Helix: Traffic Engineering for Multi-Controller SDN

Nicu Florin Zaicu
University of Waikato
Hamilton, New Zealand
nfz@wand.net.nz

Matthew Luckie
University of Waikato
Hamilton, New Zealand
mjl@wand.net.nz

Richard Nelson
University of Waikato
Hamilton, New Zealand
richardn@waikato.ac.nz

Marinho Barcellos
University of Waikato
Hamilton, New Zealand
marinho.barcellos@waikato.ac.nz

ABSTRACT

Deploying traffic engineering (TE) in the context of multi-controller SDN (MCSDN) or on WANs is challenging due to state and consistency requirements. For example, using strong consistency to ensure that information is always up-to-date introduces significant performance overheads. However, using eventual consistency to reduce synchronisation time comes at the expense of using outdated information to make decisions. We design and implement Helix, an MCSDN system that supports deployment on WANs. Helix offloads operations closer to the data plane and minimises shared state between devices, allowing it to tolerate high latency and mitigate state consistency concerns. We develop a lightweight TE algorithm that requires minimal state, making it suitable for use with Helix. Our simulation results show that Helix reduces congestion loss by up to 1.6x and performs 12x fewer path changes compared to CSPF.

CCS CONCEPTS

• **Networks → Traffic engineering algorithms**; *Network reliability*; *Network performance evaluation*.

KEYWORDS

SDN, Multi-Controller, TE, Failure Resilience, Protection Recovery

ACM Reference Format:
Nicu Florin Zaicu, Matthew Luckie, Richard Nelson, and Marinho Barcellos. 2021. Helix: Traffic Engineering for Multi-Controller SDN. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), October 11–12, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3482898.3483354

1 INTRODUCTION

Traffic Engineering (TE) is a critical task that provides efficient forwarding. Current TE optimisation methods are complex and need up-to-date network-wide information to operate. As such, deploying TE in the context of Multi-Controller SDN (MCSDN) is challenging due to state and consistency requirements.

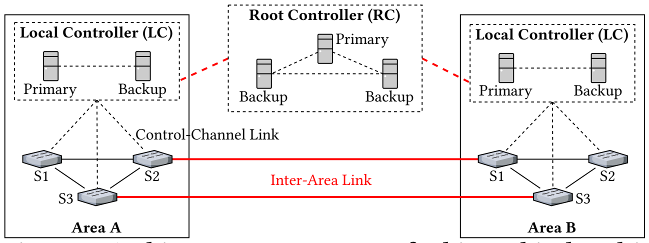

Figure 1: Architecture components of a hierarchical multi-controller SDN system.

MCSDN addresses scalability concerns by distributing the control plane across multiple devices. MCSDN controllers share network state using one of two synchronisation approaches. First, *strong consistency* [10, 21] guarantees that information is always up-to-date, which ensures that controllers always perform forwarding decisions using the current state. Strong consistency introduces performance overheads [6, 40], which delays TE optimisation and thus decreases traffic forwarding performance. Second, *eventual consistency* relaxes the guarantees of the model to improve performance [40]. Eventual consistency removes these performance overheads at the expense of using outdated information to make TE decisions, which can increase path change churn, cause policy violation, and packet loss. Issues of performance and consistency are further complicated when deploying on a Wide Area Network (WAN) because these networks contain higher inter-device latency.

In this paper, we discuss the design and implementation of Helix, a hierarchical MCSDN system that combines various techniques to address performance, robustness and consistency concerns of deploying TE on WANs. Helix offloads critical operations (such as inter-area TE and failure recovery) closer to the data plane, which allows it to tolerate high inter-device latency and improve its robustness to controller failures by removing dependencies between devices. Helix requires and shares minimal state between controllers, improving update propagation time and reducing state consistency concerns. Like other hierarchical MCSDN systems such as Espresso [54] and TurboEPC [45], Helix defines two controller types, Local Controller (LC) and Root Controller (RC), illustrated in figure 1. LCs are deployed within an area and interact with switches, while RCs connect to all LCs and coordinate inter-area operations.

Our contributions are as follows: First, we design and implement Helix, a hierarchical MCSDN system that combines various techniques to address performance, robustness and consistency concerns of deploying TE on WANs. Second, we develop a lightweight TE algorithm that requires minimal-state and is suitable for use with Helix. Third, we design and implement an emulation framework that allows comparing failure resilience performance of MCSDN systems. We use YATES [32], a simulation framework, to evaluate and compare Helix's TE algorithm performance against other algorithms. We find that Helix reduces congestion loss by up to 1.6x while performing up to 12x and 29x fewer path changes compared to CSPF and MCF TE algorithms. We make all our source code for Helix and the emulation framework available at [1].

2 RELATED WORK

Control Plane Load Balancers: Balcon [52], Elasticon [12], and others [5, 8, 13, 18, 36, 44, 56] resolve control plane load imbalances by migrating switches between controllers. Despite resolving both CPU and request imbalances, these systems increase inter-controller communication, delay forwarding state modification during switch migration, or increase control plane latency by moving switches to distant controllers. Helix addresses CPU and request load imbalances by reducing the computational intensity of controller operations (e.g. TE) and using offloaded operations.

