort(∀c ∈ candidates, sortby : c.usage)
13     con_usage ← rebuildUsage(candidates, con)
14     T' ← T.copy()
15     foreach c ∈ candidates do
16         if con_usage ≤ TE_threshold then break;
17         T_tmp ← T'.copy()
18         CSPFPrune(T_tmp, con, c.path, c.usage)
19         new_p ← recomputePath(T_tmp, c)
20         if |new_p| > 0 then
21             fix.add((c, new_p))
22             con_rate ← con_rate − c.usage // Subtract candidate send rate
23             updateUsage(T', c, new_p) // Update stats to reflect new candidate path
24         end if
25     end foreach
26     if |fix| > 0 & con_usage ≤ TE_threshold then
           // Found a valid solution. Apply path changes and update link usages
27         applyFix(fix, T)
28     end if
29 end function
```

request, decreasing the number of path changes Helix performs. After the consolidation period timer has elapsed, the module executes algorithm 5.2.

The algorithm iterates through all congested ports and attempts to resolve congestion by altering candidate paths to avoid using the port (*optimisePort*(), line 9). The algorithm will first generate a list of *candidates* that utilise the port (line 11), sorted by port usage (line 12). Next, the algorithm calculates the port's actual congestion rate as the sum of all candidate usage rates (line 13). Rebuilding the port's usage rate is essential to ensure that the algorithm resolves the detected congestion by moving sufficient traffic away, thus reducing the chances of a re-optimisation trigger occurring during the next poll interval. The initial port usage rate the module calculated from the statistics reply may not reflect the total traffic candidates are trying to send over the port. A port's actual potential usage (recomputed) can exceed its capacity.

Next, the algorithm iterates over all candidates in the sorted set, stopping once the port is no longer congested ($con_{usage} < TE_{threshold}$). For every candi-

date, the algorithm will prune unsuitable links ($CSPFPrune()$, line 1) from an object containing a copy of the topology (T_{tmp}). The algorithm's CSPF pruning operation removes any links in T_{tmp} that are congested or do not have sufficient spare capacity to carry the candidate's traffic (lines 5-6). Suppose that the current candidate does not use a particular link from the topology. If moving the candidate's traffic to that link introduces new congestion, the algorithm removes the link from the topology object. The CSPF prune function ensures that the new candidate paths do not introduce new congestion in the network. This check prevents the algorithm from shifting traffic back and forth between links (flapping), which affects the system's forwarding stability, actively disrupting the flow of packets and causing packet reordering to occur.

The algorithm will use the pruned topology object to compute a new path for the current candidate (new_p, line 19). If new_p is valid, the algorithm adds it to the potential solution path list (fix). Next, the algorithm adjusts con_{usage} by subtracting the current candidate's send rate (line 22) and updates the topology link usage rates to reflect the new candidate path (line 23).

After the algorithm has considered all candidates of an overutilised port, or if the port is no longer congested, the algorithm checks if the potential solution set (fix) is valid. If fix is not empty and the final con_{usage} is under the $TE_{threshold}$, the algorithm applies the found solution (lines 26-28).

Candidate Sort Order & Partial Solutions: Helix provides two additional features (controlled by the user) that influence how the algorithm optimises the network. The first feature allows users to specify if the algorithm should reverse sort candidates. By default, the algorithm sorts the list of candidates for a port in descending order. Sorting candidates in descending order implies that the algorithm considers heavy-hitters first (candidates generating the most traffic). In contrast, when the algorithm performs a reverse sort, it will consider heavy-hitter candidates last, decreasing its chances of modifying these candidate paths and prioritising their forwarding stability.

The second feature is whether or not to accept partial solutions. Suppose

that an area of the topology has limited spare capacity and observes a significant increase in traffic volume. Let us assume that a link's con_{usage} exceeds its capacity causing congestion loss to occur. If the algorithm can reduce con_{usage} under the link's capacity but not under the $TE_{threshold}$, the algorithm will not apply any path changes despite being able to reduce some of the experienced congestion. Helix allows the algorithm to deploy partial solutions to address this scenario. Helix defines a partial solution as a set of potential path changes that do not fully address congestion but yield an improvement. In essence, if a set of new candidate paths do not reduce con_{usage} under $TE_{threshold}$ but would overall reduce congestion across the network, the algorithm applies the candidate path modifications. We introduce support for partial solutions by changing the algorithm's acceptance criteria and the CSPF prune method behaviour to allow using links over $TE_{threshold}$ but under their capacity. Helix's default behaviour is to accept partial solutions (attribute set to true).

Configuration Attributes: The TE optimisation module provides two user-configurable attributes that control how frequently Helix will collect statistics from the network (τ_{stats}) and the amount of spare capacity to reserve on links ($TE_{threshold}$). First, τ_{stats} defines how long Helix will wait between subsequent poll intervals. In effect, τ_{stats} controls how often Helix will check for and resolve congestion in the network. A network operator can use τ_{stats} to define how aggressive they wish Helix's TE optimisation algorithm to be. For example, setting τ_{stats} to 60s implies that Helix will resolve congestion in the network every minute. Second, $TE_{threshold}$ indicates the amount of spare capacity (headroom) Helix's TE algorithm will reserve on a link. A network operator can use the $TE_{threshold}$ value to define how much of the topologies capacity is reserved to handle unexpected traffic peaks. By default, Helix defines $TE_{threshold}$ as 90%, reserving 10% headroom on links.

When deciding on values for both TE attributes, network operators should ensure that the used values do not make the TE algorithm too aggressive. For example, using a low τ_{stats} value will cause the TE algorithm to no longer

average-out short peaks in traffic rates, causing Helix to over-optimise the network, leading to decreased forwarding stability. Suppose that a network operator configured Helix to reserve 30% headroom ($TE_{threshold}$ set to 70%) and defined τ_{stats} as 1 second. Let us assume that under normal circumstances, a particular link ($link_a$) has low utilisation under the defined $TE_{threshold}$ value. Next, suppose that the network is experiencing a 1-second burst in traffic that increases the utilisation of $link_a$ over $TE_{threshold}$ (e.g. 75% usage rate). Due to the small value assigned to τ_{stats}, Helix will consider $link_a$ congested despite $link_a$ returning to low utilisation during the next poll interval. By checking for congestion too aggressively, Helix will decrease its forwarding stability, disrupting the flow of in-flight packets by causing packet reordering in the network. In comparison, setting $TE_{threshold}$ to 60s will promote better forwarding stability because Helix will not consider $link_a$ as congested (the peak only occurs for 1-second). As a result, both attributes should be assigned a large enough value to allow the TE algorithm to ignore short-lived peaks in traffic send rates, promoting better forwarding stability.

5.4 Inter-Controller Communications

The inter-controller communication module allows Helix controllers and instances to share and receive information from other devices. Helix controllers communicate via messages using the Advanced Message Queuing Protocol (AMQP) [65] with a publish-and-subscribe model. The Helix publish-and-subscribe model groups messages into channels based on a routing key. Each routing key conveys specific information about a relevant topic (e.g. local controller topology information). Helix controllers (subscribers) register to receive messages on a set of channels (create bindings), depending on the information they require. Controllers (publishers) will communicate with a device by sending a message with a specific routing key. The AMQP broker uses the routing key to forward messages to all subscribers of a channel.

Helix uses a hierarchical routing key syntax to identify a receiving layer of the control plane and the shared information type. Helix structures the routing keys to enforce the natural restrictions introduced by the architecture. For example, the Helix architecture limits the interaction of the root controller with local controller instances. The RC communicates with a cluster of LC instances (area), and not a particular device. As a result, all LC instances operating in a cluster register to receive messages on an area channel (identified by the area's ID) and broadcast channel for all local controllers.

5.5 Leader Election Module

The leader election module allows Helix controllers to perform instance failure detection, discovery, and failure recovery. This section starts by discussing the leader election module in-depth and concludes by presenting several examples to illustrate the Helix leader election process.

Failure Detection: Helix uses a keep-alive mechanism to detect instance failures. The leader election module broadcasts a keep-alive message (heartbeat) within its cluster every keep-alive interval (τ_k seconds). The module keeps track of keep-alive messages it has received from other devices and considers that an instance has failed if several consecutive heartbeats are missing. The leader election module will declare a heartbeat missing if it has not received the message within $\tau_{timeout}$ seconds of a keep-alive broadcast interval. Helix defines $\tau_{timeout}$ as half the value of τ_k.

Instance Discovery: On startup, the leader election module will enter an initiation phase. During this phase, the module broadcasts a find message within the cluster to discover other active instances. The initiation phase allows the module sufficient time to detect active instances before deciding its role, thus preventing unnecessary role change and state rebuild operations. The initiation phase lasts for τ_{init} seconds. Helix defines τ_{init} as half the value of τ_k. When a controller instance receives a find message, it will immediately

reset its internal keep-alive timer and respond with a keep-alive message that advertises its current role. This operation effectively synchronises the keep-alive intervals of all active devices within a cluster, allowing for more aggressive instance failure detection.

After the initiation phase has terminated, the leader election module will either: (a) assign itself a backup role if the module detected a primary device within the cluster; or (b) assign itself a primary role if the module did not discover another primary device.

Failure Recovery: The leader election module responds to instance failures in two ways. First, if a backup instance fails, the module removes the device from its list of active cluster devices. Second, if the primary device fails, the module enters the role reassignment phase. In the role reassignment phase, the leader election module will compare its instance ID against the IDs of the other active devices in the cluster. If the module has the lowest ID, it will promote itself to the primary device, taking control of the cluster. Without intervention, the other active backup devices will perform the same ID check and modify their roles accordingly.

Attribute Configuration: All Helix attributes are user-configurable. A network administrator should configure Helix's timeout attributes based on network conditions. For example, τ_k must account for communication delays between Helix instances. If τ_k does not account for communication latency, the leader election module may incorrectly consider that a controller instance has failed (false positive failure detection). False positive failure detection events will generate unnecessary role change and state rebuild operations, thus increasing control channel load.

A network administrator should measure the maximum round-trip-time (RTT) between instances in a cluster and use this as a minimum value for τ_k. We configured τ_k to 1 second, and implicitly $\tau_{timeout}$ and τ_{init} to 0.5s, for the control plane failure experiments described in chapter 7.

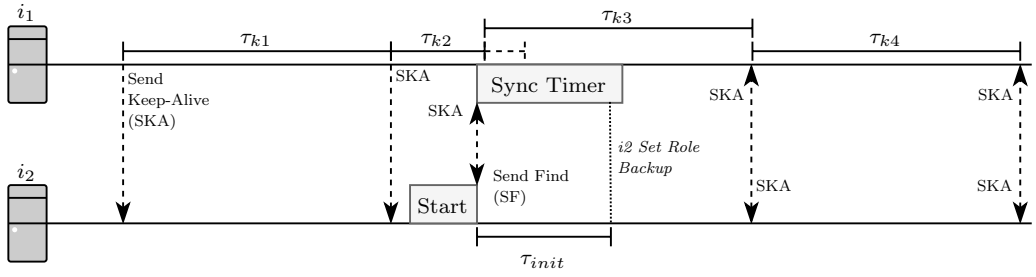


Figure 5.3: Timeline diagram showing an example of messages exchanged by the leader election module. In the initiation phase (τ_{init}) an instance will broadcast a find message to discover other active devices within a cluster.

5.5.1 Example: Instance Discovery

Figure 5.3 presents an example timeline showing an instance (i_2) starting up and joining a cluster that contains another active Helix controller instance (i_1). We separated the timeline into four-keep alive intervals (τ_{k1} to τ_{k4}).

In the example, i_2 is started during τ_{k2}. Once started, i_2 will enter its initiation phase by broadcasting a find message within the cluster. i_2's find message will be received by i_1 during the second keep-alive interval (τ_{k2}). In response to receiving i_2's message, i_1 will reset its keep-alive timer and broadcast a keep-alive message within the cluster. The broadcasted keep-alive message will tell i_2 that the cluster currently contains another instance (i_1). In essence, the broadcasted find message forces the cluster to enter a new keep-alive interval (τ_{k3}). After i_2's initiation period has elapsed (after τ_{init} seconds), i_2 will assign itself a backup role. Both i_1 and i_2 will continue to broadcast keep-alive messages every τ_k seconds.

5.5.2 Example: Failure Detection and Recovery

Figure 5.4 presents an example timeline showing the failure detection process of the leader election module. i_1 and i_2 are active at the start of the example.

In the example, i_1 will fail during τ_{k2}. During the next keep-alive interval (τ_{k3}), i_2 detects that i_1 has failed after i_1's keep-alive message is not received within $\tau_{timeout}$ seconds of the τ_{k3} timer trigger. After detecting that the i_1 has

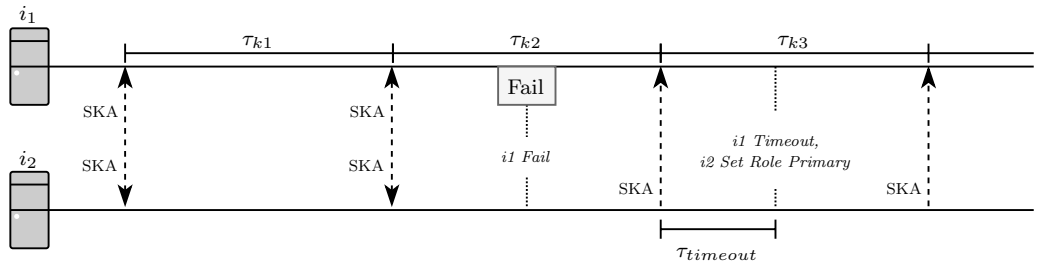


Figure 5.4: Timeline diagram showing the instance failure detection process used by Helix. The leader election module expects to receive a keep-alive message from another instance within $\tau_{timeout}$ seconds. If the module does not receive a keep-alive message, it assumes an instance has failed.

failed, i_2 will enter the role reassignment phase. i_2 will compare its instance ID against the other active device in the cluster. In the example, i_2 is the only active device in the cluster and, as such, will modify its role to take it over.

To better illustrate Helix's role assignment process, suppose that in the example outlined in figure 5.4 we have a second instance (i_3) active during τ_{k3}. Both i_2 and i_3 detect that i_1 has failed at similar times (due to the timer synchronisation process). Because i_2's ID is lower than i_3's, i_2 will take over the cluster while i_3 remains a backup device.

In the event of a cascading failure where i_2 fails after taking over the cluster (during τ_{k3}), i_3 will detect the failure of i_2 and recover during the next keep-alive interval (τ_{k4}). While the cluster is unmanaged (does not have an active primary instance), Helix will continue to forward packets and respond to data plane failures. An unmanaged Helix cluster will, however, lose the ability to perform reactive operations (e.g. TE optimisation). Because Helix executes reactive operations infrequently and at a different time scale than instance failure recovery (instance failure recovery occurs more frequently), the effect of a cluster becoming unmanaged has minimal impact on performance.

5.6 Root Controller Topology

The RC relies on the local controllers (LCs) to discover the inter-area topology. Helix reduces inter-controller communications and mitigates consistency issues

by limiting the state shared with the RC through abstraction. The LCs provide the RC with a set of hosts (LC_{host}) and a set of switches (LC_{sw}) operating within their area. The RC uses LC_{host} to compute inter-area paths and LC_{sw} to map an unknown border switch to a particular area. While we refer to the contents of LC_{host} as host nodes, these objects can be generalised to any source/destination node. We do not need to tell Helix of every individual host device in the network but instead provide the system with a source or destination for traffic. For example, a host in the Helix system can represent a group of devices connected by a non-SDN switch or any other routing protocol. We can also apply this principle to enable Helix to route traffic on the internet between organisations by considering an organisation as a single host node.

Both LC_{sw} and LC_{host} do not change frequently and as such do not require consistent synchronisation. When an LC detects an inter-area topology change, it will send a topology update message to the RC. The topology update message contains the corresponding LC_{host}, LC_{sw} and the set of already known inter-area links (LC_{ial}). The LCs encode an inter-area link as a tuple consisting of the source switch, source port number, destination switch and the link's capacity (speed).

Figure 5.5 presents an example network topology (figure 5.5a) and the corresponding RC abstracted topology (figure 5.5b). The example topology contains three areas, $Area_a$, $Area_b$ and $Area_c$. A corresponding local controller manages each area (e.g. $Area_a$ is managed by LC_a). The three LCs provide the RC with topology information. For example, when LC_b first starts up, it will send a discovery message to the RC. The discovery message contains the controller's configuration attributes (e.g. TE threshold value) and area-ID. Once LC_b has discovered its topology, it will send its host ($LC_{host} = \{H2, H3\}$) and switch ($LC_{sw} = \{SW_{10}, SW_{11}, SWb_3\}$) lists to the RC.

After receiving the discovery message from LC_b, the RC will save the new controller information in its active LC list and add a node representing the area to its topology graph (e.g. $Area_b$). From the received topology messages,

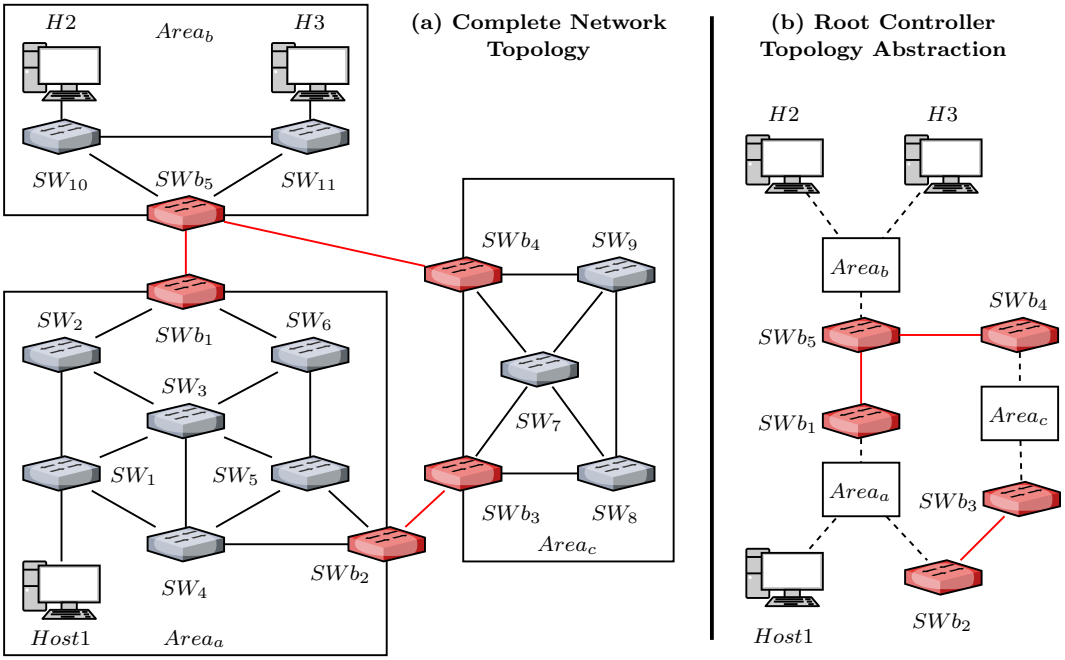


Figure 5.5: (a) Example network topology. Switches (grey) are labelled SW. Border switches (red) are labelled SWb. (b) The abstracted root controller topology for the example network. The root controller preserves the border switches and abstracts other devices in an area as a single node. Dashed links represent virtual connections which may not exist in the original topology.

the RC adds the hosts from LC_{host} to the topology graph, attaching them to the area node using a virtual link. A virtual link represents a connection that does not reflect a physical data plane link.

The RC does not add LC_{sw} to its topology graph and instead uses LC_{sw} to respond to unknown neighbour requests it receives from the LCs. For example, LC_a detects the inter-area link $SWb_1 - SWb_5$ and sends a corresponding unknown neighbour request to the RC. The RC consults its saved LC_{sw} lists and responds to the LC's request by telling the controller that SWb_5 belongs to $Area_b$. The LC saves the resolved neighbour information to allow the RC to rebuild its state after recovering from a control plane failure. The LCs do not use the resolved neighbour information to route traffic.

The RC discovers inter-area links via the LC's unknown neighbour request message. The LC's request contains the local border switch information (e.g. SWb_1 port a) and the unknown destination border switch (e.g. SWb_5). The RC will add the two border switches to its topology graph and connect them

to their respective area nodes using a virtual link. Finally, the RC adds a physical link between the border switches for the newly discovered inter-area connection. An RC physical link reflects the actual data plane connection, containing the correct ports joining the two areas on each device.

To evaluate the reduction in the RC's topology size offered by Helix's topology abstraction, we will compare the size of the RC and complete network topology outlined in figure 5.5. Helix's RC topology abstraction has reduced:

- the number of nodes by 39% (8 to 11 nodes)
- the number of links by 62% (29 to 11 links)
- the number of objects by 53% (47 to 22 objects)

Despite using a small topology for the example, Helix's topology abstraction has reduced the RC's topology graph by more than half. This reduction will grow when deploying Helix on a larger and more complex network. Moreover, most of the links in the RC's topology are virtual links. In essence, Helix's abstraction has decreased the topology by 94% (from 47 links and switches down to 3 inter-area links). The LCs only send messages to the RC related to the inter-area links. As a result, the abstraction significantly reduces the number of metrics and messages the LCs send to the RC.

5.7 Root Controller Path Computation

The root controller's path computation module computes inter-area paths. Using a distributed path computation system was a potential approach for Helix to install inter-area paths. While using distributed path computation systems such as ParaCon [72], Sparc [58] and HiDCoP [90] would have enabled this functionality, such systems introduce several problems. Distributed path computation systems share connectivity matrices (e.g. Bellman-Ford link weight matrix) or complete network state and, as such, were not suitable for use with

Helix. Moreover, these systems require strong consistency, which decreases performance by delaying operations (§3.3).

