Building Efficient and Reliable Software-Defined Networks

Naga Praveen Kumar Katta

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Jennifer Rexford

November 2016

© Copyright by Naga Praveen Kumar Katta, 2016.

All rights reserved.

Abstract

Software-defined networking (SDN) promises flexible control of computer networks by orchestrating switches in the network dataplane through a centralized controller. However, despite this promise, operators used to fast and fault-tolerant routing using traditional protocols face three important problems while deploying SDN. One, the control plane timescales are too slow to enforce effective load balancing in order to efficiently use the available network capacity. Second, the commodity SDN switches have limited memory to enforce fine-grained policy rules which undermines the promise of flexible control. Third, the centralized controller itself is a single point of failure, which is unacceptable for operators used to running distributed fault-tolerant network protocols.

This thesis aims to mitigate these problems using novel algorithms that exploit advanced data plane capabilities and enhancements to the control plane software. At the same time, we also provide simple abstractions on top of these systems so that network operators writing control programs need not worry about low-level details of the underlying implementation mechanisms.

First, we propose HULA, which gives the abstraction of one big efficient non-blocking switch. Instead of asking the control plane to choose the best path for each new flow, HULA efficiently routes traffic on least congested paths in the network. HULA uses advanced hardware data plane capabilities to infer global congestion information and uses that information to do fine-grained load balancing at RTT timescales. HULA is congestion-aware, scales to large topologies, and is robust to topology failures.

Second, we propose CacheFlow, which helps enforce fine-grained policies by proposing the abstraction of a switch with logically infinite rule space. CacheFlow uses a combination of software and hardware data paths to bring the best of both worlds to policy enforcement. By dynamically caching a small number of heavy hitting rules in the hardware switch and the rest of the rules in the software data path, it achieves both high throughput and high rule

capacity. Since cross-rule dependencies make rule caching difficult, CacheFlow uses novel algorithms to do dependency-aware, efficient rule caching that is transparent to control applications.

Finally, we propose Ravana, which gives the abstraction of one logically centralized controller. Given this abstraction, the network operator only writes programs for one controller and the Ravana runtime takes care of replicating the control logic for fault-tolerance. Since network switches carry additional state external to the controller state, Ravana uses an enhanced version of traditional replicated state machine protocols to ensure ordered and exactly-once execution of network events.

Together these systems propose a new SDN paradigm where basic routing is done efficiently at dataplane timescales, policy enforcement is done scalably with the help of software data planes, and the control plane is fault-tolerant. This new architecture has the properties of fast routing and fault-tolerance of traditional networks while delivering the promise of efficient enforcement of fine-grained control policies.

Acknowledgements

I am deeply indebted to my advisor, Jennifer Rexford, for being the most reliable guiding light throughout this journey. When I was lost in the crossroads during my first year in grad school, she gave me the career-defining opportunity to explore project ideas under her wing. Thereafter, she helped develop my taste in research and subtly inspired me to live up to her standards of doing high-quality research and maintaining academic integrity. I cannot possibly express my gratitude to her in just a few words here, but I know this experience alongside her will have a profound impact throughout my professional and personal life.

I am thankful to Dave walker whose advice was incredibly helpful in the formative years of my research related to software-defined networks. I was influenced by how he would start with smallest case for a research problem and dissect it clearly before going for the big fish. He helped me with clear and succinct writing, and introduced me to the fascinating world of programming languages.

I am grateful to Mike Freedman for kindling my interest in distributed systems and its intersection with networking. He was incredibly helpful in doing critical analyses of many replication protocols in the literature and in fleshing out the key design aspects of our Ravana system.

In addition to my academic advisors, I am grateful to have to worked with Mukesh Hira at VMware and Changhoon Kim at Barefoot Networks. Mukesh was extremely kind and patient with me during my tryst with a true transition from research to practice. He would often get his hands dirty when it comes to setting up testbeds and doing experiments. His keen insights into debugging large scale, complex systems were inspiring to me. I am also thankful to many folks at Barefoot networks — especially Changhoon Kim, who kindled my interest in programmable data planes in a big way. I benefited greatly from his advice on the design aspects of the HULA project and on my contributions to the internal demos at the company.

I thank my collaborators - Omid Alipourfard, Harris Zhang, and Anirudh Sivaraman for working with me on the projects discussed in this thesis. They have been of immense help while discussing ideas, running experiments, and during the publication process. I cherish their professional and personal friendship immensely. I thank Aarti Gupta and Nick Feamster for chatting with me through my thesis defense process and for having a conversation about research topics and career paths.

I also thank all of the cabernet family I interacted with regularly - Mojgan, Mina, Srinivas, Ori, Robert, Rob, Jennifer, and former members - Theo, Laurent, Peng, Nanxi, Xin, Kelvin, Ronaldo, Yaron, Soudeh and Josh. I cherish the friendship of Cole, Amy, Logan, Natasha, Nick, Danielle, Jordan, and many others during my PhD years. I also thank Nicki, Mitra, and Nicole for helping me with a lot of administrative work in the department.

This dissertation work is supported by NSF grats CNS-1111520, CNS-1409056, the ONR under award N00014-12-1-0757 and an Intel grant.

Finally, I am extremely grateful to my loving parents, my sister Sindhu, my good friend Karthik, and everyone else who extended emotional support during this arduous time.

"...non-existent things can be more easily and irresponsibly represented in words than existing things; for the serious and conscientious historian, it is just the reverse."

- Herman Hesse, "The Glass Bead Game"

To my parents.

Contents

Abstract iii

Acknowledgements v

List of Tables xiv

List of Figures xv

1 Introduction **1**

1.1 Traditional Networks 1

1.2 Software-Defined Networking: A New Paradigm 3

1.3 SDN Meets Reality: Challenges 4

1.3.1 Efficient Network Utilization 5

1.3.2 Flexibility 6

1.3.3 Reliability 7

1.4 Opportunities for Handling Challenges 9

1.4.1 Programmable Dataplanes for Efficient Utilization 9

1.4.2 Software Switching for Fine-Grained Policies 11

1.4.3 Replicated State Machines for Reliable Control 11

1.5 Contributions 12

2 HULA: Scalable Load Balancing Using Programmable Data Planes **15**

2.1 Introduction 16

2.2 Design Challenges for HULA 18

2.3 HULA Overview: Scalable, Proactive, Adaptive, and Programmable 20

2.4 HULA Design: Probes and Flowlets 23

2.4.1 Origin and Replication of HULA Probes 24

2.4.2 Processing Probes to Update Best Path 25

2.4.3 Flowlet Forwarding on Best Paths 28

2.4.4 Data-Plane Adaptation to Failures 29

2.4.5 Probe Overhead and Optimization 30

2.5 Programming HULA in P4 31

2.5.1 Introduction to P4 31

2.5.2 HULA in P4 32

2.5.3 Feasibility of P4 Primitives at Line Rate 36

2.6 Evaluation 37

2.6.1 Symmetric 3-tier Fat-Tree Topology 40

2.6.2 Handling Topology Asymmetry 41

2.6.3 Stability 45

2.6.4 Robustness of probe frequency 48

2.7 Related Work 48

2.8 Conclusion 50

3 CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks **51**

3.1 Introduction 52

3.2 Identifying Rule Dependencies 55
3.2.1 Rule Dependencies 55
3.2.2 Where do complex dependencies arise? 57
3.2.3 Constructing the Dependency DAG 60
3.2.4 Incrementally Updating The DAG 61
3.3 Caching Algorithms 65
3.3.1 Optimization: NP Hardness 66
3.3.2 Dependent-Set: Caching Dependent Rules 68
3.3.3 Cover-Set: Splicing Dependency Chains 69
3.3.4 Mixed-Set: An Optimal Mixture 71
3.3.5 Updating the TCAM Incrementally 71
3.4 CacheMaster Design 72
3.4.1 Scalable Processing of Cache Misses 73
3.4.2 Preserving OpenFlow Semantics 73
3.5 Commodity Switch as the Cache 75
3.6 Prototype and Evaluation 76
3.6.1 Cache-hit Rate 76
3.6.2 Incremental Algorithms 81
3.7 Related Work 82
3.8 Conclusion 84

4 Ravana: Controller Fault-Tolerance in Software-Defined Networking **85**
4.1 Introduction 86
4.2 Controller Failures in SDN 89

4.2.1 Inconsistent Event Ordering 90
4.2.2 Unreliable Event Delivery 92
4.2.3 Repetition of Commands 93
4.2.4 Handling Switch Failures 94
4.3 Ravana Protocol 95
4.3.1 Protocol Overview 97
4.3.2 Protocol Insights 99
4.4 Correctness 101
4.5 Performance Optimizations 104
4.6 Implementation of Ravana 106
4.6.1 Controller Runtime: Failover, Replication 106
4.6.2 Switch Runtime: Event/Command Buffers 107
4.6.3 Control Channel Interface: Transactions 108
4.6.4 Transparent Programming Abstraction 108
4.7 Performance Evaluation 109
4.7.1 Measuring Throughput and Latency 110
4.7.2 Sensitivity Analysis for Event Batching 112
4.7.3 Measuring Failover Time 114
4.7.4 Consistency Levels: Overhead 114
4.8 Related Work 117
4.9 Conclusion 120

5 Conclusion **121**

5.1 Summary of Contributions 122
5.2 Future Work 123

5.2.1 Heterogenous and Incremental HULA 123

5.2.2 Cooperative Caching for Efficient Resource Utilization 125

5.2.3 Ravana with Runtime Knobs for Consistency Requirements 126

5.3 Concluding Remarks 127

Bibliography **128**

List of Tables

2.1 Number of paths and forwarding entries in 3-tier Fat-Tree topologies [58] . 19

4.1 Ravana design goals and mechanisms 92

4.2 Comparing different solutions for fault-tolerant controllers 117

List of Figures

1.1 While traditional networking relies on running ossified distributed protocols, Software-Defined Networking separates the control plane from switches and unifies it in a centralized controller . . . 3

1.2 Example Switch Rule Table . . . 6

1.3 Programmable dataplane model . . . 10

1.4 Thesis contributions . . . 13

2.1 HULA probe replication logic . . . 25

2.2 HULA probe processing logic . . . 27

2.3 HULA header format and control flow . . . 33

2.4 HULA stateful packet process in P4 . . . 34

2.5 Topology used in evaluation . . . 38

2.6 Empirical traffic distribution used in evaluation . . . 39

2.7 Average flow completion times for the Web-search and data-mining workload on the *symmetric* topology. . . . 40

2.8 Average FCT for the Web-search workload on the *asymmetric* topology. . . . 42

2.9 Average FCT for the data mining workload on the *asymmetric* topology. . . . 44

2.10 99th percentile FCTs and queue growth on the asymmetric topology . . . 46

2.11 HULA resilience to link failures and probe frequency settings . . . 47

3.1 CacheFlow architecture 54

3.2 Constructing the rule dependency graph (edges annotated with reachable packets) 56

3.3 Dependent-set vs. cover-set algorithms (L_0 cache rules in red) 58

3.4 Dependent-set vs. cover-set algorithms (L_0 cache rules in red) 67

3.5 Reduction from densest k-subgraph 68

3.6 Dependent-set vs. cover-set Cost 69

3.7 TCAM Update Time 75

3.8 Cache-hit rate vs. TCAM size for three algorithms and three policies (with x-axis on log scale) 77

3.9 Cache-Miss Latency Overhead 79

3.10 Performance of Incremental Algorithms for DAG and TCAM update 80

4.1 SDN system model 89

4.2 Examples demonstrating different correctness properties maintained by Ravana and corresponding experimental results. t_1 and t_2 indicate the time when the old master controller crashes and when the new master is elected, respectively. In (f), the delivery of commands is slowed down to measure the traffic leakage effect. 90

4.3 Steps for processing a packet in Ravana. 98

4.4 Sequence diagram of event processing in controllers: steps 2–8 are in accordance with in Figure 4.3. 99

4.5 In SDN, control applications and end hosts both observe system evolution, while traditional replication techniques treat the switches (S) as observers. . 102

4.6 Optimizing performance by processing multiple transactions in parallel. The controller processes events $e1$ and $e2$, and the command for $e2$ is acknowledged before both the commands for $e1$ are acknowledged. 105
4.7 Ravana Event-Processing Throughput and Latency 111
4.8 Variance of Ravana Throughput, Latency and Failover Time 113
4.9 Throughput and Latency Overheads with Varied Levels of Correctness Guarantees 115

Chapter 1

Introduction

1.1 Traditional Networks

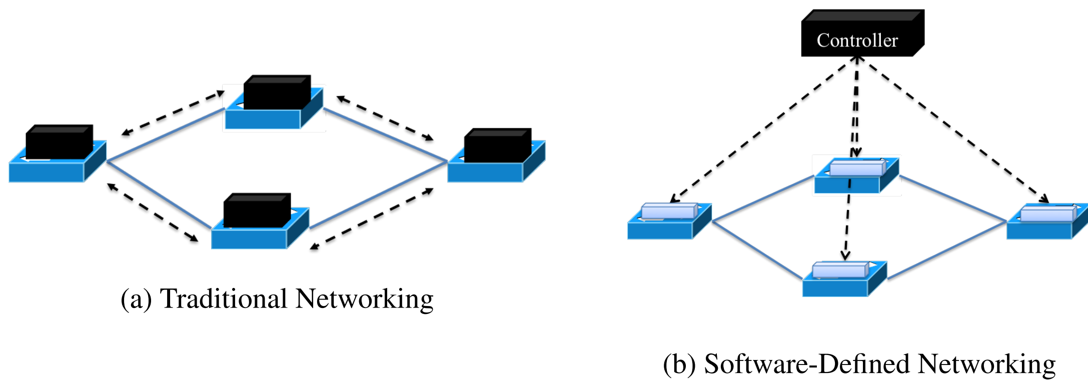
Traditionally, network operators ran distributed network protocols like OSPF [6], RIP [11], etc. in order to discover routes between network end points. An important hallmark of these protocols that was appealing to network operators was that these protocols ensure reliable routing in the face of network failures which is an important abstraction for the applications running at the end hosts. For example, switches running the OSPF protocol periodically send local link state information to all the other switches in the network. This information is then collected and processed by software running on switch CPUs to calculate global routes. When a network link fails, the protocol propagates updated link state information to calculate the new set of routes within seconds.

However, over the past few decades, while reliable routing was widely adopted, these protocols forced the operators to build their networks around rigid objectives and increasingly lose visibility into their networks. The key reason for this is that the protocols themselves are implemented as ossified software with hundreds of thousands of lines of opaque code written by multiple vendors. These implementations were proprietary and network

operators could neither access the internals for visibility, nor could they modify these implementations to achieve customized routing goals.

Traditional network switches have tightly coupled two important but separate functional components — switches discover network routes using distributed protocols in the *control plane* and use the computed routes to forward packets in the fast *data plane*. Most proprietary implementations had both these components baked into really expensive black box switch hardware that was the only choice for network operators. In addition, network operators had to manually login to each individual switch through an arcane CLI to configure them with thousands of parameters so that the overall distributed protocol discovered routing correctly. This meant the operator neither had a high level abstract view of the network nor did she have a high level interface to configure routing directly at that level of abstraction. Thus the overall network behavior was very opaque to the network operator which in turn made it difficult to either change this behavior (to achieve flexibility of routing packets based on different objectives, like traffic engineering etc.) or to debug deviation from expected behavior (like packet losses, routing loops etc.).

In recent years, the rise of geo-distributed data centers exacerbated these problems of lack of visibility and flexibility. The rapidly increasing demand on datacenter workloads meant that the datacenter network had to be fast enough to accommodate ever increasing compute capacity on short notice. In addition, the volatile nature of datacenter workloads meant that the operators had to quickly identify bottleneck network links and route data traffic around them at rapid timescales in order to fully exploit network capacity without cost prohibitive over-provisioning . This meant that network operators were willing to trade these protocol implementations for alternative architectures that are more manageable i.e., those that tackle the twin problems of lack of visibility and flexibility.



(a) Traditional Networking

(b) Software-Defined Networking

Figure 1.1: While traditional networking relies on running ossified distributed protocols, Software-Defined Networking separates the control plane from switches and unifies it in a centralized controller

1.2 Software-Defined Networking: A New Paradigm

In the place of ossified and opaque network protocols, the paradigm of Software-Defined Networking (SDN) proposes cleanly separating the control plane from network switches into a centralized server called a *controller*. The switches simply forward packets in the dataplane using *commands* sent by the controller. In addition, the switches may send *events* to the controller regarding the arrival of specific packets, flow counters, etc. The controller typically sends commands to switches in response to these events.

This decoupling of the data plane and control plane is the key to better network management in SDN. There are three important architectural changes that this separation enables. First, the operator gets a centralized view and control of the entire network from one place, the controller, instead of having to configure or poll each switch independently using different interfaces. Second, the functionality of the switches is abstracted into a much simpler match-action dataplane model (which dictates how to classify packets based on their header match) instead of having to worry about the code complexity that comes with running distributed protocols. Third, the separation and simplification of the dataplane enables a simple, unified and vendor-agnostic control interface like OpenFlow [89] across a heterogenous set of network elements from different equipment vendors.

Based on the above architectural changes, SDN promises two important properties — flexibility and efficiency. Instead of tweaking the protocols to indirectly do her bidding, the network operator can use the central controller to customize the network behavior according to her needs. This means she can flexibly route network traffic on specific paths. She can perform various actions on packets like packet forwarding, header modifications, etc. without having to use a separate protocol for each such function. In addition, the global network visibility and a unified control interface across multiple devices makes for much more efficient decision making.

Overall, SDN is seen today as a promising alternative to traditional distributed protocol implementations because of the increased flexibility and visibility. For example, Google's Software-defined WAN called B4 [63] is a result of the frustration with rigid and opaque vendor switching solutions. One of the key appeals of B4 for Google was the ability to rely on a central control software that can be flexibly customized to their needs (e.g., a WAN traffic engineering solution), the ability to efficiently change control at software speeds, and the ability to simply rely on control software testing before ensuring that the network behaves as expected when deployed in production. Over the years, in addition to large service providers like Google who built their own SDN switches, the SDN paradigm is also adopted by switching vendors like Cisco, Juniper, etc. which started rolling out support for OpenFlow-style control interface to their switches.

1.3 SDN Meets Reality: Challenges

While SDN is an attractive paradigm for network design, the practice of SDN is fraught with many challenges. While some challenges arise due to architectural shifts forcing a rethink about how specific objectives like reliability and efficiency should be achieved, some are due to unforeseen deployment issues with the new promises like flexibility.

Regardless, these challenges warrant a relook at the best practices for networking at many levels in an SDN — control plane, data plane, and the intertwining control interface. In this section, we focus on three important aspects of SDN that needed a fresh study recently — dataplane efficiency, control interface flexibility, and control plane reliability.

1.3.1 Efficient Network Utilization

Networks today typically have multiple paths connecting the network end points to avoid congestion when run at high utilization. Datacenter networks in particular have multi-rooted tree topologies with many equal cost paths that must be used effectively in order to exploit the high bisection bandwidth. This is why operators typically employ network load balancing schemes that spread incoming traffic load on the multiple paths in the network.

Traditionally, even before the advent of SDN, operators used a data-plane load-balancing technique called equal-cost multi-path routing (ECMP), which spreads traffic by assigning each flow to one of several paths at random. However, ECMP suffers from degraded performance [23, 31, 38, 68, 112] if two long-running flows are assigned to the same path. ECMP also doesn't react well to link failures and leaves the network underutilized or congested in asymmetric topologies.

The paradigm of SDN gives a new opportunity to look at dataplane efficiency by using the controller to deploy sophisticated traffic engineering algorithms. In particular, a controller can exploit global visibility into the congestion levels on various paths (say, by polling switch flow counters) and then can push commands to switches that steer traffic along multiple paths optimally. For example, schemes such as Hedera [23], SWAN [57], and B4 [63] use switch monitoring techniques to collect flow counters, solve the traffic engineering (TE) problem centrally at the controller given this information and then push routing rules to switches.

However, compared to ECMP, which would make load balancing decisions in the dataplane at line rate, control plane timescales are too slow to implement load balancing that ef-

Rule	Match	Action	Priority
R1	11*	Fwd 1	3
R2	1*0	Fwd 2	2
R3	10*	Fwd 3	1

Figure 1.2: Example Switch Rule Table

ficiently uses the available network capacity. The controller-based centralized TE schemes take on the order of minutes [57, 63] to react to changing network conditions, which is too slow for networks running volatile traffic. For example, in datacenters, most flows are short-lived, interactive, mice flows whose lifetimes span a few milliseconds. In this case, the flows either have to delayed till the central TE decision is enforced or simply use stale paths in the dataplane, both of which adversely effect application responsiveness.

This warrants a new look at designing effective load balancing schemes for SDN that are (i) responsive to volatile traffic at dataplane timescales in order to be effective and (ii) adhere to SDN principles of visibility and flexibility without resorting to opaque and rigid vendor-specific ASIC implementations.

1.3.2 Flexibility

As mentioned earlier, SDN achieves one of its fundamental objectives of flexible control by sending prioritized match-action rules to the switches using an interface like OpenFlow [89]. The match part describes the header pattern of packets that should be used to process this rule. The corresponding action may be either to forward the packet out of a switch port or to change certain header fields, or to drop it entirely, etc. The rules are prioritized in order to disambiguate when a packet matches multiple rules in the table. Figure 1.2 shows an example OpenFlow table with prioritized rules having match patterns and corresponding actions.

In modern hardware switches, these rules are stored in special hardware memory called Ternary Content Addressable Memory (TCAM) [16]. A TCAM can compare an incoming packet to the patterns in all of the rules at the same time, at line rate. However, commodity

switches support relatively few rules, in the small thousands or tens of thousands [117]. This is an order of magnitude less than the typical number of forwarding rules pushed to switches today. For example, around 500k IPv4 forwarding rules are stored in routers of the internet today.

Undoubtedly, future switches will support larger rule tables [34, 100], but TCAMs still introduce a fundamental trade-off between rule-table size and other concerns like cost and power. TCAMs introduce around 100 times greater cost [15] and 100 times greater power consumption [116], compared to conventional RAM. Plus, updating the rules in TCAM is a slow process—today's hardware switches only support around 40 to 50 rule-table updates per second [59, 66], which could easily constrain a large network with dynamic policies.

Therefore, commodity SDN switches have limited space to enforce fine-grained policy rules which undermines the promise of flexible control. The challenge here is to come up with a solution that works transparently with current controllers and switches without having to wait for the next generation of advances in switch hardware.

1.3.3 Reliability

Traditionally, network operators had to do a lot of work to manually configure various routers in the network with a myriad of parameters to run distributed routing protocols. However, once configured, switches running these protocols can discover link failures and automatically adjust routing around them. In contrast, the centralized controller in SDN is a single point of failure, which is unacceptable. Now, they need to understand and deal with the myriad issues related to consistent replication and fault-tolerance of controller instances in order to implement a logically centralized controller.

Additionally, one cannot simply deploy traditional software replication techniques directly. Maintaining consistent controller state is only part of the solution. To provide a logically centralized controller, one must also ensure that the *switch state* is handled consistently during controller failures. This is because the switch has state related to match-action

rule tables, packet buffers, link failures, etc. Broadly speaking, existing systems do not reason about this switch state; they have not rigorously studied the semantics of processing switch events and executing switch commands under failures.

For example, while the system could roll back the controller state, the switches cannot easily "roll back" to a safe checkpoint. After all, what does it mean to rollback a packet that was already sent? The alternative is for the new master to simply repeat commands, but these commands are not necessarily idempotent (per §4.2). Since an event from one switch can trigger commands to other switches, simultaneous failure of the master controller and a switch can cause inconsistency in the rest of the network. If these issues are not carefully handled, the network can witness erratic behavior like performance degradation or security breaches.

At the same time, running a consensus protocol involving the switches for every event would be prohibitively expensive, given the demand for high-speed packet processing in switches. On the other hand, using distributed storage to replicate controller state alone (for performance reasons) does not capture the switch state precisely. Therefore, after a controller crash, the new master may not know where to resume reconfiguring switch state. Simply reading the switch forwarding state would not provide enough information about all the commands sent by the old master (*e.g.*, PacketOuts, StatRequests).

Given this, the challenge is to build a fault-tolerant controller runtime that takes care of guaranteeing *transactional semantics* to the entire control loop that involves gathering events, processing them and subsequent issue of commands. This way, even under failures, the physically replicated control instances behave as one logical controller. An additional challenge is to remove the burden from the network operator of having to reason about failure and consistency issues and instead write control programs for just one controller while the runtime takes care of correctly replicating it to multiple instances.

1.4 Opportunities for Handling Challenges

While there are several challenges plaguing the current implementations of SDN, there is little doubt that the basic architectural vision behind SDN is sound and desirable in practice. In this section, we discuss how we can take advantage of recent advances in hardware and software dataplanes and past techniques from replicated state machines to tackle the three challenges mentioned earlier — efficiency, flexiblity and reliability.

1.4.1 Programmable Dataplanes for Efficient Utilization

In order to have efficient data plane forwarding that exploits multiple network paths, we need to be able to infer global congestion information and then be able to react to it at dataplane timescales. This means we need the ability to export and process link level utilization information in the dataplane itself instead of using the switch CPU. In addition, we need the ability to dynamically split traffic flows at fine granularity (in order to avoid adverse effects of collisions) and route them instantaneously based on previously gathered information.

In this context, the recent rise of programmable hardware dataplanes fits our requirements perfectly. As opposed to a programmable control plane which dictates which rules to send to switches, programmable dataplanes allow the operator to specify how hardware resources like TCAM, SRAM, packet buffers, etc. should be distributed into multiple tables and registers in a packet processing pipeline. This way, the operator can not only customize the match-action rules but also every stage of the packet processing pipeline in the dataplane. This results in a dataplane that provides a sophisticated hardware platform that can be customized 'in the field' for a wide variety of dataplane algorithms without having to wait for vendor approved ASIC upgrades.