Offloading Controller Operations: DIFANE [57], Kandoo [19], TurboEPC [45], and others [9, 46, 47] explored offloading operations closer to the data plane to improve scalability and performance. To the best of our knowledge, previous work has not explored offloading of inter-area TE in the context of MCSDN.

In-Band Load Balancers: Conga [2], Hula [28], Contra [23], and Dash [24], deploy specialised P4 [7] programs onto switches to propagate metrics hop-by-hop and resolve congestion by splitting traffic on multiple paths. These systems require hierarchical data plane structures and low latency making them unsuitable for WANs.

TE Systems: The majority of work in the literature has either not considered TE [17, 31, 41, 43], proposed architectures that make deploying TE difficult [40, 55], only considered single controller deployment [22, 33], or proposed methods that apply to specific networks [14, 15, 22, 26, 54]. Helix improves performance by reducing the amount of shared state between controllers and offloading inter-area TE closer to the data-plane. Helix uses a lightweight TE optimisation mechanism that reduces its problem search space, resulting in fewer path changes and decreasing controller load.

MCSDN Systems: MCSDN systems such as B4 [26], Kandoo [19], Espresso [54], and others [14, 17, 30, 43], divide the network into areas. Operators decide on the switch-to-controller mapping and controller locations for their network based on available resources or using a Controller Placement Problem (CPP) solver such as [20, 27, 35, 37, 53]. Logically centralised MCSDN systems such as Onix [30], ONOS [6], DISCO [41], and Orion [15] allow controllers to make end-to-end decisions with centralised scope by propagating complete network state between devices. Single controller image frameworks such as Beehive [55] and SCL [40] effectively implement a logically centralised system by taking a Single-Controller SDN (SCSDN) system and automatically deploying it on multiple devices. Both system types suffer from performance issues because they require strong consistency. Helix avoids performance overheads by not using strong consistency and reducing the amount of shared state between devices. Distributed MCSDN systems use eventual consistency and introduce the notion of local and global operations. Compared to Helix, these systems either share more information between controllers or target specific applications [43]. Furthermore, Helix mitigates eventual consistency issues by reducing the amount of shared state and offloading operations.

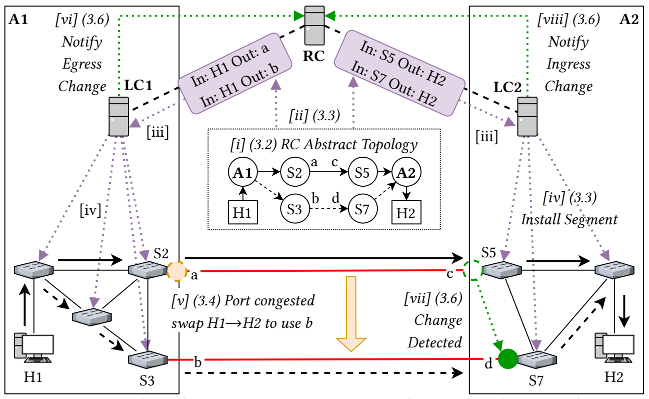

Figure 2: Helix design overview. Border switches and inter-area link ports are labelled on the diagram. The diagram shows (in order) Helix's supported operations (§3.2 - §3.6).

3 HELIX DESIGN

We designed Helix to minimise load and shared state between devices while still offering robust resilience to failures.

Helix uses a hierarchical control plane architecture, which reduces controller load and simplifies the system's design [39]. Helix deploys multiple redundant controller instances in a cluster to improve control plane failure resilience and remove single points of failure. Helix ensures consistency between cluster instances by restricting changes to a single primary device. Controller instances use a lightweight leader election process to designate the primary device and detect failures.

Helix uses eventual consistency to synchronise network information between controllers and uses abstraction to minimise information shared with the RC, which mitigates consistency concerns and improves scalability.

(3.1) LC Data Plane Recovery: Helix LC's use protection recovery to proactively address data plane failures. Protection recovery uses precomputed paths to allow switches to restore traffic forwarding without controller intervention. The LCs deploy protection recovery by computing and installing multiple minimally overlapping paths for each source-destination pair in their areas.

Despite the benefits, protection recovery raises two issues. First, switches must support fast-failover groups to deploy protection [48, 49]. Second, protection paths can lose optimality if the topology changes. To resolve the latter, the LC checks if paths are still optimal and recomputes them if necessary after detecting a topology change.

(3.2) RC Topology: Helix reduces the amount of state the RC maintains by abstracting topology information. Reducing managed state improves the scalability of the system and mitigates state consistency concerns. The RC topology contains all hosts and border switches of an area. Helix abstracts other elements of the local topology to a single node *(fig. 2 [i]).* The RC's topology includes border switches to allow the RC control over the ingress and egress port for a particular area. In the RC's topology, two border switches interconnect two areas via an inter-area link.