Helix's inter-area path computation approach differs from distributed path computation systems. The RC performs four steps to install inter-area paths:

1. The RC's path computation module uses the abstract topology to compute abstract paths for inter-area source-destination pairs.

2. The module divides the paths into instructions for each LC, advising the controller which inter-area link to use when forwarding inter-area traffic.

3. The RC sends the instructions to each LC.

4. The LCs use the instructions to compute and deploy an inter-area path segment onto the area's data plane.

After receiving the topology information from the LCs, the RC schedules a path consolidation period to group multiple topology changes into a single inter-area path update. Once the consolidation timer has elapsed, the module uses algorithm 5.1 (§5.2) to compute abstract paths for every inter-area source-destination pair. Unlike the LC, the RC will not compute path splices and instead only computes two minimally overlapping paths. The RC provides the LCs with two paths per source-destination pair to allow the LCs to deploy protection recovery and perform inter-area TE optimisation locally.

Figure 5.6 presents an example of Helix's inter-area path computation process. The RC computes a primary path between $H1$ and $H2$ (solid red arrow, figure 5.6a). Next, the RC uses the link manipulation technique to compute a minimally overlapping secondary path (blue arrows, figure 5.6b). After the module computes the two paths, the RC will separate the paths into instructions that the RC sends to each LC to process and install. For example, LC_a will receive two instructions that specify that traffic originating at $H1$ can use the inter-area links $SWb_1 - SWb_5$ (primary) or $SWb_2 - SWb_3$ (backup) as egress points to reach $H2$.

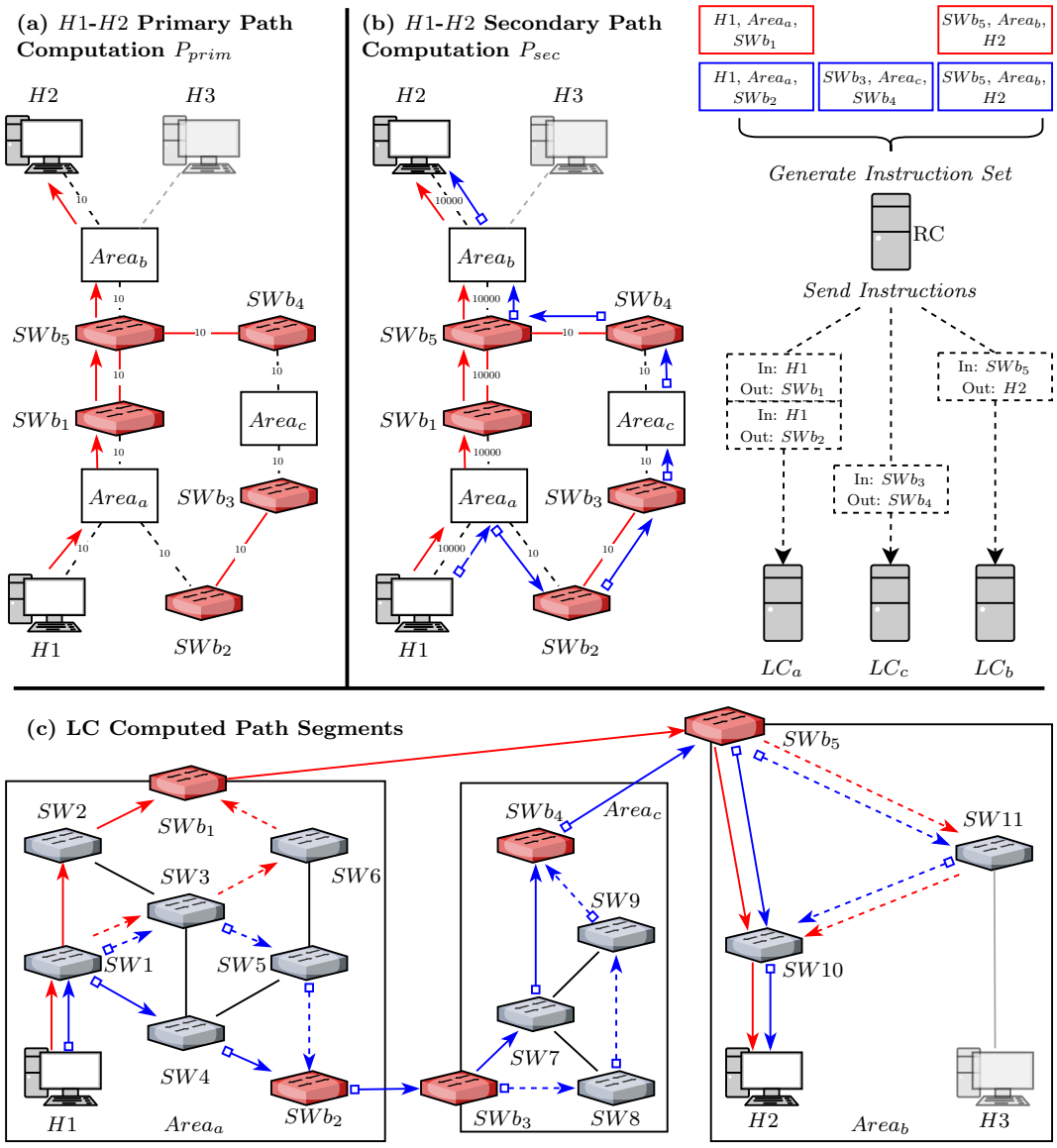


Figure 5.6: Example of a path the root controller (RC) computes for two inter-area source-destination pairs ($H1$ - $H2$). (a) First, the RC computes a primary path P_{prim} for the source-destination pair (solid red arrow). (b) Next, The RC changes the cost of links in P_{prim} to large values and recomputes the path (P_{sec}). P_{sec} is a minimally overlapping backup path (solid blue arrow with a square cap). Finally, the RC converts P_{prim} and P_{sec} into a set of instructions, which it sends to each relevant local controller (LC). (c) The LCs process the received instructions. Each LC will install a path segment onto the area's data plane. The LC computes a primary (solid arrow), backup (dashed arrows), and a set of path splices using the standard Helix protection mechanism.

Figure 5.6c shows an example of the path segments computed by the LCs from the RC's instructions. The LCs use the standard protection mechanism described in §5.2 to compute and install a primary path (solid-line arrow figure), secondary path (dashed-line arrow), and a set of splices (not shown on the figure) for every received inter-area instruction. The figure colour codes the paths to match the received instructions. For example, the figure shows the paths generated from the primary instruction by LC_a as red arrows (solid for primary and dashed for backup locally computed paths) and the secondary as blue arrows. In the example, LC_a will install the start segment of the inter-area path in $Area_a$ (contains the source), LC_b installs the destination segment in $Area_b$ (contains the destination), and LC_c installs a transit segment in $Area_c$ (contains neither the source nor the destination).

Protection Coverage: As discussed in §5.2, protection coverage is an essential factor to consider when deploying protection recovery. To increase protection coverage, the RC can compute more backup paths for each source-destination pair. The RC path computation module can reuse the link manipulation technique from algorithm 5.1 to generate multiple minimally overlapping backup paths for each inter-area pair. The RC can produce an exhaustive set of inter-area paths by generating backup paths until they are no longer unique (i.e. the new path has already been computed). When increasing the number of paths, a factor to consider is TCAM table space. Switches have limited TCAM memory, constraining the number of paths we can deploy.

5.8 Inter-Area TE

Helix implements two inter-area TE optimisation mechanisms. First, Helix offloads inter-area TE from the RC to the LCs. LCs can independently resolve inter-area congestion by changing how inter-area traffic transits their area. Second, if the offloaded LC operation fails, the LC can request a root controller TE optimisation. The RC's TE optimisation modifies inter-area paths based

on global metrics the RC receives from the LCs.

LC Inter-Area TE: The LCs perform inter-area TE optimisation using the method outlined in §5.3. In addition to monitoring internal links, an LC monitors the usage of its inter-area links to detect when they are congested. When the LC detects that an inter-area link is congested, the LC applies algorithm 5.2 to modify inter-area candidates to avoid using the overutilised link. For inter-area link congestion, the LC diverts an inter-area candidate's traffic to a backup path. The LC will not distinguish between local and inter-area candidates when optimising internal links. There are, however, three subtle differences when the LC's TE optimisation module addresses inter-area congestion. First, the module measures inter-area candidate send rates on the ingress device where the traffic enters the area. Second, when addressing inter-area congestion, the module only considers inter-area candidates. Finally, while the optimisation process is the same, the module notifies the RC of any optimisation failures.

RC Inter-Area TE: The RC reuses algorithm 5.2 to perform inter-area TE optimisation. Unlike the LC, which has a constrained network view, the RC will perform TE optimisation using a centralised scope (global view). For example, the LC cannot influence how inter-area traffic enters its area due to Helix's inter-area link management strategy. In contrast, the RC has complete management capabilities over what area the traffic will use.

To allow the RC to run algorithm 5.2, the LC will provide the RC with inter-area link usage rates and candidate send rates. The LC sends inter-area link usage rates to the RC when collecting statistics. The RC saves this information to its abstract topology. The LC will provide the RC with candidate usage rates in the TE optimisation request message. Similar to the LC's behaviour, when the RC receives a TE optimisation request, it iterates through the received candidate information, performs a CSPF style prune of the abstract topology, and recomputes the candidate path. If a valid solution to resolve the detected congestion is found, the RC converts the new candidate

paths from the solution list to instructions. The RC sends the instructions to the LCs, which modify the installed inter-area path segments.

Helix's two-step inter-area TE optimisation can be affected by lost request messages. Helix removes this problem by using an LC-based locking and retransmissions mechanism for the optimisation request sent to the RC. The mechanism deals with lost messages and ensures the RC receives optimisation requests. The mechanism requires the RC to notify the LC when it has finished optimising the network. If the LC does not receive a notification from the RC within a timeframe, the LC assumes the TE optimisation request was lost and resends it.

5.9 Area Failure Detection

The Helix RC uses a keep-alive mechanism to detect area or complete local controller cluster failures. An area without an active LC enters an unmanaged state. Unmanaged areas continue to forward traffic and respond to data plane failures but cannot perform reactive tasks (e.g. TE optimisation). To ensure good forwarding performance in case of a complete cluster failure, the RC will modify inter-area paths to avoid using such areas.

The area keep-alive mechanism is similar to the leader election module's mechanism. An LC periodically sends a keep-alive message to the RC. The RC assumes an LC has failed if several consecutive heartbeat messages are missing. When the RC detects an area as failed, the RC removes the area from its abstract topology and triggers an inter-area path recomputation.

5.10 Root Controller Path Synchronisation

Because Helix offloads inter-area operations from the RC to the LCs, the LCs can perform path changes that cause the RC's installed path information to become inconsistent with the actual data plane forwarding state. Helix ensures that the RC's path information is eventually consistent with the data plane

forwarding rules by implementing an LC-based notification mechanism. If the offloaded TE optimisation modifies an inter-area path (changes the egress link), the LC sends an egress change notification to the RC to update its state. An inter-area egress change performed by an LC causes a subsequent ingress change in a neighbouring area. To notify LCs of inter-area ingress changes, Helix does not propagate the egress change notification and instead implement a local LC-detection mechanism. A Helix LC installs flow rules to enable the controller to detect inter-area ingress changes locally. Once an inter-area ingress change is detected, the LC will notify the RC of the forwarding state changes. §4.3 discussed the benefits of using local or offloaded operations.

5.11 Summary

This chapter discussed Helix's implementation details by outlining the modules implemented in each controller type. The LC contains five modules. The *topology discovery module* (§5.1) performs area and inter-area link detection. The *path computation module* (§5.2) uses the discovered topology to compute paths and deploy protection-based recovery. The *TE module* (§5.3) is used to detect and address local and inter-area congestion. The *leader election module* (§5.5) provides instance detection, failure detection, and role assignment within an LC or RC controller cluster. The *inter-controller communication module* (§5.4) allows instances and controllers to communicate.

The Helix RC has a similar design to the LC, containing five modules. Both LC and RC controllers share a common leader election. The RC *topology module* stores the topology information the RC receives from the LCs (§5.6). The *path computation module* (§5.7) uses the received topology information to compute inter-area paths. The *TE module* (§5.8) performs inter-area TE optimisation if the offloaded LC operation fails.

Chapter 6

Evaluation: Data Plane Failure Resilience

The primary contribution of this chapter is to evaluate Helix's data plane failure recovery performance and compare it against restoration recovery. While existing work has compared protection with restoration recovery (e.g. [84, 85]), those experiments did not consider the effects of control channel latency on performance. In contrast, this evaluation collected results using latency values representative of an average WAN and a small network to highlight the impact of latency on recovery time. This evaluation provides a concrete testing methodology and introduces tools to assess the failure recovery performance of SDN systems. The conducted experiments directly compare reactive and proactive operations, showing that latency inflates the completion time of reactive operations, decreasing their performance. We extrapolate the results of our experiments to show that offloading operations improves their performance when considering deploying the system on a WAN.

This chapter is structured as follows. §6.1 presents the testing methodology and data plane failure emulation framework used to evaluate Helix. §6.2 offers a description of the performed experiments while §6.3 presents the results of this evaluation. Finally, §6.4 concludes this chapter by summarising results and discussing the applicability of our findings to offloaded versus centralised

Multi-Controller SDN (MCSDN) operations.

6.1 Testing Methodology

All evaluation results presented in this thesis were collected using a virtual machine running Ubuntu 16.04.6 LTS. The virtual machine was assigned 8GB of RAM and two cores of a four-core Intel i5-7500 CPU @ 3.40Ghz. Results were collected using OpenvSwitch 2.11.1 with DB Schema version 7.16.1.

We develop an emulation framework that compares the failure recovery performance of SDN systems. The framework uses Mininet [56] to emulate a virtual topology (T) with realistic network conditions. For example, by applying NetEm [34] attributes to links, the framework can introduce latency or loss in the network. The emulation framework produces metrics by measuring the amount of time it takes a system to recover from a set of link failures (F_{link}) defined by a failure scenario (F).

Under stable network conditions, the framework expects packets sent between two hosts (SRC and DST) to use the primary path (P_{prim}). For an experiment, the framework generates a constant stream of packets between the two source-destination nodes using Pktgen [67]. During an experiment, the framework introduces link failures on the primary path ($F_{link} \in P_{prim}$). When the SDN system detects a failure, it modifies P_{prim} to avoid using the failed link, generating a new path (P_{sec}). In the case of restoration-based recovery, the controller detects the failure and intervenes by modifying the current path. For protection recovery, both P_{prim} and P_{sec} are pre-installed onto the data plane. When F_{link} occurs, the switches modify the active path without contacting the controller, causing traffic to use P_{sec}. The framework leverages this expected behaviour of the SDN system to compute recovery metrics.

The emulation framework actively monitors Pktgen packets on both P_{prim} and P_{sec}. Figure 6.1 presents an example of the two packet capture probe locations the framework uses during an experiment. The framework defines

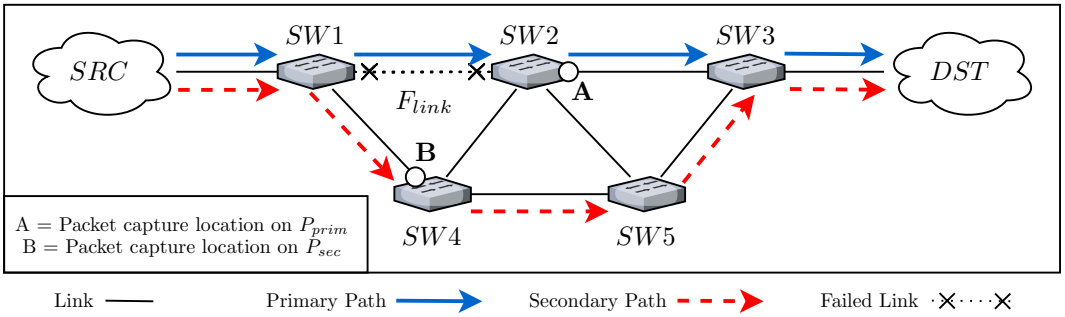


Figure 6.1: Figure showing emulation framework Pktgen packet capture locations on an example network. Under normal conditions, traffic uses P_{prim} to reach DST from SRC. The emulation framework will fail link F_{link} causing packets to use P_{sec}. The framework monitors packets on the adjacent hop after the failed link (location A) and on unique nodes of P_{sec} (location B).

location A as any unique link in path P_{prim} after the failed link (F_{link}). In other words, the failure of F_{link} should interrupt the flow of Pktgen packets to probe A. The framework defines location B as any unique link in path P_{sec}. Probe B should capture Pktgen packets only after the SDN system has recovered from the introduced failure (traffic started using P_{sec}).

We define recovery time as the difference between the last Pktgen packet timestamp observed by probe A (A_{time}) and the first timestamp observed by B (B_{time}). This represents the amount of time it takes the SDN system to divert packets from P_{prim} to P_{sec}. The recovery time metric (T_{recv}) is equal to $B_{time} - A_{time}$.

The framework uses Pktgen packet timestamps instead of packet capture time to prevent capture location and system clock differences from skewing results. For example, because capture time changes depending on how many nodes the packet has traversed, the capture locations will influence the calculated recovery metric. In contrast, Pktgen timestamps are inserted at the source host and represent the packet's creation or send time. The Pktgen metadata remains consistent while the packet travels through the network, implying that capturing packets at different locations will not influence the failure recovery results.

The emulation framework configures Pktgen to introduce a 0.1ms gap be-

tween consecutive packets (send rate of 10,000 packets per second). This gap implies that the recovery metrics produced by the framework are accurate to the closest 0.1ms. The framework also reconciles T_{recv} with the observed number of lost packets. The framework calculates the number of lost packets as the difference between packet A and B's Pktgen sequence numbers. Assuming a constant gap between consecutive packets of t_{send}, the recovery metric is also equal to $(B_{seq} - A_{seq}) * t_{send}$. B_{seq} represents the Pktgen sequence number of the first packet observed on P_{sec}, while A_{seq} represents the sequence number of the last packet observed on P_{prim}.

While the framework uses an out-of-band deployment for experiments, it allows users to emulate realistic conditions on the control channel that match in-band deployment. For example, by applying attributes to the control channel links, the framework can introduce packet loss or latency that we expect to encounter within a network. We use this functionality to emulate conditions found in two different network types for this evaluation.

6.2 Experiment Description

We built a new version of the Helix local controller (LC) that performs controller-based data plane failure recovery (restoration). The restoration controller computed a single path per source-destination pair. The restoration controller used Dijkstra's algorithm to recompute and modify the installed path after detecting a failure. The restoration controller did not apply a path consolidation period or deploy a path optimisation phase (§5.2). Instead, the restoration controller immediately recomputed the existing paths once it detected a topology change. We then compared Helix's data plane recovery performance against the restoration controller.

In this evaluation, we used three randomly generated topologies (labelled Topo 1 - Topo 3) of different sizes and five failure scenarios (labelled F1 - F5). The three topologies contained between 5-8 nodes, while the failure scenar-

Scenario	Usable on Topology			# Link Failures
	Topo 1	*Topo 2*	*Topo 3*	
F1	✓			1
F2	✓			3
F3	✓			1
F4		✓	✓	1
F5		✓	✓	1

Table 6.1: This table describes the five scenarios used to evaluate Helix's data plane resilience. The table indicates the topology used in conjunction with each failure scenario and the number of link failures defined in each.

ios specified either a single or a multi-link failure. We used topologies with small node counts to conduct our experiments to minimise the chances that emulation overheads slow down the recovery operation. All three topologies contained a single source-destination pair, $SRC/H1$ and $DST/H2$. Topo 1 contained five nodes and is illustrated in figure 6.1. Topo 2 and 3 were extended versions of Topo 1. These topologies introduced more alternative routes between $SW1$ and $SW3$ by adding a new set of links joining these nodes. Topo 2 contained six switches, while Topo 3 contained eight switches. Table 6.1 outlines the details of the failure scenarios used in this evaluation.

The framework executed multi-link failure scenarios sequentially by performing a failure action, reporting recovery time, and proceeding to the next link. The system maintained its state (controllers were not restarted) for a multi-link scenario throughout the experiment. In this evaluation, experiments were repeated 100 times with the average metric value reported (population size $N = 300$ for F2 and $N = 100$ for the other scenarios). To highlight the effects of latency on recovery time, we collected results using two control channel latencies, 4ms and 20ms.

We used a 4ms Round-Trip Time (RTT) latency to represent link characteristics encountered within an area or in a standard small network. In contrast, we calculated the latency of a WAN by using a medium-sized network from the Internet Topology Zoo Project [49], specifically the USA AT&T MPLS network. Our method to calculate latency applied the following steps:

- We calculated the edge-to-edge RTT latency of the furthest two-node pairs in the topology based on geographic distance and the speed of light travelling through a fibre cable.

- We scaled the calculated edge-to-edge latency value to account for latency inflation using the 1.5x factor observed by Singla et al. [86].

- Our estimation method assumed that an operator would deploy a controller in the centre of the network to minimise control-channel latency. Based on this assumption, we halved the calculated latency value to use for our experiments.

Applying the above method, we estimated that the halfway point latency for the AT&T MPLS network is 20ms.

To validate the above method, we reconciled estimated latency values with actual measurements taken on the ESNet topology [22], a research and education network. The ESNet network has similar geographic node placement and distances between nodes as the AT&T MPLS topology, enabling us to perform a fair comparison. We selected several random link segments from the ESNet topology, used our method to estimate the latency between the nodes, and compared our results against the measured latency of these links. Finally, we calculated the edge-to-edge latency value for the ESNet network (spanning the USA) using our method. We compared our estimated value against the measured latency across several link segments that span the entire width of the ESNet topology. Our comparison found that our method to estimate latency on a WAN is valid because our calculated latency was similar to the measured latency values. Moreover, we observed a similar latency inflation factor on the ESNet network as described in [86].