In a programmable dataplane, as shown in Figure 1.3, the switch consists of a programmable parser that parses packets from bits on the wire. Then the packets enter an

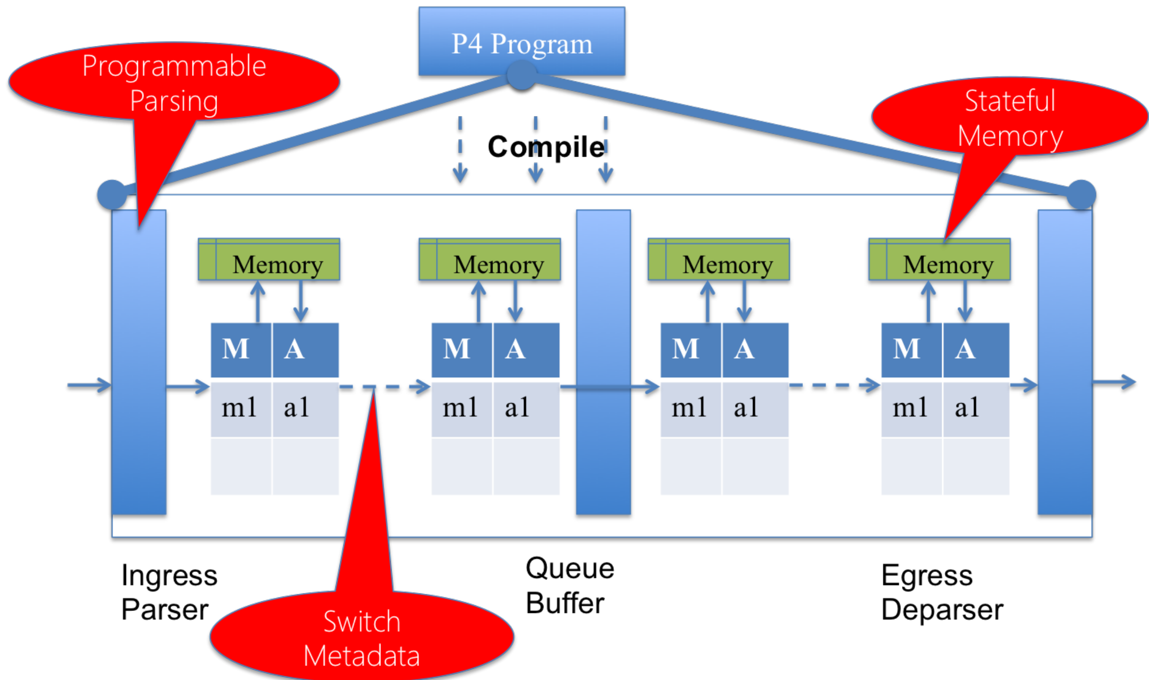


Figure 1.3: Programmable dataplane model

ingress pipeline containing a series of match-action tables that modify packets if they match on specific packet header fields. The most important aspect of the model that is of interest to us is that each table can access stateful memory registers that can be used to read and write state at line rate. This feature can be used to export network utilization values onto packets flowing through a switch. The neighboring switches that receive this packet can then store this information in their own local memory and use it to decide where to send the next set of packets flowing through them.

In this thesis, we will try to exploit such dataplane architectures that allow for global congestion visibility and stateful packet processing entirely in the dataplane. This will mean much faster reaction times for a load balancing scheme, on the order hundreds of microseconds, which matches the round trip time in modern datacenter networks. At the same time, we aim to configure the dataplane using a vendor-agnostic programming interface that can customize a heterogenous set of dataplane targets in a way that adheres to the basic principles of SDN - visibility and flexibility.

1.4.2 Software Switching for Fine-Grained Policies

Limited TCAM availability in switch hardware leads to difficulties implementing a truly flexible control policy. Fortunately, traffic tends to follow a Zipf distribution, where the vast majority of traffic matches a relatively small fraction of the rules [110]. Hence, we could leverage a small TCAM to forward the vast majority of traffic, and rely on alternative datapaths for processing the remaining traffic.

Recent advances in software switching provide one such attractive alternative. Running on commodity servers, software switches can process packets at around 40 Gbps on a quad-core machine [14, 45, 55, 104] and can store large rule tables in main memory and (to a lesser extent) in the L1 and L2 cache. In addition, software switches can update the rule table more than ten times faster than hardware switches [59]. But, supporting wildcard rules that match on many header fields is taxing for software switches, which must resort to slow processing in user space to handle the first packet of each flow [104]. As a result, they cannot match the "horsepower" of hardware switches that provide hundreds of Gbps of packet processing (and high port density).

Thus, based on the Zipf nature of the amount of traffic matching switch rules, we will try to use a combination of hardware and software switching to bring the best of both worlds – high throughput and large rule space. In addition, we need to carefully distribute one rule table into multiple tables spanning across heterogenous datapaths so that the semantics of the single-switch rule table are preserved in the distributed implementation.

1.4.3 Replicated State Machines for Reliable Control

In order to provide reliable control in the face of controller failures, we need to design failover protocols that not only handle controller state but also the switch state. In addition, we need to provide a simple programming abstraction where the network operator need only write a program for a single controller while the runtime manages proper replication.

In this context, a natural choice for such a simple abstraction is that of a replicated state machine where a single state machine is replicated across multiple physical instances for fault tolerance. An example protocol that implements such an abstraction is the Viewstamped Replication protocol [101], a replication technique that handles crash failures using a three stage request processing protocol and a view change protocol that on node failures, depends on a quorum to reconstruct the committed requests.

In the case of SDN controller failure, we need to adopt such replicated state machine protocols for control state replication and then add mechanisms for ensuring the consistency of switch state. In particular, we need the protocol to provide transactional semantics for the entire control loop that is triggered for each network event: event input replication, event processing at each instance and executing resulting commands at the switch. Instead of involving all switches in a consensus protocol, we need to design a light weight replication protocol that keeps the overhead on the switch runtime low while ensuring correctness of the transactional semantics.

1.5 Contributions

As discussed so far, this thesis aims to deal with three important issues related to the practice of Software-Defined Networking. This thesis aims to tackle dataplane efficiency and controller reliability, two issues that arise out of architectural changes imposed by SDN, and it aims to tackle policy flexibility, an issue arising out of constraints posed by switch hardware resources. The solutions proposed in this thesis aim to handle these problems at the appropriate layer in the SDN stack and at acceptable response time scales while keeping the programming abstraction simple to use. Therefore, we make three important contributions as solutions proposed for each of the three problems discussed earlier.

First, I will present HULA [74], which gives the abstraction of an efficient non-blocking switch. Instead of asking the control plane to choose the best path for each new flow, HULA

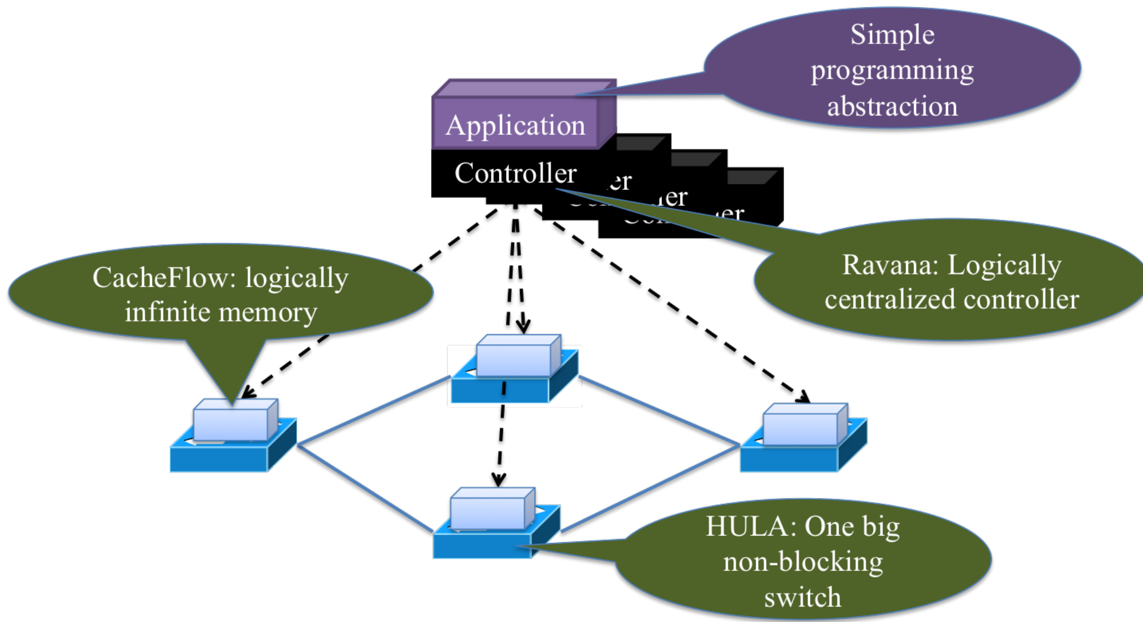


Figure 1.4: Thesis contributions

efficiently routes traffic on least congested paths in the network. HULA uses advanced hardware data plane capabilities to infer global congestion information and uses that information to do fine-grained load balancing at RTT timescales. HULA is congestion-aware, scales to large topologies, and is robust to topology failures.

Second, I will present CacheFlow [73] which helps users implement fine-grained policies by proposing the abstraction of a switch with logically infinite rule space. CacheFlow uses a combination of software and hardware data paths to bring the best of both worlds to policy enforcement. By dynamically caching a small number of heavy hitting rules in the hardware switch and the rest of the rules in the software data path, it achieves both high throughput and high rule capacity. Since cross-rule dependencies make rule caching difficult, CacheFlow uses novel algorithms to do dependency-aware, efficient and transparent rule caching.

Finally, I will present Ravana [75] which gives users the abstraction of one logically centralized controller. Given this abstraction, the network operator only writes programs for one controller and the Ravana runtime takes care of replicating the control logic for fault-tolerance. Since network switches carry additional state external to the controller

state, Ravana uses an enhanced version of traditional replicated state machine protocols to ensure ordered and exactly-once execution of network events.

Taken together, the these three abstractions help operators build network applications on top of a new network architecture where basic routing is done efficiently at dataplane timescales, policy enforcement is done scalably with the help of software data planes, and the control plane is fault-tolerant.

Chapter 2

HULA: Scalable Load Balancing Using Programmable Data Planes

Datacenter networks employ multi-rooted topologies (e.g., Leaf-Spine, Fat-Tree) to provide large bisection bandwidth. These topologies use a large degree of multipathing, and need a data-plane load-balancing mechanism to effectively utilize their bisection bandwidth. The canonical load-balancing mechanism is equal-cost multi-path routing (ECMP), which spreads traffic uniformly across multiple paths. Motivated by ECMP's shortcomings, congestion-aware load-balancing techniques such as CONGA have been developed. These techniques have two limitations. First, because switch memory is limited, they can only maintain a small amount of congestion-tracking state at the edge switches, and do not scale to large topologies. Second, because they are implemented in custom hardware, they cannot be modified in the field.

This chapter presents HULA, a data-plane load-balancing algorithm that overcomes both limitations. First, instead of having the leaf switches track congestion on *all paths* to a destination, each HULA switch tracks congestion for the *best path* to a destination through a neighboring switch . Second, we design HULA for emerging programmable switches and program it in P4 to demonstrate that HULA could be run on such programmable chipsets,

without requiring custom hardware. We evaluate HULA extensively in simulation, showing that it outperforms a scalable extension to CONGA in average flow completion time (1.6× at 50% load, 3× at 90% load).

2.1 Introduction

Data-center networks today have multi-rooted topologies (Fat-Tree, Leaf-Spine) to provide large bisection bandwidth. These topologies are characterized by a large degree of multi-pathing, where there are several routes between any two endpoints. Effectively balancing traffic load across multiple paths in the data plane is critical to fully utilizing the available bisection bandwith. Load balancing also provides the abstraction of a single large output-queued switch for the entire network [24, 69, 103], which in turn simplifies bandwidth allocation across tenants [64, 105], flows [27], or groups of flows [41].

The most commonly used data-plane load-balancing technique is equal-cost multi-path routing (ECMP), which spreads traffic by assigning each flow to one of several paths at random. However, ECMP suffers from degraded performance [23, 31, 38, 68, 112] if two long-running flows are assigned to the same path. ECMP also doesn't react well to link failures and leaves the network underutilized or congested in asymmetric topologies. CONGA [25] is a recent data-plane load-balancing technique that overcomes ECMP's limitations by using link utilization information to balance load across paths. Unlike prior work such as Hedera [23], SWAN [57], and B4 [63], which use a central controller to balance load every few minutes, CONGA is more responsive because it operates in the data plane, permitting it to make load-balancing decisions every few microseconds.

This responsiveness, however, comes at a significant implementation cost. First, CONGA is implemented in custom silicon on a switching chip, requiring several months of hardware design and verification effort. Consequently, once implemented, the CONGA algorithm cannot be modified. Second, memory on a switching chip is at a premium, implying that

CONGA's technique of maintaining per-path congestion state at the leaf switches limits its usage to topologies with a small number of paths. This hampers CONGA's scalability and as such, it is designed only for two-tier Leaf-Spine topologies.

This chapter presents HULA (Hop-by-hop Utilization-aware Load balancing Architecture), a data-plane load-balancing algorithm that addresses both issues.

First, HULA is more scalable relative to CONGA in two ways. One, each HULA switch only picks the next hop, in contrast to CONGA's leaf switches that determine the entire path, obviating the need to maintain forwarding state for a large number of tunnels (one for each path). Two, because HULA switches only choose the best next hop along what is globally the instantaneous best path to a destination, HULA switches only need to maintain congestion state for the best next hop per destination, not all paths to a destination.

Second, HULA is specifically designed for a programmable switch architecture such as the RMT [35], FlexPipe [4], or XPliant [2] architectures. To illustrate this, we prototype HULA in the recently proposed P4 language [33] that explicitly targets such programmable data planes. This allows the HULA algorithm to be inspected and modified as desired by the network operator, without the rigidity of a silicon implementation.

Concretely, HULA uses special probes (separate from the data packets) to gather global link utilization information. These probes travel periodically throughout the network and cover all desired paths for load balancing. This information is summarized and stored at each switch as a table that gives the best next hop towards any destination. Subsequently, each switch updates the HULA probe with its view of the best downstream path (where the best path is the one that minimizes the maximum utilization of all links along a path) and sends it to other upstream switches. This leads to the dissemination of best path information in the entire network similar to a distance vector protocol. In order to avoid packet reordering, HULA load balances at the granularity of *flowlets* [68]— bursts of packets separated by a significant time interval.

To compare HULA with other load-balancing algorithms, we implemented HULA in the network simulator ns-2 [62]. We find that HULA is effective in reducing switch state and in obtaining better flow-completion times compared to alternative schemes on a 3-tier topology. We also introduce asymmetry by bringing down one of the core links and study how HULA adapts to these changes. Our experiments show that HULA performs better than comparative schemes in both symmetric and asymmetric topologies.

In summary, we make the following two key contributions.

- We propose HULA, a scalable data-plane load-balancing scheme. To our knowledge, HULA is the first load balancing scheme to be explicitly designed for a programmable switch data plane.
- We implement HULA in the ns-2 packet-level simulator and evaluate it on a Fat-Tree topology [98] to show that it delivers between 1.6 to 3.3 times better flow completion times than state-of-the-art congestion-aware load balancing schemes at high network load.

2.2 Design Challenges for HULA

Large datacenter networks [21] are designed as multi-tier Fat-Tree topologies. These topologies typically consist of 2-tier Leaf-Spine pods connected by additional tiers of spines. These additional layers connecting the pods can be arbitrarily deep depending on the datacenter bandwidth capacity needed. Load balancing in such large datacenter topologies poses scalability challenges because the explosion of the number of paths between any pair of Top of Rack switches (ToRs) causes three important challenges.

Large path utilization matrix: Table 1 shows the number of paths between any pair of ToRs as the radix of a Fat-Tree topology increases. If a sender ToR needs to track link utilization on all desired paths[1] to a destination ToR in a Fat-Tree topology with radix k,

[1] A path's utilization is the maximum utilization across all its links.

Topology	# Paths between pair of ToRs	# Max forwarding entries per switch
Fat-Tree (8)	16	944
Fat-Tree (16)	64	15,808
Fat-Tree (32)	256	257,792
Fat-Tree (64)	1024	4,160,512

Table 2.1: Number of paths and forwarding entries in 3-tier Fat-Tree topologies [58]

then it needs to track k^2 paths for each destination ToR. If there are m such leaf ToRs, then it needs to keep track of $m * k^2$ entries , which can be prohibitively large. For example, CONGA [25] maintains around 48K bits of memory (512 ToRs, 16 uplinks, and 3 bits for utilization) to store the path-utilization matrix. In a topology with 10K ToRs and with 10K paths between each pair, the ASIC would require 600M bits of memory, which is prohibitively expensive (by comparison the packet data buffer of a shallow-buffered switch such as the Broadcom Trident [3] is 96 Mbits). For the ASIC to be viable and scale with large topologies, it is imperative to reduce the amount of congestion-tracking state stored in any switch.

Large forwarding state: In addition to maintaining per-path utilization at each ToR, existing approaches also need to maintain large forwarding tables in *each* switch to support a leaf-to-leaf tunnel for each path that it needs to route packets over. In particular, a Fat-Tree topology with radix 64 supports a total of 70K ToRs and requires 4 million entries [58] per switch as shown in Table 1. The situation is equally bad [58] in other topologies like VL2 [53] and BCube [54]. To remedy this, recent techniques like Xpath [58] have been designed to reduce the number of entries using compression techniques that exploit symmetry in the network. However, since these techniques rely on the control plane to update and compress the forwarding entries, they are slow to react to failures and topology asymmetry, which are common in large topologies.

Discovering uncongested paths: If the number of paths is large, when new flows enter, it takes time for reactive load balancing schemes to discover an uncongested path especially

when the network utilization is high. This increases the flow completion times of short flows because these flows finish before the load balancer can find an uncongested path. Thus, it is useful to have the utilization information conveyed to the sender in a proactive manner, before a short flow even commences.

Programmability: In addition to these challenges, implementing data-plane load-balancing schemes in hardware can be a tedious process that involves significant design and verification effort. The end product is a one-size-fits-all piece of hardware that network operators have to deploy without the ability to modify the load balancer. The operator has to wait for the next product cycle (which can be a few years) if she wants a modification or an additional feature in the load balancer. An example of such a modification is to load balance based on queue occupancy as in backpressure routing [28, 29] as opposed to link utilization.

The recent rise of programmable packet-processing pipelines [4, 35] provides an opportunity to rethink this design process. These data-plane architectures can be configured through a common programming language like P4 [33], which allow operators to program stateful data-plane packet processing at line rate. Once a load balancing scheme is written in P4, the operator can modify the program so that it fits her deployment scenario and then compile it to the underlying hardware. In the context of programmable data planes, the load-balancing scheme must be simple enough so that it can be compiled to the instruction set provided by a specific programmable switch.

2.3 HULA Overview: Scalable, Proactive, Adaptive, and Programmable

HULA combines distributed network routing with congestion-aware load balancing thus making it tunnel-free, scalable, and adaptive. Similar to how traditional distance-vector routing uses periodic messages between routers to update their routing tables, HULA uses

periodic probes that proactively update the network switches with the best path to any given leaf ToR. However, these probes are processed at line rate entirely in the data plane unlike how routers process control packets. This is done frequently enough to reflect the instantaneous global congestion in the network so that the switches make timely and effective forwarding decisions for volatile datacenter traffic. Also, unlike traditional routing, to achieve fine-grained load balancing, switches split flows into *flowlets* [68] whenever an inter-packet gap of an RTT (network round trip time) is seen within a flow. This minimizes receive-side packet-reordering when a HULA switch sends different flowlets on different paths that were deemed best at the time of their arrival respectively. HULA's basic mechanism of probe-informed forwarding and flowlet switching enables several desirable features, which we list below.

Maintaining compact path utilization: Instead of maintaining path utilization for all paths to a destination ToR, a HULA switch only maintains a table that maps the destination ToR to the best next hop as measured by path utilization. Upon receiving multiple probes coming from different paths to a destination ToR, a switch picks the hop that saw the probe with the minimum path utilization. Subsequently it sends its view of the best path to a ToR to its neighbors. Thus, even if there are multiple paths to a ToR, HULA does not need to maintain per-path utilization information for each ToR. This reduces the utilization state on any switch to the order of the number of ToRs (as opposed to the number of ToRs times the number of paths to these ToRs from the switch), effectively removing the pressure of path explosion on switch memory. Thus, HULA distributes the necessary *global* congestion information to enable scalable *local* routing.

Scalable and adaptive routing: HULA's best hop table eliminates the need for separate source routing in order to exploit multiple network paths. This is because in HULA, unlike other source-routing schemes such as CONGA [25] and XPath [58], the sender ToR isn't responsible for selecting optimal paths for data packets. Each switch independently chooses the best next hop to the destination. This has the additional advantage that switches do not

need separate forwarding-table entries to track tunnels that are necessary for source-routing schemes [58]. This switch memory could be instead be used for supporting more ToRs in the HULA best hop table. Since the best hop table is updated by probes frequently at data-plane speeds, the packet forwarding in HULA quickly adapts to datacenter dynamics, such as flow arrivals and departures.

Automatic discovery of failures: HULA relies on the periodic arrival of probes as a keep-alive heartbeat from its neighboring switches. If a switch does not receive a probe from a neighboring switch for more than a certain threshold of time, then it ages the network utilization for that hop, making sure that hop is not chosen as the best hop for any destination ToR. Since the switch will pass this information to the upstream switches, the information about the broken path will reach all the relevant switches within an RTT. Similarly, if the failed link recovers, the next time a probe is received on the link, the hop will become a best hop candidate for the reachable destinations. This makes for a very fast adaptive forwarding technique that is robust to network topology changes and an attractive alternative to slow routing schemes orchestrated by the control plane.

Proactive path discovery: In HULA, probes are sent separately from data packets instead of piggybacking on them. This lets congestion information be propagated on paths independent of the flow of data packets, unlike alternatives such as CONGA. HULA leverages this to send periodic probes on paths that are not currently used by any switch. This way, switches can instanteously pick an uncongested path on the arrival of a new flowlet without having to first explore congested paths. In HULA, the switches on the path connected to the bottleneck link are bound to divert the flowlet onto a less-congested link and hence a less-congested path. This ensures short flows quickly get diverted to uncongested paths without spending too much time on path exploration.

Programmability: Processing a packet in a HULA switch involves switch state updates at line rate in the packet processing pipeline. In particular, processing a probe involves updating the best hop table and replicating the probe to neighboring switches. Processing

a data packet involves reading the best hop table and updating a flowlet table if necessary. We demonstrate in section 2.5 that these operations can be naturally expressed in terms of reads and writes to match-action tables and register arrays in programmable data planes [7].

Topology and transport oblivious: HULA is not designed for a specific topology. It does not restrict the number of tiers in the network topology nor does it restrict the number of hops or the number of paths between any given pair of ToRs. However, as the topology becomes larger, the probe overhead can also be high and we discuss ways to minimize this overhead in section 2.4. Unlike load-balancing schemes that work best with symmetric topologies, HULA handles topology asymmetry very effectively as we demonstrate in section 3.6. This also makes incremental deployment plausible because HULA can be applied to either a subset of switches or a subset of the network traffic. HULA is also oblivious to the end-host application transport layer and hence does not require any changes to the host TCP stack.

2.4 HULA Design: Probes and Flowlets

The probes in HULA help proactively disseminate network utilization information to all switches. Probes originate at the leaf ToRs and switches replicate them as they travel through the network. This replication mechanism is governed by multicast groups set up once by the control plane. When a probe arrives on an incoming port, switches update the best path for flowlets traveling in the *opposite* direction. The probes also help discover and adapt to topology changes. HULA does all this while making sure the probe overhead is minimal.

In this section, we explain the probe replication mechanism (§2.4.1), the logic behind processing probe feedback (§2.4.2), how the feedback is used for flowlet routing (§2.4.3), how HULA adapts to topology changes (§2.4.4), and finally an estimate of the probe overhead on the network traffic and ways to minimize it (§2.4.5).

We assume that the network topology has the notion of upstream and downstream switches. Most datacenter network topologies have this notion built in them (with switches laid out in multiple tiers) and hence the notion can be exploited naturally. If a switch is in tier i, then the switches directly connected to it in tiers less than i are its *downstream* switches and the switches directly connected to it in tiers greater than i are its upstream switches. For example, in Figure 2.1, $T1$, $T2$ are the downstream switches for $A1$ and $S1$, $S2$ are its upstream switches.

2.4.1 Origin and Replication of HULA Probes

Every ToR sends HULA probes on all the uplinks that connect it to the datacenter network. The probes can be generated by either the ToR CPU, the switch data plane (if the hardware supports a packet generator), or a server attached to the ToR. These probes are sent once every T_p seconds, which is referred to as the probe frequency hereafter in this chapter. For example, in Figure 2.1, probes are sent by ToR $T1$, one on each of the uplinks connecting it to the aggregate switch $A1$.

Once the probes reach $A1$, it will forward the probe to all the other downstream ToRs ($T2$) and all the upstream spines ($S1$, $S2$). The spine $S1$ replicates the received probe onto all the other downstream aggregate switches. However, when the switch $A4$ receives a probe from $S3$, it replicates it to all its downstream ToRs (but not to other upstream spines — $S4$). This makes sure that all paths in the network are covered by the probes. This also makes sure that no probe loops forever.[2] Once a probe reaches another ToR, it ends its journey.

The control plane sets up multicast group tables in the data plane to enable the replication of probes. This is a one-time operation and does not have to deal with link failures and recoveries. This makes it easy to *incrementally* add switches to an existing set of multicast groups for replication. When a new switch is connected to the network, the control plane only needs to add the switch port to multicast groups on the adjacent upstream and

[2] Where the notion of upstream/downstream switches is ambiguous [107], mechanisms like TTL expiry can also be leveraged to make sure HULA probes do not loop forever.

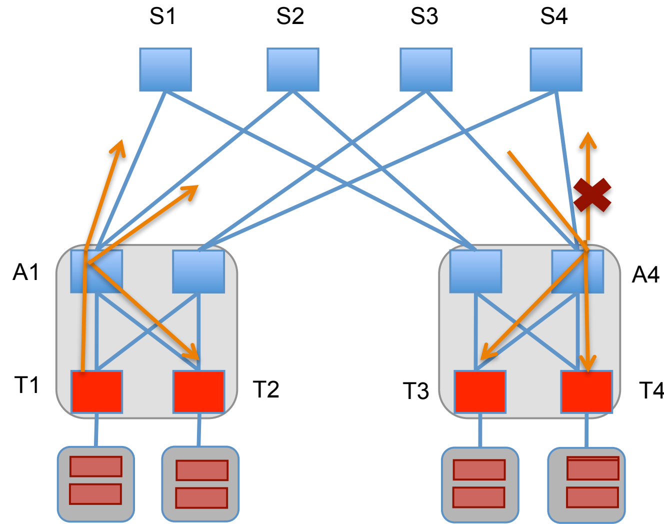


Figure 2.1: HULA probe replication logic

downstream switches, in addition to setting up the multicast mechanism on the new switch itself.

2.4.2 Processing Probes to Update Best Path

A HULA probe packet is a minimum-sized packet of 64 bytes that contains a HULA header in addition to the normal Ethernet and IP headers. The HULA header has two fields:

- **torID (24 bits):** The leaf ToR at which the probe originated. This is the destination ToR for which the probe is carrying downstream path utilization in the opposite direction.
- **minUtil (8 bits):** The utilization of the best path if the packet were to travel in the opposite direction of the probe.

Link utilization: Every switch maintains a link utilization estimator per switch port. This is based on an exponential moving average generator (EWMA) of the form $U = D + U * (1 - \frac{\Delta t}{\tau})$ where U is the link utilization estimator and D is the size of the outgoing packet that triggered the update for the estimator. Δt is the amount of time passed since the last update to the estimator and τ is a time constant that is at least twice the HULA probe frequency. In steady state, this estimator is equal to $C \times \tau$ where C is the outgoing

link bandwidth. As discussed in section 2.5, this is a low pass filter similar to the DRE estimator used in CONGA [25]. We assume that a probe can access the TX (packets sent) utilization of the port that it enters.