(3.3) RC Path Computation: The RC computes multiple abstracted minimally overlapping paths for every inter-area source-destination pair *(fig. 2 [ii])*. The computed paths specify the ingress and egress points needed in every area to forward traffic. The RC converts the paths to a sequence of instructions which it sends to the LC clusters *(fig. 2 [iii])*. The LCs use the RC instructions to install local path segments using the LC protection mechanism *(fig. 2 [iv])*. Helix's protection recovery mechanism allows restoration of data plane forwarding without LC or RC involvement.

(3.4) LC TE Optimisation: Helix attempts to minimise congestion. LCs collect and monitor link usage and path transmission rates for their respective areas. LCs consider a link as congested when its usage rate exceeds a threshold. LCs resolve congestion by modifying paths to avoid using the congested link.

Helix offloads inter-area TE optimisation to LCs, which improves performance and mitigates the state synchronisation concerns raised by using eventual consistency. LCs detect inter-area congestion by monitoring transmission rates on inter-area ports. When an inter-area link becomes congested, the LCs will shift inter-area traffic to another inter-area link (egress point) *(fig. 2 [v])*. In essence, LCs perform inter-area TE optimisation by making local decisions that influence the upstream path of traffic through other areas.

(3.5) RC TE Optimisation: Despite offloading inter-area TE to LCs, an LC cannot influence where traffic enters its area. When an area experiences congestion on all egress links, the LC will fail to resolve inter-area congestion locally because it lacks sufficient headroom on alternative inter-area links to move traffic. Helix addresses this scenario by implementing an RC TE optimisation procedure, initiated by an LC when it fails to resolve inter-area congestion. The RC and LC TE mechanisms are similar.

LCs collect and periodically provide the RC with inter-area link usage information. The LC optimisation request includes a set of source-destination pairs using the congested port and the amount of traffic each path carries observed by the ingress ports of the area. The RC's TE mechanism modifies inter-area paths to divert traffic away from a congested area.

(3.6) Updating Inter-Area Paths: Because Helix offloads inter-area TE optimisation to local controllers, LCs can change the path used by inter-area traffic. To ensure that the RC path information is up-to-date and reflects the actual installed data plane forwarding, Helix implements a path change notification mechanisms. When an LC modifies the egress port of an inter-area path, it will send a notification to the RC which will update its computed path information *(fig. 2 [vi])*. LCs detect ingress changes for inter-area traffic within their area by monitoring ingress ports *(fig. 2 [vii])*. When an ingress change is detected, the LC will notify the RC to update its information *(fig. 2 [viii])*. Helix performs local ingress change detection to remove dependencies between LC clusters and improve the control plane failure resilience of the system.

4 HELIX IMPLEMENTATION

We implemented Helix using Ryu [4], a Python OpenFlow framework. Both the Local Controller (LC) and the Root Controller (LC) are divided into several modules: inter-controller communication (§4.1), topology discovery (§4.2), path computation (§4.3, §4.4), and TE optimisation (§4.4, §4.5).

(4.1) Inter-Controller Communication Module: Helix controllers communicate via messages using the Advanced Message Queuing Protocol (AMQP) [38] with a publish-subscribe (pub-sub) model. The Helix pub-sub model groups messages into channels based on routing keys. Each routing key conveys specific information about a relevant topic (e.g. local controller topology information). Helix controllers create bindings subscribing to receive messages on a set of channels, depending on the required state.

(4.2) LC Topology Discovery Module: We extended Ryu's topology detection module to support controller roles, host discovery, and inter-area link detection. The topology discovery module performs active bidirectional link connectivity and failure detection through Link Layer Discovery Protocol (LLDP) packet flooding. LCs send LLDP packets on every port of every switch they manage. Switches send received LLDP packets to the LC, allowing the controller to observe the topology. LLDP packets also act as heartbeat messages, which enable the LC to infer data plane failures.

LCs discover inter-area links using the LLDP mechanism. Because of the packet flooding process, neighbouring area LCs will receive flooded LLDP packets from other LC clusters on inter-area links. When an LC receives an LLDP packet from a switch it does not manage, it processes it as an inter-area link discovery packet.

(4.3) LC Path Computation Module: The LC path computation module uses Dijkstra's algorithm [11] to compute two minimally overlapping paths (P_{prim} and P_{sec}) and a set of path splices for every source-destination pair in the area. The path computation module generates minimally overlapping paths through link manipulation. Links previously used in P_{prim} will be set to large weights causing the algorithm to avoid using them, generating minimally overlapping paths. After P_{prim} and P_{sec} are computed, the module generates a set of path splices to increase protection coverage.

A path splice is the shortest path between unique nodes in the first path to unique nodes in the second path such that the exit node is closest to the final destination (minimise path stretch). A path splice increases protection coverage and allows switches to recover from complex failure conditions, such as simultaneous failures on both primary and secondary paths.

The module translates the computed paths into a set of ports it installs onto the data plane using the fast-failover group type.

(4.4) LC TE Optimisation Module: The TE module detects bidirectional link congestion by polling switches for port statistics. The TE module monitors source-destination pair (candidate) transmission rates on ingress points of the area by recording send statistics of ingress rules. LCs attempt to reduce port usage by modifying candidate paths to avoid the congested ports.