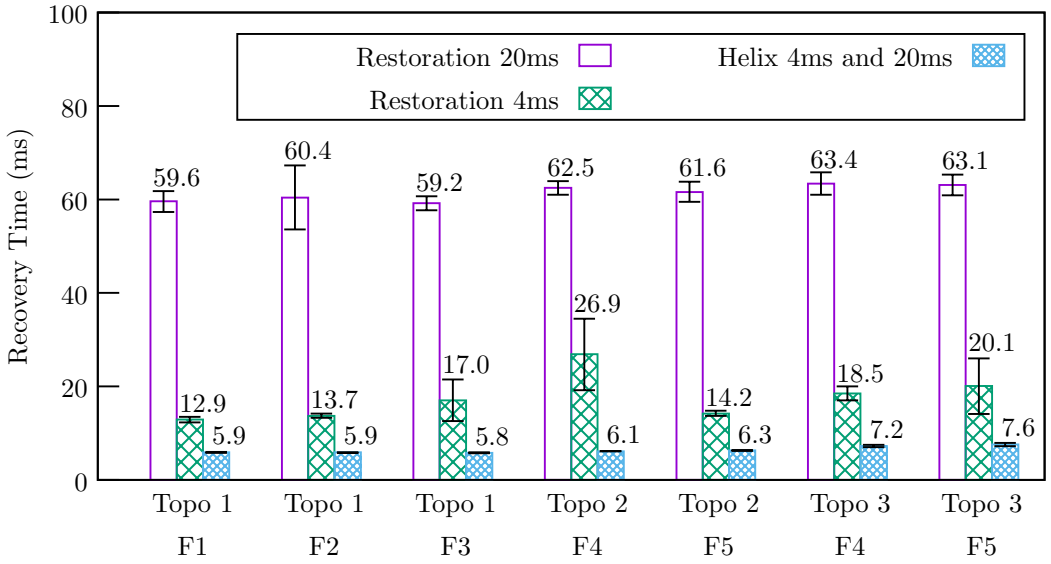


Figure 6.2: Graph of data plane failure experiment results that compared Helix against restoration recovery. Labels represent the average recovery metric across 100 experiment runs and black error bars indicate the 95% confidence interval. Restoration recovery is affected by latency while Helix's recovery time remains consistent regardless of network latency.

6.3 Evaluation Results

Figure 6.2 presents the average recovery metric for Helix and the restoration controller using the two control-channel latency values. The graph shows the 95% confidence interval as a black error bar. Helix's recovery time was on average 6ms regardless of control-channel latency. Latency does not affect protection-based recovery because switches deal with data plane failures without involving the controller. Across all experiments, we observed that Helix's recovery time had little variation, evident by the small range of the 95% confidence interval. Because of emulation overheads, Helix reported a slight increase in recovery time when evaluating the system using the larger Topo 3 topology. For Topo 3, the framework had to deploy more virtual nodes for this experiment, increasing CPU load and slowing down the switches' response.

Restoration-based recovery involves the controller in all recovery decisions. With a control channel latency of 4ms, restoration took an average of 18ms to recover from the failure or 3x longer than Helix. With a control-channel

latency of 20ms, the average restoration recovery metric increased to 64ms or 10x longer than Helix. Unsurprisingly, latency had a direct effect on recovery time for reactive operations. We also observed that the restoration recovery metric varied significantly more than Helix. The 95% confidence interval range for restoration recovery was wider across all experiments.

Limitations: Restoration recovery is affected by other factors not considered in our evaluation. For example, the number of data plane interactions, the topology size, and the load on controllers affect the system's reaction time, influencing performance. In contrast, none of these factors affects protection recovery because switches deal with failures without involving the controller.

Another factor not considered in this evaluation is the load on links. For in-band deployment, congestion on links will delay switch-to-controller communications, further increasing the performance gap between protection and restoration recovery. While we can mitigate some of this difference by using an out-of-band SDN deployment, adding extra links to carry controller traffic may be too costly in some environments [76].

6.4 Discussion and Conclusion

While this chapter evaluates Helix's failure recovery performance, the experiment and results compare reactive and proactive operations directly. Restoration and protection recovery offer prime examples of applying either a proactive or reactive strategy to implement the same functionality. Furthermore, we can extrapolate our evaluation to showcase the difference between local (offloaded) and centralised MCSDN operations. Because offloaded operations do not involve remote controllers in decisions, we can consider them as proactive (they occur locally). In contrast, centralised MCSDN operations are reactive, involving communication with a remote controller. This communication is subject to latency delays similar to restoration recovery.

Helix offloads both inter-area failure recovery and TE optimisation to the

LC clusters. We expect to observe similar inflation behaviour when comparing the performance of locally offloaded with centralised operations. If an MCSDN system executes operations on a remote controller, inter-controller latency directly affects performance, delaying operation completion time. Moreover, centralised operations have a direct impact on control plane resilience. The MCSDN system introduces a dependency between the devices by performing tasks on a remote controller. If the parent or root controller fails, the system cannot execute the centralised operation. Helix removes this dependency by offloading operations. For example, even if all Helix RC instances have failed, the LCs can perform inter-area data plane failure recovery and TE.

Despite the outlined benefits, proactive or offloaded operations raise several concerns, such as lacking global visibility. In the case of protection recovery, because paths are pre-installed, topology changes can cause the paths to no longer be optimal (i.e. offer the shortest path). Helix addresses this concern by implementing a path optimisation mechanism that ensures the installed protection paths are optimal after a topology change (§5.2).

To conclude this chapter, the presented evaluation and emulation framework provides a means to assess an essential aspect of single and multi-controller SDN systems, namely data plane failure resilience. Our evaluation results show that latency affects reactive or centralised operation performance. We observed that restoration-based recovery was up to 10x slower than Helix to restore data plane forwarding. Based on the small range in the 95% confident interval, we concluded that Helix's recovery time was consistent across experiments regardless of the network size or latency. Using proactive or offloaded operations benefits completion time and improves performance. As latency increases, the gap in performance between reactive and proactive operations grows. Based on this observation, we can conclude that using offloaded or proactive operations benefit Helix's performance, making the system suitable for deployment on a WAN.

Chapter 7

Evaluation: Control Plane Failure Resilience

The primary contribution of this chapter is to evaluate Helix's control plane failure recovery performance and provides insight into Helix's ability to cope with administrative tasks (e.g. restarting instances to perform upgrades). This chapter provides a concrete testing methodology and introduces tools that allow network operators to assess the control plane failure resilience of Multi-Controller SDN (MCSDN) systems.

This chapter is structured as follows. §7.1 introduces an emulation framework we developed to evaluate the failure recovery performance of MCSDN systems. §7.2 and §7.3 outlines the testing methodology used to collect experiment results and discusses how the framework uses collected events to compute metrics and components using an example failure scenario. §7.4 and §7.5 present an evaluation of Helix's failure resilience performance using two failure scenarios that contained simultaneous and cascading controller failures. §7.7 discusses the limitations with our evaluation. §7.8 presents two models that estimate Helix's failure recovery performance. Finally, §7.9 concludes this chapter by assessing and comparing Helix's failure recovery performance against a root controller-based failure recovery approach commonly used by MCSDN systems that have flat control plane architectures (e.g. ONOS [9]).

7.1 Emulation Framework

While existing MCSDN literature has evaluated various aspects of MCSDN system performance, control plane failure resilience has received little attention. To this end, we develop an emulation framework to evaluate the control plane failure resilience of Helix and MCSDN systems. Control plane failure resilience refers to a system's ability to recover from controller failures. The emulation framework measures an MCSDN system's reaction to control plane failures, evaluating the time it takes for the system to recover. Moreover, the framework offers insight into the startup performance of controllers, allowing network operators to plan for administrative tasks such as restarting instances to perform upgrades.

The framework also checks system behaviour under a predefined failure scenario. In essence, the framework provides a black-box testing tool that checks if an MCSDN system exhibits correct behaviour.

The framework uses Mininet [56] to conduct experiments with realistic network conditions, such as introducing latency or loss on links. To run an experiment, a user needs to provide the framework with four inputs:

- T specifies the topology to use for the experiment
- A_{map} provide the framework with a switch to controller mapping
- F specifies the failure scenario for the experiment
- $E_{expected}$ specifies a set of expected events for an experiment

The framework uses A_{map} to divide T into areas. A_{map} specifies which cluster of controllers manage each node n of the topology. A_{map} also states how many instances i the MCSDN system will deploy in each cluster. Let c represent a cluster of controllers, while LC is the set of all local controller clusters. Formally, the emulation framework defines A_{map} as:

$$\forall n \in T, \forall c \in LC \qquad A_{map} : n \mapsto c$$

$$\forall i \in c \qquad A_{map} : i \mapsto c$$

We separate failure scenarios into multiple stages (F_{stage}) to allow the framework to attribute (group) events to a sequence of actions. The framework executes the actions of a stage sequentially, maintaining the MCSDN system's control plane and data plane state throughout the experiment. The stages of a failure scenario are not independent. A previously completed F_{stage} influences the future behaviour of the MCSDN system. An F_{stage} represents a logical separation of the experiment when the framework computes metrics.

During each stage, the framework generates a timeline of observed events ($E_{observed}$). The framework groups and attributes $E_{observed}$ to an F_{stage} by waiting for several seconds of inactivity before proceeding with the experiment. Grouping events promote deterministic behaviour because the framework waits for the system to stabilise before advancing to the next stage. Moreover, grouping events allows the framework to validate the MCSDN system's behaviour. An MCSDN system exhibits correct behaviour if all event types from $E_{observed}$ are present in $E_{expected}$, and vice-versa.

7.1.1 Supported Actions

The emulation framework implements three action types:

- Fail a controller instance from a cluster
- Start or restart a controller instance in a cluster
- Introduce a delay (wait n seconds)

We emulate simultaneous and cascading instance failures by combining these action types. For example, chaining together multiple failure actions emulates simultaneous instance failures. Combining failure and delay actions

emulates cascading failures, which are often caused by overloading instances when the system migrates the request load from one controller to another in response to a previous failure event [98].

7.1.2 Collected Events

The emulation framework collects two types of events: control channel and local events. *Control channel events* describe the interaction of the MCSDN system with the data plane. The framework gathers control channel events by capturing and processing OpenFlow Protocol (OFP) packets on the control channel/switch-to-controller links. Control channel events occur irrespective of the evaluated MCSDN system. The emulation framework requires control channel events to be generated by an MCSDN system in order to compute metrics.

Local events represent internal controller state changes that occur within the MCSDN system. Unlike control channel events, local events require modification or support from the system because controllers need to push specific information to their logs when an internal state change occurs. For example, the Helix local controller will write a local event to its log once it detects the failure of a controller instance. The framework captures local events by monitoring the logs of the MCSDN system.

Event Use: The framework uses control channel events to compute metrics and local events to compute component values. As an example, the framework uses local events to calculate the failure detection (δ_{FD}) and role change (δ_{RC}) components of the instance failure recovery metric ($\Delta_{recv} = \delta_{FD} + \delta_{RC}$). Components offer better insight into a system's failure resilience performance and are subject to local event availability. We designed the emulation framework to consider local events as optional to improve compatibility with other MCSDN systems. Even if an MCSDN system does not push local events to its logs, the framework will still produce metrics based on captured control channel events.

Event Metadata: An event stored in $E_{observed}$ contains associated metadata such as the capture time, the type of event, and the ID of the instance that created the event. For control channel events, the framework uses TShark [15] to gather OFP packets and extract metadata from their payload. In contrast, the MCSDN system needs to provide the required metadata fields in its log output of local events. The framework defines a local event as a formatted variable-length CSV line where tokens map to different metadata depending on the local event type.

Latency: When calculating metrics, the framework considers the effects of latency on performance by capturing OFP packets as they arrive at a switch. In essence, the capture time of control channel events includes the time it takes for a controller's OFP request to arrive at a switch. By considering latency when collecting events, the framework generates metrics that allows users to compare the failure recovery performance of different MCSDN system architectures.

7.1.3 Generated Metrics

After completing an F_{stage}, the framework will report unexpected behaviour and output $E_{observed}$ to allow the calculation of metrics and components. A metric represents the time it takes a system to respond to an event such as a failure, while a component is the amount of time it takes an MCSDN system to perform a particular step towards responding to an event. For example, instance failure recovery time is a metric with two components, failure detection time (the time it takes the system to detect the instance failure) and role change time (the time it takes a backup instance to take over a cluster).

The framework generates two primary metrics: instance failure recovery time and area failure recovery time.

The *instance failure recovery metric* profiles an MCSDN system's ability to deal with failures in a cluster, representing the time it takes a backup instance to detect a failure and modify its role. The framework calculates the instance

failure recovery metric as the time difference between the final action executed by the framework and the last role change request event in $E_{observed}$.

An area failure occurs when a controller cluster no longer contains active instances. The *area failure recovery metric* represents the time it takes an MCSDN system to detect the failure of a controller cluster (area) and recover from the event by modifying its inter-area paths to avoid the unmanaged area. For this evaluation of Helix, we calculate the area failure recovery metric as the difference between the final action executed by the framework and the last observed path installation event generated by the local controllers.

The framework can also generate *instance and area join metrics*. These two metrics allow network operators to assess how well an MCSDN system copes with administrative tasks. Area and instance join time help characterise the overheads of the system when adding new controllers to scale the network to cope with increased traffic or when restarting devices to perform upgrades.

Performing failure recovery experiments can also provide insight into the stability of a system. Suppose we compare two systems by emulating a single instance failure that we expect not to cause an area failure. If the framework reportes that one of the systems modified its paths, this indicates that this system has poorer forwarding stability because it could not tolerate the failure without performing a path change.

7.2 Testing Methodology

This evaluation used a control channel latency of 20ms for all performed experiments. We repeated experiments 100 times and reported the average metric and component values along with the 95% confidence intervals. It was intractable to evaluate Helix under all possible failure scenarios. As a result, we defined several hard-to-solve simultaneous (§7.4) and cascading failure (§7.5) scenarios that we used to evaluate Helix's control plane failure resilience. The used failure scenarios assessed Helix's failure recovery performance, dynamic

scaling capabilities (e.g. adding an area to increase forwarding capacity), and response to administrative tasks (e.g. restarting instances to deploy upgrades). The evaluation undertaken in this chapter provided insight into the capabilities and limitations of Helix and its design choices.

This evaluation did not consider the failure of inter-controller links when performing experiments. If a connection between two controllers fails, the two devices will no longer share state and coordinate. We assume that Helix's protection mechanism will deal with such failures when using an in-band inter-controller connection strategy (evaluated in chapter 6). For this deployment approach, the system treats inter-controller communications as data plane traffic. For an out-of-band deployment, we assume these links provide enough resilience to minimise such failures.

7.3 Emulation Framework Example

This section outlines the notation used in this evaluation and presents an example failure scenario. All local and root controller instances use the same naming convention. An instance label contains the area/cluster ID (i.e. CX) followed by a dot and the instance ID.

The example failure scenario in this section assumes that T contains several areas numbered 1 through X. Suppose that $Area_1$ of T contains three local controller instances. The instances are labelled $C1.0$, $C1.1$ and $C1.2$. We assume that the framework selects the instance with the lowest ID to manage a cluster at the start of the experiment. As such, instance $C1.0$ is the primary instance of the $Area_1$ cluster, while $C1.1$ and $C1.2$ are backup instances.

Figure 7.1 shows a portion of a timeline diagram that indicates the stage completion time (recovery metric) and its relevant components. Labelled timeline points show actions and events in chronological order.

The example F_{stage} outlined in this section contains two actions, fail instance $C1.0$ and $C1.1$. The framework will begin observing and recording

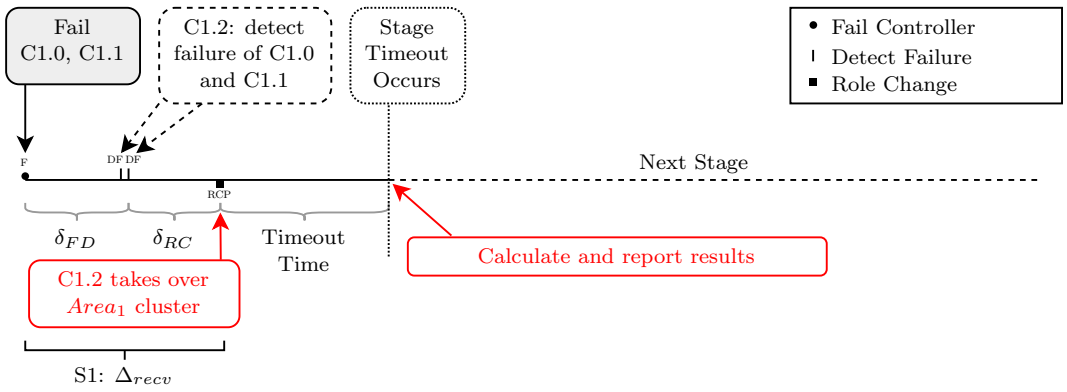


Figure 7.1: Timeline diagram showing an example experiment stage. The timeline shows the failure recovery metric (Δ_{recv}) along with its detection (δ_{FD}) and role change (δ_{RC}) component values.

events at the start of the F_{stage}. During the experiment, $C1.2$ detects the failures of $C1.0$ and $C1.1$. The framework will capture two local failure detection events generated by $C1.2$ (DF points on the timeline).

In response to the failures, $C1.2$ will modify its role to take over the $Area_1$ cluster by sending role change requests to every switch in the area (RCP point on the timeline). To group events to the current F_{stage}, the framework waits for n-seconds of inactivity (Timeout Time in figure 7.1) before generating the results and proceeding with the experiment.

For the example F_{stage}, the framework computes one metric and two-component values based on the observed events. First, δ_{FD} represents the failure detection component for $C1.2$. δ_{FD} is the time difference between the final failure action executed by the framework and the last DF event. Second, δ_{RC} represents the role change component, which the framework calculates as the difference between the last DF event and the last RCP event. Finally, the framework calculates the failure recovery metric or stage completion time (Δ_{recv}) as the difference between the final action and the last RCP event. Δ_{recv} indicates the time it took the system to detect and recover from the introduced control plane failures.

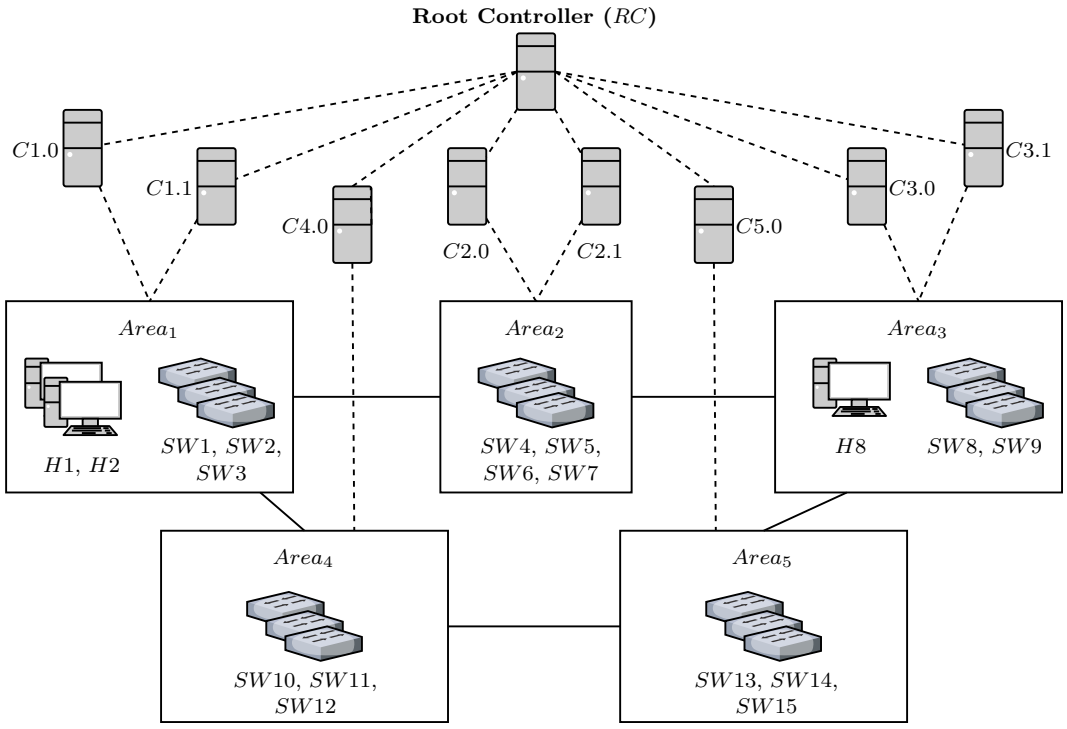


Figure 7.2: Diagram showing the topology used for the Failure Scenario 1 evaluation experiments. The network contained five areas, with two instances deployed in the $C1$, $C2$ and $C3$ cluster. The $C4$ and $C5$ local controller clusters contained a single deployed instance.