A switch uses the information on the probe header and the local link utilization to update switch state in the data plane before replicating the probe to other switches. Every switch maintains a best path utilization table ($pathUtil$) and a best hop table $bestHop$ as shown in Figure 2.2. Both the tables are indexed by a ToR ID. An entry in the $pathUtil$ table gives the utilization of the best path from the switch to a destination ToR. An entry in the $bestHop$ table is the next hop that has the minimum path utilization for the ToR in the $pathUtil$ table. When a probe with the tuple ($torID$, $probeUtil$) enters a switch on interface i, the switch calculates the min-max path utilization as follows:

- The switch calculates the maximum of $probeUtil$ and the TX link utilization of port i and assigns it to $maxUtil$.
- The switch then calculates the minimum of this $maxUtil$ and the $pathUtil$ table entry indexed by $torID$.
- If $maxUtil$ is the minimum, then it updates the $pathUtil$ entry with the newly determined best path utilization value $maxUtil$ and also updates the $bestHop$ entry for $torID$ to i.
- The probe header is updated with the latest $pathUtil$ entry for $torID$.
- The updated probe is then sent to the multicast table that replicates the probe to the appropriate neighboring switches as described earlier.

The above procedure carries out a distance-vector-like propagation of best path utilization information along all the paths destined to a particular ToR (from which the probes originate). The procedure involves each switch updating its local state and then propagating a

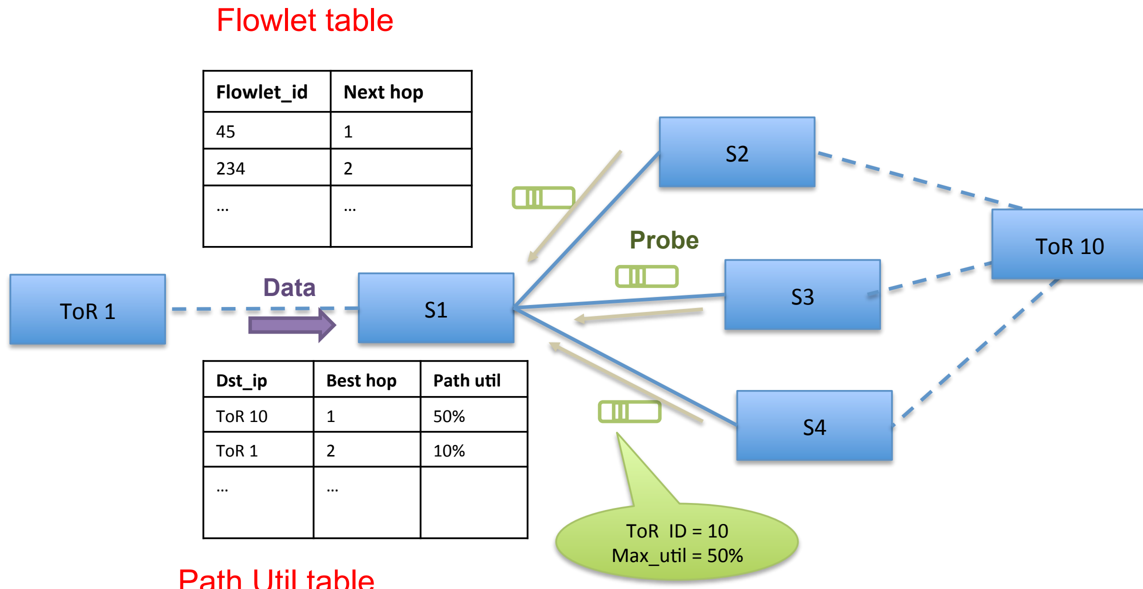


Figure 2.2: HULA probe processing logic

summary of the update to the neighboring switches. This way any switch only knows the utilization of the best path that can be reached via a best next hop and does not need to keep track of the utilization of all the paths. The probe propagation procedure ensures that if the best path changes downstream, then that information will be propagated to all the relevant upstream switches on that path.

Maintaining best hop at line rate: Ideally, we would want to maintain a path utilization matrix that is indexed by both the ToR ID *and* a next hop. This way, the best next hop for a destination ToR can be calculated by taking the minimum of all the next hop utilizations from this matrix. However, programmable data planes cannot calculate the minimum or maximum over an array of entries at line rate [113]. For this reason, instead of calculating the minimum over all hops, we maintain a current best hop and replace it in place when a better probe update is received.

This could lead to transient sub-optimal choices for the best hop entries – since HULA only tracks the current best path utilization, which could potentially go up in the future until a utilization update for the current best hop is received, HULA has no way of tracking other next hop alternatives with lower utilization that were also received within this window

of time. However, we observe that this suboptimal choice can only be transient and will eventually converge to the best choice within a few windows of probe circulation. This approximation also reduces the amount of state maintained per destination from the order of number of neighboring hops to just one hop entry.

2.4.3 Flowlet Forwarding on Best Paths

HULA load balances at the granularity of flowlets in order to avoid packet reordering in TCP. As discussed earlier, a flowlet is detected by a switch whenever the inter-packet gap (time interval between the arrival of two consecutive packets) in a flow is greater than a flowlet threshold T_f. All subsequent packets, until a similar inter-packet gap is detected, are considered part of a new flowlet. The idea here is that the time gap between consecutive flowlets will absorb any delays caused by congested paths when the flowlets are sent on different paths. This will ensure that the flowlets will still arrive in order at the receiver and thereby not cause packet reordering. Typically, T_f is of the order of the network round trip time (RTT). In datacenter networks, T_f is typically of the order of a few hundreds of microseconds but could be larger in topologies with many hops.

HULA uses a flowlet hash table to record two pieces of information:the last time a packet was seen for the flowlet, and the best hop assigned to that flowlet. When the first packet for a flow arrives at a switch, it computes the hash of the flow's 5-tuple and creates an entry in the flowlet table indexed by the hash. In order to choose the best next hop for this flowlet, the switch looks up the $bestHop$ table for the destination ToR of the packet. This best hop is stored in the flowlet table and will be used for all subsequent packets of the flowlet. For example, when the second packet of a flowlet arrives, the switch looks up the flowlet entry for the flow and checks that the inter-packet gap is below T_f. If that is the case, it will use the best hop recorded in the flowlet table. Otherwise, a new flowlet is detected and it replaces the old flowlet entry with the current best hop, which will be used for forwarding the new flowlet.

Flowlet detection and path selection happens at every hop in the network. Every switch selects only the best next hop for a flowlet. This way, HULA avoids an explicit source routing mechanism for forwarding of packets. The only forwarding state required is already part of the $bestHop$ table, which itself is periodically updated to reflect congestion in the entire network.

Bootstrapping forwarding: To begin with, we assume that the path utilization is infinity (a large number in practice) on all paths to all ToRs . This gets corrected once the initial set of probes are processed by the switch. This means that if there is no probe from a certain ToR on a certain hop, then HULA will always choose a hop on which it actually received a probe. Thereafter, once the probes begin circulating in the network before sending any data packets, valid routes are automatically discovered.

2.4.4 Data-Plane Adaptation to Failures

In addition to learning the best forwarding routes from the probes, HULA also learns about link failures from the *absence* of probes. In particular, the data plane implements an aging mechanism for the entries in the $bestHop$ table. HULA tracks the last time $bestHop$ was updated using an $updateTime$ table. If a $bestHop$ entry for a destination ToR is not refreshed within the last T_{fail} (a threshold for detecting failures), then any other probe that carries information about this ToR (from a different hop) will simply replace the $bestHop$ and $pathUtil$ entries for the ToR. When this information about the change in the best path utilization is propagated further up the path, the switches may decide to choose a completely disjoint path if necessary to avoid the bottleneck link.

This way, HULA does not need to rely on the control plane to detect and adapt to failures. Instead HULA's failure-recovery mechanism is much faster than control-plane-orchestrated recovery, and happens at network RTT timescales. Also, note that this mechanism is better than having pre-coded backup routes because the flowlets immediately get forwarded on the next best alternative path as opposed to congestion-oblivious pre-installed backup

paths. This in turn helps avoid sending flowlets on failed network paths and results in better network utilization and flow-completion times.

2.4.5 Probe Overhead and Optimization

The ToRs in the network need to send HULA probes frequently enough so that the network receives fine-grained information about global congestion state. However, the frequency should be low enough so that the network is not overwhelmed by probe traffic alone.

Setting probe frequency: We observe that even though network feedback is received on every packet, CONGA [25] makes flowlet routing decisions with probe feedback that is stale by an RTT because it takes a round trip time for the (receiver-reflected) feedback to reach the sender. In addition to this, the network switches only use the congestion information to make load balancing decisions when a new flowlet arrives at the switch. For a flow scheduled between any pair of ToRs, the best path information between these ToRs is used only when a new flowlet is seen in the flow, which happens at most once every T_f seconds. While it is true that flowlets for different flows arrive at different times, any flowlet routing decision is still made with probe feedback that is stale by at least an RTT. Thus, a reasonable sweet spot is to set the probe frequency to the order of the network RTT. In this case, the HULA probe information will be stale by at most a few RTTs and will still be useful for making quick decisions.

Optimization for probe replication: HULA also optimizes the number of probes sent from any switch A to an adjacent switch B. In the naive probe replication model, A sends a probe to neighbor B whenever it receives a probe on another incoming interface. So in a time window of length T_p (probe frequency), there can be multiple probes from A to B carrying the best path utilization information for a given ToR T, if there are multiple paths from T to A. HULA suppresses this redundancy to make sure that for any given ToR T, only one probe is sent by A to B within a time window of T_p. HULA maintains a *lastSent* table indexed by ToR IDs. A replicates a probe update for a ToR T to B only if the last

probe for T was sent more than T_p seconds ago. Note that this operation is similar to the calculation of a flowlet gap and can be done in constant time in the data plane.[3] Thus, by making sure that on any link, only one probe is sent per destination ToR within this time window, the total number of probes that are sent on any link is proportional to the number of ToRs in the network alone and is not dependent on the number of possible paths the probes may take.

Overhead: Given the above parameter setting for the probe frequency and the optimization for probe replication, the probe overhead on any given network link is $\frac{probeSize*numToRs*100}{probeFreq*linkBandwidth}$ where *probeSize* is 64 bytes, *numTors* is the total number of leaf ToRs supported in the network and *probeFreq* is the HULA probe frequency. Therefore, in a network with 40G links supporting a total of 1000 ToRs, with probe frequency of $1ms$, the overhead comes to be 1.28%.

2.5 Programming HULA in P4

2.5.1 Introduction to P4

P4 is a packet-processing language designed for programmable data-plane architectures like RMT [35], Intel Flexpipe [4], and Cavium Xpliant [2]. The language is based on an abstract forwarding model called protocol-independent switch archtecture (PISA) [9]. In this model, the switch consists of a programmable parser that parses packets from bits on the wire. Then the packets enter an ingress pipeline containing a series of match-action tables that modify packets if they match on specific packet header fields. The packets are then switched to the output ports. Subsequently, the packets are processed by another sequence of match-action tables in the egress pipeline before they are serialized into bytes and transmitted.

[3] If a probe arrives with the latest best path (after this bit is set), we are still assured that this best path information will be replicated (and propagated) in the next window assuming it still remains the best path.

A P4 program specifies the the protocol *header format*, a *parse graph* for the various headers, the definitions of tables with their match and action formats and finally the *control flow* that defines the order in which these tables process packets. This program defines the configuration of the hardware at compile time. At runtime, the tables are populated with entries by the control plane and network packets are processed using these rules. The programmer writes P4 programs in the syntax described by the P4 specification [7].

Programming HULA in P4 allows a network operator to compile HULA to any P4 supported hardware target. Additionally, network operators have the flexibility to modify and recompile their HULA P4 program as desired (changing parameters and the core HULA logic) without having to invest in new hardware. The wide industry interest in P4 [5] suggests that many switch vendors will soon have P4 compilers from P4 to their switch hardware, permitting operators to program HULA on such switches in the future.

2.5.2 HULA in P4

We describe the HULA packet processing pipeline using version 1.1 of P4 [7]. We make two minor modifications to the specification for the purpose of programming HULA.

1. We assume that the link utilization for any output port is available in the ingress pipeline. This link utilization can be computed using a low-pass filter applied to packets leaving a particular output port, similar to the Discounting Rate Estimator (DRE) used by CONGA [25]. At the language level, a link utilization object is syntactically similar to counter/meter objects in P4.

2. Based on recent proposals [8] to modify P4, we assume support for the conditional operator within P4 actions.[4]

[4]For ease of exposition, we replace conditional operators with equivalent if-else statements in Figure 2.4.

```
header_type hula_header {
     fields{
          dst_tor : 24;
          path_util : 8;
     }
}

header_type metadata{
     fields{
          nxt_hop : 8;
          self_id : 32;
          dst_tor : 32;
     }
}

control ingress {
     apply(get_dst_tor)
     apply(hula_logic)
     if(ipv4.protocol == PROTO_HULA){
          apply(hula_mcast);
     }
     else if(metadata.dst_tor
               === metadata.self_id) {
          apply(send_to_host);
     }
}
```

Figure 2.3: HULA header format and control flow

We now describe various components of the HULA P4 program in Figure 2.4. The P4 program has two main components: one, the HULA probe header format and parser specification, and two, packet control flow, which describes the main HULA logic.

Header format and parsing: We define the P4 header format for the probe packet and the parser state machine as shown in Figure 2.4(a). The header consists of two fields and is of size 4 bytes. The parser parses the HULA header immediately after the IPv4 header based on the special HULA protocol number in the IPv4 protocol field. Thereafter, the header fields are accessible in the pipeline through the header instance. The metadata header is used to access packet fields that have special meaning to a switch pipeline (e.g., the next hop) and local variables to be carried across multiple tables (e.g, a data packet's destination ToR or the current switch ID).

The control flow in Figure 2.4(a) shows that the processing pipeline first finds the ToR that the incoming packet is destined to. This is done by the `get_dst_tor` table that matches on the destination IP address and retrieves the destination ToR ID. Then the packet is processed by the `hula_logic` table whose actions are defined in Figure 2.4(b). Subsequently, the

```
1  action hula_logic{
2    if(ipv4_header.protocol == IP_PROTOCOLS_HULA){
3           /*HULA Probe Processing
4        if(hula_hdr.path_util < tx_util)
5            hula_hdr.path_util = tx_util;
6        if(hula_hdr.path_util < min_path_util[hula_hdr.dst_tor] ||
7            curr_time - update_time[dst_tor] > KEEP_ALIVE_THRESH)
8        {
9            min_path_util[dst_tor] = hula_hdr.path_util;
10           best_hop[dst_tor] = metadata.in_port;
11           update_time[dst_tor] = curr_time;
12       }
13       hula_header.path_util = min_path_util[hula_hdr.dst_tor];
14   }
15   else { /*Flowlet routing of data */
16       if(curr_time - flowlet_time[flow_hash]> FLOWLET_TOUT) {
17           flowlet_hop[flow_hash] = best_hop[metadata.dst_tor];
18       }
19       metadata.nxt_hop = flowlet_hop[flow_hash];
20       flowlet_time[flow_hash] = curr_time;
21   }
22 }
```

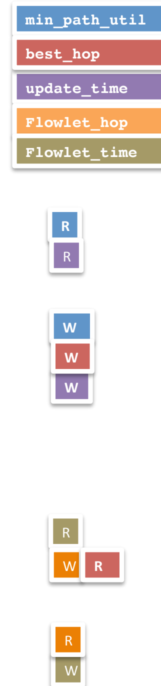


Figure 2.4: HULA stateful packet process in P4

probe is sent to the `hula_mcast` table that matches on the in_port the probe came in and then assigns the appropriate set of multicast ports for replication.

HULA pipeline logic: Figure 2.4(b) shows the main HULA table where a series of actions perform two important pieces of HULA logic — (i) Processing HULA *probes* and (ii) Flowlet forwarding for *data* packets. We briefly describe how these two are expressed in P4. At a high level, the `hula_logic` table reads and writes to five register data structures shown in Figure 2.4(b) — `path_util`, `best_hop`, `update_time`, `flowlet_hop` and `flowlet_time`. The reads and writes performed by each action are color coded in the figure. For example, the red colored write tagging line 9 indicates that the action makes a write access to the `best_hop` register array.

1. Processing HULA probes: In step 1, the path utilization being carried by the HULA probe is updated (lines 4-5) with the the maximum of the local link utilization (`tx_util`) and the probe utilization. This gives the path utilization across all the hops including the

link connecting the switch to its next hop. Subsequently, the current best path utilization value for the ToR is read from the `min_path_util` register into a temporary metadata variable (line 5).

In the next step, if either the probe utilization is less than the current best path utilization (line 6) or if the best hop was not refreshed in the last failure detection window (line 7), then three updates take place - (i) The best path utilization is updated with the probe utilization (line 9), (ii) the best hop value is updated with the incoming interface of the probe (line 10), and (iii) the best hop refresh time is updated with the current timestamp (line 11). Finally, the probe utilization itself is updated with the final best hop utilization (line 13). Subsequently the probe is processed by the `hula_mcast` match-action table that matches on the probe's input port and then assigns the appropriate multicast group for replication.

2. Flowlet forwarding: If the incoming packet is a data packet (line 15), first we detect new flowlets by checking if the inter-packet gap for that flow is above the flowlet threshold (line 16). If that is the case, then we use the current best hop to reach the destination ToR (line 17). Subsequently, we populate the next hop metadata with the final flowlet hop (line 19). Finally, the arrival time of the packet is noted as the last seen time for the flowlet (line 20).

The benefits of programmability: Writing a HULA program in P4 gives multiple advantages to a network operator compared to a dedicated ASIC implementation. The operator could modify the sizes of various registers according to her workload demands. For example, she could change the sizes of the `best_hop` and `flowlet` register arrays based on her requirements. More importantly, she could change the way the algorithm works by modifying the HULA header to carry and process queue occupancy instead of link utilization to implement backpressure routing [28, 29].

2.5.3 Feasibility of P4 Primitives at Line Rate

In the P4 program shown in Figure 2.4, we require both stateless (i.e., operations that only read or write packet fields) and stateful (i.e., operations that may also maniupulate switch state in addition to packet fields) operations to program HULA's logic. We briefly comment on the hardware feasibility of each kind of operation below.

The stateless operations used in the program (like the assignment operation in line 4) are relatively easy to implement and have been discussed before [35]. In particular, Table 1 of the RMT paper [35] lists many stateless operations that are feasible on a programmable switch architecture with forwarding performance competitive with the highest-end fixed-function switches.

For determining the feasibility of stateful operations, we use techniques developed in Domino [113], a recent system that allows stateful data-plane algorithms such as HULA to be compiled to line-rate switches. The Domino compiler takes as inputs a data-plane algorithm and a set of *atoms*, which represent a programmable switch's instruction set. Based on the atoms supported by a programmable switch, Domino determines if a data-plane algorithm can be run on a line-rate switch. The same paper also proposes atoms that are expressive enough for a variety of data-plane algorithms, while incurring $< 15\%$ estimated chip area overhead. Table 3 of the Domino paper [113] lists these atoms.

We now discuss how the stateful operations required by each of HULA's five state variables `min_path_util`, `best_hop`, `update_time`, `flowlet_hop`, and `flowlet_time`, can be supported by Domino's atoms (the atom names used here are from Table 3 of the Domino paper [113]).

1. Both `flowlet_time` and `update_time` track the last time at which some event happened, and require only a simply read/write capability to a state variable (the Read/Write atom).

2. The `flowlet_hop` variable is conditionally updated whenever the flowlet threshold is exceeded. This requires the ability to predicate a write to a state variable based on some condition (the PRAW atom).

3. The variables `min_path_util` and `best_hop` are mutually dependent on one another: `min_path_util` (the utilization on the best hop) needs to be updated if a new probe is received for the current `best_hop` (the variable tracking the best next hop) ; conversely, the `best_hop` variable needs to be updated if a probe for another path indicates a utilization lesser than the current `min_path_util`. This mutually dependence requires hardware support for updating a *pair* of state variables depending on the previous values of the pair (the Pairs atom).

The most complex of these three atoms (Read/Write, PRAW, and Pairs) is the Pairs atom. However, even the Pairs atom only incurs modest estimated chip area overhead based on synthesis results from a 32 nm standard-cell library. Further, this atom is useful for other algorithms besides HULA as well (Table 4 of the Domino paper describes several more examples). We conclude based on these results that it is feasible to implement the instructions required by HULA without sacrificing the performance of a line-rate switch.

2.6 Evaluation

In this section, we illustrate the effectiveness of the HULA load balancer by implementing it in the ns-2 discrete event simulator and comparing it with the following alternative load balancing schemes:

1. **ECMP:** Each flow's next hop is determined by taking a hash of the flow's five tuple (src IP, dest IP, src port, dest port, protocol).

2. **CONGA':** CONGA [25] is the closest alternative to HULA for congestion-aware data-plane load balancing. However, CONGA is designed specifically for 2-tier

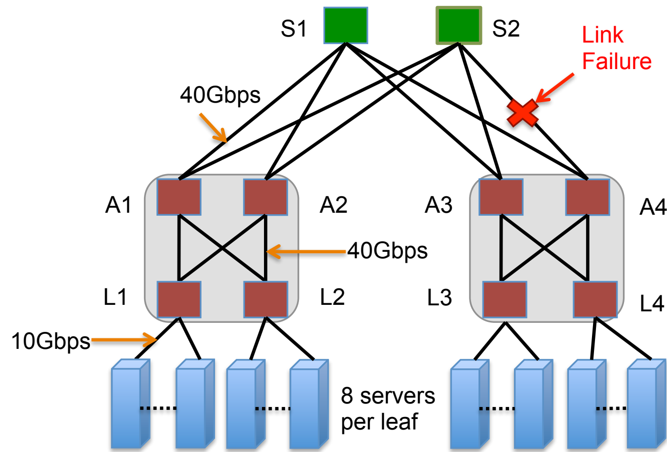


Figure 2.5: Topology used in evaluation

Leaf-Spine topologies. However, according to the authors [1], if CONGA is to be extended to larger topologies, CONGA should be applied within each pod and for cross-pod traffic, ECMP should be applied at the flowlet level. This method involves taking a hash of the six tuple that includes the flow's five tuple and the flowlet ID (which is incremented every time a new flowlet is detected at a switch). This hash is subsequently used by all the switches in the network to find the next hop for each flowlet. We refer to this load balancing scheme as CONGA' in our evaluation results.

We use our experiments to answer the following questions:

- How does HULA perform in the baseline topology compared to other schemes?
- How does HULA perform when there is asymmetry in the network?
- How quickly does HULA adapt to changes in the network like link failures?
- How robust is HULA to various parameters settings?

Topology: We simulated a 3-tier Fat-Tree topology as shown in Figure 2.5, with two spines (S1 and S2) connecting two *pods*. Each pod contains two aggregate switches connected to two leaf ToRs with 40G links. Each ToR is connected to 8 servers with 10G links. This ensures that the network is not oversubscribed: the 16 servers in one pod can

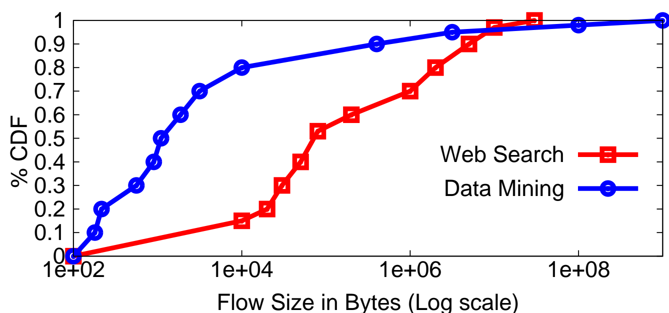
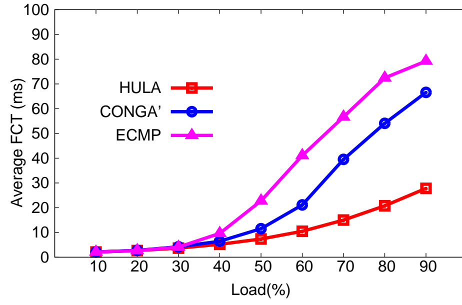


Figure 2.6: Empirical traffic distribution used in evaluation

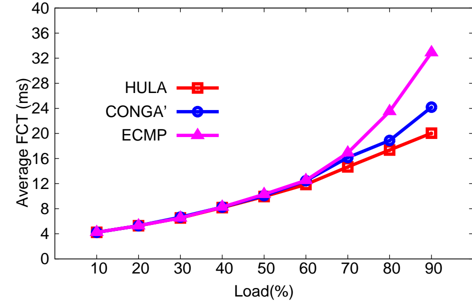
together use the 160G bandwidth available for traffic across the two pods. In this topology, even though there are only two uplinks from any given ToR, there are a total of 8 different paths available between a pair of ToRs sitting in different pods. To simulate asymmetry in the baseline symmetric topology, we disable the 40G link connecting the spine $S2$ with the aggregate switch $A4$.

Empirical workload: We use two realistic workloads to generate traffic for our experiments - (i) A Web-search workload [26] and (ii) a data-mining workload [53]. Both of these workloads are obtained from production datacenters. Figure 2.6 shows the cumulative distribution of flow sizes seen in these two workloads. Note that flow sizes in the CDF are in log scale. Both the workloads are heavy tailed: most flows are small, while a small number of large flows contribute to a substantial portion of the traffic. For example, in the data mining workload, 80% of the flows are of size less than 10KB.

We simulate a simple client-server communication model where each client chooses a server at random and initiates three persistent TCP connections to the server. The client sends a flow with size drawn from the empirical CDF of one of the two workloads. The inter-arrival rate of the flows on a connection is also taken from an exponential distribution whose mean is tuned to achieve a desired load on the network. Similar to previous work [25, 27], we look at the average flow completion time (FCT) as the overall performance metric



(a) Web-search overall avg FCT



(b) Data-mining overall avg FCT

Figure 2.7: Average flow completion times for the Web-search and data-mining workload on the *symmetric* topology.

so that all flows including the majority of small flows are given equal consideration. We run each experiment with three random seeds and then measure the average of the three runs.

Parameters: In our experimental setting, there are two important parameters that determine the system behavior. First, the flowlet inter-packet gap, as is recommended in previous work [25, 68], is set to be of the order of the network RTT so that packet reordering at the receiver is minimized. In our experiments, we used a flowlet gap of 100 µs . The second parameter is the probe frequency, which (as mentioned in §2.4.5) is set to few times the RTT so that it is frequent enough to quickly react to congestion but does not overwhelm the network. In our experiments, unless stated explicitly, the probe frequency was set to 200 µs.

2.6.1 Symmetric 3-tier Fat-Tree Topology

Figure 2.7 shows the average completion time for all flows as the load on the network is varied. HULA performs better than ECMP and CONGA' for both the workloads at higher loads. At lower loads, the performance of all three load balancing schemes is nearly the same because when there is enough bandwidth available in the network, there is a greater

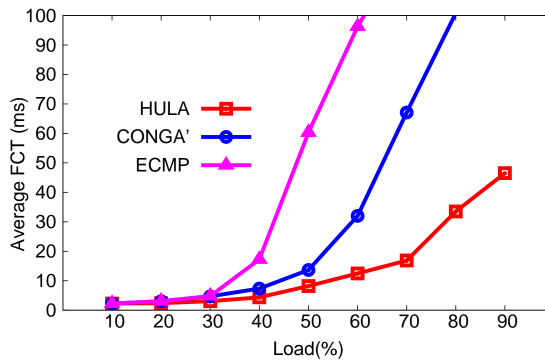
tolerance for congestion-oblivious path forwarding. However, as the network load becomes higher, the flows have to be carefully assigned to paths such that collisions do not occur. Given that flow characteristics change frequently, at high network load, the load balancing scheme has to adapt quickly to changes in link utilizations throughout the network.