First, the module recomputes the congested port usage rate based on the candidate traffic send rates to account for congestion loss. The TE optimisation module considers candidates in descending order of generated traffic. Second, the algorithm uses a CSPF-style recomputation to move the candidate's traffic away from the congested port while not introducing new congestion to the network. The algorithm prunes the topology of links that do not have sufficient capacity to carry the candidate traffic and recomputes the path using Dijkstra's algorithm. The module will keep track of all valid candidate path change and apply them if it successfully resolved the detected congestion

(4.5) RC TE Optimisation Module: The RC TE optimisation module recomputes inter-area source-destination paths (candidates) to avoid a congested inter-area link. The RC uses the same CSPF-style recomputation used by the LCs to recompute candidate paths and resolve congestion. If the RC finds a solution to resolve the detected congestion, the RC converts the new paths into instructions, which it sends to the LCs to be installed onto the data plane.

5 EMULATION FRAMEWORK

We developed an emulation framework to evaluate control plane failure resilience. The framework provides two contributions. First, the framework computes metrics that allow comparing an MCSDN system's failure recovery performance. Second, the framework checks the behaviour of an MCSDN system by monitoring its interaction with the data plane. In essence, the framework provides a black-box testing tool that ensures an MCSDN system exhibits correct behaviour under predefined failure conditions.

The emulation framework is built on top of Mininet [34] and requires four inputs. First, a topology T specifies the network to use for the experiment. Second, F defines a failure scenario that outlines the actions to emulate. Third, A_{map} outlines the areas in T and the switch-to-controller assignment. A_{map} describes the set of nodes (n) every controller cluster (c) manages ($\forall n \in T, \forall c \in T; A_{map} : n \mapsto c$) and the set of redundant instances (i) in each cluster ($\forall i \in c; A_{map} : i \mapsto c$). Finally, $E_{expected}$ contains a set of event types the framework should observe during an experiment.

We divide failure scenarios into multiple stages (F_{stage}) to allow the framework to attribute observed events to a sequence of actions. An F_{stage} contains a set of actions such that $F_{action} \in F_{stage}$. The framework implements three action types: (1) fail a control plane device, (2) start a control plane device, and (3) introduce a delay.

During an experiment, the framework monitors the MCSDN system's behaviour, generating a set/timeline of events ($E_{observed}$). The framework collects two event types. (1) Control-channel events, which describe the interaction between the controller and switches, are collected by capturing and processing control plane traffic. (2) Local events, which represent internal controller state changes, are gathered by monitoring the log files of the MCSDN system. Local event collection requires support from the MCSDN system because controllers need to push information to their respective logs when a relevant state change occurs.

The framework uses local events to separate metrics into components. For example, the framework uses local events to calculate the failure detection (δ_{FD}) and role change (δ_{RC}) component of the failure recovery metric ($\Delta_{recv} = \delta_{FD} + \delta_{RC}$). We designed the framework to produce metrics that allow comparing performance without the MCSDN system providing local event support. As a result, local events are optional.

6 EVALUATION: RESILIENCE

It is intractable to evaluate Helix under all possible failure scenarios. As a result, we define several hard-to-solve failure conditions (simultaneous failures) across a failure scenario to characterise Helix's failure resilience performance. We repeat experiments 100 times

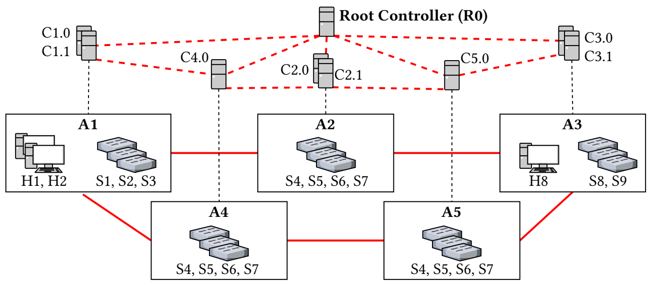

Figure 3: The topology used to evaluate Helix's control plane failure resilience. Dashed lines represent control-channel links and solid lines inter-area connections.

and report the average metric values along with the 95% confidence interval. We used a control-channel latency of 20ms to simulate conditions found in WANs.

We calculated the edge-to-edge latency of a WAN by using a medium-sized network from the Internet Topology Zoo Project [29], specifically the AT&T MPLS network, based on the geographic distance scaled to account for latency inflation as described in [50]. We assumed that controllers are deployed in the centre of an area to minimise inter-device communication time, so we used a latency of 20ms (half the edge-to-edge latency) to represent deployment conditions encountered on an average WAN.

For our experiments, we use a topology (T) illustrated in figure 3 consisting of five areas labelled $A1$ - $A5$. In T, a controller cluster Cn manages the area An where n represents a unique ID (1 - 5). $A1$, $A2$ and $A3$ contain two controller instances per cluster. $A4$ and $A5$ contain one instance per cluster. We limit the number of switches and hosts in each area to minimise the likelihood of encountering emulation overheads. All areas in T contain three switches. $A1$ contains two hosts labelled $H1$ and $H2$ while $A3$ contains the final host $H8$.