7.4 Scenario 1: Simultaneous Device Failures

Failure Scenario 1 emulated simultaneous device failures within a topology separated into five areas, illustrated in figure 7.2. $Area_1$, $Area_2$, and $Area_3$ each contain two controller instances. $Area_4$ and $Area_5$ contain a single instance per controller cluster. We separated Failure Scenario 1 into five stages containing actions that targeted the $Area_1$, $Area_2$ and $Area_3$ controller clusters. The actions of the failure scenario stages (1 through 5) were as follow:

1. Fail the primary instances of the $C1$, $C2$, and $C3$ clusters (e.g. $C3.0$)
2. Restart instance $C2.0$
3. Fail instance $C2.1$
4. Fail instance $C2.0$
5. Restart instance $C2.1$

Metric (Time)	Time (s) over 100 iterations		
	Average	*CI (95%)*	CI Range
(a) Multi-Device: Instance Failure Recovery	1.609	0.181	1.428 - 1.790
Failure Detection	1.003	0.033	0.970 - 1.036
Role Change	0.036	0.002	0.034 - 0.038
(b) Instance Join Time	0.959	0.056	0.903 - 1.015
Controller Start	0.426	0.005	0.421 - 0.431
Initiation Phase	0.005	0.000	0.005 - 0.005
Switch Enter	0.522	0.056	0.466 - 0.578
Role Change	0.011	0.000	0.011 - 0.011
(c) Single-Device: Instance Failure Recovery	1.106	0.116	0.990 - 1.222
Failure Detection	0.958	0.058	0.900 - 1.016
Role Change	0.037	0.003	0.034 - 0.040
(d) Area Failure Recovery	3.426	0.084	3.342 - 3.510
Failure Detection	2.365	0.084	2.281 - 2.449
Root Compute Path	1.002	0.000	1.002 - 1.002
Root Path Installation	0.059	0.002	0.057 - 0.061
(e) Area Join Time	13.228	0.035	13.193 - 13.263
Controller Start	0.411	0.004	0.407 - 0.415
Initiation Phase	0.501	0.000	0.501 - 0.501
Switch Enter	0.586	0.058	0.528 - 0.644
Role Change	0.026	0.004	0.022 - 0.03
Root Compute Path	13.164	0.035	13.129 - 13.199
Root Path Installation	0.064	0.002	0.062 - 0.066

Table 7.1: Average (over 100 iterations) Helix metric and component values collected using Failure Scenario 1 (simultaneous device failures). Failure detection made up over 95% of Helix's failure recovery metrics. Helix delayed area recovery and instance join operations to improve forwarding stability.

Table 7.1 contains Helix's average metric and component values recorded over 100 experiment iterations. Each stage of the failure scenario evaluated a particular aspect of Helix's control plane resilience, producing different metrics. §7.4.1 - §7.4.5 discuss each stage individually.

7.4.1 Stage 1: Multi-Device Instance Failure Recovery

Stage 1 (S1) emulated simultaneous controller failures in multiple clusters. This stage evaluated the efficiency of Helix's instance abstraction, which shields the root controller from local instance recovery events. As long as a cluster contained an active instance, Helix did not propagate failure events to the root controller and did not recompute paths. During the experiment, instances $C3.1$, $C2.1$ and $C1.1$ detected the failure of the primary instances and modified their roles to take over their clusters.

For S1, the framework recorded three failure recovery events. Helix's average failure recovery metric was 1.609 seconds (metric a). The average failure

detection component of the metric was 1.003s, while the average role change time was 0.036s. The failure recovery metric (a) represents the time it took Helix to recover from all three instance failures. Conversely, the component values represent an individual cluster's failure detection and role change time averaged across all three affected areas.

We observed that failure detection made up most of Helix's failure recovery metric, accounting for over 95% of its value. In our experiments, the configured leader election module attributes and stage alignment with internal controller timers affected Helix's failure detection metric (discussed in §7.8).

7.4.2 Stage 2: Instance Join Time

Stage 2 (S2) emulated adding a new instance to a cluster, providing insight into the time it takes to add or restart a Helix controller instance. A system administrator may add a new instance to a controller cluster to improve resilience, respond to a failure (restart a failed instance), or perform upgrades.

During this stage, instance $C2.0$ started up and entered its initiation phase. $C2.0$ established a connection with every switch in $Area_2$ after the initiation phase elapsed. Next, $C2.0$ assigned itself a backup role.

Helix's average instance join metric (b) was 0.959s. It took $C2.0$ an average of 0.426s to start up, while its initiation phase lasted for 0.011s. Usually, an instance's initiation phase should take at least τ_{init} seconds (0.5s); however, during S2, $C2.0$ exited its initiation phase early because the $Area_2$ cluster contained other active instances, triggering a startup optimisation condition. This optimisation improves an instance's startup time, allowing it to join its cluster faster to avoid false-positive area failure detection events. All $Area_2$ switches took an average of 0.522s from the controller start action to establish a connection with the new local controller. During S2, $C2.0$'s average role change time was 0.011s.

The instance join metric is essential in planning administrative tasks. For example, based on our measurements, if a network operator performs an up-

grade, we expect the controller to take 1s to restart and join its cluster. While the initiation phase is a component of the instance join metric, because $C2.0$ exited its initiation phase early, the used configuration attributes did influence Helix's performance.

7.4.3 Stage 3: Single-Device Instance Failure Recovery

Stage 3 (S3) emulated a single instance failure and produced the same metric as S1 (§7.4.1). Collecting results for both stages enabled us to identify any emulation overheads or factors that can delay Helix's failure recovery process when multiple areas are involved. S1 emulated a more complex failure scenario compared to S3, performing three simultaneous instance failures. Helix's behaviour during this stage was consistent with the observed behaviour during S1 (§7.4.1).

Helix's average instance failure recovery metric was 1.106s (c). The average failure detection component was 0.958s, while the average role change component was 0.037s. Similar to our findings in §7.4.1, failure detection made up a large portion of Helix's instance failure recovery metric (97%). The component and metric values observed during this stage were consistent with §7.4.1, despite those experiments emulating a more complex failure. Based on these observations, we draw two conclusions regarding Helix and the emulation framework. First, the framework did not introduce significant emulation overheads to inflate the recovery metric. Second, a Helix LC was not affected by failures in other clusters.

7.4.4 Stage 4: Area Failure Recovery

Stage 4 (S4) emulated a complete cluster failure, evaluating the root controller's area failure detection and path recomputation performance. For S4, we considered that Helix recovered from an area failure once the system modified the installed inter-area paths. In normal circumstances, responding to area failures is a manual process. A network operator will respond to area

failures by restarting failed controller instances (evaluated in the next stage).

After $C2.0$ had failed, the $Area_2$ cluster no longer contained an active instance, the area becoming unmanaged. During this time, $Area_2$'s data plane continued to forward packets and respond to data plane failures; however, Helix lost the ability to perform reactive operations in the area, such as TE and topology discovery. As a result, the $Area_1$ and $Area_3$ controllers no longer received heartbeat packets on inter-area links leading to $Area_2$, causing them to eventually timeout the links and notify the root controller ($R0$) of the failure. In response to the failed link notifications, $R0$ removed the failed inter-area links, recomputed its paths to avoid using $Area_2$ (after applying a 1-second path consolidation period), and sent instructions to the local controllers. Although $Area_2$ still maintained some capabilities, Helix opted to avoid using the area because it could not guarantee optimal traffic forwarding. After the LCs applied $R0$'s inter-area path changes onto the data plane, we considered Helix recovered from the area failure.

In the evaluation experiments, $R0$ detected that $Area_2$ failed after Helix recovered from the introduced control plane failure (recomputed the inter-area paths to avoid using $Area_2$). The configuration attributes we assigned to Helix for the evaluation caused this behaviour. We configured Helix to recover from inter-area link failures aggressively. Detecting area failures via the root controller's timeout mechanism or using dead inter-area link notifications produces the same result (discussed further in §7.8).

Across all 100 experiment iterations, the average Helix area recovery metric (d) was 3.426s. Helix's average failure detection time was 2.365s or 70% of (d)'s value. It took Helix an average of 0.059s to modify the inter-area paths to avoid the unmanaged area. $R0$ recomputed its inter-area paths within 1.002s from detecting the failure of $Area_2$ (last received dead inter-area link notifications). Consistent with the findings in §7.4.1 and §7.4.3, failure detection time made up a large portion of the recovery metric (98% when we ignoring the 1s path consolidation period).

7.4.5 Stage 5: Area Join Time

Stage 5 (S5) evaluated Helix's ability to detect and use a new area. Like S2, S5 generated metrics that are useful for planning administrative tasks, such as adding a new cluster to increase the network's forwarding capacity.

During this stage, $C2.0$ started up and entered its initiation phase. $C2.0$ went through the complete initiation phase during S5 because the $Area_2$ cluster did not contain other active instances. After the initiation phase elapsed, $C2.0$ established connections with the switches of $Area_2$ and modified its role to take over the cluster. After taking over the cluster, $C2.0$ advertised its presence to neighbouring areas and the root controller. $R0$ discovered the new area and recomputed its inter-area paths to use $Area_2$.

Helix's average startup and initiation components for S5 were 0.411s and 0.501s. On average, the new Helix instance modified its role in 0.026s. Helix's average area join metric (e) was 13.164s. Consistent with §7.4.4, Helix's average inter-area path change time was 0.064s.

For this evaluation, we configured Helix to respond quickly to area failures but delay detecting and using new areas to improve the system's forwarding stability. Helix's average area join metric was 3.8x larger than its area recovery metric. An aggressive area failure recovery is desirable to ensure consistent TE policy application. Helix cannot perform TE optimisation in a particular area when a cluster failure occurs. As a direct consequence, the system cannot guarantee the forwarding performance of transiting traffic. We opted to configure Helix to recover from cluster failures quickly to maintain the system's TE capabilities, sacrificing stability for performance.

Delaying adding a new area has the opposite effect, improving forwarding stability by reducing the number of path changes. Helix assumes that the currently active areas provide adequate capacity to forward traffic. Due to this assumption, detecting and using a new area is not critical to providing congestion-free forwarding. Moreover, techniques such as capacity planning of links minimise packet loss due to congestion. We can consider area failures

as rare events because they entail multiple devices failing. Because Helix reduces the load on controllers, it minimises the chances that recovering from an instance failure will cascade into an area failure. Based on this observation, we can view S5 as having assessed Helix's performance when faced with a worst-case scenario.

7.4.6 Summary of Results

Helix took up to 1.6s to recover from instance failures, with failure detection making up over 95% (1s) of the metric value and role change time taking only 0.035s. Helix's average instance join metric was under 1s when a new instance joined a cluster with active devices. When recovering from area failures, Helix took an average of 3.5s to recover from the failure, with the detection component making up 2.365s or 70% of the metric (98% if we ignore the path consolidation period). Helix deployed an inter-area path modification to the data plane in under 0.065s.

The framework calculated the role change component as the difference between the final role change request sent by a controller and the last switch connection event. Because switches established connections with the controller in parallel and independently, the framework observed slight variation in switch connection events between experiment runs. We observed a slight variation in role change time in our experiments, which we can attribute to differences in the switch connection events.

7.5 Scenario 2: Cascading Device Failures

Cascading failures can occur when migrating load from one instance to another, overwhelming the primary instance and eventually causing all instances within a controller cluster to fail. The cascading failure actions defined in this scenario verified Helix's behaviour and capability to compartmentalise complex failures to clusters without needing root controller intervention. For this scenario, we

Metric (Time)	Time (s) over 100 iterations		
	Average	*CI (95%)*	CI Range
(a) Cascading Failure: Stage Completion	5.500	0.120	5.620 - 5.380
Failure Detection	1.074	0.031	1.105 - 1.043
Role Change	0.036	0.002	0.038 - 0.034
Controller Start (Restart)	0.431	0.005	0.436 - 0.426
Initiation Phase (Restart)	0.501	0.000	0.501 - 0.501
Switch Enter (Restart)	0.593	0.067	0.660 - 0.526
Role Change (Restart)	0.026	0.003	0.029 - 0.023
(b) Multi-Device: Instance Join Time	1.095	0.097	1.192 - 0.998
Controller Start	0.512	0.008	0.520 - 0.504
Initiation Phase	0.006	0.000	0.006 - 0.006
Switch Enter	0.485	0.040	0.525 - 0.445
Role Change	0.011	0.000	0.011 - 0.011
(c) Multi-Device: Area Failure Recovery	3.178	0.026	3.204 - 3.152
Failure Detection	2.117	0.026	2.143 - 2.091
Root Compute path	1.002	0.000	1.002 - 1.002
Root Path Installation	0.060	0.002	0.062 - 0.058

Table 7.2: Average (over 100 iterations) Helix metric and component values collected using Failure Scenario 2 (cascading device failures). The results and trends were consistent with the first scenario.

used a similar topology to the previous experiments (figure 7.2). The $Area_1$ and $Area_3$ controller clusters contained two instances in this evaluation, while $Area_2$ contained three instances. We separated Scenario 2 into three stages:

1. Fail the primary instance of the $C2$ cluster ($C2.0$). After 2 seconds, fail $C2.1$. After a further 2 seconds fail instance $C2.2$ and simultaneously restart $C2.0$.

2. Restart instances $C2.1$ and $C2.2$.

3. Fail instance $C2.0$ and $C2.1$. After 4 seconds, fail instance $C2.2$.

Table 7.2 contains Helix's average metric and component values calculated over 100 experiment iterations using Failure Scenario 2.

7.5.1 Stage 1: Cascading Failure Recovery

Stage 1 (S1) emulated a cascading multi-device controller failure. This stage evaluated Helix's ability to respond to multiple failure events without causing a false-positive area failure detection.

After the first failure action, both $C2.1$ and $C2.2$ detected the failure of $C2.0$. Because $C2.1$ had the lowest instance ID, it took over the $Area_2$ cluster,

while $C2.2$ maintained its backup role. After the framework executed the second failure action, $C2.2$ took over the cluster.

The final part of S1 assessed if a Helix instance could take over the cluster without the root controller declaring the area as failed. This part of the experiment validated the used configuration attributes, ensuring we correctly configured the area failure detection timeouts to prevent false-positive detection. During the experiments, $C2.0$ started up and took over its cluster before $R0$ declared $Area_2$ as failed.

The average stage completion time (a) for the experiments was 5.5s. Helix's average failure recovery metric across all three failure events was 1.11s. Consistent with the trend observed in §7.4, failure detection made up a large portion of the metric value (97% or 1.074s), while role change time contributed a small amount (0.036s). We observed an average controller startup and role change time after the initiation phase of 0.431s and 0.026s. $C2.0$ went through the complete initiation phase during this stage, which lasted 0.501s. The average startup metric and component values were consistent with §7.4.

7.5.2 Stage 2: Multi-device Instance Join Time

Stage 2 (S2) emulated restarting several failed instances from a cluster.

Helix's average instance join metric (b) for both instances was 1.095s. The two instances took an average of 0.512s to startup, with their initiation phase lasting 0.006s. Consistent with the behaviour observed in §7.4.2, $C2.1$ and $C2.2$ exited their initiation phase early because the $Area_2$ cluster contained active instances. Helix's average role change time for S2 was 0.011s.

Helix's behaviour, metric, and component values were consistent with the results observed in §7.4.2, despite this stage emulating more actions. Based on this observation, we draw two conclusions. First, we can verify the previous observation made in §7.4 that the framework did not introduce significant overheads during emulation. While the experiments performed in S2 contained more actions increasing the load on the framework, the results were consistent

with previous experiments. Second, adding a new instance to a Helix cluster did not affect the other active instances, and simultaneously restarting multiple instances did not affect their startup performance.

7.5.3 Stage 3: Multi-Device Area Failure Recovery

Stage 3 (S3) emulated a cascading failure that triggered a complete cluster failure. S3 may be encountered in real-world situations when moving resources from several failed instances to a backup instance.

After the first failure action, $C2.2$ detected the instance failure and took over the $Area_2$ cluster. After 4 seconds, the framework failed instance $C2.2$, causing the $Area_2$ cluster to become unmanaged. Similar to the behaviour observed in §7.4.4, $R0$ detected the failure of the area via failed inter-area link notifications received from the $Area_1$ and $Area_3$ clusters. The root controller applied a 1s path consolidation period, recomputed the inter-area paths to avoid using $Area_2$, and sent instructions to the LC. Consistent with the behaviour observed in §7.4.4, Helix recovered from the area failure before $R0$ declared $Area_2$ as failed.

Helix's average area failure recovery metric (c) was 3.178s, with an average detection component of 2.117s and an average path installation component of 0.060s. Consistent with the trend observed in §7.4, failure detection time made up a large portion of the area recovery metric (97% if we ignore $R0$'s path consolidation period).

7.5.4 Summary of Results

Despite emulating more complex cascading failures, Helix's metric and component values for these experiments were consistent with §7.4. Helix's instance failure recovery metric was 1.11s, with failure detection making up 98% of the metric value (1.074s) and role change taking only 0.036s. Similarly, the area failure recovery metric was 3.178s, with the detection component making up 66% (98% if we ignore the consolidation timeout) and path installation taking

just 0.060s. Based on the observed results during these experiments, we can verify the previous assertions made for Helix and draw several conclusions:

- Failure detection time made up a large portion of Helix's instance and area failure recovery metrics.
- Helix's architecture and design choices enabled instances to quickly recover from area failures and modify their role.
- Helix's instance/role abstraction mechanism promoted stability by avoiding unnecessary path changes.

7.6 Root Controller Cluster Failure

The Helix RC performs three distinct operations: (a) inter-area path computation, (b) inter-area TE optimisation, and (c) area failure detection and recovery. Because Helix uses replication to improve the failure resilience of all controller clusters, an RC cluster failure is rare as it will involve multiple devices failing simultaneously. Despite this, Helix's design choices and architecture limit the effects of an RC cluster failure. Helix offloads critical inter-area operations to the LCs to improve the failure resilience of the system by removing dependencies between devices. The RC offloads inter-area data plane failure recovery and TE optimisation to the LCs. In the event of a complete RC cluster failure, the LCs maintain their ability to forward inter-area traffic, resolve inter-area data plane failures, and perform TE optimisation. While Helix still implements an RC-based TE optimisation operation, this operation occurs less frequently. Moreover, the TE evaluation experiments outlined in chapter 8 found that the Helix LCs will address most inter-area congestion locally without RC intervention. Because Helix will still perform inter-area TE optimisation and resolve data plane failures, a complete RC cluster failure has minimal impact on traffic forwarding performance.

When a complete RC cluster failure occurs, Helix will lose the ability to

respond to area failures (complete LC cluster failures). If both an area and the RC fail, Helix will not be able to guarantee the performance of traffic transiting the area. The affected area will become unmanaged, Helix losing its ability to perform TE optimisation. While Helix cannot improve the forwarding performance of traffic in an unmanaged area, the system will still deal with data plane failures. It is also worth noting that this scenario is very unlikely as all instances in an LC and the RC cluster need to fail. Moreover, good capacity planning of links will minimise the chances that an unmanaged area experiences congestion even if there is a peak in traffic volumes.

7.7 Limitations of Experiments

The computed role change component of the experiments measured how long it took a local instance to send role change requests to every connected switch. A limitation of our evaluation was that it did not consider how the number of switches in an area affected the role change component. The number of switches connected to a controller will influence the observed role change time. Because the local controllers use asynchronous mechanisms when generating and sending OFP packets, intuition would suggest that role change time will slightly increase as the number of switches in an area grows.

7.8 Modelling Helix's Performance

The used configuration attributes and experiment stage alignment with controller timers affected several of Helix's metrics and component values. This section discusses these effects and proposes two models to estimate Helix's failure recovery performance.

7.8.1 Instance Failure Recovery

The notation used in this section is as follows. τ_k represent the leader election keep-alive interval. $\tau_{timeout}$ represents the amount of time the module waits before considering a keep-alive message as missing. Finally, τ_{init} represents the local controller's initiation phase length. The local controller applies the initiation phase on startup to allow the device to detect active instances before assigning itself a role. For the evaluation experiments, we used a τ_k of 1 second. Helix defines $\tau_{timeout}$ as half the keep-alive interval (0.5 seconds). Finally, the initiation phase was configured to the same value as $\tau_{timeout}$.

The Helix leader election module broadcasted a keep-alive message (heartbeat) every keep-alive interval (τ_k seconds). The keep-alive messages enabled the module to discover other active instances and detect failures. Because Helix synchronised the keep-alive send-timers of all devices, we assume that each cluster had a single keep-alive interval. In essence, all controller instances broadcasted heartbeats at the same time. Despite this, Helix implemented mechanisms to prevent false-positive failure detection, tolerating slight timer variations and inter-instance latency. The module waited up to $\tau_{timeout}$ seconds before declaring a heartbeat as missing. When N_k consecutive heartbeats for a device were missing, the module considered the instance failed.

We configured Helix to declare an instance as failed after missing one heartbeat. We were able to use an $N_k = 1$ without encountering false-positive failure detection in our experiments due to two factors. First, Helix deployed instances within the same cluster to minimise inter-instance latency by ensuring that the module received heartbeats within the allocated $\tau_{timeout}$. Second, Helix removed variation in the keep-alive timers by ensuring instances broadcasted heartbeat messages at similar times.

The emulation framework treated MCSDN systems as black-boxes and, as such, did not align stages with internal controller timers. Suppose that an instance failed before the keep-alive interval elapsed. In this situation, the cluster would detect the failure after $\tau_{timeout}$ seconds, implying the failure

detection time is small. Next, suppose that an instance failed after a new keep-alive interval starts. In this case, the cluster would detect the instance's failure during the next τ_k, implying that the failure detection time is larger.

Because of this variability, we cannot calculate an exact value for the failure detection component. We can, however, define a maximum expected failure detection time based on the used Helix configuration attributes. Both τ_k and implicitly $\tau_{timeout}$ affect the failure detection component of the instance recovery metric. Effectively, Helix has a maximum instance failure detection time of $\tau k * N_k + \tau_{timeout}$. Assuming that $\tau_{timeout} = \frac{\tau_k}{2}$, we expect that Helix instances will take at most $(\tau_k * N_k) + \frac{\tau_k}{2}$ seconds to detect a local instance failure. Based on the configuration attributes used in this evaluation, instance failure detection should take no more than 1.5 seconds. Referring back to §7.4 and §7.5, we see that all instance failure detection components adhere to this maximum constraint.