ECMP performs the worst because it performs congestion-oblivious load balancing at a very coarse granularity. CONGA' does slightly better because it still does congestion-oblivious ECMP (across pods) but at the granularity of flowlets. In particular, flows sent on congested paths see more inter-flowlet gaps being created due to the delay caused by queue growth. Hence, compared to ECMP, CONGA' has additional opportunities to find an uncongested path when new flowlets are hashed. HULA performs the best because of its fine-grained congestion-aware load balancing. For the Web-search workload, HULA achieves 3.7x lower FCT (better performance) compared to ECMP and 2.7x better compared to CONGA' at 70% network load. The performance of HULA is slightly less apparent in the data mining workload because a vast portion of the flows in the workload are really small (50% are just 1 packet flows) and HULA does not often get a chance to better load balance large flows with multiple flowlets. Nevertheless, HULA achieves 1.35x better performance than ECMP at 80% network load.

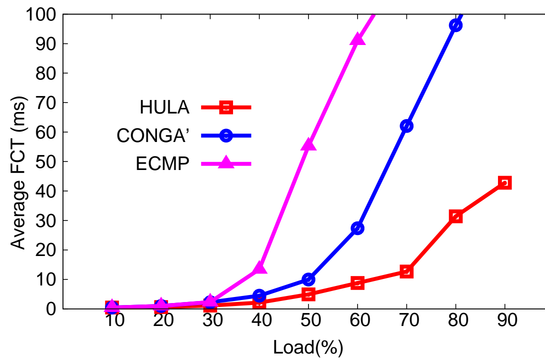
2.6.2 Handling Topology Asymmetry

When the link between the spine switch $S2$ and switch $A4$ is removed, the effective bandwidth of the network drops by 25% for traffic going across the pods. This means that the load balancing schemes have to carefully balance paths at even lower network loads compared to the baseline topology scenario. In particular, the load balancing scheme has to make sure that the bottleneck link connecting $S2$ to $A3$ is not overwhelmed with a disproportionate amount of traffic.

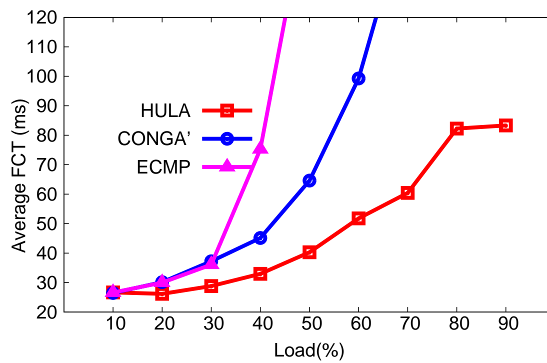
Figure 2.8 shows how various schemes perform with the Web-search workload as the network load is varied. The overall FCT for ECMP rises quickly and goes off the charts



(a) Overall Average FCT



(b) Small Flows (of size <100KB)



(c) Large Flows (of size >10MB)

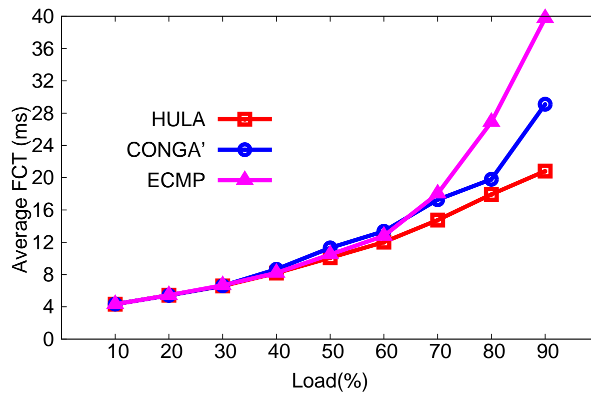
Figure 2.8: Average FCT for the Web-search workload on the *asymmetric* topology.

beyond a 60% network load. Once the network load reaches 50%, the bottleneck link incurs pressure from the flows hashed to go through $S2$. This is why ECMP and CONGA' have bad performance at high network loads. CONGA' does slightly better than ECMP here because the network sees more flowlets being created on congested paths (due to the delays caused by queue growth) and hence has a slightly higher chance of finding the uncongested paths for new flowlets. Because of this, CONGA' is 3x better than ECMP at 60% load. However, HULA performs the best because of its proactive utilization-aware path selection, which avoids pressure on the bottleneck link. This helps HULA achieve 8x better performance at 60% network load.

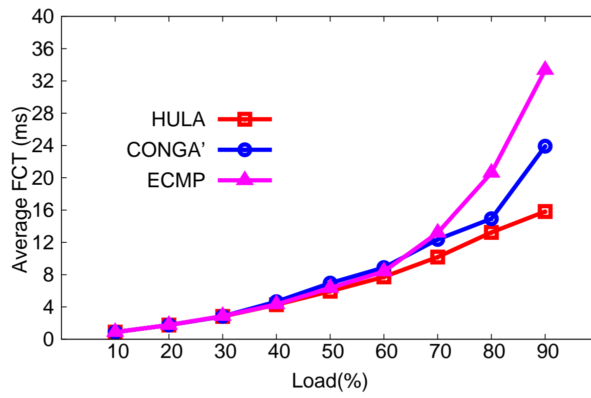
Figure 2.8(b) shows the average FCTs for small flows of size less than 100KB and Figure 2.8(c) shows the average FCTs for large flows of size greater than 10MB. HULA's gains are most pronounced on the large number of small flows where it does 10x better than ECMP at 60% load. Even for large flows, HULA is 4x better than ECMP at 60% load.

HULA prevents queue growth: The superior performance of HULA can be understood by looking at the growth of switch queues. As described earlier, in the link failure scenario, all the traffic that crosses the pod through the spine $S2$ has to go through the link connecting it to $A3$, which becomes the bottleneck link at high network load. Figure 2.10c shows the CDF of queue depth at the bottleneck link. The queue was monitored every 100 microseconds and the instantaneous queue depth was plotted. ECMP has high depth most of the time and frequently sees packet drops as well. HULA on the other hand maintains zero queue depth 90% of the time and sees no packet drops. In addition, the 95th percentile queue depth for HULA is 8x smaller compared to CONGA' and 19x smaller compared to ECMP.

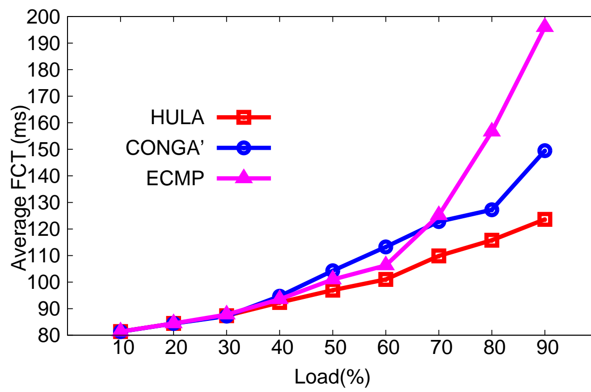
Figure 2.9 shows that HULA's gains are less pronounced with the data mining workload similar to what was seen with the baseline topology. Due to the extremely large number of small flows, the effect of congestion-aware load balancing is less pronounced. Nevertheless, HULA does the best with small flows having 1.53x better performance than ECMP at



(a) Overall Average FCT



(b) Small Flows (of size <100KB)



(c) Large Flows (of size >10MB)

Figure 2.9: Average FCT for the data mining workload on the *asymmetric* topology.

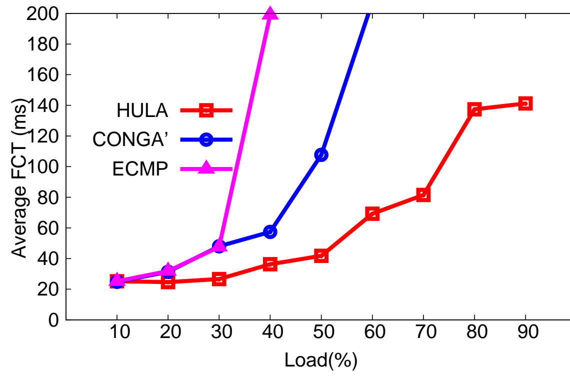
80% load. With large flows, it does 1.35x better than ECMP. Overall, HULA does 1.52x better than ECMP and 1.17x better than CONGA'.

HULA achieves better tail latency: In addition to performing better on average FCT, HULA also achieves good tail latency for both workloads. Figure 2.10 shows the 99th percentile FCT for all the flows. For the Web-search workload, HULA achieves 10x better 99th percentile FCT compared to ECMP and 3x better compared to CONGA' at 60% load. For the data mining workload, HULA achieves 1.53x better tail latency compared to ECMP.

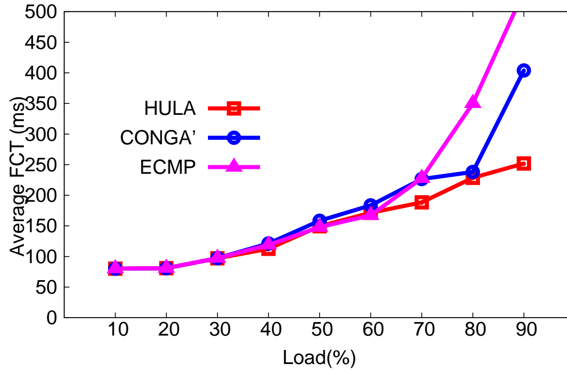
2.6.3 Stability

In order to study HULA's stability in response to topology changes, we monitored the link utilization of the links that connect the spine to the aggregate switches in the asymmetric topology while the Web-search workload is running. We then brought down the bottleneck link at 0.2 milliseconds from the beginning of the experiment. As Figure 4.8c(a) shows, HULA quickly adapts to the failure and redistributes the load onto the two links going through $S1$ within a millisecond. Then when the failed link comes up later, HULA quickly goes back to the original utilization values on all the links. This demonstrates that HULA is robust to changes in the network topology and also shows that the load is distributed almost equally on all the available paths at any given time regardless of the topology.

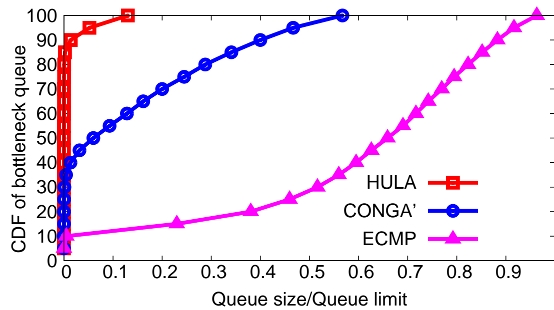
Figure 4.8c(b) shows a similar experiment but run with long-running flows as opposed to the empirical workload. Long-running flows allow us to study HULA's stability better than empirical workloads, because in an empricial workload the link utilizations may fluctuate depending on flow arrivals and departures. As the figure shows, when the link connecting a spine to an aggregate switch fails, HULA quickly deflects the affected flows onto another available path within half a millisecond. Further, while doing this, it does not disturb the bottleneck link and cause instability in the network.



(a) 99th percentile FCT for Web-search workload

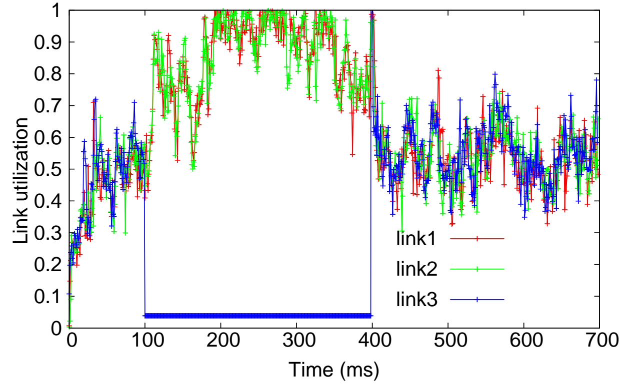


(b) 99th percentile FCT for datamining

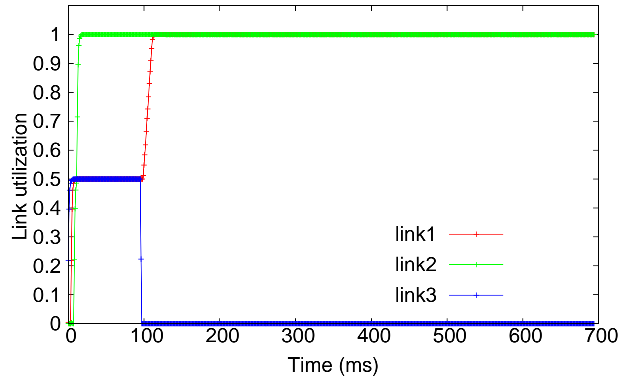


(c) Bottleneck queue length (S2->A3) in the link failure scenario

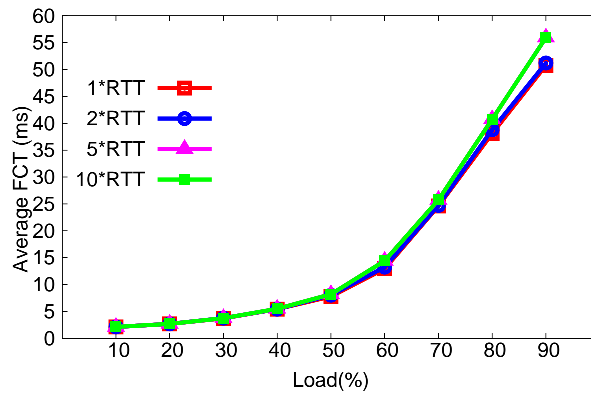
Figure 2.10: 99th percentile FCTs and queue growth on the asymmetric topology



(a) Link utilization on failures with Web-search workload



(b) Link utilization on failures with long running flows



(c) Effect of decreasing probe frequency

Figure 2.11: HULA resilience to link failures and probe frequency settings

2.6.4 Robustness of probe frequency

As discussed earlier, carrying probes too frequently can reduce the effective network bandwidth available for data traffic. While we argued that the ideal frequency is of the order of the network RTT, we found that HULA is robust to change in probe frequency. Figure 4.8c(c) shows the average FCT with the Web-search workload running on the asymmetric topology. When the network load is below 70%, increasing the probe frequency to 10 times its ideal has no effect on the performance. Even at 90% load, the average FCT for 10x frequency is only 1.15x higher. In addition, compared with ECMP and CONGA', these numbers are much better. Therefore, we believe HULA probes can be circulated with moderately low frequency so that the effective bandwidth is not affected while still achieving utilization-aware load balancing.

2.7 Related Work

Stateless or local load balancing: Equal-Cost Multi-Path routing (ECMP) is a simple hash-based load-balancing scheme that is implemented widely in switch ASICs today. However, it is congestion-agnostic and only splits traffic at the flow level, which causes collisions at high network load. Further, ECMP is shown to have degraded performance during link failures that cause asymmetric topologies [25]. DRB [38] is a per-packet load balancing scheme that sprays packets effectively in a round robin fashion. More recently, PRESTO [56] proposed splitting flows into TSO (TCP Segment Offload) segments of size 64KB and sending them on multiple paths. On the receive side GRO (General Receive Offload), the packets are buffered temporarily to prevent reordering. Neither DRB nor Presto is congestion aware, which causes degraded performance during link failures. Flare [68] and Localflow [112] discuss switch-local solutions that balance the load on all switch ports but do not take global congestion information into account.

Centralized load balancing: B4 [63] and SWAN [57] propose centralized load balancing for wide-area networks connecting their data centers. They collect statistics from network switches at a central controller and push forwarding rules to balance network load. The control plane operates at the timescale of minutes because of relatively predictable traffic patterns. Hedera [23] and MicroTE [31] propose similar solutions for datacenter networks but still suffer from high control-loop latency in the critical path and cannot handle highly volatile datacenter traffic in time.

Modified transport layer: MPTCP [108] is a modified version of TCP that uses multiple subflows to split traffic over different paths. However, the multiple subflows cause burstiness and perform poorly under Incast-like conditions [25]. In addition, it is difficult to deploy MPTCP in datacenters because it requires change to all the tenant VMs, each of which might be running a different operating system. DCTCP [26], pFabric [27] and PIAS [30] reduce the tail flow completion times using modified end-host transport stacks but do not focus on load balancing. DeTail [125] proposes a per-packet adaptive load balancing scheme that adapts to topology asymmetry but requires a complex cross-layer network stack including end-host modifications.

Global utilization-aware load balancing: TeXCP [67] and MATE [47] are adaptive traffic-engineering proposals that load balance across multiple ingress-egress paths in a wide-area network based on per-path congestion metrics. TeXCP also does load balancing at the granularity of flowlets but uses router software to collect utilization information and uses a modified transport layer to react to this information. HALO [92], inspired by a long line of work beginning with Minimum Delay Routing [50], studies load-sensitive adaptive routing as an optimization problem and implements it in the router software. Relative to these systems, HULA is a routing mechanism that balances load at finer granularity and is simple enough to be implemented entirely in the data plane.

As discussed earlier, CONGA [25] is the closest alternative to HULA for global congestion-aware fine-grained load balancing. However, it is designed for specific 2-

tier Leaf-Spine topologies in a custom ASIC. HULA, on the other hand, scales better than CONGA by distributing the relevant utilization information across all switches. In addition, unlike CONGA, HULA reacts to topology changes like link failures almost instantaneously using data-plane mechanisms. Lastly, HULA's design is tailored towards programmable switches—a first for data-plane load balancing schemes.

2.8 Conclusion

In this chapter, we design HULA (Hop-by-hop Utilization-aware Load balancing Architecture), a scalable load-balancing scheme designed for programmable data planes. HULA uses periodic probes to perform a distance-vector style distribution of network utilization information to switches in the network. Switches track the next hop for the best path and its corresponding utilization for a given destination, instead of maintaining per-path utilization congestion information for each destination. Further, because HULA performs forwarding locally by determining the next hop and not an entire path, it eliminates the need for a separate source routing mechanism (and the associated forwarding table state required to maintain source routing tunnels). When failures occur, utilization information is automatically updated so that broken paths are avoided.

We evaluate HULA against existing load balancing schemes and find that it is more effective and scalable. While HULA is effective enough to quickly adapt to the volatility of datacenter workloads, it is also simple enough to be implemented at line rate in the data plane on emerging programmable switch architectures. While the performance and stability of HULA is studied empirically in this chapter, an analytical study of its optimality and stability will provide further insights into its dynamic behavior.

Chapter 3

CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks

Software-Defined Networking (SDN) allows control applications to install fine-grained forwarding policies in the underlying switches. While Ternary Content Addressable Memory (TCAM) enables fast lookups in hardware switches with flexible wildcard rule patterns, the cost and power requirements limit the number of rules the switches can support. To make matters worse, these hardware switches cannot sustain a high rate of updates to the rule table.

In this chapter, we show how to give applications the illusion of high-speed forwarding, large rule tables, and fast updates by combining the best of hardware and software processing. Our CacheFlow system "caches" the most popular rules in the small TCAM, while relying on software to handle the small amount of "cache miss" traffic. However, we cannot blindly apply existing cache-replacement algorithms, because of dependencies between rules with overlapping patterns. Rather than cache large chains of dependent rules, we "splice" long dependency chains to cache smaller groups of rules while preserving the

semantics of the policy. Experiments with our CacheFlow prototype—on both real and synthetic workloads and policies—demonstrate that rule splicing makes effective use of limited TCAM space, while adapting quickly to changes in the policy and the traffic demands.

3.1 Introduction

In a Software-Defined Network (SDN), a logically centralized controller manages the flow of traffic by installing simple packet-processing rules in the underlying switches. These rules can match on a wide variety of packet-header fields, and perform simple actions such as forwarding, flooding, modifying the headers, and directing packets to the controller. This flexibility allows SDN-enabled switches to behave as firewalls, server load balancers, network address translators, Ethernet switches, routers, or anything in between. However, fine-grained forwarding policies lead to a large number of rules in the underlying switches.

In modern hardware switches, these rules are stored in Ternary Content Addressable Memory (TCAM) [16]. A TCAM can compare an incoming packet to the patterns in all of the rules at the same time, at line rate. However, commodity switches support relatively few rules, in the small thousands or tens of thousands [117]. Undoubtedly, future switches will support larger rule tables, but TCAMs still introduce a fundamental trade-off between rule-table size and other concerns like cost and power. TCAMs introduce around 100 times greater cost [15] and 100 times greater power consumption [116], compared to conventional RAM. Plus, updating the rules in TCAM is a slow process—today's hardware switches only support around 40 to 50 rule-table updates per second [59, 66], which could easily constrain a large network with dynamic policies.

Software switches may seem like an attractive alternative. Running on commodity servers, software switches can process packets at around 40 Gbps on a quad-core machine [14, 45, 55, 104] and can store large rule tables in main memory and (to a lesser extent) in the L1 and L2 cache. In addition, software switches can update the rule table more than ten times faster than hardware switches [59]. But, supporting wildcard rules that match on many header fields is taxing for software switches, which must resort to slow processing (such as a linear scan) in user space to handle the first packet of each flow [104]. So, they cannot match the "horsepower" of hardware switches that provide hundreds of Gbps of packet processing (and high port density).

Fortunately, traffic tends to follow a Zipf distribution, where the vast majority of traffic matches a relatively small fraction of the rules [110]. Hence, we could leverage a small TCAM to forward the vast majority of traffic, and rely on software switches for the remaining traffic. For example, an 800 Gbps hardware switch, together with a single 40 Gbps software packet processor could easily handle traffic with a 5% "miss rate" in the TCAM. In addition, most rule-table updates could go to the slow-path components, while promoting very popular rules to hardware relatively infrequently. Together, the hardware and software processing would give controller applications the illusion of high-speed packet forwarding, large rule tables, and fast rule updates.

Our CacheFlow architecture consists of a TCAM and a *sharded* collection of software switches, as shown in Figure 3.1. The software switches can run on CPUs in the data plane (e.g., network processors), as part of the software agent on the hardware switch, or on separate servers. CacheFlow consists of a CacheMaster module that receives OpenFlow commands from an *unmodified* SDN controller. CacheMaster preserves the semantics of the OpenFlow interface, including the ability to update rules, query counters, etc. CacheMaster uses the OpenFlow protocol to distribute rules to *unmodified* commodity hardware and software switches. CacheMaster is a purely *control-plane* component, with control sessions shown as dashed lines and data-plane forwarding shown by solid lines.

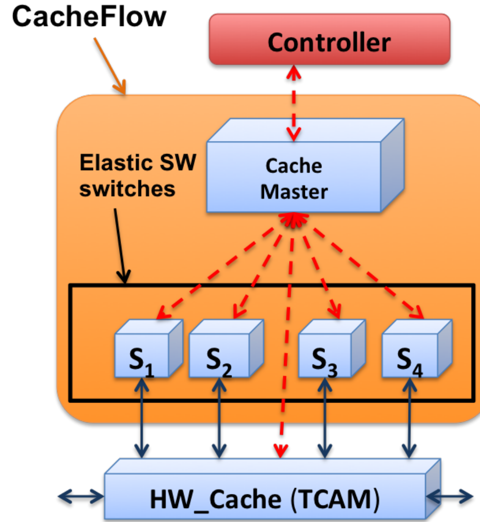


Figure 3.1: CacheFlow architecture

As the name suggests, CacheFlow treats the TCAM as a *cache* that stores the most popular rules. However, we cannot simply apply existing cache-replacement algorithms, because the rules can match on overlapping sets of packets, leading to dependencies between multiple rules. Indeed, the switch we used for our experiments makes just this mistake (See §5)—a bug now addressed by our new system! Moreover, while earlier work on IP route caching [48, 79, 85, 110] considered rule dependencies, IP prefixes only have simple "containment" relationships, rather than patterns that *partially* overlap. The partial overlaps can also lead to long dependency chains, and this problem is exacerbated by applications that combine multiple functions (like server load balancing and routing, as can be done in Frenetic [49] and CoVisor [65]) to generate many more rules.

To handle rule dependencies, we construct a compact representation of the given prioritized list of rules as an annotated directed acyclic graph (DAG), and design incremental algorithms for adding and removing rules to this data structure. Our cache-replacement algorithms use the DAG to decide which rules to place in the TCAM. To preserve rule-table space for the rules that match a large fraction of the traffic, we design a novel "splicing" technique that breaks long dependency chains. Splicing creates a few new rules that "cover"

a large number of unpopular rules, to avoid polluting the cache. The technique extends to handle changes in the rules, as well as changes in their popularity over time.

In summary, we make the following key technical contributions:

- **Incremental rule-dependency analysis:** We develop an algorithm for incrementally analyzing and maintaining rule dependencies.
- **Novel cache-replacement strategies:** We develop new algorithms that only cache heavy-hitting rules along with a small set of dependencies.
- **Implementation and evaluation:** We discuss how CacheFlow preserves the semantics of the OpenFlow interface. Our experiments on both synthetic and real workloads show a cache-hit rate of 90% of the traffic by caching less than 5% of the rules.

A preliminary version of this work appeared as a workshop paper [72] which briefly discussed ideas about rule dependencies and caching algorithms. In this chapter, we develop novel algorithms that help efficiently deal with practical deployment constraints like incremental updates to policies and high TCAM update times. We also evaluate CacheFlow by implementing large policies on actual hardware.

3.2 Identifying Rule Dependencies

In this section, we show how rule dependencies affect the correctness of rule-caching techniques and where such dependencies occur. We show how to represent cross-rule dependencies as a graph, and present efficient algorithms for incrementally computing the graph.

3.2.1 Rule Dependencies

The OpenFlow policy on a switch consists of a set of packet-processing rules. Each rule has a pattern, a priority, a set of actions, and counters. When a packet arrives, the switch identifies the highest-priority matching rules, performs the associated actions and incre-

Rule	(dst_ip, dst_port)	Ternary Match	Action	Priority	Weight
R1	(10.10.10.10/32, 10)	000	Fwd 1	6	10
R2	(10.10.10.10/32, *)	00*	Fwd 2	5	60
R3	(10.10.0.0/16, *)	0**	Fwd 3	4	30
R4	(11.11.11.11/32, *)	11*	Fwd 4	3	5
R5	(11.11.0.0/16, 10)	1*0	Fwd 5	2	10
R6	(11.11.10.10/32, *)	10*	Fwd 6	1	120

(a) Example rule table

(b) Incremental DAG insert

(c) Incremental DAG delete

Figure 3.2: Constructing the rule dependency graph (edges annotated with reachable packets)

ments the counters. CacheFlow implements these policies by splitting the set of rules into two groups—one residing in the TCAM and another in a software switch.

The semantics of CacheFlow is that (1) the highest-priority matching rule in the TCAM is applied, if such a rule exists, and (2) if no matching rule exists in the TCAM, then the highest-priority rule in the software switch is applied. As such, not all splits of the set of rules lead to valid implementations. If we do not cache rules in the TCAM *correctly*, packets that should hit rules in the software switch may instead hit a cached rule in the TCAM, leading to incorrect processing.