We also evaluated Helix's control-plane failure resilience performance against cascading failures. Cascading failures often occur due to increased load caused by shifting resources from a failed instance to another. Due to space constraints, we omitted these results from the paper. Our cascading failure results and observations were consistent with the results presented in this section.

Table 1 presents the average metric values when evaluating Helix using the defined failure scenario (simultaneous failures). The stages of the scenario are as follows: (a) emulates a simultaneous failure of the primary controller instance across the $C1$, $C2$ and $C3$ clusters; (b) emulates either adding a new controller instance to an existing cluster or restarting a failed instance; (c) emulates a single controller instance failure; (d) emulates the failure of an area (i.e. a controller instance failure that Helix will propagate to the RC).

Local Cluster Recovery: For stages (a) and (c), Helix did not modify any paths and did not propagate the failure events to $R0$. The average failure recovery metric for (a) and (c) was 1.6 and 1.1 seconds. The average observed failure detection time (a component of failure recovery) is 1 second for both stages, while role change time took an average of 36ms and 37ms. For Helix, failure detection made up over 90% of the recovery metric. The significant contribution of failure detection to recovery time is unsurprising because Helix uses a timeout mechanism to detect instance failures. The timeout mechanism introduced a long enough delay to account for

Metric	Time (s) *Average ; CI*	Metric	Time (s) *Average ; CI*
(a) Failure Recovery	1.609 ; 0.181	(c) Failure Recovery	1.106 ; 0.116
Failure Detection	1.003 ; 0.033	Failure Detection	0.958 ; 0.058
Role Change	0.036 ; 0.002	Role Change	0.037 ; 0.003
(b) Cluster Join Time	0.959 ; 0.056	(d) Area Failure Recovery	3.426 ; 0.084
Controller Start	0.426 ; 0.005	Failure Detection	2.365 ; 0.084
Initiation Phase	0.005 ; 0.000	$R0$ Compute Path	1.002 ; 0.000
Switch Enter	0.522 ; 0.056	$R0$ Path Installation	0.059 ; 0.002
Role Change	0.011 ; 0.000		

Table 1: Control plane failure resilience results (average over 100 iterations) for Helix. Failure detection made up over 90% of Helix's failure recovery metric.

latency and timer trigger differences between instances. However, role change requests are not delayed and only affected by latency, contributing less to the metric value.

In our experiments, failure detection was affected by the keep-alive (τ_k) and timeout ($\tau_{timeout}$) configuration values and the alignment of the failure events to keep-alive intervals. We expect that Helix detects a failure within $\tau_k + \tau_{timeout}$ from the failure action.

Both instance failure recovery metrics and component values were consistent, despite (a) defining a more complex failure scenario. Based on the observed consistency, we draw two conclusions: (1) the framework did not introduce significant overheads to inflate results, and (2) Helix's leader election module did not significantly increase CPU load.

Area Failure Recovery: Once Helix detects a complete cluster failure, $R0$ recomputes inter-area paths to avoid using the failed area. Because of the used configuration attributes, area recovery is slower compared to local instance failure recovery. However, switches will still forward traffic and deal with data plane failures during a complete cluster failure. Helix will delay area failure recovery to promote better forwarding stability (prevent disruptions and packet reordering) and prevent inter-area path flapping. If desired, modifying Helix's configuration attributes will change this behaviour and make area failure detection more aggressive.

The average failure recovery time was 3.4 seconds for (d). $R0$ detected the failure of $A2$ via dead inter-area port messages (triggered by LC LLDP timeouts) received from $C1$ and $C3$. The messages indicated to $R0$ that $A2$ has become isolated (all inter-area links failed). On average, Helix detected the area failure in 2.4 seconds. We observed that $R0$ recomputed inter-area paths after a 1-second delay. The delay represents Helix's path consolidation timeout ($\tau_{RConsolidate}$), which groups multiple changes to a single update to reduce path change churn. Helix's average path installation time was 60ms, which represents the time it took to compute and apply the inter-area path modifications to the data plane.

Instance Join Time: During stage (b), we observed that Helix's average cluster join time was 1.0 seconds, while the instance's startup time was 426ms. After starting up, the controller instance entered the initiation phase, which terminated early in under 5ms. Early termination is a Helix optimisation for the leader election process. A Helix controller exits its initiation phase once $\tau_{init} = \tau_k/2$ seconds have elapsed or a primary instance for the cluster was detected (early termination). The switch enter time, which measures the time it took all the switches to connect to the controller instance from startup, was 522ms. The role change time was 11ms, which is closer to the configured experiment latency. In our experiments, the role change time for the failure recovery stages (a and c) was larger (35ms) due to increased control-channel load during the scenario. The internal controller threading and Ryu's asynchronous message processing mechanism introduced delays when dealing with more control-channel messages, causing the observed slight increase in role change time.