Helix's instance failure recovery metric can be approximated as $\Delta_{recv} \approx \delta_{FD}+\delta_{RC}$, where δ_{FD} represents the instance failure detection time and δ_{RC} the role change time. In our evaluation, the instance role change time was under 0.04s (40ms). While role change time depended on factors such as latency and the size of the area, the role change component represented a small portion of the overall recovery metric value (2%). As such, errors in estimating role change time will have a small impact on the overall estimated metric value. We can approximate Helix's local instance failure recovery metric as:

$$\Delta_{recv} \approx (\tau_k * N_k + \frac{\tau_k}{2}) + 0.04$$

7.8.2 Area Failure Recovery

The notation used in this section is as follows. τ_{iap} represent Helix's inter-area link timeout, while τ_{lldp} the local controller's LLDP packet-send interval. τ_{rk} represents the root controller's keep-alive interval. $\tau_{rtimeout}$ represents the amount of time the root controller (RC) waits before considering a keep-alive

message from a local controller as missing.

Similar to instance failure detection time, stage alignment and Helix's configuration attribute values affected Helix's area failure detection time. Our experiments showed that Helix detected area failures using failed inter-area link notifications. Helix employed two mechanisms to detect area failures:

First, the Helix root controller detected area failures via failed inter-area link notifications received from the local controllers. The Helix local controller used the standard LLDP mechanism to detect inter-area link failures (discussed in §5.1). The topology discovery module expected to receive a heartbeat from a neighbouring area on an inter-area link within τ_{iap} seconds. Helix assigned τ_{iap} as five LLDP send-intervals, implying that a local controller considered an inter-area link as failed after five consecutive heartbeats were missing. After an area lost all controller instances, the area stopped sending LLDP packets to neighbouring areas, causing neighbouring controllers to declare the inter-area links as failed and notify the root controller.

Second, the RC employed a timeout mechanism similar to the instance failure detection mechanism. Local controllers send keep-alive messages to the root controller every τ_{rk} seconds. The root controller waited up to $\tau_{rtimeout}$ seconds before declaring a heartbeat as missing. We assigned $\tau_{rtimeout}$ a large enough value to account for inter-controller communication latency and timer differences to prevent false-positive area failure detection events. The root controller considered an area had failed when N_{rk} consecutive heartbeats for a controller were missing. Helix defined $\tau_{rtimeout}$ as half the keep-alive interval.

For this evaluation, we used a τ_{lldp} of 0.4s, implying that τ_{iap} was 2 seconds. In contrast, we configured τ_{rk} to 4 seconds and N_{rk} to 1.

Because Helix local controllers did not synchronise LLDP-send intervals, the controllers generated LLDP packets at different times. We expect that the variation in LLDP timers is at most τ_{lldp} seconds. We can approximate Helix's maximum inter-area link failure detection as $\tau_{iap} + \tau_{lldp}$ or 2.4 seconds. Referring back to §7.4 and §7.5, all area failure detection components adhered

to this maximum constraint.

When using the second area failure detection method, Helix should detect an area failure within $\tau_{rk} * N_{rk} + \tau_{rtimeout}$ seconds (6s based on the used configuration attributes for the experiments). We can estimate Helix's maximum area failure detection time as the smallest value of either $f(x)$ or $g(x)$:

$$f(x) = (\tau_{rk} * \tau_{nk}) + \frac{\tau_{rk}}{2} \qquad\qquad g(x) = (\tau_{iap} + \tau_{lldp})$$

We can approximate Helix's area failure recovery metric as $\Delta_{Arecv} \approx \delta_{AFD} + \tau_{RConsolidate} + \delta_{PI}$, where δ_{AFD} represents the area failure detection time, $\tau_{RConsolidate}$ the root controller path computation timeout and δ_{PI} Helix's inter-area path installation time. In this evaluation, Helix took under 0.07 seconds to modify the inter-area paths in response to failures. We used a $\tau_{RConsolidate}$ of 1 for our experiments. Based on the more aggressive inter-area link failure detection, we can approximate Helix's local area failure recovery metric as:

$$\Delta_{Arecv} \approx (\tau_{iap} + \tau_{lldp}) + 1.4$$

7.9 Conclusion

This section concludes the control plane evaluation chapter by assessing Helix's design choices related to the failure recovery of areas and instances. We use the experiment results and models to discuss how different design decisions impact recovery performance. Assuming a hierarchical control plane architecture, we can identify two possible methods to detect and recover from instance failures:

- *Replication*: A system can perform instance failure detection and recovery locally within a cluster (i.e. Helix's approach).
- *Reassignment:* A system can use the RC as an orchestrator.

The *replication method* describes an offloaded or proactive operation. In contrast, the *reassignment method* is a centralised or reactive operation that

involves a remote entity in the decision process. Helix uses replication to deal with controller failures. Using the root controller as an orchestrator (reassignment method) is a common approach used by systems that deploy a flat control plane architecture (e.g. ONOS [9]) and deal with controller failures by allowing any controller to take over any switch in the network.

We used a control channel latency of 20ms (average WAN latency) and conservative Helix timeout attributes for the conducted experiments. Despite Helix deploying instances within the same cluster, the emulation framework applied the defined control channel latency to all controller devices. As such, instances deployed in the same cluster experienced inter-area WAN latency when communicating. Based on this factor, we assume that our evaluation results would mimic the performance we expect to see from the reassignment method. If Helix were to apply the reassignment method, we would need to configure Helix's timeout attributes to large enough values to prevent false-positive failure detection. For example, if $\tau_{timeout}$ is too low, the backup instances may take over the cluster despite the primary device still functioning, causing back-and-forth role change operations.

In our evaluation experiments, Helix's average instance recovery time was 1.6s. We can improve Helix's instance failure recovery performance by using a lower τ_k and $\tau_{timeout}$ value. Because Helix deployed instances in the same cluster (low communication latency) and synchronised keep-alive timers, we can configure a more aggressive failure detection without encountering false-positive detection events. For example, decreasing τ_k to 0.2s will reduce Helix's failure detection time to 0.3s. The new failure detection time is 5x times faster compared to the average δ_{FD} we observed for the experiments. Using the lower τ_k will decrease instance failure recovery to 0.34s (4.5x faster). Comparing the two failure recovery methods, we assume that using the root controller as an orchestrator would yield a failure recovery metric of 1.6s. By adjusting Helix's configuration attributes, the system can recover from failures up to 4.5x faster without experiencing false-positive failure detection events.

If Helix experienced false-positive instance failure detection, Helix's design choices would minimise the impact of these events. Helix instances recover state from the data plane to prevent unnecessary path changes. When a Helix instance takes over a cluster, it will not recompute paths. As such, false-positive failure detection will only cause a role change operation without affecting the system's forwarding stability.

Area Failure Recovery: We can apply a similar approach to improve Helix's area failure recovery. Unlike instance recovery, accounting for inter-controller latency in Helix's configuration attributes is critical for this operation. False-positive area failure detection affects the system's forwarding stability because the RC recomputes inter-area paths in response to area failures. We can improve Helix's area failure recovery by targeting the inter-area link failure detection mechanism. Inter-area link failure detection involves neighbouring devices. Because areas are adjacent, inter-area latency will be smaller than the LC-RC communication latency, allowing us to configure a more aggressive failure detection. In contrast, the RC's keep-alive mechanism will take longer to detect failures as it needs to account for higher latency. By setting τ_{lldp} to 0.1s, we can reduce Helix's inter-area link failure detection to 0.6s. The new inter-area link failure detection is 4x faster compared to Helix's δ_{AFD} observed in the experiments. If we use a $\tau_{RConsolidate}$ of 1 second, Helix will recover from area failures within 1.67s (2x faster).

An MCSDN system that detects area failures using a keep-alive mechanism (similar to the Helix RC) will have slower failure recovery performance compared to Helix. These systems need to account for higher inter-area latency between controllers to prevent false-positive detection. If Helix used its keep-alive mechanism to detect area failures, Helix's area recovery time would increase to 7.4s (4.4x slower). Even if we decrease τ_{rk} to 1s, area failure recovery will still be slower (2.4s or 1.5x higher). Because of Helix's design choices, we can configure Helix to quickly recover from area failures while avoiding the risk of introducing false-positive area failure detection events.

Chapter 8

Evaluation: TE Optimisation

The primary contribution of this chapter is to evaluate Helix's Traffic Engineering (TE) performance and compare Helix against several other systems. This chapter evaluates TE performance by conducting experiments using YATES [52], a simulation framework. This chapter also provides a concrete testing methodology and presents our extensions to YATES to add support for evaluating reactive TE optimisation systems (e.g. Helix).

Helix offloads inter-area TE from the root to the local controllers. The main concern with offloaded operations is the loss of global visibility, which may cause Helix to make suboptimal TE decisions. To this end, this chapter sets out to answer two research questions: (1) "What are the effects of using multiple versus a single controller on TE performance?"; (2) "What are the effects of loss of centralised scope on TE optimisation performance?".

This chapter is structured as follows. First, §8.1 contains a brief introduction of YATES and discusses our modifications to the framework to add support for evaluating reactive TE optimisation systems. §8.2 discusses our testing methodology, topologies, and algorithms used in our experiments. Next, §8.3 presents the evaluation results collected across three topologies, and §8.4 discusses the limitations of this evaluation. Finally, §8.5 concludes this chapter.

8.1 Simulation Framework

YATES is a simulation framework that enables network operators to develop and assess offline TE optimisation systems. The framework simulates a network's forwarding behaviour by keeping track of aggregated traffic on links. YATES divides experiments into runs, which last for a predetermined number of iterations ($T_{simtime}$). In every iteration, the framework advances packets to their next hop and applies fair queueing, keeping track of metrics such as dropped packets. YATES requires four user inputs to run an experiment:

- a topology graph that specifies the network to use for the simulation
- a TE algorithm to evaluate
- a set of traffic demands to use during the experiment (sent traffic)
- a set of predicted traffic demands to provide to the TE algorithm

YATES expects demands encoded as a list of traffic matrices where each row represents one TE matrix that coincides with a particular experiment run. The columns of the demand TE matrix represent the amount of traffic a source node from the topology will send to a destination node during the experiment. In contrast, the prediction matrix specifies how much traffic we expect each source-destination pair to send. YATES provides the prediction matrices to the TE algorithm, which uses them to compute routing schemes.

YATES only supports simulating offline TE systems because the framework calls the TE algorithm it is evaluating to generate routing schemes between experiment runs. YATES does not collect metrics or introduce traffic into the topology while the algorithm computes its paths. In contrast, Helix's TE algorithm makes real-time decisions based on collected metrics. As such, we added several new functions to YATES' default algorithm interface. These new functions enable a TE algorithm to gather live metrics from the topology and perform decisions during the experiment. The new functions are as follows:

- f_A provides an algorithm with the current link usage information. The framework triggers the method for every link during an iteration.

- f_B notifies the system when the current iteration has finished.

- f_C allows YATES to check for and apply a routing scheme update generated by the algorithm. If a routing scheme update is available, the framework uses it to forward traffic during the next iteration. The framework triggers f_C after f_B.

We built a new YATES algorithm module for Helix to conduct our experiments. The algorithm module maintains local network information (e.g. metrics) and simulates the system's behaviour. For example, the module simulates Helix's polling behaviour by aggregating and averaging link usage statistics after a predetermined number of iterations have elapsed (i.e. τ_{stats}). Unlike the other algorithm modules provided by YATES, the Helix module implements a routing scheme update timeout to simulate the effects of control channel latency on decisions. Both the statistics polling and routing update timeouts are user-configurable. Finally, we built a shallow wrapper of each Helix controller type that runs Helix's path computation and TE optimisation algorithms. The YATES module calls the controller wrappers to trigger specific events. We will go through the six steps involved when resolving congestion to explain how YATES, the module, and the controller wrapper interact:

1. f_A provides the Helix module with the current traffic on links. The module adds the traffic information to a link's total observed usage.

2. After the framework forwards traffic on every link, it will trigger f_B. The method tells the Helix module that the current iteration has finished. If the current poll interval has elapsed, the module averages the link statistics and checks for congestion.

3. If the module has detected congestion in the network, it will call the Helix controller wrapper to resolve it. The module provides the wrapper

with all relevant metrics required to perform TE optimisation.

4. The wrapper executes Helix's TE algorithm, providing a new routing scheme (path changes) to the module.

5. The module processes the path changes and triggers the update timeout.

6. Once the timeout elapses, f_C returns the new routing scheme to the framework. The framework uses the new scheme to forward traffic during the next iteration.

8.2 Testing Methodology

This evaluation introduced congestion in a network and measured the percentage of traffic lost during an experiment to evaluate the TE performance of systems by assessing how much of the introduced congestion the system resolved. We also considered forwarding stability in our experiments by looking at the path change churn metric or the number of path modifications the system performed. Frequently changing paths disrupts the flow of packets by causing packet reordering, which negatively affects performance. A TE system needs to balance TE performance with stability to avoid such problems.

For these experiments, we used a $T_{simtime}$ of 500, a TE threshold of 95%, and a Helix poll interval of 100. Helix processed link statistics five times per experiment run. The TE threshold limited TE algorithms to use up to 95% of a link's capacity before considering the link congested.

Evaluated Algorithms: We compared the performance of Helix against three load balancing (ECMP [37], VLB [94], and Raeke [73]) and three TE algorithms (CSPF, MCF [28], and Semi-MCF-KSP) provided by YATES. Semi-MCF-KSP is an approximation of a centralised TE optimisation system (e.g. SWAN [36]). This algorithm formulated a constrained MCF problem to generate traffic ratios, which the system used to split and forward traffic on multiple paths.

Experiment Topologies: We collected results using three topologies of different complexity and size from the Topology Zoo Project [49]. First, we collected results using the AT&T MPLS topology, which is representative of an average-sized Wide Area Network (WAN). The AT&T MPLS topology contained 25 nodes and 56 links. Second, we collected results using the Abilene topology, representative of a small-sized network. The Abilene topology had 11 nodes and 14 links. Finally, we collected results using the Hibernia Atlantic network, an example of a large inter-continental WAN. The Hibernia network contained 55 nodes and 81 links. Using topologies that vary in size allowed us to evaluate Helix's ability to scale and cope with an increasing number of metrics and links. For example, the Hibernia topology contains 25x more source-destination pairs than the Abilene network (3025 vs 121) and almost 5x more than the AT&T MPLS network (3025 versus 625). Using more source-destination pairs increased the TE algorithm's search space and the number of factors it considered when computing new paths.

Experiment Demand: We used YATES to generate synthetic demands for all three topologies in our experiments. YATES produced TE matrices with realistic traffic patterns using the gravity model [74]. For example, figure 8.1 shows the total traffic sent between source-destination pairs of the AT&T MPLS topology. The used demands contained diurnal patterns, simulating peak and off-peak usage commonly encountered in real networks. All the tested TE algorithms used predictions to compute an offline routing scheme.

YATES allows scaling the provided traffic demand for an experiment by a specific multiplication factor. The framework applies the traffic multiplier to the generated traffic and prediction matrices. To perform our experiments, we found a traffic multiplier that was sufficiently large to introduce congestion into a specific network. We use this multiplier to collect results and compare the performance of the TE algorithms. To showcase the ability of the tested systems to scale and adapt to increasing traffic demands, we gathered results using three gradually increasing scale factors. Increasing the traffic multipliers

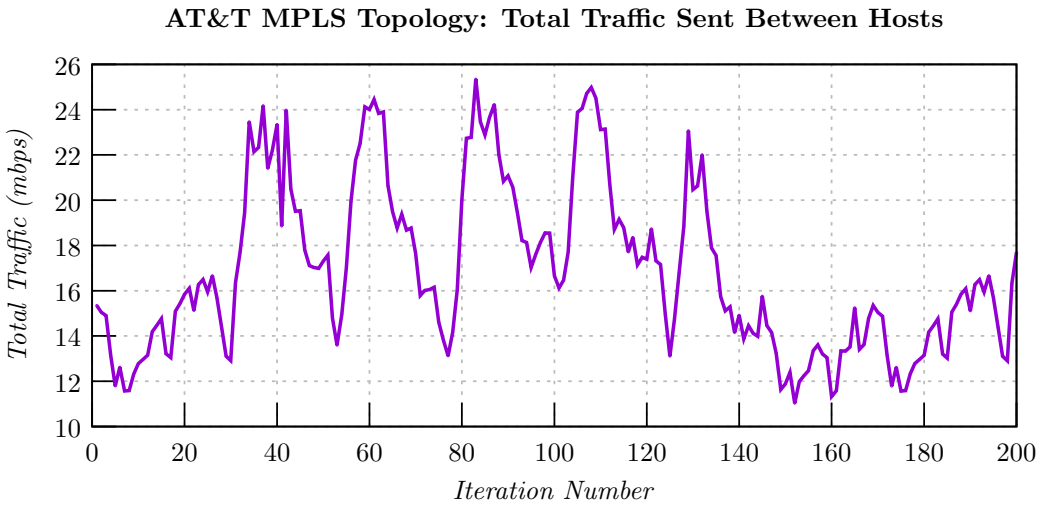


Figure 8.1: Graph showing the total traffic sent between hosts for the AT&T MPLS Topology evaluation results. Demand contains diurnal patterns, simulating peak and off-peak usage.

used for the experiments caused the framework to send more traffic through the topology, placing more strain on the TE systems.

8.3 Evaluation Results

This section discusses results for the AT&T MPLS (§8.3.1), Abilene (§8.3.2), and the Hibernia Global topologies (§8.3.3). We repeated each experiment three times using gradually increasing traffic multipliers. Because YATES is a simulation framework, it did not introduce delays when executing TE algorithms. Moreover, YATES maintained consistent traffic rates based on the provided demands implying that the evaluated systems had consistent behaviour and performance during experiments. Unlike the previous evaluation chapters that used emulation frameworks to collect results, we did not need to repeat experiments as repeated experiments would produce the same results. All results presented in this chapter are reproducible. To encourage further development in this area, we provide all of our collection scripts, along with raw and processed results for all experiments outlined in this section [2].

All load balancing algorithms performed poorly in our experiments com-

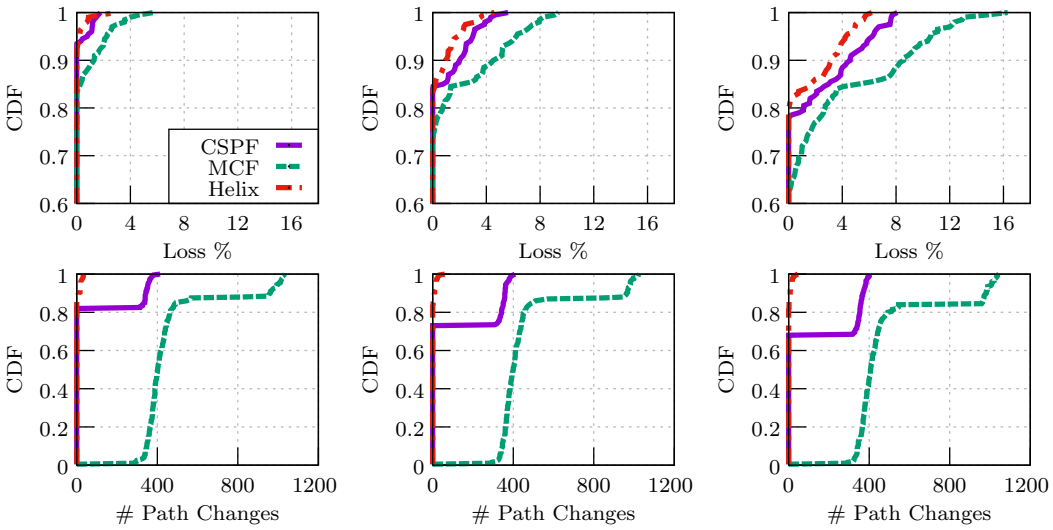
pared to the evaluated TE systems. This poor performance is unsurprising because load balancing algorithms did not divide traffic based on predicted demands or congestion information. In our experiment, the load balancing algorithms split traffic equally (i.e. ECMP) or used random node indirection to randomly forward traffic on several paths (i.e. VLB and Raeke). For example, ECMP experienced congestion loss during all experiments with a maximum loss rate of 21% for the AT&T MPLS topology. In contrast, SWAN (i.e. Semi-MCF-KSP) did not experience any congestion loss for 56% of runs and had a maximum loss rate of 14.1%. Because we observed a significant gap in system performance across our evaluation, this chapter will discuss and present results for the top-three best performing TE systems.

8.3.1 AT&T MPLS Network Results

Figure 8.2 shows a CDF graph of congestion loss (top row) and the number of path changes (bottom row) observed on the AT&T MPLS topology.

TE Performance: With a 500x traffic multiplier (figure 8.2a) Helix experienced no congestion loss for 90.5% of experiment runs compared to 92.5% for CSPF. While Helix encountered congestion during more runs, Helix performed better than CSPF. The average loss rate across all runs for Helix was 0.5%, with 99% of its runs reporting up to 0.9% loss. In comparison, CSPF's average loss rate was 0.07%, with 99% of its runs reporting up to 1.5% loss (1.6x higher). Despite this, Helix's maximum recorded loss rate was higher than CSPF's (3.0% versus 1.8%). This loss rate was recorded at the start of the experiment (first run) when Helix started with unoptimised paths. During this run, Helix forwarded 20% of all traffic using shortest path forwarding before modifying the routing scheme.

In contrast, because CSPF is an offline TE algorithm, it performed its optimisation before YATES forwarded traffic through the topology. The second-highest loss rate value for Helix was only 1.7%. Both Helix and CSPF performed better than MCF. MCF reported no congestion loss for 83.5% of ex-



(a) 500x (b) 550x (c) 600x

Figure 8.2: AT&T MPLS Topology: CDF graphs showing congestion loss rate and the number of path changes per run using three traffic multipliers. Helix experienced less congestion loss and performed up to 12x fewer changes compared to CSPF and 29x compared to MCF.