In particular, *dependencies* may exist between rules with differing priorities, as shown in the example in Figure 3.2(a). If the TCAM can store four rules, we cannot select the four rules with highest traffic volume (i.e., R_2, R_3, R_5, and R_6), because packets that should match R_1 (with pattern 000) would match R_2 (with pattern 00*); similarly, some packets

(say with header 110) that should match R_4 would match R_5 (with pattern 1*0). That is, rules R_2 and R_5 *depend* on rules R_1 and R_4, respectively. In other words, there is a dependency from rule R_1 to R_2 and from rule R_4 to R_5. If R_2 is cached in the TCAM, R_1 should also be cached to preserve the semantics of the policy, similarly with R_5 and R_4.

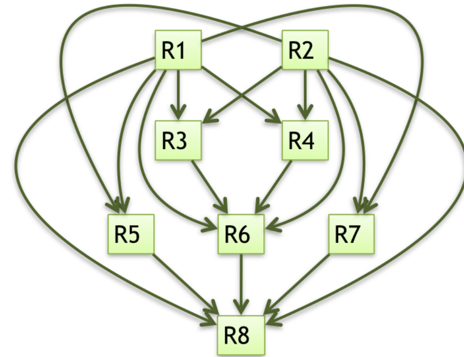
A *direct* dependency exists between two rules if the patterns in the rules intersect (e.g., R_2 is dependent on R_1). When a rule is cached in the TCAM, the corresponding dependent rules should also move to the TCAM. However, simply checking for intersecting patterns does *not* capture all of the policy dependencies. For example, going by this definition, the rule R_6 only depends on rule R_5. However, if the TCAM stored only R_5 and R_6, packets (with header 110) that should match R_4 would inadvertently match R_5 and hence would be incorrectly processed by the switch. In this case, R_6 also depends *indirectly* on R_4 (even though the matches of R_4 and R_6 do *not* intersect), because the match for R_4 overlaps with that of R_5. Therefore we need to define carefully what constitutes a dependency to handle such cases properly.

3.2.2 Where do complex dependencies arise?

Partial overlaps. Complex dependencies do not arise in traditional destination prefix forwarding because a prefix is dependent only on prefixes that are strict subsets of itself (nothing else). However, in an OpenFlow rule table, where rules have priorities and can match on multiple header fields, indirect dependencies occur because of partial overlaps between rules—both R_4 and R_6 only partially overlap with R_5. Hence, even though R_4 and R_6 do not have a direct dependency, they have an indirect dependency due to R_5's own dependence on R_6. The second column of Figure 3.2(a) illustrates such a situation. Here, one might interpret the first two bits of R_4, R_5, and R_6 as matching a destination IP, and the last bit as matching a port. Note that it is not possible to simply transform these rules into large FIB tables that are supported by today's switches because FIB lookups match on a single

Rule	Priority	Match
R1	32800	tcp;dport:179
R2	32800	tcp;dport:646
R3	16640	dstip:111.221.69.0/24
R4	16640	dstip:111.221.66.0/24
R5	16630	dstip:111.221.78.0/23
R6	16610	dstip:111.221.64.0/21
R7	16610	dstip:111.221.112.0/21
R8	16580	dstip:111.221.64.0/18

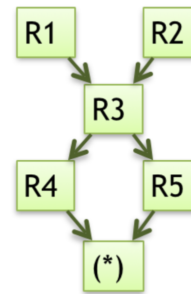
(a) Reanzz Rule Table



(b) Reanzz Subgraph

Rule	Priority	Match
R1	5	srcip = 1.0.0.0/24 dstip = 2.0.0.1
R2	4	srcip = 1.0.0.0/24 dstip = 2.0.0.2
R3	3	srcip = 1.0.0.0/24
R4	2	dstip = 2.0.0.1
R5	1	dstip = 2.0.0.2

(c) CoVisor Example



(d) CoVisor Graph

Figure 3.3: Dependent-set vs. cover-set algorithms (L_0 cache rules in red)

header field. Even if one were to somehow split a wild card rule into multiple FIB rules, it does not preserve counters.

Policy composition. Frenetic [49], Pyretic [96], CoVisor [65] and other high-level SDN programming platforms support abstractions for constructing complex network policies from a collection of simpler components. While the separate components may exhibit few dependencies, when they are compiled together, the composite rule tables may contain many complex dependencies. For instance, Figure 3.3(c) presents an illustrative rule table drawn from the CoVisor project [65]. The corresponding dependency graph for the rules is shown in Figure 3.3(d). Here, the dependency between rules R_3 and R_4 was not seen in the

two separate components that defined the high-level policy, but does arise when they are composed.

Dependencies in REANNZ policies. We also analyzed a number of policies drawn from real networks to determine the nature of the dependencies they exhibit. As an example, Figure 3.3(a) shows part of an OF policy in use at the REANNZ research network [10]. The corresponding dependency graph can be seen in Figure 3.3(b). Now, one might conjecture that a network operator could manually rewrite the policy to reduce the number of dependencies and thereby facilitate caching. However, doing so is bound to be extremely tedious and highly error prone. Moreover, a good split may depend on the dynamic properties of network traffic. We argue that such tasks are much better left to algorithms, such as the ones we propose in this chapter. An expert can develop a single caching algorithm, validate it and then deploy it on any policy. Such a solution is bound to be more reliable than asking operators to manually rewrite policies.

Is this a temporary problem? Even if the TCAM available on future switches grows, network operators will only become greedier in utilizing these resources—in the same way that with increasing amounts of DRAM, user applications have begun to consume increasing amounts of memory in conventional computers. Newer switches have multi-table pipelines [34] that can help avoid rule blowup from policy composition but the number of rules is still limited by the available TCAM. Our algorithms can be used to cache rules independently in each table (which maximizes the cache-hit traffic across all tables). Further, even high-end software switches like the OpenVSwitch (OVS) spend considerable effort [104] to cache popular OpenFlow rules in the kernel so that majority of the traffic does not get processed by the user-level classifier which is very slow. Thus, the rise of software switches may not completely avoid the problem of correctly splitting rule dependencies to cache them in the faster classifier process. Thus we believe our efforts are widely applicable and are going to be relevant in the long term despite the near term industry trends.

3.2.3 Constructing the Dependency DAG

Algorithm 1: Building the dependency graph

```
   // Add dependency edges
1  func addParents(R:Rule, P:Parents) begin
2      deps = (∅);
       // p.o : priority order
3      packets = R.match;
4      for each R_j in P in descending p.o: do
5          if (packets ∩ R_j.match) != ∅ then
6              deps = deps ∪ {(R,R_j)};
7              reaches(R,R_j) = packets ∩ R_j;
8              packets = packets - R_j.match;
9      return deps;
10 for each R:Rule in Pol:Policy do
11     potentialParents = [R_j in Pol — R_j.p.o ≤ R.p.o];
12     addParentEdges(R, potentialParents);
```

A concise way to capture all the dependencies in a rule table is to construct a directed graph where each rule is a node, and each edge captures a direct dependency between a pair of rules as shown in Figure 3.2(b). A direct dependency exists between a child rule R_i and a parent rule R_j under the following condition—if R_i is removed from the rule table, packets that are supposed to hit R_i will now hit rule R_j. The edge between the rules in the graph is annotated by the set of packets that reach the parent from the child. Then, the dependencies of a rule consist of all descendants of that rule (e.g., R_1 and R_2 are the dependencies for R_3). The rule R_0 is the default *match-all* rule (matches all packets with priority 0) added to maintain a connected rooted graph without altering the overall policy.

To identify the edges in the graph, for any given child rule R, we need to find out all the parent rules that the packets matching R can reach. This can be done by taking the symbolic set of packets matching R and iterating them through all of the rules with lower priority than R that the packets might hit.

To find the rules that depend directly on R, Algorithm 1 scans the rules R_i with lower priority than R (line 14) in order of decreasing priority. The algorithm keeps track of the set of packets that can reach each successive rule (the variable `packets`). For each such new rule, it determines whether the predicate associated with that rule intersects[1] the set of packets that can reach that rule (line 5). If it does, there is a dependency. The arrow in the dependency edge points from the child R to the parent R_i. In line 7, the dependency edge also stores the packet space that actually reaches the parent R_i. In line 8, before searching for the next parent, because the rule R_i will now occlude some packets from the current `reaches` set, we subtract R_i's predicate from it.

This compact data structure captures *all* dependencies because we track the flow of all the packets that are processed by any rule in the rule table. The data structure is a directed acyclic graph (DAG) because if there is an edge from R_i to R_j then the priority of R_i is always strictly greater than priority of R_j. Note that the DAG described here is *not* a topological sort (we are not imposing a total order on vertices of a graph but are computing the edges themselves). Once such a dependency graph is constructed, if a rule R is to be cached in the TCAM, then all the descendants of R in the dependency graph should also be cached for correctness.

3.2.4 Incrementally Updating The DAG

Algorithm 1 runs in $\mathcal{O}(n^2)$ time where n is the number of rules. As we show in Section 3.6, running the static algorithm on a real policy with 180K rules takes around 15 minutes, which is unacceptable if the network needs to push a rule into the switches as quickly as possible (say, to mitigate a DDoS attack). Hence we describe an incremental algorithm that has considerably smaller running time in most practical scenarios—just a few milliseconds for the policy with 180K rules.

[1] Symbolic intersection and subtraction of packets can be done using existing techniques [76].

Figure 3.2(b) shows the changes in the dependency graph when the rule R_5 is inserted. All the changes occur only in the right half of the DAG because the left half is not affected by the packets that hit the new rule. A rule insertion results in three sets of updates to the DAG: (i) existing dependencies (like (R_4,R_0)) change because packets defining an existing dependency are impacted by the newly inserted rule, (ii) creation of dependencies with the new rule as the parent (like (R_4,R_5)) because packets from old rules (R_4) are now hitting the new rule (R_5), and (iii) creation of dependencies (like (R_5,R_6)) because the packets from the new rule (R_5) are now hitting an old rule (R_6). Algorithm 1 takes care of all three dependencies by it rebuilding *all* dependencies from scratch. The challenge for the incremental algorithm is to do the same set of updates without touching the irrelevant parts of the DAG — In the example, the left half of the DAG is not affected by packets that hit the newly inserted rule.

Incremental Insert

In the incremental algorithm, the intuition is to use the `reaches` variable (packets reaching the parent from the child) cached for each existing edge to recursively traverse only the necessary edges that need to be updated. Algorithm 2 proceeds in three phases:

(i) Updating existing edges (lines 1–10): While finding the affected edges, the algorithm recursively traverses the dependency graph beginning with the default rule R_0. It checks if the newRule intersects any edge between the current node and its children. It updates the intersecting edge and adds it to the set of affected edges (line 4). However, if newRule is higher in the priority chain, then the recursion proceeds exploring the edges of the next level (line 9). It also collects the rules that could potentially be the parents as it climbs up the graph (line 8). This way, we end up only exploring the relevant edges and rules in the graph.

(ii) Adding directly dependent children (lines 11-15): In the second phase, the set of affected edges collected in the first phase are grouped by their children. For each child, an

Algorithm 2: Incremental DAG insert

```
1  func FindAffectedEdges(rule, newRule) begin
2      for each C in Children(rule) do
3          if Priority(C) > priority(newRule) then
4              if reaches(C,rule) ∩ newRule.match != ∅ then
5                  reaches(C, rule) -= newRule.match;
6                  add (C, Node) to affEdges
7          else
8              if Pred(C) ∪ newRule.match != ∅ then
9                  add C to potentialParents;
10                 FindAffectedEdges(C, newRule);
11 func processAffectedEdges(affEdges) begin
12     for each childList in groupByChild(affEdges) do
13         deps = deps ∪ {(child, newRule)};
14         edgeList = sortByParent(childList);
15         reaches(child, newRule) = reaches(edgeList[0]);
16 func Insert(G=(V, E), newNode) begin
17     affEdges = { };
18     potentialParents = [R0];
19     FindAffectedEdges(R0, newNode);
20     ProcessAffectedEdges(affEdges);
21     addParents(newNode, potentialParents);
```

edge is created from the child to the newRule using the packets from the child that used to reach its highest priority parent (line 14). Thus all the edges from the new rule to its children are created.

(iii) Adding directly dependent parents (line 21): In the third phase, all the edges that have newRule as the child are created using the `addParents` method described in Algorithm 1 on all the potential parents collected in the first phase.

In terms of the example, in phase 1, the edge (R_4, R_0) is the affected edge and is updated with `reaches` that is equal to 111 (11* - 1*0). The rules R_0 and R_6 are added to the new rule's potential parents. In phase 2, the edge (R_4, R_5) is created. In phase 3, the function `addParents` is executed on parents R_6 and R_0. This results in the creation of edges (R_5, R_6) and (R_5, R_0).

Algorithm 3: Incremental DAG delete

```
1 func Delete(G=(V, E), oldRule) begin
2     for each c in Children(oldRule) do
3         potentialParents = Parents(c) - {oldRule};
4         for each p in Parents(oldRule) do
5             if reaches(c, oldRule) ∩ p.match != ∅ then
6                 add p to potentialParents
7         addParents(C, potentialParents)
8     Remove all edges involving oldRule
```

Running Time: Algorithm 2 clearly avoids traversing the left half of the graph which is not relevant to the new rule. While in the worst case, the running time is linear in the number of edges in the graph, for most practical policies, the running time is linear in the number of closely related dependency groups[2].

Incremental Delete

The deletion of a rule leads to three sets of updates to a dependency graph: (i) new edges are created between other rules whose packets used to hit the removed rule, (ii) existing edges are updated because more packets are reaching this dependency because of the absence of the removed rule, and (iii) finally, old edges having the removed rule as a direct dependency are deleted.

For the example shown in Figure 3.2(c), where the rule R_5 is deleted from the DAG, existing edges (like (R_4, R_0)) are updated and all three involving R_5 are created. In this example, however, no new edge is created. But it is potentially possible in other cases (consider the case where rule R_2 is deleted which would result in a new edge between R_1 and R_3).

An important observation is that unlike an incremental insertion (where we recursively traverse the DAG beginning with R_0), incremental deletion of a rule can be done local to the rule being removed. This is because all three sets of updates involve only the children

[2]Since the dependency graph usually has a wide bush of isolated prefix dependency chains—like the left half and right half in the example DAG—which makes the insertion cost equal to the number of such chains.

or parents of the removed rule. For example, a new edge can only be created between a child and a parent of the removed rule[3].

Algorithm 3 incrementally updates the graph when a new rule is deleted. First, in lines 2-6, the algorithm checks if there is a new edge possible between any child-parent pair by checking whether the packets on the edge (child, oldRule) reach any parent of oldRule (line 5). Second, in lines 3 and 7, the algorithm also collects the parents of all the existing edges that may have to be updated (line 3). It finally constructs the new set of edges by running the `addParents` method described in Algorithm 1 to find the exact edges between the child c and its parents (line 7). Third, in line 8, the rules involving the removed rule as either a parent or a child are removed from the DAG.

Running time: This algorithm is dominated by the two `for` loops (in lines 2 and 4) and may also have a worst case $\mathcal{O}(n^2)$ running time (where n is the number of rules) but in most practical policy scenarios, the running time is much smaller (owing to the small number of children/parents for any given rule in the DAG).

3.3 Caching Algorithms

In this section, we present CacheFlow's algorithm for placing rules in a TCAM with limited space. CacheFlow selects a set of important rules from among the rules given by the controller to be cached in the TCAM, while redirecting the cache misses to the software switches.

We first present a simple strawman algorithm to build intuition, and then present new algorithms that avoids caching low-weight rules. Each rule is assigned a "cost" corresponding to the number of rules that must be installed together and a "weight" corresponding to the number of packets expected to hit that rule [4]. Continuing with the running example

[3] In the example where R_2 is deleted, a new rule can only appear between R_1 and R_3. Similarly when R_5 is deleted, a new rule could have appeared between R_4 and R_6 but does not because the rules do not overlap.

[4] In practice, weights for rules are updated in an online fashion based on the packet count in a sliding window of time.

from the previous section, R_6 depends on R_4 and R_5, leading to a cost of 3, as shown in Figure 3.4(a). In this situation, R_2 and R_6 hold the majority of the weight, but cannot be installed simultaneously on a TCAM with capacity 4, as installing R_6 has a cost of 3 and R_2 bears a cost of 2. Hence together they do not fit. The best we can do is to install rules R_1, R_4, R_5, and R_6 which maximizes total weight, subject to respecting all dependencies.

3.3.1 Optimization: NP Hardness

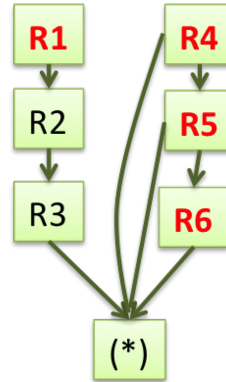
The input to the rule-caching problem is a dependency graph of n rules $R_1, R_2, \ldots, R_n$, where rule R_i has higher priority than rule R_j for $i < j$. Each rule has a match and action, and a weight w_i that captures the volume of traffic matching the rule. There are dependency edges between pairs of rules as defined in the previous section. The output is a prioritized list of C rules to store in the TCAM[5]. The objective is to maximize the sum of the weights for traffic that "hits" in the TCAM, while processing "hit" packets according to the semantics of the original rule table.

$$
\begin{array}{ll}
\textit{Maximize} & \sum_{i=1}^{n} w_i c_i \\
\text{subject to} & \sum_{i=1}^{n} c_i \leq C;\ c_i \in \{0,1\} \\
 & c_i - c_j \geq 0 \text{ if } R_i.\textit{is_descendant}(R_j)
\end{array}
$$

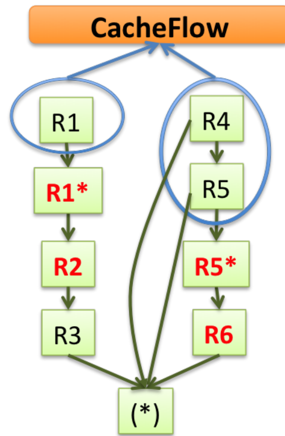
The above optimization problem is NP-hard in n and k. It can be reduced from the densest k-subgraph problem which is known to be NP-hard. We outline a sketch of the reduction here between the decision versions of the two problems. Consider the decision problem for the caching problem: Is there a subset of C rules from the rule table which respect the directed dependencies and have a combined weight of atleast W. The decision problem for the densest k-subgraph problem is to ask if there is a subgraph incident on k vertices that

[5]Note that CacheFlow does *not* simply install rules on a cache miss. Instead, CacheFlow makes decisions based on traffic measurements over the recent past. This is important to defend against cache-thrashing attacks where an adversary generates low-volume traffic spread across the rules.

Rule	Cost	Hits
R1	1	10
R2	2	0
R3	3	0
R4	1	5
R5	2	10
R6	3	120

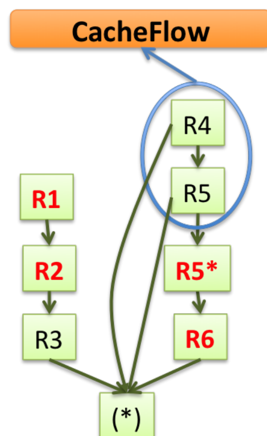


(a) Dependent Set Algo.



Rule	Match	Action	Hits
R1	000	Fwd 1	0
R1*	000	To_SS	0
R2	00*	Fwd 2	60
R3	0**	Fwd 3	0
R4	11*	Fwd 4	0
R5	1*0	Fwd 5	0
R5*	1*0	To_SS	0
R6	10*	Fwd 6	120

(b) Cover Set Algo.



Rule	Match	Action	Hits
R1	000	Fwd 1	10
R2	00*	Fwd 2	60
R3	0**	Fwd 3	0
R4	11*	Fwd 4	0
R5	1*0	Fwd 5	0
R5*	1*0	To_SS	0
R6	10*	Fwd 6	120

(c) Mixed Set Algo

Figure 3.4: Dependent-set vs. cover-set algorithms (L_0 cache rules in red)

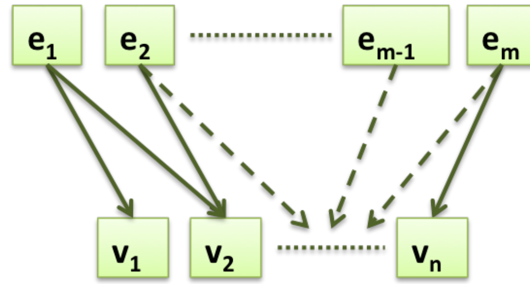


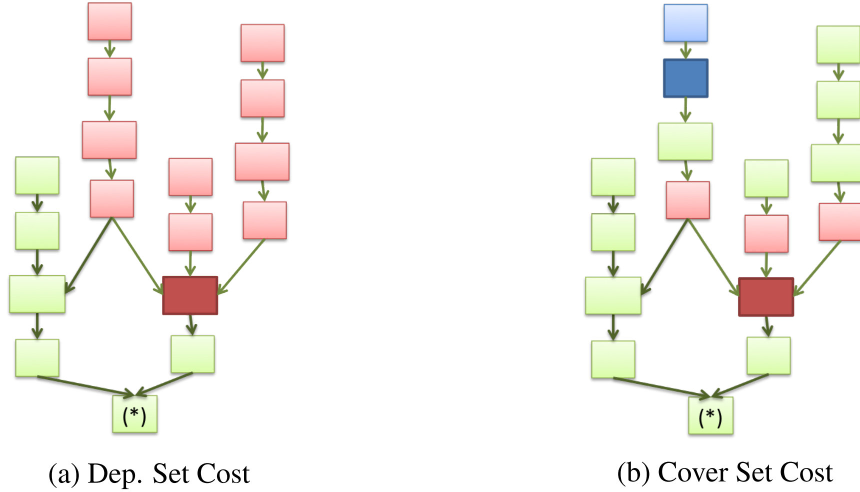
Figure 3.5: Reduction from densest k-subgraph

has at least d edges in a given undirected graph G=(V,E) (This generalizes the well known CLIQUE problem for d=$\binom{k}{2}$, hence is hard).

Consider the reduction shown in Figure 3.5. For a given instance of the densest k-subgraph problem with parameters k and d, we construct an instance of the cache-optimization problem in the following manner. Let the vertices of G' be nodes indexed by the vertices and edges of G. The edges of G' are constructed as follows: for every undirected edge $e = (v_i, v_j)$ in G, there is a directed edge from e to v_i and v_j. This way, if e is chosen to include in the cache, v_i and v_j should also be chosen. Now we assign weights to nodes in V' as follows : $w(v) = 1$ for all $v \in V$ and $w(e) = n+1$ for all $e \in E$. Now let $C = k+d$ and $W = d(n+1)$. If you can solve this instance of the cache optimization problem, then you have to choose at least d of the edges $e \in E$ because you cannot reach the weight threshold with less than d edge nodes (since their weight is much larger than nodes indexed by V). Since C cannot exceed $d+k$, because of dependencies, one will also end up choosing less than k vertices $v \in V$ to include in the cache. Thus this will solve the densest k-subgraph instance.

3.3.2 Dependent-Set: Caching Dependent Rules

No polynomial time approximation scheme (PTAS) is known yet for the densest k-subgraph problem. It is also not clear whether a PTAS for our optimization problem can be derived directly from a PTAS for the densest subgraph problem. Hence, we use a heuristic that is



(a) Dep. Set Cost (b) Cover Set Cost

Figure 3.6: Dependent-set vs. cover-set Cost

modeled on a greedy PTAS for the Budgeted Maximum Coverage problem [77] which is similar to the formulation of our problem. In our greedy heuristic, at each stage, the algorithm chooses a set of rules that maximizes the ratio of combined rule weight to combined rule cost ($\frac{\Delta W}{\Delta C}$), until the total cost reaches k. This algorithm runs in $O(nk)$ time.

On the example rule table in Figure 3.4(a), the greedy algorithm selects R_6 first (and its dependent set $\{R_4, R_5\}$), and then R_1 which brings the total cost to 4. Thus the set of rules in the TCAM are R_1, R_4, R_5, and R_6 which is the optimal. We refer to this algorithm as the *dependent-set* algorithm.

3.3.3 Cover-Set: Splicing Dependency Chains

Respecting rule dependencies can lead to high costs, especially if a high-weight rule depends on many low-weight rules. For example, consider a firewall that has a single low-priority "accept" rule that depends on many high-priority "deny" rules that match relatively little traffic. Caching the one "accept" rule would require caching many "deny" rules. We can do better than past algorithms by modifying the rules in various semantics-preserving ways, instead of simply packing the existing rules into the available space—this is the key observation that leads to our superior algorithm. In particular, we "splice" the dependency

chain by creating a small number of new rules that *cover* many low-weight rules and send the affected packets to the software switch.

For the example in Figure 3.4(a), instead of selecting all dependent rules for R_6, we calculate new rules that cover the packets that would otherwise incorrectly hit R_6. The extra rules direct these packets to the software switches, thereby breaking the dependency chain. For example, we can install a high-priority rule R_5^* with match 1*1* and action `forward_to_Soft_switch`,[6] along with the low-priority rule R_6. Similarly, we can create a new rule R_1^* to break dependencies on R_2. We avoid installing higher-priority, low-weight rules like R_4, and instead have the high-weight rules R_2 and R_6 inhabit the cache simultaneously, as shown in Figure 3.4(b).

More generally, the algorithm must calculate the *cover set* for each rule R. To do so, we find the immediate ancestors of R in the dependency graph and replace the actions in these rules with a `forward_to_Soft_Switch` action. For example, the cover set for rule R_6 is the rule R_5^* in Figure 3.4(b); similarly, R_1^* is the cover set for R_2. The rules defining these `forward_to_Soft_switch` actions may also be merged, if necessary.[7] The cardinality of the cover set defines the new cost value for each chosen rule. This new cost is strictly less than or equal to the cost in the dependent set algorithm. The new cost value is *much* less for rules with long chains of dependencies. For example, the old dependent set cost for the rule R_6 in Figure 3.4(a) is 3 as shown in the rule cost table whereas the cost for the new cover set for R_6 in Figure 3.4(b) is only 2 since we only need to cache R_5^* and R_6. To take a more general case, the old cost for the red rule in Figure 3.6(a) was the entire set of ancestors (in light red), but the new cost (in Figure 3.6(b)) is defined just by the immediate ancestors (in light red).

---

[6] This is just a standard forwarding action out some port connected to a software switch.

[7] To preserve OpenFlow semantics pertaining to hardware packet counters, policy rules cannot be compressed. However, we can compress the intermediary rules used for forwarding cache misses, since the software switch can track the per-rule traffic counters.

3.3.4 Mixed-Set: An Optimal Mixture

Despite decreasing the cost of caching a rule, the cover-set algorithm may also decrease the weight by redirecting the spliced traffic to the software switch. For example, for caching the rule R_2 in Figure 3.4(c), the dependent-set algorithm is a better choice because the traffic volume processed by the dependent set in the TCAM is higher, while the cost is the same as a cover set. In general, as shown in Figure 3.6(b), cover set seems to be a better choice for caching a higher dependency rule (like the red node) compared to a lower dependency rule (like the blue node).

In order to deal with cases where one algorithm may do better than the other, we designed a heuristic that chooses the best of the two alternatives at each iteration. As such, we consider a metric that chooses the *best* of the two sets i.e., $\max(\frac{\Delta W_{dep}}{\Delta C_{dep}}, \frac{\Delta W_{cover}}{\Delta C_{cover}})$. Then we can apply the same greedy covering algorithm with this new metric to choose the best set of candidate rules to cache. We refer to this version as the *mixed-set* algorithm.