7 EVALUATION: TE OPTIMISATION

We evaluated the performance of Helix's TE optimisation algorithm using YATES [32], a simulation framework. YATES models link forwarding behaviour by calculating the total traffic on every link of a topology. YATES simulates network traffic using a TE matrix that contains multiple iterations (rows) and specifies the amount of traffic each source-destination pair is sending (column). YATES runs every iteration for $T_{simtime}$ simulation units. A simulation unit represents a repetition of the simulation loop.

YATES provides support to simulate offline TE algorithm behaviour. At the start of each iteration, YATES calls the algorithm to compute a routing scheme based on estimated traffic demands (prediction TE matrix). In contrast, Helix uses a reactive TE algorithm that performs TE optimisation during runtime. We, therefore, extended YATES to support simulating reactive TE algorithms.

Other TE Algorithms: We compared the performance of Helix's TE optimisation algorithm against three commonly used load balancing (ECMP, VLB [51], Raeke [42]) and three TE algorithms (CSPF, MCF [16, 22, 25, 54] and SWAN [22]) provided by YATES.

Testing Methodology: Our experiment configuration attributes were a $T_{simtime}$ of 500 simulation units with a Helix TE/max link usage threshold of 95%, and a Helix poll interval of 100 units.

We collected results using the AT&T MPLS topology (size: 25 nodes and 56 links), which is representative of a medium-sized WAN, from the Internet Topology Zoo Project [29]. We used YATES to generate TE matrices with realistic traffic patterns [32]. All three of the YATES TE algorithms performed offline TE optimisation using a prediction matrix. In our evaluation, we used prediction matrices that contained no errors. Prediction errors can increase congestion and degrade performance.

We evaluated TE optimisation performance by introducing congestion into the network and measuring the percentage of traffic lost during an iteration. We also evaluated the overall forwarding stability of the TE algorithms by looking at the number of path changes performed per iteration. Performing frequent path changes decreases system stability and throughput by causing packet reordering. In our experiments, we showcased the ability of the tested algorithms to deal with more congestion by collecting results with three gradually increasing matrix multipliers. To do this, we first found a multiplier that scaled the TE matrix sufficiently to cause congestion loss and then gradually increased its value.

Results: Unsurprisingly, because load balancing algorithms do not split traffic based on congestion or a prediction matrix, all tested load balancing algorithms performed poorly. With a multiplier of 500x, VLB (the best performing load-balancing algorithm) experienced congestion loss during all iterations with a max loss rate of 20% compared to no loss for 57% of iterations and a max loss rate of 14.1% for SWAN (the worst-performing TE algorithm). Despite outperforming the load balancing algorithms, SWAN experienced more congestion loss compared to Helix, CSPF and MCF. Due to

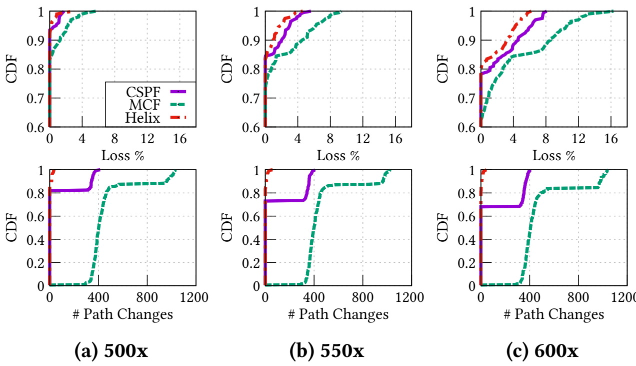

Figure 4: AT&T MPLS - CDF graphs showing congestion loss % and the number of path changes per iteration (top three) using three TE matrix traffic multipliers for 200 iterations.

space constraints, this section will only present results for the top three performing algorithms (Helix, CSPF and MCF).

Figure 4 shows a CDF graph of congestion loss (top row) and path change churn (bottom row) observed on the AT&T topology. With a multiplier of 500x (figure 4a), Helix performed similarly to CSPF, with no congestion loss reported for 91% of iteration versus 92.5% for CSPF. For 99% of iterations, Helix observed a 1.6x decrease in congestion loss compared to CSPF (0.9% loss rate compared to 1.4%). The maximum loss rate was higher for Helix, 3.0% compared to 1.8% for CSPF (reported during the first iteration when Helix started with non-optimised paths). Helix outperformed MCF, which experienced no congestion loss for 85% of iterations and 4.6% maximum loss rate compared to Helix.

Increasing the multiplier to 550x (figure 4b), we observed a slight performance improvement for Helix compared to the other algorithms (1.6x decrease in congestion loss for 97.5% of iterations). Helix's maximum congestion loss rate was also 1.3x lower (4.6% versus 5.6% for CSPF). The performance difference between Helix and CSPF slightly widened with a 600x multiplier (figure 4c).

While we observed a marginal decrease in congestion loss when using Helix compared to the other TE algorithms, Helix performed significantly fewer path changes, 12x fewer compared to CSPF (36 versus 414) and 29x fewer compared to MCF (36 versus 1034). This pattern carried on when increasing the multiplier, Helix performing at most 60 and 45 path changes (per iteration) compared to 408 and 400 for CSPF. The slight drop in path change churn when increasing the traffic multiplier was caused by the TE algorithms not being able to resolve all congestion due to the increased load (implying fewer path modifications). We can confirm this inference by looking at the congestion loss results.