periment runs and had a maximum loss rate of 5.7%.

When we increased the traffic multiplier to 550x (figure 8.2b), we observed a slight performance improvement for Helix. Helix had a loss rate of up to 2.4% for 97.5% of runs compared to 3.9% for CSPF (1.6x higher) and 7.5% for MCF (2.5x higher). Moreover, when increasing traffic in the experiment, Helix had a lower maximum loss rate than CSPF (4.6% versus 5.6%). The lower loss rate suggests that Helix's TE algorithm was better at finding solutions for congestion when dealing with higher traffic demands.

Consistent with the trend observed in the 500x traffic multiplier results, CSPF and Helix resolved more congestion than MCF, despite MCF being a more complex algorithm. While this may not seem intuitive, we attribute the poor performance of MCF to MCF's accuracy targets. MCF algorithms attempt to resolve all congestion in the topology using complex path manipulation techniques, while algorithms such as CSPF are simpler. Because our experiments placed a high load on links, MCF routed packets on longer paths, dispersing the introduced congestion across multiple links rather than resolv-

ing it. MCF's longest routing scheme path had 17 hops compared to 9 for CSPF and Helix. MCF introduced minor loss across multiple links by using longer paths, hindering its ability to deal with high traffic loads.

The difference in performance between Helix and the other systems continued when using a 600x traffic multiplier (figure 8.2c). Helix's average loss rate was 0.62% compared to 0.94% for CSPF (1.5x higher). Consistent with the 550x results, Helix had the lowest maximum loss rate reported during a run (6.2% versus 8.1%). Based on the trend in performance, we conclude that Helix adapted well to increasing traffic demands.

Forwarding Stability: While Helix marginally outperformed the other evaluated algorithms in terms of addressing congestion, Helix had significantly better forwarding stability. With a 500x traffic scale factor, Helix performed up to 36 path changes during a run compared to 414 path changes made by CSPF (12x higher) and 1034 by MCF (29x higher). We observed a similar difference when increasing the traffic demands of the experiment. With a 550x and 600x traffic scale factor, Helix's maximum path change churn metrics were 60 and 46, respectively. In contrast, CSPF's maximum churn metric values were 408 and 400. Like the previous results, MCF performed the most path modifications. We observed a slight drop in path change churn for Helix and CSPF in the 600x experiment results. This decrease was caused by the experiment placing more load on links, implying that Helix and CSPF could not address all of the introduced congestion. As such, Helix and CSPF performed slightly fewer path modifications. We can confirm this observation by looking at the recorded loss rate, which was higher for this experiment (4.6% versus 5.2% for Helix).

Discussion: We can attribute Helix's good forwarding stability to its TE algorithm, which minimises its solution search space. Helix's TE algorithm considered and modified only the source-destination paths that used an over-utilised port, allowing it to perform fewer path modifications. In contrast, CSPF and MCF recomputed all source-destination pairs in the network, exploring a more substantial search space. When recomputing all paths to re-

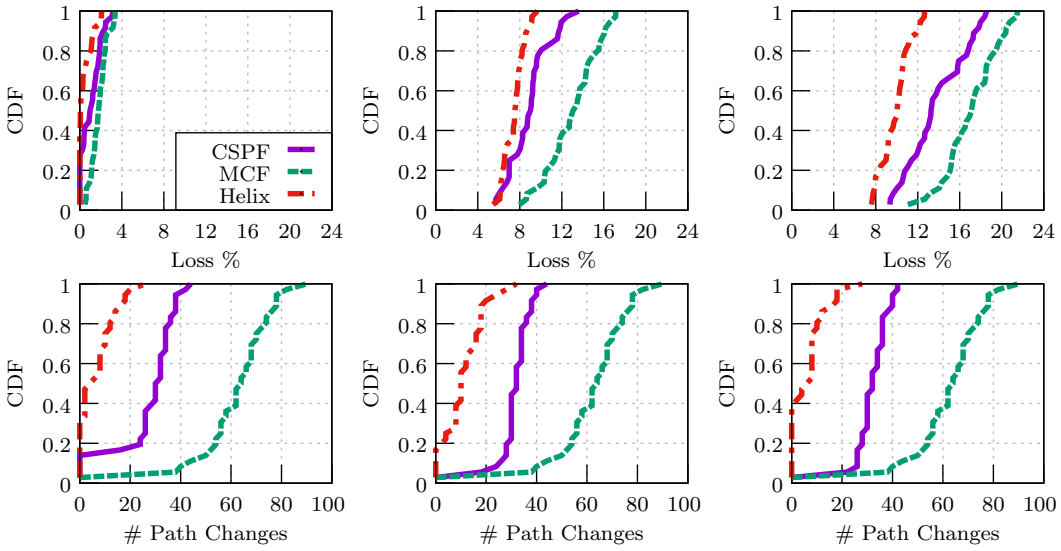
generate the routing scheme based on new traffic demands, there is a higher chance that the system will modify a candidate's path. Because Helix constrained its search space, most forwarding paths were unmodified, increasing the system's forwarding stability by reducing path change churn.

Furthermore, the smaller search space allowed Helix to find better solutions to congestion. The algorithm had to consider moving fewer flows, increasing the chances of finding a set of potential path changes to reduce usage on an over-utilised port (fewer variables in the optimisation problem). Helix's TE algorithm treated congestion on each link as an independent problem, accepting partial solutions to address the detected congestion. In contrast, CSPF tried to simultaneously resolve congestion on every link by recomputing the network's routing scheme. CSPF may have ignored potential path changes that reduced but did not fully resolve all detected congestion, as it could not fix every congested link in the network.

To better explain the reason for Helix's improved performance, we will use an example situation. Figure 8.4 illustrates the Hibernia global network. Suppose that link a of node A from Area 1 becomes congested. Because node A is at the edge of the network and is isolated, we assume that link a is only used by traffic originating from or going to node A. In other words, we assume that other paths generated by Helix will not use link a as a transit link to reach a destination. In this example, the algorithm's search space contains 108 candidates implying that Helix will modify at most 108 paths (worst case).

In contrast, because CSPF recomputes all candidates in the network, it can perform up to 3025 path changes (worst case). While CSPF may not need to modify all paths, it is likely to modify more paths than Helix due to the higher number of recomputations. Moreover, Helix is more likely to resolve congestion because it considers moving fewer flows than CSPF.

Let us assume that we encounter congestion on two links in the network, a from the example and b. In this situation, CSPF will modify the routing scheme (resolve congestion) only if it can address congestion on both links a



(a) 2.2x (b) 2.8x (c) 3.0x

Figure 8.3: Abilene Topology: CDF graphs showing congestion loss and the number of path changes per run. Results were consistent with §8.3.1.

and b. Suppose there is no viable solution to address congestion on b, but there is a solution for a. CSPF will not perform any path changes and discard the generated routing scheme in this situation. In contrast, Helix treats congestion on a and b as separate problems. Helix will still address congestion on a even if it could not find a solution for b.

8.3.2 Abilene Network Results

Figure 8.3 shows a CDF graph of congestion loss (top row) and the number of path changes (bottom row) observed on the Abilene topology.

TE Performance: Our results were consistent with §8.3.1. With a traffic multiplier of 2.2x (figure 8.3a), 36% of Helix's runs reported no loss compared to 28% for CSPF and none for MCF. Helix's maximum loss rate was 2.1% compared to 3.2% for CSPF (1.5x higher) and 3.4% for MCF (1.6x higher). Helix outperformed CSPF and MCF when increasing the traffic demands of the experiments. Helix's maximum loss rate was 9.7% when using a traffic multiplier of 2.8x (figure 8.3b) and 12.7% when using a multiplier of 3.0x (figure 8.3c). In contrast, CSPF had a maximum loss rate of 13.6% (1.4x

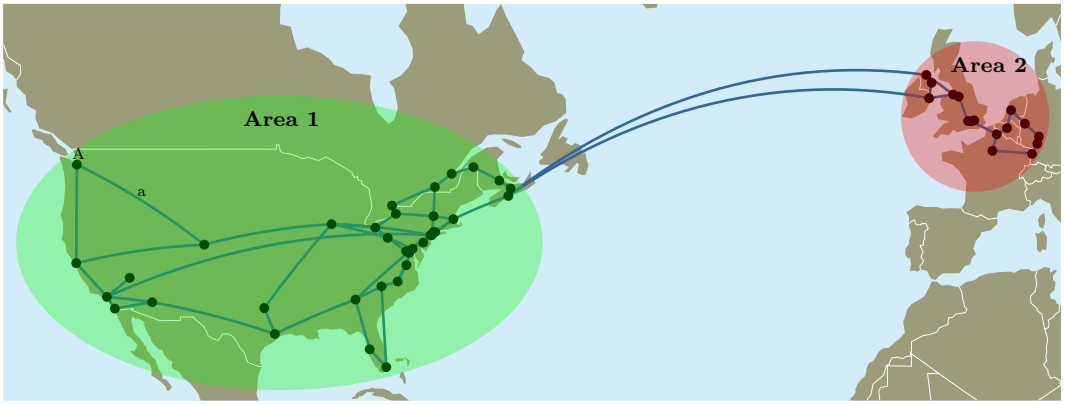


Figure 8.4: Hibernia Global topology graph showing the two Helix areas (green and blue circles) used for the multi-controller evaluation results. We divided the network into two areas, one for each continent.

higher) and 18.6% (1.5x higher), respectively.

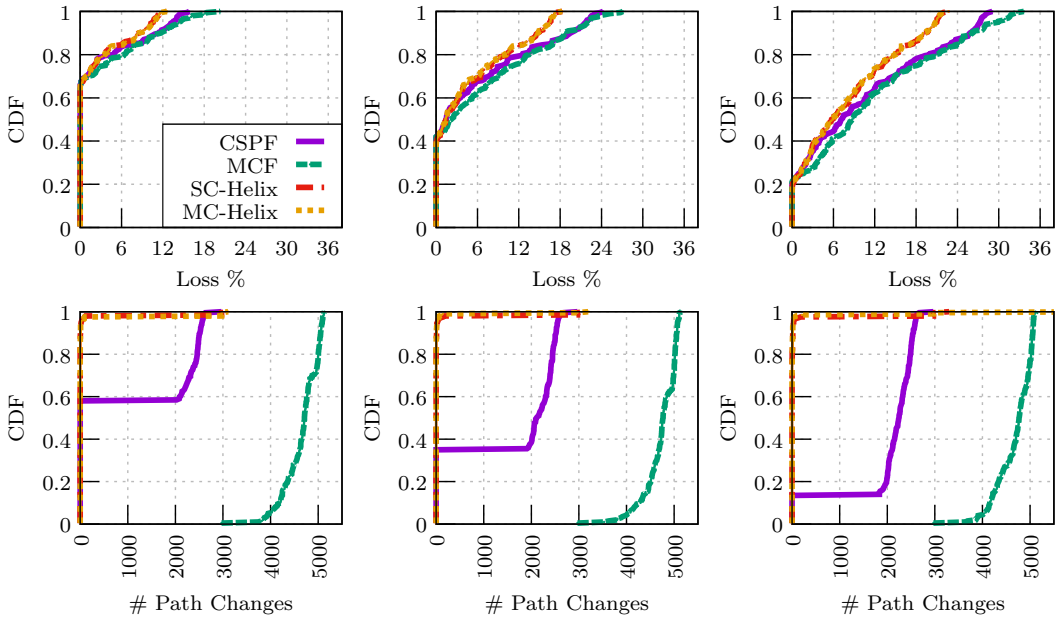
Forwarding Stability: Consistent with §8.3.1, Helix had the best forwarding stability of all evaluated systems. Overall, Helix modified fewer paths compared to CSPF and MCF. With a traffic multiplier of 2.2x, Helix performed a maximum of 26 path changes during a run compared to 44 maximum changes for CSPF (2x higher) and 90 for MCF (3.5x higher). Helix's maximum path change churn was 32 with a traffic multiplier of 2.8x and 28 with a multiplier of 3.0x. Similar to §8.3.1, we observed a slight drop in path change churn for Helix when using a higher traffic multiplier. While Helix still performed the least number of path modifications, the difference between Helix and the other systems decreased. We attribute this reduction in path change churn difference to the simplicity of the Abilene network. The Abilene topology contained 4x fewer path pairs than the AT&T MPLS network (144 versus 625), significantly reducing the search space for all algorithms. As a result, CSPF and MCF recomputed fewer candidate paths, decreasing the number of path changes. Moreover, because the search space was already constrained, Helix's TE algorithm could not reduce it further, diminishing the improvements Helix offered to stability.

8.3.3 Hibernia Global Network Results

Figure 8.5 presents a CDF graph of the results collected using the Hibernia Global Network. For our experiments, we collected two sets of Helix results. One of the experiments used a single controller to manage the complete topology (SC-Helix), and the second used a multi-controller deployment (MC-Helix). The Hibernia Global network is an example of an inter-continental WAN. From a latency standpoint, it is logical to divide the topology into areas based on the proximity of nodes. MC-Helix separated the topology into two areas deployed on each continent. Figure 8.4 shows how we divided the topology for the MC-Helix deployment. Area 1 (green circle) covered all US-based nodes, while Area 2 (red circle) covered the nodes deployed in Europe.

TE Performance: Consistent with §8.3.1 and §8.3.2, SC-Helix outperformed all tested algorithms. With a 300x traffic multiplier (figure 8.5a), SC-Helix's average congestion loss rate was 1.89% with a maximum rate of 12.53%. In contrast, CSPF's average loss rate was 2.38% (1.3x higher) with a maximum of 15.84% (1.3x higher). With a traffic multiplier of 350x (figure 8.5b) and 400x (figure 8.5c), SC-Helix addressed more of the introduced congestion compared to CSPF and MCF. SC-Helix's average loss rate was 4.47% and 7.72%, respectively. CSPF's average loss rate was 5.60% (1.3x higher) and 9.87% (1.3x higher), while MCF's had an average loss rate of 6.23% (1.4x higher) and 10.90% (1.4x higher).

Forwarding Stability: SC-Helix also had the best forwarding stability. With a 300x traffic multiplier, SC-Helix performed an average of 64 path changes during the entire experiment, while CSPF's average was 1010 (16x higher) and MCF's average was 4660 (74x higher). SC-Helix's maximum number of path changes performed during a run was slightly higher than CSPF (3078 versus 2970). The peak in SC-Helix's path change churn was recorded during the first run of the experiment when the system started forwarding traffic using unoptimised paths. As a result, SC-Helix recomputed all paths during the first run, causing the observed peak in path change churn. Despite



(a) 300x (b) 350x (c) 400x

Figure 8.5: Hibernia Network: CDF graphs showing results collected across 200 experiment runs. We used a single-controller (SC-Helix) and multi-controller (MC-Helix) Helix deployment. The trend in results was consistent with §8.3.1 and §8.3.2. Both Helix deployments had similar performance.

the observed peak in the metric value, SC-Helix still modified significantly fewer paths overall. SC-Helix did not perform any changes for 83% of all runs, modifying at most 20 paths for 95% of runs. In contrast, CSPF did not modify any paths during 58% of runs, performing between 2026 and 2562 path changes during the 59th and 95th percentiles. This trend in path change churn continued when increasing the experiment's traffic. SC-Helix performed an average of 64 (350x traffic multiplier results) and 80 path changes (400x traffic multiplier results) across all experiment runs. In contrast, CSPF's average path change churn was 1511 (24x higher) and 1979 (25x higher), respectively.

SC-Helix vs MC-Helix: In our experiments, SC-Helix and MC-Helix performed almost identically. With a traffic multiplier of 300x, MC-Helix's average loss rate was 1.91% compared to 1.90% recorded for SC-Helix, with a maximum of 12.59% versus 12.53%. This trend continued when comparing forwarding stability, with both deployments having a similar average path change churn metric value (63 for SC-Helix versus 77 for MC-Helix).

The trend observed for MC-Helix's performance continued across the 350x and 400x results. MC-Helix's average loss rates were 4.38% (versus 4.47%) and 7.64% (versus 7.72%), respectively. MC-Helix and SC-Helix also had similar average path change churn metric values. Despite this, MC-Helix's maximum number of path changes performed during a run was significantly higher than SC-Helix's for the 400x scale factor results (6312 versus 3204). While MC-Helix modifies more paths, a higher path change metric for MC-Helix may not indicate poorer forwarding stability. YATES computed the path change churn metric for MC-Helix as the total number of path modifications performed by all controllers. Suppose that SC-Helix changed two links of a candidate's path, interacting with two switches. Despite changing two links, YATES reported a single path change performed by the system. Let us assume that MC-Helix made the same overall path modification, with the two links spanning different areas. Because two controllers were involved in the path change process, YATES reported that the system performed two path modifications. As such, a higher path change churn metric value for MC-Helix did not necessarily correlate with less stable forwarding. Moreover, we conclude that both SC-Helix and MC-Helix had similar forwarding stability due to two factors: First, this difference in path change churn was only visible in one of the experiments. Second, the increase in churn occurred during the first run when Helix needed to optimise all paths.

Summary: Based on the collected results, we conclude that Helix performed better than the other evaluated systems. This trend in Helix's performance was consistent throughout our evaluation. This section also demonstrated that both single and multi-controller Helix deployments had similar performance. As such, we draw three main conclusions from this evaluation:

1. Using multiple Helix controllers does not negatively affect TE optimisation performance or forwarding stability.

2. MC-Helix did not perform any root controller TE optimisation requests.

As such, we can conclude that Helix's offloaded inter-area TE optimisation could address the majority of inter-area congestion locally.

3. TE optimisation performance seems to be minimally affected by the loss of centralised scope.

8.4 Limitations of Results

We evaluated TE optimisation performance and forwarding stability using YATES, a simulation framework. YATES did not simulate or consider how latency and state distribution affected TE performance. Instead, the framework opted to collect metrics that provided insight into how well a TE algorithm satisfied a set of traffic demands. Due to this limitation, we assume that YATES overestimated the performance of the evaluated systems.

YATES did not evaluate how convergence from one routing scheme to another affected the performance of offline TE systems. The time it takes to compute and deploy the new routing scheme delays how fast the systems respond to congestion, increasing the loss rate. In contrast, because Helix performed real-time traffic steering decisions, our collected results accounted for convergence time, offering a more realistic evaluation of Helix's performance.

Researchers have identified that prediction TE matrices often contain errors [4]. Our experiments collected results with prediction matrices that contained no estimation errors, implying that all offline TE algorithms were aware of the actual demand they would encounter, and as such made better TE decisions. Prediction errors can have a detrimental effect on TE performance. For example, underestimating traffic demands can cause congestion loss if a TE system does not reserve sufficient spare capacity on links to cope with extra traffic. On the other hand, overestimating the demand can cause the TE algorithm to reserve too much capacity, constraining links and potentially delaying forwarding by routing traffic on longer than necessary paths.

8.5 Conclusion

This chapter evaluated Helix's TE optimisation performance and presented experiments that provided insight into the effects of using multiple controllers on TE performance. This chapter also presented a concrete testing methodology to evaluate the TE optimisation performance and forwarding stability of reactive MCSDN systems.

Across all conducted experiments, we found that Helix performed better than the other evaluated systems and coped well with increases in traffic demands. Helix's average loss rate was up to 1.6x lower compared to CSPF while performing up to 29x fewer path changes. We attribute Helix's improved forwarding stability and performance to the system's TE algorithm, which reduced its search space and deployed partial solutions by treating congestion on links as independent problems.

Because Helix offloads inter-area TE to local controllers, we set out to answer two research questions in this evaluation: (1) "What are the effects of using multiple versus a single controller on TE performance?"; (2) "What are the effects of loss of centralised scope on TE optimisation performance?". We collected results using two Helix deployments on the Hibernia Global Network to answer these questions. Our experiments found that both single (SC-Helix) and multi (MC-Helix) controller Helix deployments had almost identical TE optimisation performance and forwarding stability. Based on our results, we drew two conclusions. First, because both Helix deployments had similar performance and forwarding stability, using multiple controllers to manage an SDN system had minimal effects on TE optimisation performance (first research question). Second, because MC-Helix did not perform any root controller-based TE optimisations, we conclude that loss of global visibility had minimal impact on Helix's TE performance because the local controllers resolved most inter-area congestion locally (second research question).

Chapter 9

Application to Other SDN Systems

This thesis identified and proposed solutions to address the challenges faced by Multi-Controller SDN (MCSDN) systems when deploying Traffic Engineering (TE) on Wide Area Networks (WANs). While this thesis evaluated the design choices using an OpenFlow-based system, we can apply this work to any existing and emerging SDN-based network management approaches as all SDN systems face similar challenges to those discussed in this thesis.