3.3.5 Updating the TCAM Incrementally

As the traffic distribution over the rules changes over time, the set of cached rules chosen by our caching algorithms also change. This would mean periodically updating the TCAM with a new version of the policy cache. Simply deleting the old cache and inserting the new cache from scratch is not an option because of the enormous TCAM rule insertion time. It is important to minimize the churn in the TCAM when we periodically update the cached rules.

Updating just the difference will not work Simply taking the difference between the two sets of cached rules—and replacing the stale rules in the TCAM with new rules (while retaining the common set of rules)—can result in incorrect policy snapshots on the TCAM during the transition. This is mainly because TCAM rule update takes time and hence packets can be processed incorrectly by an incomplete policy snapshot during transition.

For example, consider the case where the mixed-set algorithm decides to change the cover-set of rule R_6 to its dependent set. If we simply remove the cover rule (R_5^*) and then install the dependent rules (R_5,R_4), there will be a time period when only the rule R_6 is in the TCAM without either its cover rules or the dependent rules. This is a policy snapshot that can incorrectly process packets while the transition is going on.

Exploiting composition of mixed sets A key property of the algorithms discussed so far is that each chosen rule along with its mixed (cover or dependent) set can be added/removed from the TCAM independently of the rest of the rules. In other words, the mixed-sets for any two rules are easily composable and decomposable. For example, in Figure 3.6(b), the red rule and its cover set can be easily added/removed without disturbing the blue rule and its dependent set. In order to push the new cache in to the TCAM, we first decompose/remove the old mixed-sets (that are not cached anymore) from the TCAM and then compose the TCAM with the new mixed sets. We also maintain reference counts from various mixed sets to the rules on TCAM so that we can track rules in overlapping mixed sets. Composing two candidate rules to build a cache would simply involve merging their corresponding mixed-sets (and incrementing appropriate reference counters for each rule) and decomposing would involve checking the reference counters before removing a rule from the TCAM [8]. In the example discussed above, if we want to change the cover-set of rule R_6 to its dependent set on the TCAM, we first delete the entire cover-set rules (including rule R_6) and then install the entire dependent-set of R_6, in priority order.

3.4 CacheMaster Design

As shown in Figure 3.1, CacheFlow has a CacheMaster module that implements its control-plane logic. In this section, we describe how CacheMaster directs "cache-miss" packets

[8] The intuition is that if a rule has a positive reference count, then either its dependent-set or the cover-set is also present on the TCAM and hence is safe to leave behind during the decomposition phase

from the TCAM to the software switches, using existing switch mechanisms and preserves the semantics of OpenFlow.

3.4.1 Scalable Processing of Cache Misses

CacheMaster runs the algorithms in Section 3.3 to compute the rules to cache in the TCAM. The cache misses are sent to one of the software switches, which each store a copy of the entire policy. CacheMaster can shard the cache-miss load over the software switches.

Using the group tables in OpenFlow 1.1+, the hardware switch can apply a simple load-balancing policy. Thus the `forward_to_SW_switch` action (used in Figure 3.4) forwards the cache-miss traffic—say, matching a low-priority "catch-all" rule—to this load-balancing group table in the switch pipeline, whereupon the cache-miss traffic can be distributed over the software switches.

3.4.2 Preserving OpenFlow Semantics

To work with unmodified controllers and switches, CacheFlow preserves semantics of the OpenFlow interface, including rule priorities and counters, as well as features like `packet_ins`, barriers, and rule timeouts.

Preserving inports and outports: CacheMaster installs three kinds of rules in the hardware switch: (i) fine-grained rules that apply the cached part of the policy (cache-hit rules), (ii) coarse-grained rules that forward packets to a software switch (cache-miss rules), and (iii) coarse-grained rules that handle return traffic from the software switches, similar to mechanisms used in DIFANE [123]. In addition to matching on packet-header fields, an OpenFlow policy may match on the inport where the packet arrives. Therefore, the hardware switch *tags* cache-miss packets with the input port (e.g., using a VLAN tag) so that the software switches can apply rules that depend on the inport[9]. The rules in

[9] Tagging the cache-miss packets with the inport can lead to extra rules in the hardware switch. In several practical settings, the extra rules are not necessary. For example, in a switch used only for layer-3 processing, the destination MAC address uniquely identifies the input port, obviating the need for a separate tag. Newer

the software switches apply any “drop” or “modify” actions, tag the packets for proper forwarding at the hardware switch, and direct the packet back to the hardware switch. Upon receiving the return packet, the hardware switch simply matches on the tag, pops the tag, and forwards to the designated output port(s).

Packet-in messages: If a rule in the TCAM has an action that sends the packet to the controller, CacheMaster simply forwards the the `packet_in` message to the controller. However, for rules on the software switch, CacheMaster must transform the `packet_in` message by (i) copying the inport from the packet tag into the inport field of the `packet_in` message and (ii) stripping the tag from the packet before sending to the controller.

Traffic counts, barrier messages, and rule timeouts: CacheFlow preserves the semantics of OpenFlow constructs like queries on traffic statistics, barrier messages, and rule timeouts by having CacheMaster emulate these features. For example, CacheMaster maintains packet and byte counts for each rule installed by the controller, updating its local information each time a rule moves to a different part of the “cache hierarchy.” The CacheMaster maintains three counters per rule. A hardware counter periodically polls and maintains the current TCAM counter for the rule if it cached. Similarly, a software counter maintains the current software switch counters. A persistent hardware count accumulates the hardware counter whenever the rule is removed from the hardware cache and resets the hardware counter to zero. Thus, when an application asks for a rule counter, CacheMaster simply returns the sum of the three counters associated with that rule.

Similarly, CacheMaster emulates [51] rule timeouts by installing rules *without* timeouts, and explicitly removing the rules when the software timeout expires. For barrier messages, CacheMaster first sends a barrier request to all the switches, and waits for all of them to respond before sending a barrier reply to the controller. In the meantime, CacheMaster buffers all messages from the controller before distributing them among the switches.

---

version of OpenFlow support switches with multiple stages of tables, allowing us to use one table to push the tag and another to apply the (cached) policy.

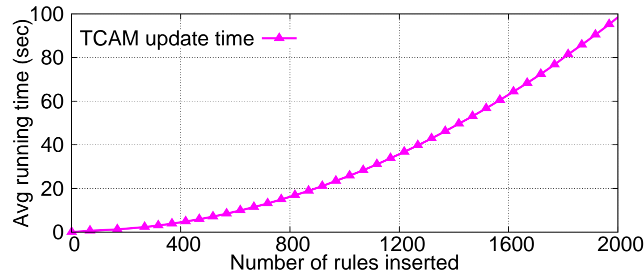


Figure 3.7: TCAM Update Time

3.5 Commodity Switch as the Cache

The hardware switch used as a cache in our system is a Pronto-Pica8 3290 switch running PicOS 2.1.3 supporting OpenFlow. We uncovered several limitations of the switch that we had to address in our experiments:

Incorrect handling of large rule tables: The switch has an ASIC that can hold 2000 OpenFlow rules. If more than 2000 rules are sent to the switch, 2000 of the rules are installed in the TCAM and the rest in the software agent. However, the switch does not respect the cross-rule dependencies when updating the TCAM, leading to incorrect forwarding behavior! Since we cannot modify the (proprietary) software agent, we simply avoid triggering this bug by assuming the rule capacity is limited to 2000 rules. Interestingly, the techniques presented in this chapter are exactly what the software agent can use to fix this issue.

Slow processing of control commands: The switch is slow at updating the TCAM and querying the traffic counters. The time required to update the TCAM is a non-linear function of the number of rules being added or deleted, as shown in Figure 3.7. While the first 500 rules take 6 seconds to add, the next 1500 rules takes almost 2 minutes to install. During this time, querying the switch counters easily led to the switch CPU hitting 100% utilization and, subsequently, to the switch disconnecting from the controller. In order to

get around this, we wait till the set of installed rules is relatively stable to start querying the counters at regular intervals and rely on counters in the software switch in the meantime.

3.6 Prototype and Evaluation

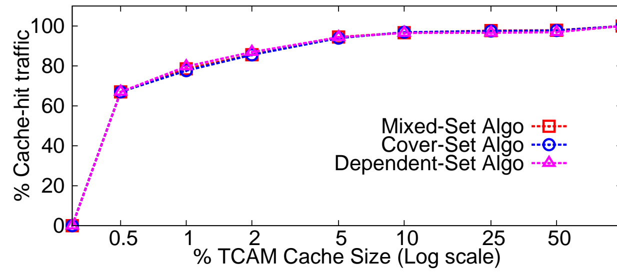
We implemented a prototype of CacheFlow in Python using the Ryu controller library so that it speaks OpenFlow to the switches. On the north side, CacheFlow provides an interface which control applications can use to send `FlowMod`s to CacheFlow, which then distributes them to the switches. At the moment, our prototype supports the semantics of the OpenFlow 1.0 features mentioned earlier (except for rule timeouts) transparently to both the control applications and the switches.

We use the Pica8 switch as the hardware cache, connected to an Open vSwitch 2.1.2 multithread software switch running on an AMD 8-core machine with 6GB RAM. To generate data traffic, we connected two host machines to the Pica8 switch and use `tcpreplay` to send packets from one host to the other.

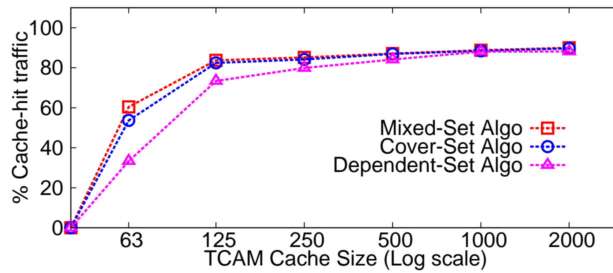
3.6.1 Cache-hit Rate

We evaluate our prototype against three policies and their corresponding packet traces: (i) A publicly available packet trace from a real data center and a synthetic policy, (ii) An educational campus network routing policy and a synthetic packet trace, and (iii) a real OpenFlow policy and the corresponding packet trace from an Internet eXchange Point (IXP). We measure the cache-hit rate achieved on these policies using three caching algorithms (dependent-set, cover-set, and mixed-set). The cache misses are measured by using `ifconfig` on the software switch port and then the cache hits are calculated by subtracting the cache misses from the total packets sent as reported by `tcpreplay`. All the results reported here are made by running the Python code using PyPy to make the code run faster.

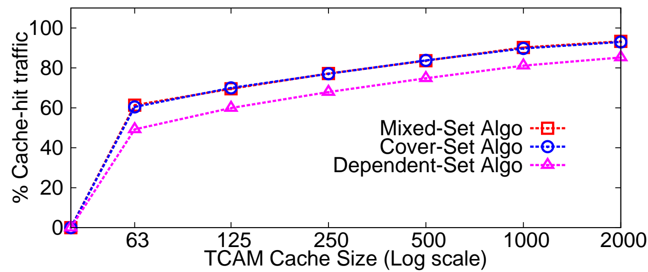
REANNZ. Figure 3.8(a) shows results for an SDN-enabled IXP that supported the REANNZ research and education network [10]. This real-world policy has 460 OpenFlow 1.0



(a) REANNZ IXP switch



(b) Stanford backbone router



(c) CAIDA packet trace

Figure 3.8: Cache-hit rate vs. TCAM size for three algorithms and three policies (with x-axis on log scale)

rules matching on multiple packet headers like `inport`, `dst_ip`, `eth_type`, `src_mac`, etc. Most dependency chains have depth 1 (some lightweight rules have complex dependencies as shown in Figure 3.3(b)). We replayed a two-day traffic trace from the IXP, and updated the cache every two minutes and measured the cache-hit rate over the two-day period. Because of the many shallow dependencies, all three algorithms have the same

performance. The mixed-set algorithm sees a cache hit rate of 84% with a hardware cache of just 2% of the rules; with just 10% of the rules, the cache hit rate increases to as much as 97%.

Stanford Backbone. Figure 3.8(b) shows results for a real-world Cisco router configuration on a Stanford backbone router [12]. which we transformed into an OpenFlow policy. The policy has 180K OpenFlow 1.0 rules that match on the destination IP address, with dependency chains varying in depth from 1 to 8. We generated a packet trace matching the routing policy by assigning traffic volume to each rule drawn from a Zipf [110] distribution. The resulting packet trace had around 30 million packets randomly shuffled over 15 minutes. The mixed-set algorithm does the best among all three and dependent-set does the worst because there is a mixture of shallow and deep dependencies. While there are differences in the cache-hit rate, all three algorithms achieve at least 88% hit rate at the total capacity of 2000 rules (which is just 1.1% of the total rule table). Note that CacheFlow was able to react effectively to changes in the traffic distribution for such a large number of rules (180K in total) and the software switch was also able to process all the cache misses at line rate. Note that installing the same number of rules in the TCAM of a hardware switch, assuming that TCAMs are 80 times more expensive than DRAMs, requires one to spend 14 times more money on the memory unit.

CAIDA. The third experiment was done using the publicly available CAIDA packet trace taken from the Equinix datacenter in Chicago [99]. The packet trace had a total of 610 million packets sent over 30 minutes. Since CAIDA does not publish the policy used to process these packets, we built a policy by extracting forwarding rules based on the destination IP addresses of the packets in the trace. We obtained around 14000 `/20` IP destination based forwarding rules. This was then *sequentially composed* [96] with an access-control policy that matches on fields other than just the destination IP address. The ACL was a chain of 5 rules that match on the source IP, the destination TCP port and inport of the packets which introduce a dependency chain of depth 5 for each destination IP prefix.

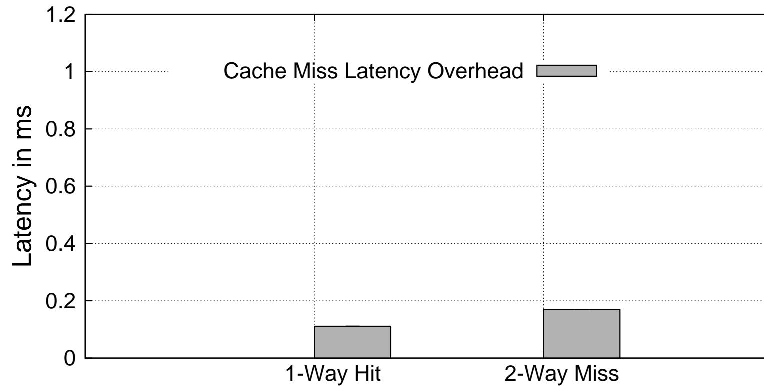
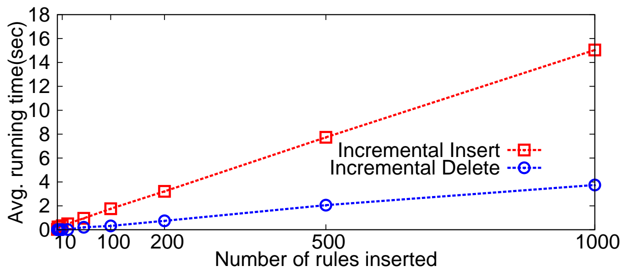


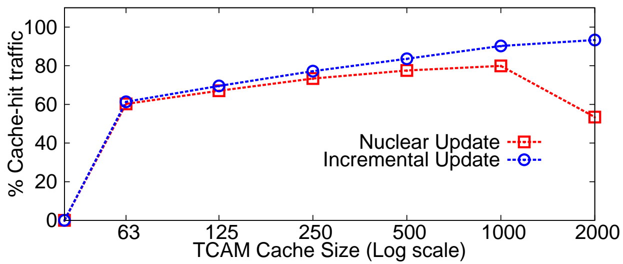
Figure 3.9: Cache-Miss Latency Overhead

This composition resulted in a total of 70K OpenFlow rules that match on multiple header fields. This experiment is meant to show the dependencies that arise from matching on various fields of a packet and also the explosion of dependencies that may arise out of more sophisticated policies. Figure 3.8(c) shows the cache-hit percentage under various TCAM rule capacity restrictions. The mixed-set and cover-set algorithms have similar cache-hit rates and do much better than the dependent-set algorithm consistently because they splice every single dependency chain in the policy. For any given TCAM size, mixed-set seems to have at least 9% lead on the cache-hit rate. While mixed-set and cover-set have a hit rate of around 94% at the full capacity of 2000 rules (which is just 3% of the total rule table), all three algorithms achieve at least an 85% cache-hit rate.

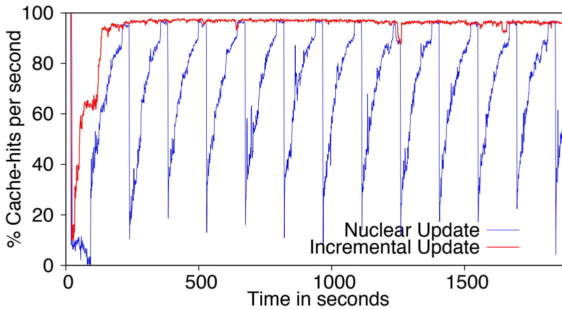
Latency overhead. Figure 3.9 shows the latency incurred on a cache-hit versus a cache-miss. The latency was measured by attaching two extra hosts to the switch while the previously CAIDA packet trace was being run. Extra rules initialized with heavy volume were added to the policy to process the ping packets in the TCAM. The average round-trip latency when the ping packets were cache-hits in both directions was $0.71ms$ while the latency for 1-way cache miss was $0.81ms$. Thus, the cost of a 1-way cache miss was $100\mu s$; for comparison, a hardware switch adds $25\mu s$ [94] to the 1-way latency of the packets.



(a) Incremental DAG update



(b) Incremental/nuclear cache-hit rate



(c) Incremental vs. nuclear stability

Figure 3.10: Performance of Incremental Algorithms for DAG and TCAM update

If an application cannot accept the additional cost of going to the software switch, it can request the CacheMaster to install its rules in the fast path. The CacheMaster can do this by assigning "infinite" weight to these rules.

3.6.2 Incremental Algorithms

In order to measure the effectiveness of the incremental update algorithms, we conducted two experiments designed to evaluate (i) the algorithms to incrementally update the dependency graph on insertion or deletion of rules and (ii) algorithms to incrementally update the TCAM when traffic distribution shifts over time.

Figure 3.10(a) shows the time taken to insert/delete rules incrementally on top of the Stanford routing policy of 180K rules. While an incremental insert takes about 15 milliseconds on average to update the dependency graph, an incremental delete takes around 3.7 milliseconds on average. As the linear graphs show, at least for about a few thousand inserts and deletes, the amount of time taken is strictly proportional to the number of flowmods. Also, an incremental delete is about 4 times faster on average owing to the very local set of dependency changes that occur on deletion of a rule while an insert has to explore a lot more branches starting with the root to find the correct position to insert the rule. We also measured the time taken to statically build the graph on a rule insertion which took around 16 minutes for 180K rules. Thus, the incremental versions for updating the dependency graph are $\sim$60000 times faster than the static version.

In order to measure the advantage of using the incremental TCAM update algorithms, we measured the cache-hit rate for mixed-set algorithm using the two options for updating the TCAM. Figure 3.10(b) shows that the cache-hit rate for the incremental algorithm is substantially higher as the TCAM size grows towards 2000 rules. For 2000 rules in the TCAM, while the incremental update achieves 93% cache-hit rate, the nuclear update achieves only 53% cache-hit rate. As expected, the nuclear update mechanism sees diminishing returns beyond 1000 rules because of the high rule installation time required to install more than 1000 rules as shown earlier in Figure 3.7.

Figure 3.10(c) shows how the cache-hit rate is affected by the naive version of doing a nuclear update on the TCAM whenever CacheFlow decides to update the cache. The figure

shows the number of cache misses seen over time when the CAIDA packet trace is replayed at 330k packets per second. The incremental update algorithm stabilizes quite quickly and achieves a cache-hit rate of 95% in about 3 minutes. However, the nuclear update version that deletes all the old rules and inserts the new cache periodically suffers a lot of cache-misses while it is updating the TCAM. While the cache-hits go up to 90% once the new cache is fully installed, the hit rate goes down to near 0% every time the rules are deleted and it takes around 2 minutes to get back to the high cache-hit rate. This instability in the cache-miss rate makes the nuclear installation a bad option for updating the TCAM.

3.7 Related Work

While route caching is discussed widely in the context of IP destination prefix forwarding, SDN introduces new constraints on rule caching. We divide the route caching literature into three wide areas: (i) IP route Caching (ii) TCAM optimization, and (iii) SDN rule caching.

IP Route Caching. Earlier work on traditional IP route caching [48, 79, 85, 86, 110] talks about storing only a small number of IP prefixes in the switch line cards and storing the rest in inexpensive slow memory. Most of them exploit the fact that IP traffic exhibits both temporal and spatial locality to implement route caching. For example, Sarrar et.al [110] show that packets hitting IP routes collected at an ISP follow a Zipf distribution resulting in effective caching of small number of heavy hitter routes. However, most of them do not deal with cross-rule dependencies and none of them deal with complex multidimensional packet-classification. For example, Liu et.al [86] talk about efficient FIB caching while handling the problem of *cache-hiding* for IP prefixes. However, their solution cannot handle multiple header fields or wildcards and does not have the notion of packet counters associated with rules. This chapter, on the other hand, deals with the analogue of the cache-hiding problem for more general and complex packet-classification patterns and also preserves packet counters associated with these rules.

TCAM Rule Optimization. The TCAM Razor [84, 90, 91] line of work compresses multi-dimensional packet-classification rules to minimal TCAM rules using decision trees and multi-dimensional topological transformation. Dong et. al. [46] propose a caching technique for ternary rules by constructing compressed rules for evolving flows. Their solution requires special hardware and does not preserve counters. In general, these techniques that use compression to reduce TCAM space also suffer from not being able to make incremental changes quickly to their data-structures.

DAG for TCAM Rule Updates. The idea of using DAGs for representing TCAM rule dependencies is discussed in the literature in the context of efficient TCAM rule updates [115, 120]. In particular, their aim was to optimize the time taken to install a TCAM rule by minimizing the number of existing entries that need to be reshuffled to make way for a new rule. They do so by building a DAG that captures how different rules are placed in different TCAM banks for reducing the update churn. However, the resulting DAG is not suitable for caching purposes as it is difficult to answer the question we ask: if a rule is to be cached, which other rules should go along with it? Our DAG data structure on the other hand is constructed in such a way that given any rule, the corresponding cover set to be cached can be inferred easily. This also leads to novel incremental algorithms that keep track of additional metadata for each edge in the DAG, which is absent in existing work.

SDN Rule Caching. There is some recent work on dealing with limited switch rule space in the SDN community. DIFANE [123] advocates caching of ternary rules, but uses more TCAM to handle cache misses—leading to a TCAM-hungry solution. Other work [70, 71, 97] shows how to distribute rules over multiple switches along a path, but cannot handle rule sets larger than the aggregate table size. Devoflow [43] introduces the idea of rule "cloning" to reduce the volume of traffic processed by the TCAM, by having each match in the TCAM trigger the creation of an exact-match rules (in SRAM) the handle the remaining packets of that microflow. However, Devoflow does not address the limitations on the total size of the TCAM. Lu et.al. [87] use the switch CPU as a traffic co-processing unit

where the ASIC is used as a cache but they only handle microflow rules and hence do not handle complex dependencies. The Open vSwitch [104] caches “megaflows” (derived from wildcard rules) to avoid the slow lookup time in the user space classifier. However, their technique does not assume high-throughput wildcard lookup in the fast path and hence cannot be used directly for optimal caching in TCAMs.

3.8 Conclusion

In this chapter, we define a hardware-software hybrid switch design called CacheFlow that relies on rule caching to provide large rule tables at low cost. Unlike traditional caching solutions, we neither cache individual rules (to respect rule dependencies) nor compress rules (to preserve the per-rule traffic counts). Instead we “splice” long dependency chains to cache smaller groups of rules while preserving the semantics of the network policy. Our design satisfies four core criteria: (1) *elasticity* (combining the best of hardware and software switches), (2) *transparency* (faithfully supporting native OpenFlow semantics, including traffic counters), (3) *fine-grained* rule caching (placing popular rules in the TCAM, despite dependencies on less-popular rules), and (4) *adaptability* (to enable incremental changes to the rule caching as the policy changes).

Chapter 4

Ravana: Controller Fault-Tolerance in Software-Defined Networking

Software-defined networking (SDN) offers greater flexibility than traditional distributed architectures, at the risk of the controller being a single point-of-failure. Unfortunately, existing fault-tolerance techniques, such as replicated state machine, are insufficient to ensure correct network behavior under controller failures. The challenge is that, in addition to the application state of the controllers, the switches maintain hard state that must be handled consistently. Thus, it is necessary to incorporate switch state into the system model to correctly offer a "logically centralized" controller.

In this chapter, we introduce Ravana, a fault-tolerant SDN controller platform that processes the control messages transactionally and exactly once (at both the controllers and the switches). Ravana maintains these guarantees in the face of both controller and switch crashes. The key insight in Ravana is that replicated state machines can be extended with lightweight switch-side mechanisms to guarantee correctness, without involving the switches in an elaborate consensus protocol. Our prototype implementation of Ravana enables *unmodified* controller applications to execute in a fault-tolerant fashion. Experiments

show that Ravana achieves high throughput with reasonable overhead, compared to a single controller, with a failover time under 100ms.

4.1 Introduction

In Software-Defined Networking (SDN), a logically centralized *controller* orchestrates a distributed set of switches to provide higher-level networking services to end-host applications. The controller can reconfigure the switches (though *commands*) to adapt to traffic demands and equipment failures (observed through *events*). For example, an SDN controller receives events concerning topology changes, traffic statistics, and packets requiring special attention, and it responds with commands that install new forwarding rules on the switches. Global visibility of network events and direct control over the switch logic enables easy implementation of policies like globally optimal traffic engineering, load balancing, and security applications on commodity switches.

Despite the conceptual simplicity of centralized control, a single controller easily becomes a single point of failure, leading to service disruptions or incorrect packet processing [39, 111]. In this chapter, we study SDN fault-tolerance under crash (fail-stop) failures. Ideally, a fault-tolerant SDN should behave the same way as a fault-free SDN from the viewpoint of controller applications and end-hosts. This ensures that controller failures do not adversely affect the network administrator's goals or the end users. Further, the right abstractions and protocols should free controller applications from the burden of handling controller crashes.

It may be tempting to simply apply established techniques from the distributed systems literature. For example, multiple controllers could utilize a distributed storage system to replicate durable state (*e.g.*, either via protocols like two-phase commit or simple primary/backup methods with journaling and rollback), as done by Onix [82] and ONOS [32]. Or, they could model each controller as a replicated state machine (RSM) and instead con-

sistently replicate the set of *inputs* to each controller. Provided each replicated controller executes these inputs deterministically and in an identical order, their internal state would remain consistent.

But maintaining consistent controller state is only part of the solution. To provide a logically centralized controller, one must also ensure that the *switch state* is handled consistently during controller failures. And the semantics of switch state, as well as the interactions between controllers and switches, is complicated. Broadly speaking, existing systems do not reason about switch state; they have not rigorously studied the semantics of processing switch events and executing switch commands under failures.

Yet one cannot simply extend traditional replication techniques to include the network switches. For example, running a consensus protocol involving the switches for every event would be prohibitively expensive, given the demand for high-speed packet processing in switches. On the other hand, using distributed storage to replicate controller state alone (for performance reasons) does not capture the switch state precisely. Therefore, after a controller crash, the new master may not know where to resume reconfiguring switch state. Simply reading the switch forwarding state would not provide enough information about all the commands sent by the old master (*e.g.*, PacketOuts, StatRequests).