Limitations of Results: YATES was not intended to test decentralised systems. YATES cannot reason about the convergence of distributed protocols or performance during state distribution [32]. As such, we consider that YATES overestimates performance. We would also like to point out that methods such as CSPF and MCF, use a prediction matrix to compute paths. Prediction matrices are difficult to generate and may include errors [3]. We collected results using prediction matrices that contained no errors, implying that the algorithms were fully aware of the exact amount of traffic sent between hosts. Prediction errors can lead to congestion loss if the TE algorithm does not reserve sufficient spare capacity on links to deal with the extra traffic, while reserving too much can prevent the algorithm from finding a solution to resolve congestion.

8 CONCLUSION & FUTURE WORK

Deploying TE in the context of MCSDN is challenging due to state and consistency requirements. In this paper, we designed and implemented Helix, a Hierarchical MCSDN system that addresses performance, robustness, and consistency concerns of deploying TE on WANs. Helix tolerates high inter-device latency and improves its robustness to controller failures by offloading operations (such as inter-area TE and failure recovery) closer to the data plane. Helix uses a proactive failure recovery mechanism that decreases recovery time and reduces packet loss.

Our experiments showed that failure detection made up over 90% of Helix's instance failure recovery time and over 70% of the area (cluster) recovery metric. Helix's failure resilience performance is affected by the selected configuration. Decreasing τ_k (keep-alive timer) to 200ms will reduce Helix's failure recovery time to 340ms (4x faster compared to our results) while setting τ_{lldp} to 100ms (setting τ_{iap} to 500ms) will reduce area failure recovery to 1.6s.

Under heavy traffic load, Helix reduces congestion loss by up to 1.6x compared to CSPF while performing 12x fewer path changes. We can attribute the improvement in performance to Helix's TE algorithm, which reduces its candidate search space. This reduction decreases the number of path changes performed and enables Helix to find solutions under heavy traffic loads, decreasing congestion. Helix's approach to TE provides three benefits that make it suitable for deployment on WANs. (1) performing fewer path changes improves the forwarding stability of the system as fewer disruptions of in-flight packets occur. (2) constraining the candidate search space reduces strain on controllers by decreasing CPU load (improves scalability). (3) Helix's TE algorithm supports offloading, which mitigates state consistency and performance problems.

While we have comprehensively evaluated Helix, we omitted parts of our evaluation from this paper due to space constraints. We evaluated Helix's data plane failure resilience (protection mechanism) and compared it against restoration recovery. We found that Helix outperformed restoration irrespective of latency and thus is a well-suited technique for use in WANs. With a control-channel latency of 20ms, Helix recovered from failures 10x faster compared to restoration recovery. As stated in the failure resilience section, we also defined and evaluated Helix's performance under cascading failures. Our cascading failure results and observations were consistent with the results presented in this paper. The complete failure resilience results also included an evaluation of Helix's new area detection performance. Finally, we also assessed Helix's TE optimisation performance using the Abilene network from the Internet Topology Zoo Project [29]. Similar to the results presented in this paper, Helix experienced less congestion loss and performed fewer path changes compared to the other tested algorithms. We leave publishing these results and expanding on Helix's evaluation as future work. To support reproducibility and encourage further research, we make the emulation framework and Helix's source code available at [1].

REFERENCES

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

[2] 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 (2014), 503–514. https://doi.org/10.1145/2740070.2626316

[3] 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* (Karlsruhe, Germany) *(SIGCOMM '03)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/863955.863991

[4] RYU Authors. 2011. RYU SDN Framework. https://github.com/faucetsdn/ryu v4.25.

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

[6] 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* (Chicago, Illinois, USA) *(HotSDN '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2620728.2620744

[7] 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. https://doi.org/10.1145/2656877.2656890

[8] 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)* (Vancouver, BC, Canada) *(IEEE IC2E 2017)*. IEEE, New York, NY, USA, 40–50. https://doi.org/10.1109/IC2E.2017.33

[9] 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 *ACM SIGCOMM Computer Communication Review* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/2018436.2018466

[10] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (Santa Clara, California) *(SOSR '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2774993.2774999

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

[12] 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)* (Marina del Rey, CA, USA) *(IEEE ANCS 2014)*. IEEE, New York, NY, USA, 17–27. https://ieeexplore.ieee.org/abstract/document/7856398

[13] 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)* (Xi'an, China, China). IEEE, New York, NY, USA, 1322–1326. https://doi.org/10.1109/ICCT46805.2019.8947182

[14] 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* (Santa Clara, California) *(SOSR '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 13 pages. https://doi.org/10.1145/2774993.2775009

[15] 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, 83–98. https://www.usenix.org/conference/nsdi21/presentation/ferguson

[16] Lester Randolph Ford Jr and Delbert R Fulkerson. 1958. A suggested computation for maximal multi-commodity network flows. *Management Science* 5, 1 (1958), 97–101. https://doi.org/10.1287/opre.11.3.344