We can demonstrate the applicability of this work by discussing how Helix can be adapted and deployed in conjunction with other non-OpenFlow SDN approaches. We developed Helix as a modular system that enables easy usage, replacement of components, and partial deployment of features. Figure 9.1a presents a simplified overview of the Helix Local Controller (LC), while figure 9.1b presents an overview of the Helix Root Controller (RC). A core part of all Helix controllers is the module coordinator/event handler component. The module coordinator is an asynchronous event handler that registers to receive specific events raised by the underlying data plane framework. The data plane framework acts as a proxy between the data plane devices and the Helix local controller. The LC uses Ryu [75] as its data plane framework. Ryu implements the OpenFlow protocol (OFP) and handles all connectivity and

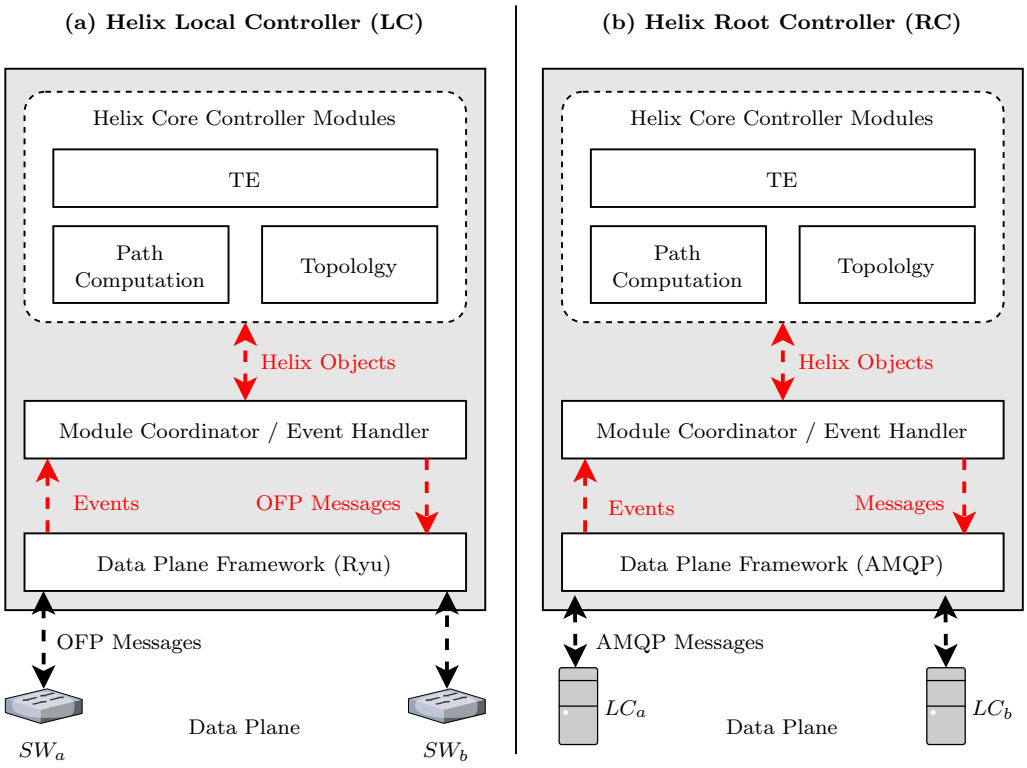


Figure 9.1: (a) Components that make up the Helix local controller (LC) and (b) root controller (RC). The Module coordinator component interacts with the core Helix modules and transforms Helix controller objects (e.g. paths) into messages. The module coordinator component is implemented as an asynchronous message handler that receives and forwards messages to and from the data plane framework component. The data plane framework module (e.g. Ryu for LC) interacts with the data plane devices.

communication with the physical data plane switches.

When a data plane device sends a message to the Helix controller, the data plane framework sends an event to the module coordinator, triggering a specific event handler method. The event handler will transform the event's information into a Helix object (e.g. link information) and forward it to a Helix core module, triggering a particular action (e.g. adding a new link to the topology). In essence, the Helix module coordinator acts as an interface between the Helix core modules and the data plane framework, performing any relevant conversion and mapping of objects. For example, when the Helix path computation module computes a path, the module coordinator will translate the Helix path object (encoded as a list of switch IDs) into a set of ports telling

each switch how to forward traffic.

9.1 Example: Interaction of LC Components

To better explain the interaction between the various components, we will discuss the ten steps that occurred when the LC performs TE optimisation. Before addressing congestion, the Helix LC will poll the network for statistics:

1. For every poll interval, the module coordinator will generate a statistics request OFP message for every active data plane device.

2. The OFP messages are sent to each switch through the data plane framework (Ryu).

3. Once a data plane device receives the statistics request, it will send a statistics reply OFP message back to the LC.

4. The OFP message is received by Ryu, which triggers the statistics reply event handler of the module coordinator.

5. The module coordinator extracts relevant usage information for each link from the OFP message.

6. The module coordinator will send the usage information to the topology module (which will update the link usage metrics of the topology object) and the TE module (which checks for congestion).

7. When the TE module detects congestion, it will execute Helix's TE optimisation algorithm (§5.3).

8. The TE module will produce a set of candidate Helix path objects which it sends to the module coordinator to install. A Helix path object is a list of switches used to forward traffic for a particular source-destination pair (e.g. $\{SRC; SW_a; SW_b; DEST\}$).

9. The module coordinator will convert the Helix path object into a list of OFP group modification messages that tell each switch how to forward traffic (e.g. SW_a should output traffic on port 2).

10. The OFP messages are sent to Ryu, which forwards the messages to the physical switches to apply the candidate path changes to the data plane.

9.2 Extending Helix

To use Helix on a non-OpenFlow data plane, we need to modify the event handler and data plane framework components of the LC. In the current LC example, the data plane framework component is Ryu, which facilitates connectivity and communication between the control plane (LC) and the data plane devices. In the current design, Ryu raises primitive events/messages that the module coordinator registers to receive via an event handler method. For example, the module coordinator will register to receive statistics reply events raised by the Ryu framework when a data plane device responds to a statistics request.

When the module coordinator interacts with a data plane device, it generates an OFP message object which it sends to the data plane framework. Ryu will encode the received OFP message object into an OpenFlow packet and forward it to the relevant data plane switch. The module coordinator generates message objects and directly interacts with events defined by the data plane framework. As a result, when we replace the data plane framework component, we also need to alter the module coordinator. The modified version of the module coordinator should register to receive events raised by the new framework and generate message objects that the framework can understand and encode. It is worth noting that when changing the data plane framework and coordinator components, we do not need to modify the core Helix modules as they are oblivious to the used module coordinator. The helix modules expose standardised interfaces that require Helix objects. The event handler

performs all translations to and from Helix objects, allowing us to keep the core modules unmodified.

For example, the Helix RC is a specialised version of the LC that uses a different data plane framework and module coordinator compared to the LC (shown in figure 9.1b). In the case of the Helix RC, the actual data plane devices are the local controllers. The RC uses an AMQP client (RabbitMQ) as its data plane framework. The RC's module coordinator registers to receive AMQP messages from the LCs and performs all deserialisation and transformation of the AMQP messages into Helix objects. The deserialised Helix objects are forwarded to the core Helix modules to trigger specific actions (e.g. add a link to the topology graph object). The module coordinator will also serialise any Helix objects into messages, which it sends to the LCs via the data plane framework. The RC provides an example of how we can modify Helix to support non-OpenFlow data planes.

When extending Helix, the module coordinator needs to provide three capabilities (requirements) to allow deploying Helix:

1. Helix needs a way to detect the current network topology. We can either:
   (a) Allow Helix to flood packets to deploy its topology discovery mechanisms
   (b) Provide topology information directly to Helix
2. Helix's path computation and TE optimisation modules compute routing schemes and modify paths. Helix needs to either:
   (a) Interact directly with the data plane to install and modify paths
   (b) Provide a routing scheme to an external system that handles forwarding
3. Helix's TE optimisation algorithm requires link usage information and candidate send rates to optimise the topology and detect congestion. We can either:

(a) Allow Helix to query the data plane for these metrics directly

(b) Provide this information to the system.

Based on requirements 1-3, we have two concurrent deployment pathways for Helix. Option (a) of all requirements outlines a *standalone deployment* pathway where Helix interacts with the data plane. The module coordinator for a standalone deployment will use the data plane framework to interact with the data plane devices. This deployment pathway characterises the current LC implementation that uses Ryu as the data plane framework.

In contrast, option (b) shows an *integrated deployment* pathway where Helix works alongside other systems or protocols that interact with the physical data plane devices. The module coordinator for an integrated deployment will not directly interact with the data plane devices. Instead, the module coordinator will provide information to a separate system that deploys Helix's paths on a network. The Helix RC provides an example of an integrated deployment. For the Helix RC, the module coordinator uses AMQP to communicate with an external protocol or system information such as paths (the Helix LC). The external system (i.e. LCs) processes and applies the received instructions from the RC to the physical data plane.

An integrated Helix deployment allows using Helix on a hybrid-SDN network (e.g. similar deployment as Espresso [99]) where traditional protocols (e.g. OSPF and BGP) handle routing.

For example, the integrated deployment pathway enables network operators to use Helix's TE optimisation on a Segment Routing (SR) network, where an SR system will handle packet forwarding and interaction with the data plane. Because the SR system forwards packets on segments to improve scalability, and not detract from this benefit, Helix should be oblivious to the links in the topology, keeping both systems separate. As such, the SR system will decide on the links of a particular segment, while Helix will select the segment that traffic will use. For this deployment, Helix should use an abstract topology graph similar to the one described in §5.6. Helix's abstract topology will contain

virtual links that map to a particular segment, enabling Helix to decide what segment traffic will use when performing TE optimisation.

When computing paths or performing TE optimisation, Helix will generate a routing scheme consisting of a list of segments. Helix should provide the routing scheme to the SR system, which assigns traffic to the segments based on the computed paths. The SR system will monitor the traffic on segments, providing Helix with the required metrics the system needs to perform TE optimisation. After Helix has resolved congestion in the network, the SR system should modify the active paths to deploy Helix's updated routing scheme. The integrated Helix deployment pathway is similar to a Path Computation Element (PCE) system [24].

The YATES Helix controller wrappers described in §8.1 offer a concrete example of an integrated Helix deployment. We built the controller wrappers to allow us to evaluate Helix's TE performance in YATES. The Helix controller wrappers are shallow copies of the controller that receive information from an external system (i.e. YATES), perform specific operations (e.g. TE optimisation), and return a routing scheme. The Helix wrappers provide the routing scheme to YATES, which uses the computed paths to forward traffic during an experiment.

Summary: When extending Helix to add support for non-OpenFlow data planes, we need to modify the module coordinator component of the Helix LC to work in conjunction with the new data plane type. For a standalone deployment, we need to use a data plane framework that enables Helix to send packets on the network to deploy its LLDP mechanism, query the data plane devices for statistics, and install multiple paths on each device. The integrated deployment simplifies the requirements for the module coordinator. In the integrated deployment, the module coordinator does not directly interact with the data plane and instead sends abstract information to an external system that processes the requests and applies the instructions to the data plane.

Chapter 10

Conclusion

10.1 Summary of Thesis

This thesis explored and proposed solutions to address the challenges faced by Multi-Controller SDN (MCSDN) systems when deploying TE optimisation on WANs. While MCSDN has received significant research attention, existing work presents performance, scalability, and resilience problems when deploying TE because they considered the four challenges faced by MCSDN systems in isolation (chapter 3). Moreover, the majority of work in the literature has either not considered TE (e.g. [78, 30, 51, 9]), explored deploying TE on SCSDN architectures (e.g. [36, 53]), or made design choices that make deploying TE difficult due to architectural constraints (e.g. [70, 100]).

This thesis presented Helix, a complete MCSDN system that offered solutions to address the challenges faced when deploying TE on WANs. Helix provides better scalability, performance and failure resilience compared to existing systems by using abstraction to reduce the amount of state shared between controllers, offloading inter-area operations to local controllers, and using computationally lightweight tasks.

A challenge that we faced when building Helix was that existing TE algorithms are computationally intensive and require network-wide state to operate. To meet Helix's design choices, we developed a CSPF-based TE algorithm

that requires less state to operate and supports offloading inter-area TE.

This thesis contained three chapters that evaluated Helix's data plane failure recovery, control plane failure recovery, and TE optimisation performance. These chapters also provided concrete testing methodologies and tools that enable researchers to assess MCSDN system performance.

Chapter 6 evaluated Helix's data plane failure recovery performance and compared Helix against restoration-based recovery. The evaluation found that Helix recovered from data plane failures up to 10x faster compared to restoration recovery. The performed experiments also demonstrated that latency inflated the completion time of reactive operations, decreasing their performance. Using proactive or offloading operations (design choice of Helix) enabled the system to tolerate high latency, which is vital for deploying TE on WANs.

Chapter 7 evaluated Helix's control plane failure recovery performance. The experiments identified that stage alignment and the used configuration attributes affected Helix's performance. Based on these two factors, this chapter presented two models that approximated Helix's failure recovery time. The chapter used the two models to compare Helix against a reassignment-based recovery approach (e.g. [9]). Helix's design choices enabled the system to recover up to 4.5x faster from instance failures than the proposed alternative.

Chapter 8 evaluated and compared Helix's TE optimisation performance and forwarding stability against several other TE systems. In our experiments, Helix's average loss rate was up to 1.6x lower compared to CSPF while performing up to 29x fewer path changes. The evaluation experiments demonstrated that Helix could scale and adapt well to an increase in traffic demand. We can attribute Helix's improved forwarding stability and performance to the system's TE algorithm, which reduced its search space and considered congestion on links as independent problems.

Based on the conducted evaluation, we conclude that the proposed design choices will benefit Helix when deploying the system on a WAN.

The presented evaluation experiments also gathered results towards an-

swering two research questions: (1) "What are the effects of loss of centralised scope on TE optimisation performance?". (2) "What are the effects of using multiple versus a single controller on system performance?". §10.2 will discuss the effects of using multiple controllers (i.e. loss of centralised scope) on system performance, answering these two research questions.

10.2 Effects of MCSDN on Performance

Helix offloads inter-area data plane failure recovery and TE optimisation from the root to the local controllers. Offloaded operations lose centralised scope, which can potentially cause Helix to make poor decisions, decreasing the system's performance. This section will discuss the effects of using multiple controllers on performance by considering the two inter-area operations that Helix offloads to the local controllers.

Data Plane Failure Recovery: When considering data plane failure recovery, loss of centralised scope will not affect Helix's failure resilience performance. Helix deals with data plane failures by pre-installing multiple paths onto the data plane to deploy protection-based recovery. Because switches make decisions independent from the controller, using multiple controllers will not influence a switch's ability to detect and recover from failures.

Despite this, questions arise about the effects of using multiple controllers on protection coverage and path optimality. The root controller (RC) computes inter-area paths. Because Helix abstracts areas and only exposes inter-area links to the RC, Helix may lose some protection coverage. We can improve protection coverage for inter-area paths by increasing the number of paths the RC computes (discussed in §5.7).

While using multiple controllers will have a minimal impact on protection coverage (which we can mitigate), Helix may route traffic across longer than necessary inter-area paths. Because Helix abstracts the RC's topology, the RC is unaware of an area's diameter or the number of nodes between its inter-area

links. Helix's abstraction mechanism may cause the RC to generate paths that transit areas with wider diameters, increasing the overall inter-area path length. We observed the effects of this increase in the TE evaluation experiments when comparing the single (SC-Helix) and multi-controller (MC-Helix) Helix deployment (§8.3.3). For our experiment results, MC-Helix's maximum computed path length was up to 2 nodes longer than SC-Helix's (22 versus 24). We can minimise the inter-area path lengths by providing the RC with an area's diameter (similar to Orion [30]). We leave exploring the effects of abstraction on inter-area path length as future work.

TE Optimisation: Helix offloads inter-area TE to the local controllers (LCs). Because the LCs are oblivious to congestion in other areas, the LCs can make sub-optimal inter-area TE decisions. §8.3.3 contained experiments that analysed the effects of using multiple controllers on performance. The experiments found that both SC-Helix and MC-Helix had identical TE performance and forwarding stability. We concluded that using multiple controllers did not negatively impact Helix's TE performance. Moreover, the experiments found that MC-Helix did not perform any RC TE optimisations, implying that the LCs could address most inter-area congestion locally.

Conclusion: For a set of representative topologies, we can conclude that loss of centralised scope has little effect on Helix's TE optimisation performance (first research question).

For the second research question, using multiple controllers will have little impact on Helix's failure resilience capabilities as the system uses protection-based recovery to recover from failures. Loss of global optimality may increase Helix's inter-area path lengths as the system abstracts the root controller's topology, implying the RC is unaware of an area's diameter. We can minimise inter-area path length by providing the RC with the average number of nodes between an area's egress ports. Using multiple versus a single Helix controller has virtually no impact on TE performance and forwarding stability. Although some operations are affected by the loss of centralised scope, we can deploy

mechanisms to mitigate these effects. Based on this factor, we conclude that using multiple controllers will have minimal on performance.

10.3 Future Work

While the loss of centralised scope is not detrimental to protection-based recovery performance, questions related to path optimality remain. For example, the Helix root controller is oblivious to an area's internal topology and generates inter-area paths based on how areas are inter-connected. The paths computed by the RC can lose global optimality by no longer offering the shortest route to a destination. Future research should further investigate the effects of loss of centralised scope on protection coverage and path optimality.

Chapter 9 discussed the applicability of this work to other non-OpenFlow SDN management approaches. A potential future research direction is to extend Helix to support other SDN systems, enabling us to confirm that the observations and benefits identified in this thesis for an OpenFlow-based deployment of Helix hold for other SDN management approaches.

Chapter 7 presented a control plane failure resilience emulation framework that enables network operators to evaluate an MCSDN system's failure recovery and startup performance. A limitation of this framework is its reliance on Mininet and emulation to conduct experiments. A future research direction for this work is to further investigate the effects of latency on control plane failure resilience. To conduct this research, we need to extend the framework to support evaluating MCSDN systems deployed in production environments. At its core, the emulation framework is a specialised packet capture and processing utility that uses different packets to detect events. We can add support for evaluating other systems by adapting the framework into a passive evaluation tool.

Bibliography

[1] Helix Authors. 2021. Helix and emulation framework implementation [Source Code]. (2021). https://github.com/wandsdn/helix.

[2] Helix Authors. 2021. YATES framework extension and Helix module [Source Code]. (2021). https://github.com/wandsdn/helix-te-evaluation.

[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. Conga: distributed congestion-aware load balancing for datacenters. *SIGCOMM Comput. Commun. Rev.*, 44, 4, 503–514. ISSN: 0146-4833. https://doi.org/10.1145/2740070.2626316.

[4] David Applegate and Edith Cohen. 2003. Making intra-domain routing robust to changing and uncertain traffic demands: understanding fundamental tradeoffs. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (SIGCOMM '03). Association for Computing Machinery, Karlsruhe, Germany, 313–324. ISBN: 1581137354. https://doi.org/10.1145/863955.863991.

[5] Mohamed Aslan and Ashraf Matrawy. 2018. A clustering-based consistency adaptation strategy for distributed sdn controllers. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, New York, NY, USA, 441–448. https://doi.org/10.1109/NETSOFT.2018.8460120.

[6] A. Atlas and Zinin A. 2008. Basic Specification for IP Fast Reroute: Loop-Free Alternates. RFC 8518. (September 2008). https://doi.org/10.17487/RFC5286.

[7] Daniel Awduche, Joe Malcolm, Johnson Agogbua, Mike O'Dell, and Jim McManus. 1999. Requirements for Traffic Engineering Over MPLS. Technical report 2702. RFC 2702. (September 1999). https://www.rfc-editor.org/info/rfc2702.

[8] Anuradha Banerjee and D. M. Akbar Hussain. 2020. Exprl: experience and prediction based load balancing strategy for multi-controller software defined networks. *International Journal of Information Technology*, 1–15. https://doi.org/10.1007/s41870-019-00408-5.

[9] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. 2014. 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, Chicago, Illinois, USA, 1–6. http://doi.acm.org/10.1145/2620728.2620744.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44, 3, (July 2014), 87–95. ISSN: 0146-4833. https://doi.org/10.1145/2656877.2656890.

[11] Antonio Capone, Carmelo Cascone, Alessandro Q. T. Nguyen, and Brunilde Sansò. 2015. Detour planning for fast and reliable failure recovery in sdn with openstate. In *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*. IEEE, New York, NY, USA, 25–32. https://doi.org/10.1109/DRCN.2015.7148981.

[12] Carmelo Cascone, Luca Pollini, Davide Sanvito, Antonio Capone, and Brunilde Sansó. 2016. Spider: fault resilient sdn pipeline with recovery delay guarantees. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE, New York, NY, USA, 296–302. https://doi.org/10.1109/NETSOFT.2016.7502425.

[13] Marco Cello, Yang Xu, Anwar Walid, Gordon Wilfong, H Jonathan Chao, and Mario Marchese. 2017. Balcon: a distributed elastic sdn control via efficient switch migration. In *2017 IEEE International Conference on Cloud Engineering (IC2E)* (IEEE IC2E 2017). IEEE, Vancouver, BC, Canada, 40–50. https://doi.org/10.1109/IC2E.2017.33.

[14] Kenjiro Cho, Kensuke Fukuda, Hiroshi Esaki, and Akira Kato. 2008. Observing slow crustal movement in residential user traffic. In *Proceedings of the 2008 ACM CoNEXT Conference* (CoNEXT '08) Article 12. Association for Computing Machinery, Madrid, Spain, 12 pages. https://doi.org/10.1145/1544012.1544024.

[15] Gerald Combs. 1998. Tshark packet capture and protocol analyzer utility. v2.6.10. (1998). https://www.wireshark.org/docs/man-pages/tshark.html.

[16] Maja Curic, Zoran Despotovic, Artur Hecker, and Georg Carle. 2019. Fitsdn: flexible integrated transactional sdn. In *2019 IEEE 44th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*. IEEE, New York, NY, USA, 1–9. https://doi.org/10.1109/LCNSymposium47956.2019.9000677.

[17] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. Devoflow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* (SIGCOMM '11). Association for Computing Machinery, Toronto, Ontario, Canada, 254–265. https://doi.org/10.1145/2018436.2018466.

[18] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik*, 1, 1, 269–271. https://doi.org/10.1007/BF01386390.

[19] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. 2013. Towards an elastic distributed sdn controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (HotSDN '13). Association for Computing Machinery, Hong Kong, China, 7–12. https://doi.org/10.1145/2491185.2491193.