In addition, while the system could roll back the controller state, the switches cannot easily "roll back" to a safe checkpoint. After all, what does it mean to rollback a packet that was already sent? The alternative is for the new master to simply repeat commands, but these commands are not necessarily idempotent (per §4.2). Since an event from one switch can trigger commands to other switches, simultaneous failure of the master controller and a switch can cause inconsistency in the rest of the network. We believe that these issues can lead to erratic behavior in existing SDN platforms.

Ravana. In this chapter, we present Ravana, an SDN controller platform that offers the abstraction of a fault-free centralized controller to control applications. Instead of just keeping the controller state consistent, we handle the *entire event-processing cycle* (includ-

ing event delivery from switches, event processing on controllers, and command execution on switches) as a *transaction*—either all or none of the components of this transaction are executed. Ravana ensures that transactions are totally ordered across replicas and executed exactly once across the entire system. This enables Ravana to correctly handle switch state, without resorting to rollbacks or repeated execution of commands.

Ravana adopts replicated state machines for control state replication and adds mechanisms for ensuring the consistency of switch state. Ravana uses a two-stage replication protocol across the controllers. The master replica decides the total order in which input events are received in the first stage and then indicates which events were processed in the second stage. On failover, the new master resumes transactions for "unprocessed" events from a shared log. The two stages isolate the effects of a switch failure on the execution of a transaction on other switches—a setting unique to SDN. Instead of involving all switches in a consensus protocol, Ravana extends the OpenFlow interface with techniques like explicit acknowledgment, retransmission, and filtering from traditional RPC protocols to ensure that any event transaction is executed *exactly once* on the switches.

While the various techniques we adopt are well known in distributed systems literature, ours is the first system that applies these techniques comprehensively in the SDN setting to design a correct, faul-tolerant controller. We also describe the safety and liveness guarantees provided by the Ravana protocol and argue that it ensures *observational indistinguishability* (per §4.4) between an ideal central controller and a replicated controller platform. Our prototype implementation allows unmodified control applications, written for a single controller, to run in a replicated and fault-tolerant manner. Our prototype achieves these properties with low overhead on controller throughput, latency, and failover time.

Contributions. Our fault-tolerant controller system makes the following technical contributions:

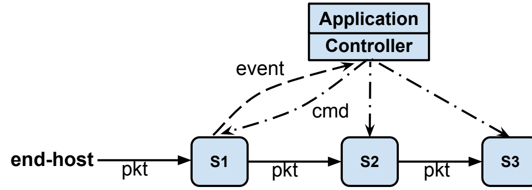


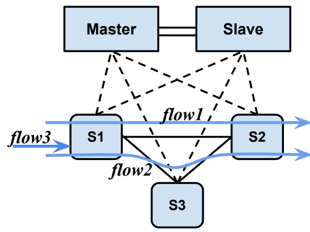
Figure 4.1: SDN system model

- We propose a two-phase replication protocol that extends replicated state machines to handle consistency of *external* switch state under controller failures.
- We propose extensions to the OpenFlow interface that are necessary to handle controller failures like RPC-level ACKs, retransmission of un-ACKed events and filtering of duplicate commands.
- We precisely define correctness properties of a logically centralized controller and argue that the Ravana protocol provides these guarantees.
- We present a prototype of a transparent Ravana runtime and demonstrate our solution has low overhead.

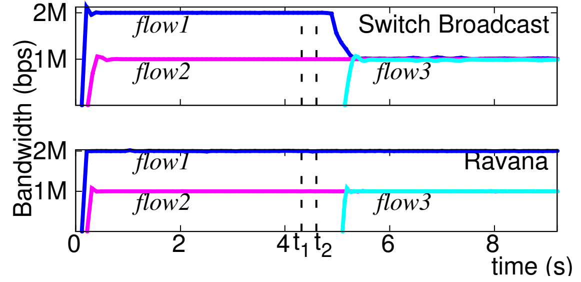
4.2 Controller Failures in SDN

Figure 4.1 shows the normal execution of an SDN in the absence of controller failures. Recent versions of OpenFlow [89], the widely used control channel protocol between controllers and switches, have some limited mechanisms for supporting multiple controllers. In particular, a controller can register with a switch in the role of *master* or *slave*, which defines the types of events and commands exchanged between the controller and the switch. A switch sends all events to the master and executes all commands received from the master, while it sends only a limited set of events (for example, `switch_features`) to slaves and does not accept commands from them.

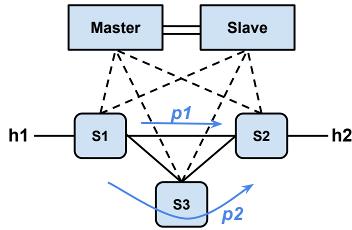
Combining existing OpenFlow protocols with traditional techniques for replicating controllers does not ensure correct network behavior, however. This section illustrates the reasons with concrete experiments. The results are summarized in Table 4.2.



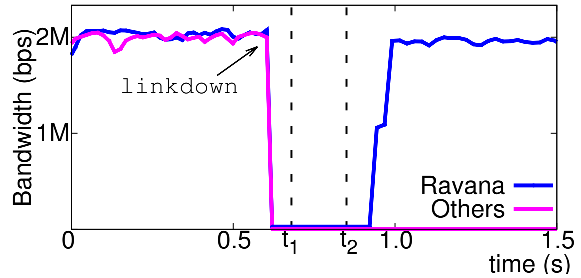
(a) Total Event Ordering



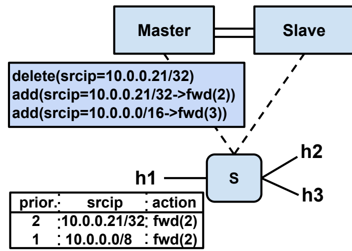
(b) Bandwidth Allocation



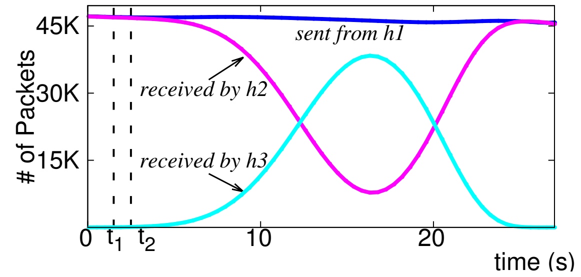
(c) Exactly-Once Event Delivery



(d) `linkdown` Under Failures



(e) Exactly-Once Command Execution



(f) Routing Under Repeated Commands

Figure 4.2: Examples demonstrating different correctness properties maintained by Ravana and corresponding experimental results. t_1 and t_2 indicate the time when the old master controller crashes and when the new master is elected, respectively. In (f), the delivery of commands is slowed down to measure the traffic leakage effect.

4.2.1 Inconsistent Event Ordering

OpenFlow 1.3 allows switches to connect to multiple controllers. If we directly use the protocol to have switches broadcast their events to every controller replica independently, each replica builds application state based on the stream of events it receives. Aside from the additional overhead this places on switches, controller replicas would have an inconsistent

ordering of events from different switches. This can lead to incorrect packet-processing decisions, as illustrated in the following example:

Experiment 1: In Figure 4.2a, consider a controller application that allocates incoming flow requests to paths in order. There are two disjoint paths in the network; each has a bandwidth of 2Mbps. Assume that two flows, with a demand of 2Mbps and 1Mbps respectively, arrive at the controller replicas in different order (due to network latencies). If the replicas assign paths to flows in the order they arrive, each replica will end up with 1Mbps free bandwidth but on different paths. Now, consider that the master crashes and the slave becomes the new master. If a new flow with 1Mbps arrives, the new master assigns the flow to the path which it thinks has 1Mbps free. But this congests an already fully utilized path, as the new master's view of the network diverged from its actual state (as dictated by the old master).

Figure 4.2b compares the measured flow bandwidths for the switch-broadcast and Ravana solutions. Ravana keeps consistent state in controller replicas, and the new master can install the flows in an optimal manner. Drawing a lesson from this experiment, a fault-tolerant control platform should offer the following design goal:

Total Event Ordering: Controller replicas should process events in the same order and subsequently all controller application instances should reach the same internal state.

Note that while in this specific example, the newly elected master can try to query the flow state from switches after failure; in general, simply reading switch state is not enough to infer sophisticated application state. This also defeats the argument for transparency because the programmer has to explicitly define how application state is related to switch state under failures. Also, information about PacketOuts and events lost during failures cannot be inferred by simply querying switch state.

Property	Description	Mechanism
At least once events	Switch events are not lost	Buffering and retransmission of switch events
At most once events	No event is processed more than once	Event IDs and filtering in the log
Total event order	Replicas process events in same order	Master serializes events to a shared log
Replicated control state	Replicas build same internal state	Two-stage replication and deterministic replay of event log
At least once commands	Controller commands are not lost	RPC acknowledgments from switches
At most once commands	Commands are not executed repeatedly	Command IDs and filtering at switches

Table 4.1: Ravana design goals and mechanisms

4.2.2 Unreliable Event Delivery

Two existing approaches can ensure a consistent ordering of events in replicated controllers: (i) The master can store shared application state in an *external* consistent storage system (e.g., as in Onix and ONOS), or (ii) the controller's *internal* state can be kept consistent via replicated state machine (RSM) protocols. However, the former approach may fail to persist the controller state when the master fails during the event processing, and the latter approach may fail to log an event when the master fails right after receiving it. These scenarios may cause serious problems.

Experiment 2: Consider a controller program that runs a shortest-path routing algorithm, as shown in Figure 4.2c. Assume the master installed a flow on path $p1$, and after a while the link between $s1$ and $s2$ fails. The incident switches send a `linkdown` event to the master. Suppose the master crashes before replicating this event. If the controller replicas are using a traditional RSM protocol with unmodified OpenFlow switches, the event is lost and will never be seen by the slave. Upon becoming the new master, the slave will have an inconsistent view of the network, and cannot promptly update the switches to reroute packets around the failed link.

Figure 4.2d compares the measured bandwidth for the flow $h1 \rightarrow h2$ with an unmodified OpenFlow switch and with Ravana. With an unmodified switch, the controller loses the link failure event which leads to throughput loss, and it is sustained even after the new master is elected. In contrast, with Ravana, events are reliably delivered to all replicas even during failures, ensuring that the new master switches to the alternate path, as shown by the blue curve. From this experiment, we see that it is important to ensure reliable event delivery. Similarly, event repetition will also lead to inconsistent network views, which can further result in erroneous network behaviors. This leads to our second design goal:

Exactly-Once Event Processing: All the events are processed, and are neither lost nor processed repeatedly.

4.2.3 Repetition of Commands

With traditional RSM or consistent storage approaches, a newly elected master may send repeated commands to the switches because the old master sent some commands but crashed before telling the slaves about its progress. As a result, these approaches cannot guarantee that commands are executed exactly once, leading to serious problems when commands are not idempotent.

Experiment 3: Consider a controller application that installs rules with overlapping patterns. The rule that a packet matches depends on the presence or absence of other higher-priority rules. As shown in Figure 4.2e, the switch starts with a forwarding table with two rules that both match on the source address and forward packets to host $h2$. Suppose host $h1$ has address 10.0.0.21, which matches the first rule. Now assume that the master sends a set of three commands to the switch to redirect traffic from the $/16$ subnet to $h3$. After these commands, the rule table becomes the following:

```
3  10.0.0.21/32  fwd(2)
2  10.0.0.0/16   fwd(3)
```

```
1   10.0.0.0/8     fwd(2)
```

If the master crashes before replicating the information about commands it already issued, the new master would repeat these commands. When that happens, the switch first removes the first rule in the new table. Before the switch executes the second command, traffic sent by $h1$ can match the rule for `10.0.0.0/16` and be forwarded erroneously to $h3$. If there is no controller failure and the set of commands are executed exactly once, $h3$ would never have received traffic from $h1$; thus, in the failure case, the correctness property is violated. The duration of this erratic behavior may be large owing to the slow rule-installation times on switches. Leaking traffic to an unexpected receiver $h3$ could lead to security or privacy problems.

Figure 4.2f shows the traffic received by $h2$ and $h3$ when sending traffic from $h1$ at a constant rate. When commands are repeated by the new master, $h3$ starts receiving packets from $h1$. No traffic leakage occurs under Ravana. While missing commands will obviously cause trouble in the network, from this experiment we see that command repetition can also lead to unexpected behaviors. As a result, a correct protocol must meet the third design goal:

Exactly-Once Execution of Commands: Any given series of commands are executed once and only once on the switches.

4.2.4 Handling Switch Failures

Unlike traditional client-server models where the server processes a client request and sends a reply to the same client, the event-processing cycle is more complex in the SDN context: when a switch sends an event, the controller may respond by issuing *multiple* commands to *other* switches. As a result, we need additional mechanisms when adapting replication protocols to build fault-tolerant control platforms.

Suppose that an event generated at a switch is received at the master controller. Existing fault-tolerant controller platforms take one of two possible approaches for replication. First, the master replicates the event to other replicas immediately, leaving the slave replicas unsure whether the event is completely processed by the master. In fact, when an old master fails, the new master may not know whether the commands triggered by past events have been executed on the switches. The second alternative is that the master might choose to replicate an event only after it is completely processed (*i.e.*, all commands for the event are executed on the switches). However, if the original switch *and* later the master fail while the master is processing the event, some of the commands triggered by the event may have been executed on several switches, but the new master would never see the original event (because of the failed switch) and would not know about the affected switches. The situation could be worse if the old master left these switches in some transitional state before failing. Therefore, it is necessary to take care of these cases if one were to ensure a consistent switch state under failures.

In conclusion, the examples show that a correct protocol should meet all the aforementioned design goals. We further summarize the desired properties and the corresponding mechanisms to achieve them in Table 4.1.

4.3 Ravana Protocol

Ravana Approach: Ravana makes two main contributions. First, Ravana has a novel two-phase replication protocol that extends replicated state machines to deal with switch state consistency. Each phase involves adding event-processing information to a replicated in-memory log (built using traditional RSM mechanisms like viewstamped replication [101]). The first stage ensures that every received event is reliably replicated, and the second stage conveys whether the event-processing transaction has completed. When the master fails, another replica can use this information to continue processing events where the old master left off. Since events from a switch can trigger commands to multiple other switches,

separating the two stages (event reception and event completion) ensures that the failure of a switch along with the master does not corrupt the state on other switches.

Second, Ravana extends the existing control channel interface between controllers and switches (the OpenFlow protocol) with mechanisms that mitigate missing or repeated control messages during controller failures. In particular, (i) to ensure that messages are delivered *at least once* under failures, Ravana uses RPC-level acknowledgments and retransmission mechanisms and (ii) to guarantee *at most once* messages, Ravana associates messages with unique IDs, and performs receive-side filtering.

Thus, our protocol adopts well known distributed systems techniques as shown in Table 4.1 but combines them in a unique way to maintain consistency of both the controller and switch state under failures. To our knowledge, this enables Ravana to provide the first fault-tolerant SDN controller platform with concrete correctness properties. Also, our protocol employs novel optimizations to execute commands belonging to multiple events in parallel to decrease overhead, without compromising correctness. In addition, Ravana provides a transparent programming platform—unmodified control applications written for a single controller can be made automatically fault-tolerant without the programmer having to worry about replica failures, as discussed in Section 4.6.

Ravana has two main components—(i) a controller runtime for each controller replica and (ii) a switch runtime for each switch. These components together make sure that the SDN is fault-tolerant if at most f of the $2f+1$ controller replicas crash. This is a direct result of the fact that each phase of the controller replication protocol in turn uses Viewstamped Replication [101]. Note that we only handle crash-stop failures of controller replicas and do not focus on recovery of failed nodes. Similarly we assume that when a failed switch recovers, it starts afresh on a clean slate and is analogous to a new switch joining the network. In this section, we describe the steps for processing events in our protocol, and further discuss how the two runtime components function together to achieve our design goals.

4.3.1 Protocol Overview

To illustrate the operation of a protocol, we present an example of handling a specific event—a packet-in event. A packet arriving at a switch is processed in several steps, as shown in Figure 4.3. First, we discuss the handling of packets during normal execution without controller failures:

1. A switch receives a packet and after processing the packet, it may direct the packet to other switches.
2. If processing the packet triggers an event, the switch runtime buffers the event temporarily, and sends a copy to the master controller runtime.
3. The master runtime stores the event in a replicated in-memory log that imposes a total order on the logged events. The slave runtimes do not yet release the event to their application instances for processing.
4. After replicating the event into the log, the master acknowledges the switch. This implies that the buffered event has been reliably received by the controllers, so the switch can safely delete it.
5. The master feeds the replicated events in the log order to the controller application, where they get processed. The application updates the necessary internal state and responds with zero or more commands.
6. The master runtime sends these commands out to the corresponding switches, and waits to receive acknowledgments for the commands sent, before informing the replicas that the event is processed.
7. The switch runtimes buffer the received commands, and send acknowledgment messages back to the master controller. The switches apply the commands subsequently.
8. After all the commands are acknowledged, the master puts an *event-processed* message into the log.

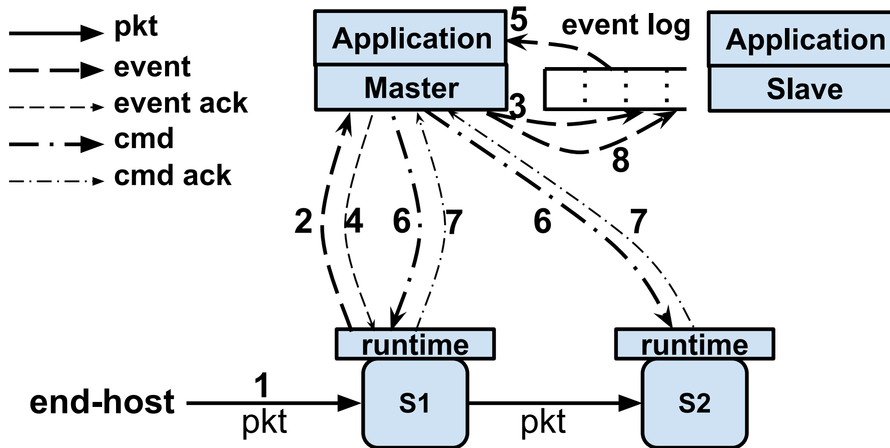


Figure 4.3: Steps for processing a packet in Ravana.

A slave runtime does not feed an event to its application instance until *after* the event-processed message is logged. The slave runtime delivers events to the application in order, waiting until each event in the log has a corresponding event-processed message before proceeding. The slave runtimes also filter the outgoing commands from their application instances, rather than actually sending these commands to switches; that is, the slaves merely simulate the processing of events to update the internal application state.

When the master controller fails, a standby slave controller will replace it following these steps:

1. A leader election component running on all the slaves elects one of them to be the new master.

2. The new master finishes processing any logged events that have their *event-processed* messages logged. These events are processed in slave mode to bring its application state up-to-date without sending any commands.

3. The new master sends *role request* messages to register with the switches in the role of the new master. All switches send a *role response* message as acknowledgment and then begin sending previously buffered events to the new master.

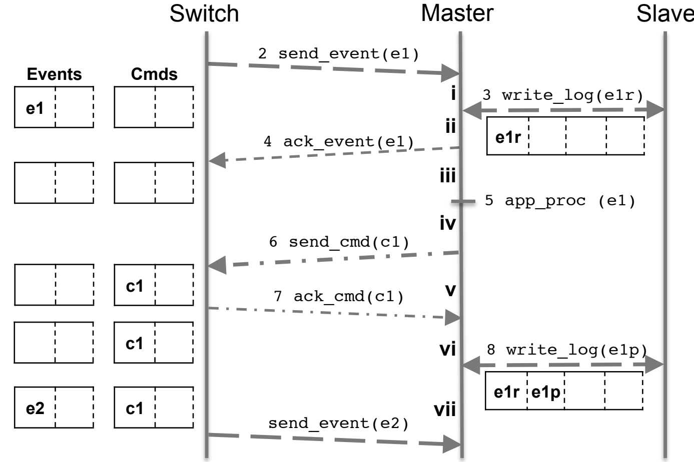


Figure 4.4: Sequence diagram of event processing in controllers: steps 2–8 are in accordance with in Figure 4.3.

4. The new master starts to receive events from the switches, and processes events (including events logged by the old master without a corresponding *event processed* message), in master mode.

4.3.2 Protocol Insights

The Ravana protocol can be viewed as a combination of mechanisms that achieve the design goals set in the previous section. By exploring the full range of controller crash scenarios (cases (i) to (vii) in Figure 4.4), we describe the key insights behind the protocol mechanisms.

Exactly-Once Event Processing: A combination of temporary event buffering on the switches and explicit acknowledgment from the controller ensures *at-least* once delivery of events. When sending an event $e1$ to the master, the switch runtime temporarily stores the event in a local event buffer (Note that this is different from the notion of buffering PacketIn payloads in OpenFlow switches). If the master crashes before replicating this event in the shared log (case (i) in Figure 4.4), the failover mechanism ensures that the switch runtime resends the buffered event to the new master. Thus the events are delivered at least once to all of the replicas. To suppress repeated events, the replicas keep track of the IDs of

the logged events. If the master crashes after the event is replicated in the log but before sending an acknowledgment (case (ii)), the switch retransmits the event to the new master controller. The new controller's runtime recognizes the duplicate eventID and filters the event. Together, these two mechanisms ensure *exactly once* processing of events at all of the replicas.

Total Event Ordering: A shared log across the controller replicas (implemented using viewstamped replication) ensures that the events received at the master are replicated in a consistent (linearized) order. Even if the old master fails (cases (iii) and (iv)), the new master preserves that order and only adds new events to the log. In addition, the controller runtime ensures exact replication of control program state by propagating information about non-deterministic primitives like timers as special events in the replicated log.

Exactly-Once Command Execution: The switches explicitly acknowledge the commands to ensure *at-least* once delivery. This way the controller runtime does not mistakenly log the event-processed message (thinking the command was received by the switch), when it is still sitting in the controller runtime's network stack (case (iv)). Similarly, if the command is indeed received by the switch but the master crashes before writing the event-processed message into the log (cases (v) and (vi)), the new master processes the event $e1$ and sends the command $c1$ again to the switch. At this time, the switch runtime filters repeated commands by looking up the local command buffer. This ensures *at-most* once execution of commands. Together these mechanisms ensure *exactly-once* execution of commands.

Consistency Under Joint Switch and Controller Failure: The Ravana protocol relies on switches retransmitting events and acknowledging commands. Therefore, the protocol must be aware of switch failure to ensure that faulty switches do not break the Ravana protocol. If there is no controller failure, the master controller treats a switch failure the same way a single controller system would treat such a failure – it relays the network port status updates to the controller application which will route the traffic around the failed switch.

Note that when a switch fails, the controller does not fail the entire transaction. Since this is a plausible scenario in the fault-free case, the runtime completes the transaction by executing commands on the set of available switches. Specifically, the controller runtime has timeout mechanisms that ensure a transaction is not stuck because of commands not being acknowledged by a failed switch. However, the Ravana protocol needs to carefully handle the case where a switch failure occurs along with a controller failure because it relies on the switch to retransmit lost events under controller failures.

Suppose the master and a switch fail sometime *after* the master receives the event from that switch but *before* the transaction completes. Ravana must ensure that the new master sees the event, so the new master can update its internal application state and issue any remaining commands to the rest of the switches. However, in this case, since the failed switch is no longer available to retransmit the event, unless the old master reliably logged the event *before* issuing any commands, the new master could not take over correctly. This is the reason why the Ravana protocol involves *two* stages of replication. The first stage captures the fact that event e is received by the master. The second stage captures the fact that the master has completely processed e, which is important to know during failures to ensure the exactly-once semantics. Thus the event-transaction dependencies across switches, a property unique to SDN, leads to this two-stage replication protocol.

4.4 Correctness

While the protocol described in the previous section intuitively gives us necessary guarantees for processing of events and execution of commands during controller failures, it is not clear if they are sufficient to ensure the abstraction of a logically centralized controller. This is also the question that recent work in this space has left unanswered. This led to a lot of subtle bugs in their approaches that have erroneous effect on the network state as illustrated in section 2.

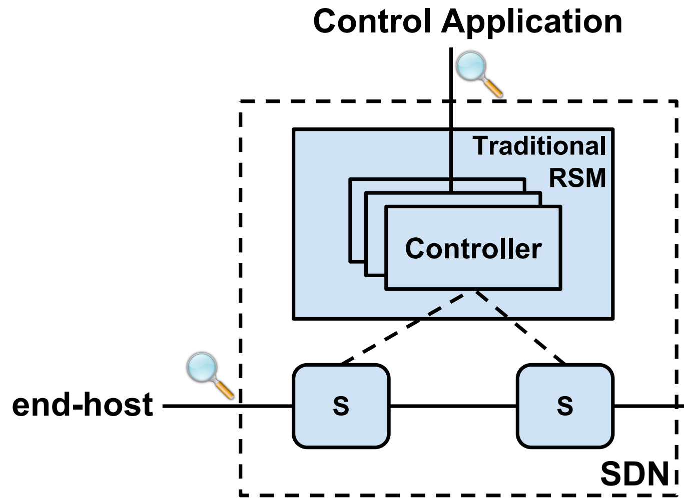


Figure 4.5: In SDN, control applications and end hosts both observe system evolution, while traditional replication techniques treat the switches (S) as observers.

Thus, we strongly believe it is important to concretely define what it means to have a logically centralized controller and then analyze whether the proposed solution does indeed guarantee such an abstraction. Ideally, a fault-tolerant SDN should behave the same way as a fault-free SDN from the viewpoint of all the users of the system.

Observational indistinguishability in SDN: We believe the correctness of a fault-tolerant SDN relies on the users—the end-host and controller applications—seeing a system that always behaves like there is a single, reliable controller, as shown in Figure 4.5. This is what it means to be a logically centralized controller. Of course, controller failures could affect performance, in the form of additional delays, packet drops, or the timing and ordering of future events. But, these kinds of variations can occur even in a fault-free setting. Instead, our goal is that the fault-tolerant system evolves in a way that *could have happened* in a fault-free execution, using observational indistinguishability [93], a common paradigm for comparing behavior of computer programs:

Definition of observational indistinguishability: If the trace of observations made by users in the fault-tolerant system is a possible trace in the fault-free system, then the fault-tolerant system is observationally indistinguishable from a fault-free system.

An observation describes the interaction between an application and an SDN component. Typically, SDN exposes *two* kinds of observations to its users: (i) end hosts observe requests and responses (and use them to evolve their own application state) and (ii) control applications observe events from switches (and use them to adapt the system to obey a high-level service policy, such as load-balancing requests over multiple switches). For example, as illustrated in section 2, under controller failures, while controllers fail to observe network failure events, end-hosts observe a drop in packet throughput compared to what is expected or they observe packets not intended for them.

Commands decide observational indistinguishability: The observations of *both* kinds of users (Figure 4.5) are preserved in a fault-tolerant SDN if the series of *commands* executed on the switches are executed just as they could have been executed in the fault-free system. The reason is that the commands from a controller can (i) modify the switch (packet processing) logic and (ii) query the switch state. Thus the commands executed on a switch determine not only what responses an end host receives, but also what events the control application sees. Hence, we can achieve observational indistinguishability by ensuring "command trace indistinguishability". This leads to the following correctness criteria for a fault-tolerant protocol:

Safety: For any given series of switch events, the resulting series of commands executed on the switches in the fault-tolerant system *could have been* executed in the fault-free system.