[17] 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

[18] 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

[19] 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* (Helsinki, Finland) *(HotSDN '12)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/2342441.2342446

[20] 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* (Helsinki, Finland) *(HotSDN '12)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/2342441.2342444

[21] C. Ho, K. Wang, and Y. 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

[22] 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* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2486001.2486012

[23] 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. https://www.usenix.org/conference/nsdi20/presentation/hsu

[24] 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* (San Jose, CA, USA) *(SOSR '20)*. Association for Computing Machinery, New York, NY, USA, 103–109. https://doi.org/10.1145/3373360.3380841

[25] Jingyu Hua, Laiping Zhao, Suohao Zhang, Yangyang Liu, Xin Ge, and Sheng Zhong. 2018. Topology-preserving traffic engineering for hierarchical multi-domain SDN. *Computer Networks* 140 (2018), 62–77. https://doi.org/10.1016/j.comnet.2018.04.011

[26] 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* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2486001.2486019

[27] 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* (Trondheim, Norway) *(IFIP Networking 2014)*. IEEE, New York, NY, USA, 1–9. https://doi.org/10.1109/IFIPNetworking.2014.6857117

[28] 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* (Santa Clara, CA, USA) *(SOSR '16)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/2890955.2890968

[29] 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 (2011), 1765–1775. https://doi.org/10.1109/JSAC.2011.111002

[30] 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*, Vol. 10. USENIX Association, Santa Clara, CA, 1–6. https://www.usenix.org/conference/osdi10/onix-distributed-control-platform-large-scale-production-networks

[31] 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* (Chicago, Illinois, USA) *(HotSDN '14)*. Association for Computing Machinery, New York, NY, USA, 133–138. https://doi.org/10.1145/2620728.2620748

[32] 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* (Los Angeles, CA, USA) *(SOSR '18)*. ACM, New York, NY, USA, Article 11, 7 pages. https://doi.org/10.1145/3185467.3185498

[33] 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

[34] 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* (Monterey, California) *(Hotnets-IX)*. Association for Computing Machinery, New York, NY, USA, Article 19, 6 pages. https://doi.org/10.1145/1868447.1868466

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

[36] 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* (Amman, Jordan) *(ICFNDS '18)*. Association for Computing Machinery, New York, NY, USA, Article 57, 6 pages. https://doi.org/10.1145/3231053.3231124

[37] L. F. Müller, R. R. Oliveira, M. C. Luizelli, L. P. Gaspary, and M. 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

[38] 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

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

[40] 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, 329–345. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-aurojit-scl

[41] K. Phemius, M. Bouet, and J. 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

[42] Harald Räcke. 2008. Optimal Hierarchical Decompositions for Congestion Minimization in Networks. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing* (Victoria, British Columbia, Canada) *(STOC '08)*. Association for Computing Machinery, New York, NY, USA, 255–264. https://doi.org/10.1145/1374376.1374415

[43] Mateus AS Santos, Bruno AA 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* (Edmonton, AB, Canada) *(IEEE LCN 2014)*. IEEE, New York, NY, USA, 402–405. https://doi.org/10.1109/LCN.2014.6925802

[44] H. Selvi, G. Gür, and F. 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

[45] 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* (San Jose, CA, USA) *(SOSR '20)*. Association for Computing Machinery, New York, NY, USA, 83–95. https://doi.org/10.1145/3373360.3380839

[46] 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* (Hong Kong, China) *(APNet'17)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3106989.3107001

[47] R. Shah, M. Vutukuru, and P. 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

[48] 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/abstract/document/6529882

[49] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. 2013. OpenFlow: Meeting carrier-grade recovery requirements. *Computer Communications* 36, 6 (2013), 656–665. https://doi.org/10.1016/j.comcom.2012.09.011

[50] 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* (Los Angeles, CA, USA) *(HotNets-XIII)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/2670518.2673876

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

[52] Y. Xu, M. Cello, I. Wang, A. Walid, G. Wilfong, C. H. . Wen, M. Marchese, and H. J. Chao. 2019. Dynamic Switch Migration in Distributed Software-Defined Networks to Achieve Controller Load Balance. *IEEE Journal on Selected Areas in Communications* 37, 3 (2019), 515–529. https://doi.org/10.1109/JSAC.2019.2894237

[53] Kongzhe Yang, Daoxing Guo, Bangning Zhang, and Bing Zhao. 2019. Multi-Controller Placement for Load Balancing in SDWAN. *IEEE Access* 7 (2019), 167278–167289. https://doi.org/10.1109/ACCESS.2019.2953723

[54] 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* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 432–445. https://doi.org/10.1145/3098822.3098854

[55] Soheil Hassas Yeganeh and Yashar Ganjali. 2016. Beehive: Simple Distributed Programming in Software-Defined Networks. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/2890955.2890958

[56] J. Yu, Y. Wang, K. Pei, S. Zhang, and J. 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

[57] 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* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 351–362. https://doi.org/10.1145/1851182.1851224