[20] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Rao Kompella. 2014. Elasticon; an elastic distributed sdn controller. In *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (IEEE ANCS 2014). IEEE, Marina del Rey, CA, USA, 17–27. https://ieeexplore.ieee.org/abstract/document/7856398.

[21] Phan The Duy, Huynh Phu Qui, Van-Hau Pham, et al. 2019. Aloba: a mechanism of adaptive load balancing and failure recovery in distributed sdn controllers. In *2019 IEEE 19th International Conference on Communication Technology (ICCT)*. IEEE, Xi'an, China, China, 1322–1326. https://doi.org/10.1109/ICCT46805.2019.8947182.

[22] 2021. Esnet: energy science research network. Dashboard. (2021). https://my.es.net/.

[23] Luyuan Fang, Fabio Chiussi, Deepak Bansal, Vijay Gill, Tony Lin, Jeff Cox, and Gary Ratterree. 2015. Hierarchical sdn for the hyper-scale, hyper-elastic data center and cloud. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (SOSR '15) Article 7. Association for Computing Machinery, Santa Clara, California, 13 pages. ISBN: 9781450334518. https://doi.org/10.1145/2774993.2775009.

[24] A. Farrel, J.-P. Vasseur, and J. Ash. 2006. A Path Computation Element (PCE)-Based Architecture. RFC 4655. (August 2006). https://www.rfc-editor.org/info/rfc4655.

[25] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. 2021. Orion: google's software-defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Boston, MA, (April 2021), 83–98. https://www.usenix.org/conference/nsdi21/presentation/ferguson.

[26] Giuseppe Fioccola, Alessandro Capello, Mauro Cociglio, Luca Castaldelli, Mach Chen, Lianshu Zheng, Greg Mirsky, and Tal Mizrahi. 2018. Alternate-Marking Method for Passive and Hybrid Performance Monitoring. RFC 8321. (January 2018). https://doi.org/10.17487/RFC8321.

[27] Marina Fomenkov, Ken Keys, David Moore, and K Claffy. 2004. Longitudinal study of internet traffic in 1998-2003. In *Proceedings of the Winter International Synposium on Information and Communication Technologies* (WISICT '04). Trinity College Dublin, Cancun, Mexico, 1–6. https://dl.acm.org/doi/abs/10.5555/984720.984747.

[28] Lester Randolph Ford Jr. and Delbert R. Fulkerson. 1958. A suggested computation for maximal multi-commodity network flows. *Management Science*, 5, 1, 97–101. https://doi.org/10.1287/mnsc.1040.026.

[29] Aleš Friedl, Sven Ubik, Alexandros Kapravelos, Michalis Polychronakis, and Evangelos P. Markatos. 2009. Realistic passive packet loss measurement for high-speed networks. In *International Workshop on Traffic Monitoring and Analysis*. Springer, Berlin, Heidelberg, 1–7. https://doi.org/10.1007/978-3-642-01645-5_1.

[30] Yonghong Fu, Jun Bi, Ze Chen, Kai Gao, Baobao Zhang, Guangxu Chen, and Jianping Wu. 2015. A hybrid hierarchical control plane for flow-based large-scale software-defined networks. *IEEE Transactions on Network and Service Management*, 12, 2, (June 2015), 117–131. https://doi.org/10.1109/TNSM.2015.2434612.

[31] R. Gouareb, V. Friderikos, A. H. Aghvami, and M. Tatipamula. 2019. Joint reactive and proactive sdn controller assignment for load balancing. In *2019 IEEE Globecom Workshops (GC Wkshps)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/GCWkshps45667.2019.9024555.

[32] Soheil Hassas Yeganeh and Yashar Ganjali. 2012. 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, Helsinki, Finland, 19–24. https://doi.org/10.1145/2342441.2342446.

[33] Brandon Heller, Rob Sherwood, and Nick McKeown. 2012. The controller placement problem. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (HotSDN '12). Association for Computing Machinery, Helsinki, Finland, 7–12. ISBN: 9781450314770. https://doi.org/10.1145/2342441.2342444.

[34] Stephen Hemminger. 2005. Network emulation with netem. In *Linux conf au*. Volume 5. Citeseer, 2005. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.1687&rep=rep1&type=pdf.

[35] Chia-Chen Ho, Kuochen Wang, and Yi-Huai Hsu. 2016. A fast consensus algorithm for multiple controllers in software-defined networks. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*. IEEE, New York, NY, USA, 112–116. https://doi.org/10.1109/ICACT.2016.7423294.

[36] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (SIGCOMM '13). Association for Computing Machinery, Hong Kong, China, 15–26. ISBN: 9781450320566. https://doi.org/10.1145/2486001.2486012.

[37] C. Hopps. 2000. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992. (November 2000). https://www.rfc-editor.org/info/rfc2992.

[38] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: a programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 701–721. ISBN: 978-1-939133-13-7. https://www.usenix.org/conference/nsdi20/presentation/hsu.

[39] Kuo-Feng Hsu, Praveen Tammana, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Adaptive weighted traffic splitting in programmable data planes. In *Proceedings of the Symposium on SDN Research* (SOSR '20). Association for Computing Machinery, San Jose, CA, USA, 103–109. ISBN: 9781450371018. https://doi.org/10.1145/3373360.3380841.

[40] Jie Hu, Chuang Lin, Xiangyang Li, and Jiwei Huang. 2014. Scalability of control planes for software defined networks: modeling and evaluation. In *2014 IEEE 22nd International Symposium of Quality of Service (IWQoS)*. IEEE, New York, NY, USA, 147–152. https://doi.org/10.1109/IWQoS.2014.6914314.

[41] Jingyu Hua, Laiping Zhao, Suohao Zhang, Yangyang Liu, Xin Ge, and Sheng Zhong. 2018. Topology-preserving traffic engineering for hier-

archical multi-domain sdn. *Computer Networks*, 140, 62–77. https://doi.org/10.1016/j.comnet.2018.04.011.

[42] IEEE Computer Society. 2005. IEEE Standard for Local and metropolitan area networks – Station and Media Access Control Connectivity Discovery. New York, NY, USA, (May 2005). https://doi.org/10.1109/IEEESTD.2005.96285.

[43] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (SIGCOMM '13). Association for Computing Machinery, Hong Kong, China, 3–14. https://doi.org/10.1145/2486001.2486019.

[44] Yury Jiménez, Cristina Cervelló-Pastor, and Aurelio J. García. 2014. On the controller placement for designing a distributed sdn control layer. In *2014 IFIP Networking Conference* (IFIP Networking 2014). IEEE, Trondheim, Norway, 1–9. https://doi.org/10.1109/IFIPNetworking.2014.6857117.

[45] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research* (SOSR '16) Article 10. Association for Computing Machinery, Santa Clara, CA, USA, 12 pages. ISBN: 9781450342117. https://doi.org/10.1145/2890955.2890968.

[46] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. 2015. Ravana: controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (SOSR '15). Association for Computing

Machinery, Santa Clara, California, 4:1–4:12. http://doi.acm.org/10.1145/2774993.2774996.

[47] Dave Katz, Kireeti Kompella, and Derek Yeung. 2003. Traffic Engineering (TE) Extensions to OSPF Version 2. RFC 3630. (October 2003). https://www.rfc-editor.org/info/rfc3630.

[48] Dave Katz and David Ward. 2010. Bidirectional Forwarding Detection (BFD). RFC 5880. (June 2010). https://www.rfc-editor.org/info/rfc5880.

[49] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29, 9, 1765–1775. https://doi.org/10.1109/JSAC.2011.111002.

[50] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. 2010. Onix: a distributed control platform for large-scale production networks. In *9th USENIX OSDI Symposium on Operating Systems Design and Implementation*. Volume 10. USENIX Association, Santa Clara, CA, (October 2010), 1–6. https://www.usenix.org/conference/osdi10/onix-distributed-control-platform-large-scale-production-networks.

[51] Anand Krishnamurthy, Shoban P. Chandrabose, and Aaron Gember-Jacobson. 2014. Pratyaastha: an efficient elastic distributed sdn control plane. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (HotSDN '14). Association for Computing Machinery, Chicago, Illinois, USA, 133–138. ISBN: 9781450329897. https://doi.org/10.1145/2620728.2620748.

[52] Praveen Kumar, Chris Yu, Yang Yuan, Nate Foster, Robert Kleinberg, and Robert Soulé. 2018. Yates: rapid prototyping for traffic engineering systems. In *Proceedings of the Symposium on SDN Research* (SOSR '18)

Article 11. ACM, Los Angeles, CA, USA, 11:1–11:7. http://doi.acm.org/10.1145/3185467.3185498.

[53] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. 2018. Semi-oblivious traffic engineering: the road not taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 157–170. https://www.usenix.org/conference/nsdi18/presentation/kumar.

[54] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44, 2, (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922.

[55] 2019. *The byzantine generals problem. Concurrency: The Works of Leslie Lamport*. Association for Computing Machinery, New York, NY, USA, 203–226. ISBN: 9781450372701. https://doi.org/10.1145/3335772.3335936.

[56] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (Hotnets-IX) Article 19. Association for Computing Machinery, Monterey, California, 6 pages. https://doi.org/10.1145/1868447.1868466.

[57] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. 2012. Logically centralized? state distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (HotSDN '12). Association for Computing Machinery, Helsinki, Finland, 1–6. https://doi.org/10.1145/2342441.2342443.

[58] Mingzheng Li, Xiaodong Wang, Haojie Tong, Tong Liu, and Ye Tian. 2019. Sparc: towards a scalable distributed control plane architecture

for protocol-oblivious sdn networks. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, New York, NY, USA, 1–9. https://doi.org/10.1109/ICCCN.2019.8846931.

[59] Ying-Dar Lin, Hung-Yi Teng, Chia-Rong Hsu, Chun-Chieh Liao, and Yuan-Cheng Lai. 2016. Fast failover and switchover for link failures and congestion in software defined networks. In *2016 IEEE International Conference on Communications (ICC)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/ICC.2016.7510886.

[60] Shaoteng Liu, Rebecca Steinert, and Dejan Kostic. 2018. Flexible distributed control plane deployment. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Taipei, Taiwan, 1–7. https://doi.org/10.1109/NOMS.2018.8406150.

[61] Alaitz Mendiola, Jasone Astorga, Eduardo Jacob, and Marivi Higuero. 2017. A survey on the contributions of software-defined networking to traffic engineering. *IEEE Communications Surveys & Tutorials*, 19, 2, 918–953. https://doi.org/10.1109/COMST.2016.2633579.

[62] Lucas F. Müller, Rodrigo R. Oliveira, Marcelo C. Luizelli, Luciano P. Gaspary, and Marinho P. Barcellos. 2014. Survivor: an enhanced controller placement strategy for improving sdn survivability. In *2014 IEEE Global Communications Conference*. IEEE, New York, NY, USA, 1909–1915. https://doi.org/10.1109/GLOCOM.2014.7037087.

[63] Ammar Muthanna, Abdelhamied A. Ateya, Maria Makolkina, Anastasia Vybornova, Ekaterina Markova, Alexander Gogol, and Andrey Koucheryavy. 2018. Sdn multi-controller networks with load balanced. In *Proceedings of the 2nd International Conference on Future Networks and Distributed Systems* (ICFNDS '18) Article Article 57. Association for Computing Machinery, Amman, Jordan, 6 pages. https://doi.org/10.1145/3231053.3231124.

[64] Ben Niven-Jenkins, Deborah Brungard, Malcolm Betts, Nurit Sprecher, and Satoshi Ueno. 2009. Requirements of an MPLS Transport Profile. RFC 5654. (September 2009). https://doi.org/10.17487/RFC5654.

[65] John O'Hara. 2007. Toward a commodity enterprise middleware: can amqp enable a new era in messaging middleware? a look inside standards-based messaging with amqp. *Queue*, 5, 4, (May 2007), 48–55. https://doi.org/10.1145/1255421.1255424.

[66] Yustus Eko Oktian, SangGon Lee, HoonJae Lee, and JunHuy Lam. 2017. Distributed sdn controller system: a survey on design choice. *computer networks*, 121, 100–111. https://doi.org/10.1016/j.comnet.2017.04.038.

[67] Robert Olsson. 2005. Pktgen the linux packet generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*. Volume 2, 11–24.

[68] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, (June 2014), 305–319. ISBN: 978-1-931971-10-2. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[69] Hamid Ould-Brahim, Don Fedyk, and Yakov Rekhter. 2009. BGP Traffic Engineering Attribute. RFC 5543. (May 2009). https://www.rfc-editor.org/info/rfc5543.

[70] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. 2017. SCL: simplifying distributed SDN control planes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, (March 2017), 329–345. ISBN: 978-1-931971-37-9. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-aurojit-scl.

[71] Kévin Phemius, Mathieu Bouet, and Jérémie Leguay. 2014. Disco: distributed multi-domain sdn controllers. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, New York, NY, USA, 1–4. https://doi.org/10.1109/NOMS.2014.6838330.

[72] Kun Qiu, Siyuan Huang, Qiongwen Xu, Jin Zhao, Xin Wang, and Stefano Secci. 2017. Paracon: a parallel control plane for scaling up path computation in sdn. *IEEE Transactions on Network and Service Management*, 14, 4, 978–990. https://doi.org/10.1109/TNSM.2017.2761777.

[73] Harald Räcke. 2008. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing* (STOC '08). Association for Computing Machinery, Victoria, British Columbia, Canada, 255–264. ISBN: 9781605580470. https://doi.org/10.1145/1374376.1374415.

[74] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. 2002. Experience in measuring backbone traffic variability: models, metrics, measurements and meaning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment* (IMW '02). Association for Computing Machinery, Marseille, France, 91–92. https://doi.org/10.1145/637201.637213.

[75] RYU Authors. 2011. Ryu sdn framework. v4.25. (2011). https://github.com/faucetsdn/ryu.

[76] Ermin Sakic, Mirza Avdic, Amaury Van Bemten, and Wolfgang Kellerer. 2020. Automated bootstrapping of a fault-resilient in-band control plane. In *Proceedings of the Symposium on SDN Research* (SOSR '20). Association for Computing Machinery, San Jose, CA, USA, 1–13. https://doi.org/10.1145/3373360.3380829.

[77] Ermin Sakic and Wolfgang Kellerer. 2018. Impact of adaptive consistency on distributed sdn applications: an empirical study. *IEEE Jour-*

nal on Selected Areas in Communications, 36, 12, 2702–2715. https://doi.org/10.1109/JSAC.2018.2871309.

[78] Mateus A. S. Santos, Bruno A. A. Nunes, Katia Obraczka, Thierry Turletti, Bruno T. de Oliveira, and Cintia B. Margi. 2014. Decentralizing sdn's control plane. In *39th Annual IEEE Conference on Local Computer Networks* (IEEE LCN 2014). IEEE, Edmonton, AB, Canada, 402–405. https://doi.org/10.1109/LCN.2014.6925802.

[79] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. 2016. In-band synchronization for distributed sdn control planes. *SIGCOMM Comput. Commun. Rev.*, 46, 1, (January 2016), 37–43. ISSN: 0146-4833. https://doi.org/10.1145/2875951.2875957.

[80] Hakan Selvi, Gürkan Gür, and Fatih Alagöz. 2016. Cooperative load balancing for hierarchical sdn controllers. In *2016 IEEE 17th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, New York, NY, USA, 100–105. https://doi.org/10.1109/HPSR.2016.7525646.

[81] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. 2020. Turboepc: leveraging dataplane programmability to accelerate the mobile packet core. In *Proceedings of the Symposium on SDN Research* (SOSR '20). Association for Computing Machinery, San Jose, CA, USA, 83–95. ISBN: 9781450371018. https://doi.org/10.1145/3373360.3380839.

[82] Rinku Shah, Mythili Vutukuru, and Purushottam Kulkarni. 2017. Devolve-redeem: hierarchical sdn controllers with adaptive offloading. In *Proceedings of the First Asia-Pacific Workshop on Networking* (APNet '17). Association for Computing Machinery, Hong Kong, China, 8–14. ISBN: 9781450352444. https://doi.org/10.1145/3106989.3107001.

[83] Rinku Shah, Mythili Vutukuru, and Purushottam Kulkarni. 2018. Cuttlefish: hierarchical sdn controllers with adaptive offload. In *2018 IEEE*

26th International Conference on Network Protocols (ICNP). IEEE, New York, NY, USA, 198–208. https://doi.org/10.1109/ICNP.2018.00029.

[84] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. 2013. Fast failure recovery for in-band openflow networks. In *2013 9th International Conference on the Design of Reliable Communication Networks (DRCN)*. IEEE, New York, NY, USA, 52–59. https://ieeexplore.ieee.org/document/6529882.

[85] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. 2013. Openflow: meeting carrier-grade recovery requirements. *Comput. Commun.*, 36, 6, (March 2013), 656–665. https://doi.org/10.1016/j.comcom.2012.09.011.

[86] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. 2014. The internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks* (HotNets-XIII). Association for Computing Machinery, Los Angeles, CA, USA, 1–7. ISBN: 9781450332569. https://doi.org/10.1145/2670518.2673876.

[87] Cheenu Srinivasan, Arun Viswanathan, and Thomas Nadeau. 2004. Multiprotocol Label Switching (MPLS) Traffic Engineering (TE) Management Information Base (MIB). RFC 3812. (June 2004). https://www.rfc-editor.org/info/rfc3812.

[88] Dimitri Staessens, Sachin Sharma, Didier Colle, Mario Pickavet, and Piet Demeester. 2011. Software defined networking: meeting carrier grade requirements. In *2011 18th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/LANMAN.2011.6076935.

[89] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M Frans Kaashoek, and Robert Tappan Morris. 2009. Flexible, wide-area storage for distributed systems with wheelfs. In *6th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 09). Volume 9. USENIX Association, Philadelphia, PA, 43–58. https://www.usenix.org/legacy/events/nsdi09/tech/full_papers/stribling/stribling.pdf.

[90] Mohammed Amine Togou, Djabir Abdeldjalil Chekired, Lyes Khoukhi, and Gabriel-Miro Muntean. 2019. A hierarchical distributed control plane for path computation scalability in large scale software-defined networks. *IEEE Transactions on Network and Service Management*, 16, 3, 1019–1031. https://doi.org/10.1109/TNSM.2019.2913771.

[91] Amin Tootoonchian and Yashar Ganjali. 2010. Hyperflow: a distributed control plane for openflow. In *2010 Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN '10)*. Volume 3. USENIX Association, Philadelphia, PA. https://www.usenix.org/legacy/event/inmwren10/tech/full_papers/Tootoonchian.pdf.

[92] Ramona Trestian, Gabriel-Miro Muntean, and Kostas Katrinis. 2013. Micetrap: scalable traffic engineering of datacenter mice flows using openflow. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, New York, NY, USA, 904–907. https://ieeexplore.ieee.org/abstract/document/6573108.

[93] Martino Trevisan, Danilo Giordano, Idilio Drago, Maurizio Matteo Munafò, and Marco Mellia. 2020. Five years at the edge: watching internet from the isp network. *IEEE/ACM Transactions on Networking*, 28, 2, 561–574. https://doi.org/10.1109/TNET.2020.2967588.

[94] Leslie G. Valiant. 1982. A scheme for fast parallel communication. *SIAM journal on computing*, 11, 2, 350–361. https://doi.org/10.1137/0211027.

[95] Niels L. M. Van Adrichem, Benjamin J. Van Asten, and Fernando A. Kuipers. 2014. Fast recovery in software-defined networks. In *2014*

Third European Workshop on Software Defined Networks. IEEE, New York, NY, USA, 61–66. https://doi.org/10.1109/EWSDN.2014.13.

[96] Yang Xu, Marco Cello, I-Chih Wang, Anwar Walid, Gordon Wilfong, Charles H.-P. Wen, Mario Marchese, and H. Jonathan Chao. 2019. Dynamic switch migration in distributed software-defined networks to achieve controller load balance. *IEEE Journal on Selected Areas in Communications*, 37, 3, 515–529. https://doi.org/10.1109/JSAC.2019.2894237.

[97] Kongzhe Yang, Daoxing Guo, Bangning Zhang, and Bing Zhao. 2019. Multi-controller placement for load balancing in sdwan. *IEEE Access*, 7, 167278–167289. https://doi.org/10.1109/ACCESS.2019.2953723.

[98] Guang Yao, Jun Bi, and Luyi Guo. 2013. On the cascading failures of multi-controllers in software defined networks. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE, New York, NY, USA, 1–2. https://doi.org/10.1109/ICNP.2013.6733624.

[99] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2017. Taking the edge off with espresso: scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (SIGCOMM '17). Association for Computing Machinery, Los Angeles, CA, USA, 432–445. https://doi.org/10.1145/3098822.3098854.

[100] Soheil Hassas Yeganeh and Yashar Ganjali. 2016. Beehive: simple distributed programming in software-defined networks. In *Proceedings of the Symposium on SDN Research* (SOSR '16) Article 4. Association

for Computing Machinery, Santa Clara, CA, USA, 12 pages. ISBN: 9781450342117. https://doi.org/10.1145/2890955.2890958.

[101] Jinke Yu, Ying Wang, Keke Pei, Shujuan Zhang, and Jiacong Li. 2016. A load balancing mechanism for multiple sdn controllers based on load informing strategy. In *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, New York, NY, USA, 1–4. https://doi.org/10.1109/APNOMS.2016.7737283.

[102] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. 2010. Scalable flow-based networking with difane. In *Proceedings of the ACM SIGCOMM 2010 Conference* (SIGCOMM '10). Association for Computing Machinery, New Delhi, India, 351–362. ISBN: 9781450302012. https://doi.org/10.1145/1851182.1851224.