Liveness: Every event sent by a switch is eventually processed by the controller application, and every resulting command sent from the controller application is eventually executed on its corresponding switch.

Transactional and exactly-once event cycle: To ensure the above safety and liveness properties of observational indistinguishability, we need to guarantee that the controller replicas output a series of commands "indistinguishable" from that of a fault-free controller for any given set of input events. Hence, we must ensure that the same input is processed

by all the replicas and that no input is missing because of failures. Also, the replicas should process all input events in the same order, and the commands issued should be neither missing nor repeated in the event of replica failure.

In other words, Ravana provides transactional semantics to the entire "control loop" of (i) event delivery, (ii) event ordering, (iii) event processing, and (iv) command execution. (If the command execution results in more events, the subsequent event-processing cycles are considered separate transactions.) In addition, we ensure that any given transaction happens exactly once—it is not aborted or rolled back under controller failures. That is, once an event is sent by a switch, the entire event-processing cycle is executed till completion, and the transaction affects the network state exactly once. Therefore, our protocol that is designed around the goals listed in Table 4.1 will ensure observational indistinguishability between an ideal fault-free controller and a logically centralized but physically replicated controller. While we provide an informal argument for correctness, modeling the Ravana protocol using a formal specification tool and proving formally that the protocol is indeed sufficient to guarantee the safety and liveness properties is out of scope for this chapter and is considered part of potential future work.

4.5 Performance Optimizations

In this section, we discuss several approaches that can optimize the performance of the protocol while retaining its strong correctness guarantees.

Parallel logging of events: Ravana protocol enforces a consistent ordering of all events among the controller replicas. This is easy if the master were to replicate the events one after the other sequentially but this approach is too slow when logging tens of thousands of events. Hence, the Ravana runtime first imposes a total order on the switch events by giving them monotonically increasing log IDs and then does parallel logging of events where multiple threads write switch events to the log in parallel. After an event is reliably

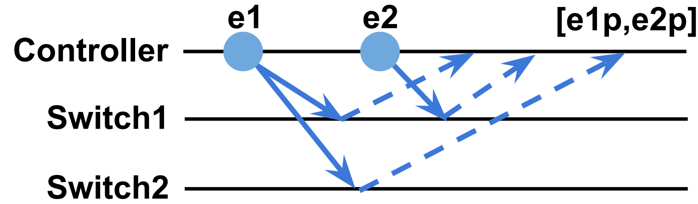


Figure 4.6: Optimizing performance by processing multiple transactions in parallel. The controller processes events $e1$ and $e2$, and the command for $e2$ is acknowledged before both the commands for $e1$ are acknowledged.

logged, the master runtime feeds the event to its application instance, but it still follows the total order. The slaves infer the total order from the log IDs assigned to the replicated events by the master.

Processing multiple transactions in parallel: In Ravana, one way to maintain consistency between controller and switch state is to send the commands for each event transaction one after the other (and waiting for switches' acknowledgments) before replicating the event processed message to the replicas. Since this approach can be too slow, we can optimize the performance by pipelining multiple commands in parallel without waiting for the ACKs. The runtime also interleaves commands generated from *multiple* independent event transactions. An internal data structure maps the outstanding commands to events and traces the progress of processing events. Figure 4.6 shows an example of sending commands for two events in parallel. In this example, the controller runtime sends the commands resulting from processing $e2$ while the commands from processing $e1$ are still outstanding.

Sending commands in parallel does not break the ordering of event processing. For example, the commands from the controller to any given individual switch (the commands for $e1$) are ordered by the reliable control-plane channel (e.g., via TCP). Thus at a given switch, the sequence of commands received from the controller must be consistent with the order of events processed by the controller. For multiple transactions in parallel, the runtime buffers the completed events till the events earlier in the total order are also completed. For example, even though the commands for $e2$ are acknowledged first, the runtime waits

till all the commands for $e1$ are acknowledged and then replicates the event processed messages for both $e1$ and $e2$ in that order. Despite this optimization, since the event processed messages are written in log order, we make sure that the slaves also process them in the same order.

Clearing switch buffers: The switch runtime maintains both an event buffer (`EBuf`) and a command buffer (`CBuf`). We add *buffer clear* messages that help garbage collect these buffers. As soon as the event is durably replicated in the distributed log, the master controller sends an `EBuf_CLEAR` message to confirm that the event is persistent. However, a `CBuf_CLEAR` is sent only when its corresponding event is done processing. An event processed message is logged only when all processing is done in the current protocol, so a slave controller gets to know that all the commands associated with the event are received by switches, and it should never send the commands out again when it becomes a master. As a result, when an event is logged, the controller sends an event acknowledgment, and at the same time piggybacks both `EBuf_CLEAR` and `CBuf_CLEAR`.

4.6 Implementation of Ravana

Implementing Ravana in SDN involves changing three important components: (i) instead of controller applications grappling with controller failures, a *controller runtime* handles them transparently, (ii) a *switch runtime* replays events under controller failures and filters repeated commands, and (iii) a modified *control channel* supports additional message types for event-processing transactions.

4.6.1 Controller Runtime: Failover, Replication

Each replica has a runtime component which handles the controller failure logic transparent to the application. The same application program runs on all of the replicas. Our prototype controller runtime uses the Ryu [19] message-parsing library to transform the raw messages on the wire into corresponding OpenFlow messages.

Leader election: The controllers elect one of them as master using a leader election component written using ZooKeeper [61], a synchronization service that exposes an atomic broadcast protocol. Much like in Google's use of Chubby [36], Ravana leader election involves the replicas contending for a ZooKeeper lock; whoever successfully gains the lock becomes the master. Master failure is detected using the ZooKeeper failure-detection service which relies on counting missed heartbeat messages. A new master is elected by having the current slaves retry gaining the master lock.

Event logging: The master saves each event in ZooKeeper's distributed in-memory log. Slaves monitor the log by registering a trigger for it. When a new event is propagated to a slave's log, the trigger is activated so that the slave can read the newly arrived event locally.

Event batching: Even though its in-memory design makes the distributed log efficient, latency during event replication can still degrade throughput under high load. In particular, the master's write call returns only after it is propagated to more than half of all replicas. To reduce this overhead, we batch multiple messages into an ordered group and write the grouped event as a whole to the log. On the other side, a slave unpacks the grouped events and processes them individually and in order.

4.6.2 Switch Runtime: Event/Command Buffers

We implement our switch runtime by modifying the Open vSwitch (version 1.10) [13], which is the most widely used software OpenFlow switch. We implement the event and command buffers as additional data structures in the OVS connection manager. If a master fails, the connection manager sends events buffered in `EBuf` to the new master as soon as it registers its new role. The command buffer `CBuf` is used by the switch processing loop to check whether a command received (uniquely identified by its transaction ID) has already been executed. These transaction IDs are remembered till they can be safely garbage collected by the corresponding `CBuf_CLEAR` message from the controller.

4.6.3 Control Channel Interface: Transactions

Changes to OpenFlow: We modified the OpenFlow 1.3 controller-switch interface to enable the two parties to exchange additional Ravana-specific metadata: `EVENT_ACK`, `CMD_ACK`, `EBuf_CLEAR`, and `CBuf_CLEAR`. The ACK messages acknowledge the receipt of events and commands, while CLEAR help reduce the memory footprint of the two switch buffers by periodically cleaning them. As in OpenFlow, all messages carry a transaction ID to specify the event or command to which it should be applied.

Unique transaction IDs: The controller runtime associates every command with a unique transaction ID (XID). The XIDs are monotonically increasing and identical across all replicas, so that duplicate commands can be identified. This arises from the controllers' deterministic ordered operations and does not require an additional agreement protocol. In addition, the switch also needs to ensure that unique XIDs are assigned to events sent to the controller. We modified Open vSwitch to increment the XID field whenever a new event is sent to the controller. Thus, we use 32-bit unique XIDs (with wrap around) for both events and commands.

4.6.4 Transparent Programming Abstraction

Ravana provides a fault-tolerant controller runtime that is completely transparent to control applications. The Ravana runtime intercepts all switch events destined to the Ryu application, enforces a total order on them, stores them in a distributed in-memory log, and only then delivers them to the application. The application updates the controller internal state, and generates one or more commands for each event. Ravana also intercepts the outgoing commands — it keeps track of the set of commands generated for each event in order to trace the progress of processing each event. After that, the commands are delivered to the corresponding switches. Since Ravana does all this from inside Ryu, existing single-

threaded Ryu applications can directly run on Ravana without modifying a single line of code.

To demonstrate the transparency of programming abstraction, we have tested a variety of Ryu applications [20]: a MAC learning switch, a simple traffic monitor, a MAC table management app, a link aggregation (LAG) app, and a spanning tree app. These applications are written using the Ryu API, and they run on our fault-tolerant control platform without any changes.

Currently we expect programmers to write controller applications that are single-threaded and deterministic, similar to most replicated state machine systems available today. An application can introduce nondeterminism by using timers and random numbers. Our prototype supports timers and random numbers through a standard library interface. The master runtime treats function calls through this interface as special events and persists the event metadata (timer begin/end, random seeds, etc.) into the log. The slave runtimes extract this information from the log so their application instances execute the same way as the master's. State-machine replication with multi-threaded programming has been studied [114], and supporting it in Ravana is future work.

4.7 Performance Evaluation

To understand Ravana's performance, we evaluate our prototype to answer the following questions:

- What is the overhead of Ravana's fault-tolerant runtime on event-processing throughput?
- What is the effect of the various optimizations on Ravana's event-processing throughput and latency?
- Can Ravana respond quickly to controller failure?

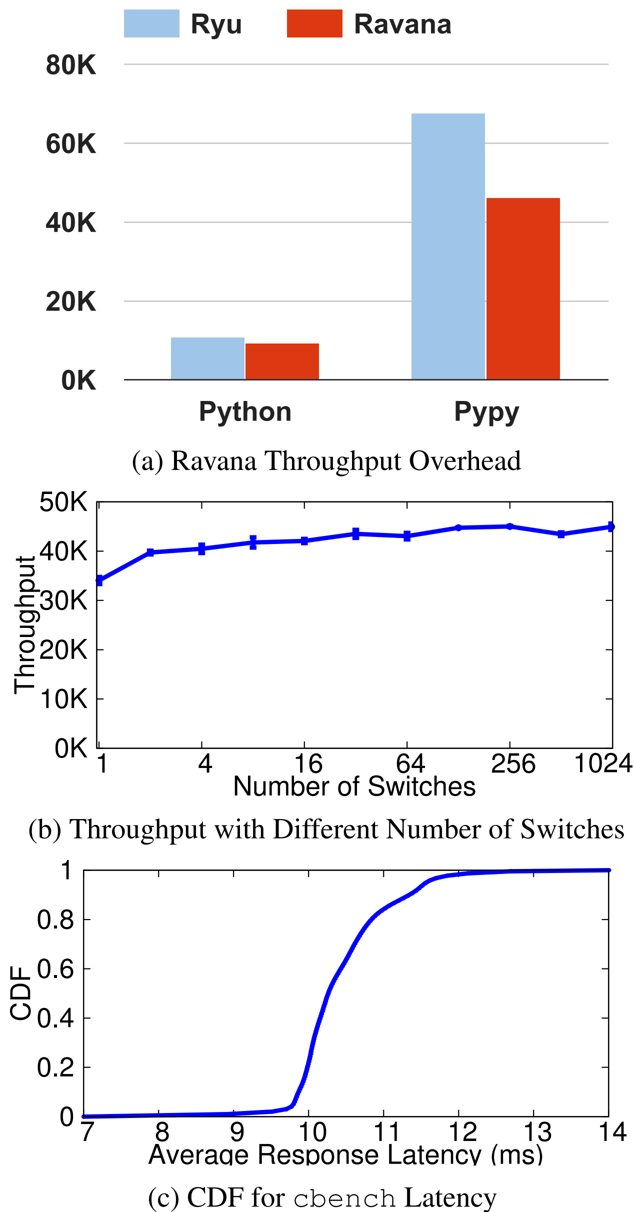
- What are the throughput and latency trade-offs for various correctness guarantees?

We run experiments on three machines connected by 1Gbps links. Each machine has 12GB memory and an Intel Xeon 2.4GHz CPU. We use ZooKeeper 3.4.6 for event logging and leader election. We use the Ryu 3.8 controller platform as our non-fault-tolerant baseline.

4.7.1 Measuring Throughput and Latency

We first compare the throughput (in terms of flow responses per second) achieved by the vanilla Ryu controller and the Ravana prototype we implemented on top of Ryu, in order to characterize Ravana's overhead. Measurements are done using the `cbench` [17] performance test suite: the test program spawns a number of processes that act as OpenFlow switches. In `cbench`'s throughput mode, the processes send PacketIn events to the controller as fast as possible. Upon receiving a PacketIn event from a switch, the controller sends a command with a forwarding decision for this packet. The controller application is designed to be simple enough to give responses without much computation, so that the experiment can effectively benchmark the Ravana protocol stack.

Figure 4.7a shows the event-processing throughput of the vanilla Ryu controller and our prototype in a fault-free execution. We used both the standard Python interpreter and PyPy (version 2.2.1), a fast Just-in-Time interpreter for Python. We enable batching with a buffer of 1000 events and 0.1s buffer time limit. Using standard Python, the Ryu controller achieves a throughput of 11.0K responses per second (rps), while the Ravana controller achieves 9.2K, with an overhead of 16.4%. With PyPy, the event-processing throughput of Ryu and Ravana are 67.6K rps and 46.4K rps, respectively, with an overhead of 31.4%. This overhead includes the time of serializing and propagating all the events among the three controller replicas in a failure-free execution. We consider this as a reasonable overhead given the correctness guarantee and replication mechanisms Ravana added.



(a) Ravana Throughput Overhead

(b) Throughput with Different Number of Switches

(c) CDF for `cbench` Latency

Figure 4.7: Ravana Event-Processing Throughput and Latency

To evaluate the runtime's scalability with increasing number of switch connections, we ran `cbench` in throughput mode with a large number of simulated switch connections. A series of throughput measurements are shown in Figure 4.7b. Since the simulated switches send events at the highest possible rate, as the number of switches become reasonably large, the controller processing rate saturates but does not go down. The event-processing

throughput remains high even when we connect over one thousand simulated switches. The result shows that the controller runtime can manage a large number of parallel switch connections efficiently.

Figure 4.7c shows the latency CDF of our system when tested with `cbench`. In this experiment, we run `cbench` and the master controller on the same machine, in order to benchmark the Ravana event-processing time without introducing extra network latency. The latency distribution is drawn using the average latency calculated by `cbench` over 100 runs. The figure shows that most of the events can be processed within 12ms.

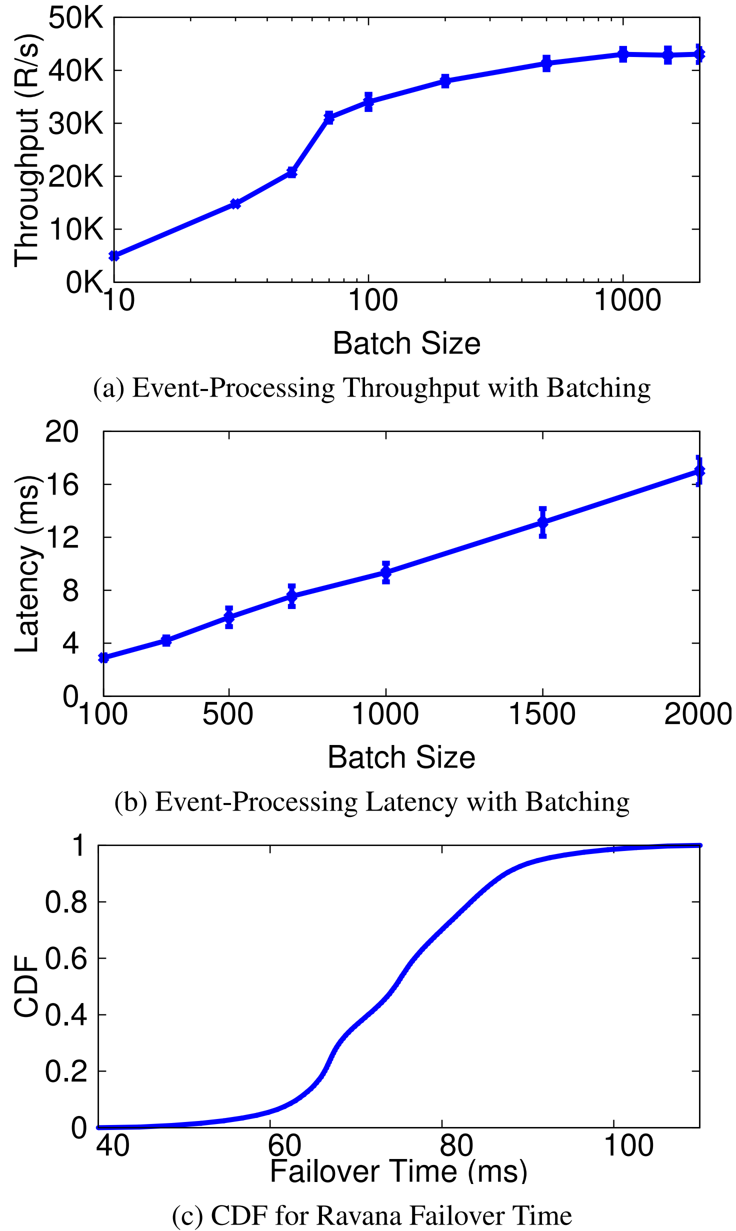
4.7.2 Sensitivity Analysis for Event Batching

The Ravana controller runtime batches events to reduce the overhead for writing several events in the ZooKeeper event log. Network operators need to tune the *batching size* parameter to achieve the best performance. A batch of events are flushed to the replicated log either when the batch reaches the size limit or when no event arrives within a certain time limit.

Figure 4.8a shows the effect of batching sizes on event processing throughput measured with `cbench`. As batching size increases, throughput increases due to reduction in the number of RPC calls needed to replicate events. However, when batching size increases beyond a certain number, the throughput saturates because the performance is bounded by other system components (marshalling and unmarshalling OpenFlow messages, event processing functions, etc.)

While increasing batching size can improve throughput under high demand, it also increases event response latency. Figure 4.8b shows the effect of varying batch sizes on the latency overhead. The average event processing latency increases almost linearly with the batching size, due to the time spent in filling the batch before it is written to the log.

The experiment results shown in Figure 4.8a and 4.8b allow network operators to better understand how to set an appropriate batching size parameter based on different require-



(a) Event-Processing Throughput with Batching

(b) Event-Processing Latency with Batching

(c) CDF for Ravana Failover Time

Figure 4.8: Variance of Ravana Throughput, Latency and Failover Time

ments. If the application needs to process a large number of events and can tolerant relatively high latency, then a large batch size is helpful; if the events need to be instantly processed and the number of events is not a big concern, then a small batching size will be more appropriate.

4.7.3 Measuring Failover Time

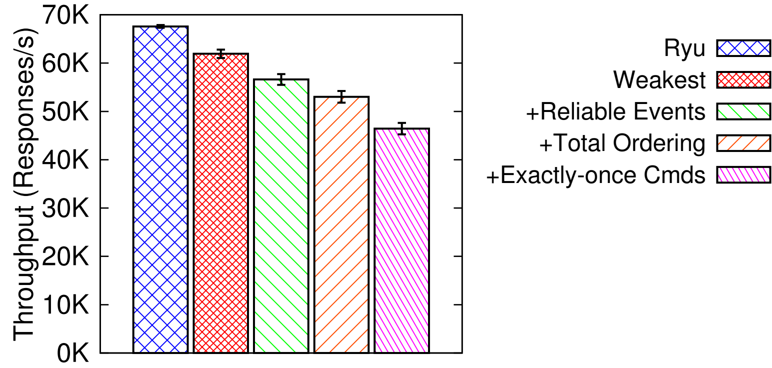
When the master controller crashes, it takes some time for the new controller to take over. To evaluate the efficiency of Ravana controller failover mechanism, we conducted a series of tests to measure the failover time distribution, as shown in Figure 4.8c. In this experiment, a software switch connects two hosts which continuously exchange packets that are processed by the controller in the middle. We bring down the master. The end hosts measure the time for which no traffic is received during the failover period. The result shows that the average failover time is 75ms, with a standard deviation of 9ms. This includes around 40ms to detect failure and elect a new leader (with the help of ZooKeeper), around 25ms to catch up with the old master (can be reduced further with optimistic processing of the event log at the slave) and around 10ms to register the new role on the switch. The short failover time ensures that the network events generated during this period will not be delayed for a long time before getting processed by the new master.

4.7.4 Consistency Levels: Overhead

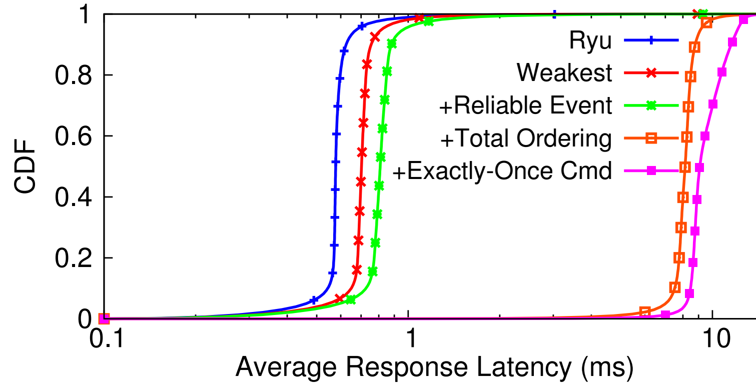
Our design goals ensure strict correctness in terms of observational indistinguishability for general SDN policies. However, as show in Figure 4.9, some guarantees are costlier to ensure (in terms of through/latency) than the others. In particular, we looked at each of the three design goals that adds overhead to the system compared to the weakest guarantee.

The weakest consistency included in this study is the same as what existing fault-tolerant controller systems provide — the master immediately processes events once they arrive, and replicates them lazily in the background. Naturally this avoids the overhead of all the three design goals we aim for and hence has only a small throughput overhead of 8.4%.

The second consistency level is enabled by guaranteeing exactly-once processing of switch events received at the controller. This involves making sure that the master syn-



(a) Throughput Overhead for Correctness Guarantees



(b) Latency Overheads for Correctness Guarantees

Figure 4.9: Throughput and Latency Overheads with Varied Levels of Correctness Guarantees

chronously logs the events and explicitly sends event ACKs to corresponding switches. This has an additional throughput overhead of 7.8%.

The third consistency level ensures total event ordering in addition to exactly-once events. This makes sure that the order in which the events are written to the log is the same as the order in which the master processes them, and hence involves mechanisms to strictly synchronize the two. Ensuring this consistency incurs an additional overhead of 5.3%.

The fourth and strongest consistency level ensures exactly-once execution of controller commands. It requires the switch runtime to explicitly ACK each command from the controller and to filter repeated commands. This adds an additional overhead of 9.7% on the controller throughput. Thus adding all these mechanisms ensures the strongest collection of

correctness guarantees possible under Ravana and the cumulative overhead is 31%. While some of the overheads shown above can be reduced further with implementation specific optimizations like cumulative and piggybacked message ACKs, we believe the overheads related to replicating the log and maintaining total ordering are unavoidable to guarantee protocol correctness.

Latency. Figure 4.9b shows the CDF for the average time it takes for the controller to send a command in response to a switch event. Our study reveals that the main contributing factor to the latency overhead is the synchronization mechanism that ties event logging to event processing. This means that the master has to wait till a switch event is properly replicated and only then processes the event. This is why all the consistency levels that do not involve this guarantee have a latency of around 0.8ms on average but those that involve the total event ordering guarantee have a latency of 11ms on average.

Relaxed Consistency. The Ravana protocol described in this chapter is oblivious to the nature of control application state or the various types of control messages processed by the application. This is what led to the design of a truly transparent runtime that works with unmodified control applications. However, given the breakdown in terms of throughput and latency overheads for various correctness guarantees, it is natural to ask if there are control applications that can benefit from relaxed consistency requirements.

For example, a Valiant load balancing application that processes flow requests (PacketIn events) from switches and assigns paths to flows randomly is essentially a stateless application. So the constraint on total event ordering can be relaxed entirely for this application. But if this application is run in conjunction with a module that also reacts to topology changes (PortStatus events), then it makes sense to enable to constraint just for the topology events and disable it for the rest of the event types. This way, both the throughput and latency of the application can be improved significantly.

A complete study of which applications benefit from relaxing which correctness constraints and how to enable programmatic control of runtime knobs is out of scope for this

	Total Event Ordering	Exactly-Once Events	Exactly-Once Commands	Transparency
Distributed Storage	✓	✗	✗	✗
Switch Broadcast	✗	✓	✗	✓
State Replication	✓	✗	✗	✓
Ravana	✓	✓	✓	✓

Table 4.2: Comparing different solutions for fault-tolerant controllers

chapter. However, from a preliminary analysis, many applications seem to benefit from either completely disabling certain correctness mechanisms or only partially disabling them for certain kinds of OpenFlow messages.

4.8 Related Work

Distributed SDN control with consistent reliable storage: The Onix [82] distributed controller partitions application and network state across multiple controllers using distributed storage. Switch state is stored in a strongly consistent Network Information Base (NIB). Controllers subscribe to switch events in the NIB and the NIB publishes new events to subscribed controllers independently and in an eventually consistent manner. This could violate the total event ordering correctness constraint. Since the paper is underspecified on some details, it is not clear how Onix handles simultaneous occurrence of controller crashes and network events (like link/switch failures) that can affect the commands sent to other switches. In addition, programming control applications is difficult since the applications have to be conscious of the controller fault-tolerance logic. Onix does however handle continued distributed control under network partitioning for both scalability and performance, while Ravana is concerned only with reliability. ONOS [32] is an experimental controller platform that provides a distributed, but logically centralized, global network view; scale-out; and fault tolerance by using a consistent store for replicating application state. However, owing to its similarities to Onix, it also suffers from reliability guarantees as Onix does.

Distributed SDN control with state machine replication: HyperFlow [119] is an SDN controller where network events are replicated to the controller replicas using a publish-subscribe messaging paradigm among the controllers. The controller application publishes and receives events on subscribed channels to other controllers and builds its local state solely from the network events. In this sense, the approach to building application state is similar to Ravana but the application model is non-transparent because the application bears the burden of replicating events. In addition, HyperFlow also does not deal with the correctness properties related to the switch state.

Distributed SDN with weaker ordering requirements: Early work on software-defined BGP route control [37, 121] allowed distributed controllers to make routing decisions for an Autonomous System. These works do not ensure a total ordering on events from different switches, and instead rely on the fact that the final outcome of the BGP decision process does not depend on the relative ordering of messages from different switches. This assumption does not hold for arbitrary applications.

Traditional fault-tolerance techniques: A well-known protocol for replicating state machines in client-server models for reliable service is Viewstamped Replication (VSR) [101]. VSR is not directly applicable in the context of SDN, where switch state is as important as the controller state. In particular, this leads to missing events or duplicate commands under controller failures, which can lead to incorrect switch state. Similarly, Paxos [83] and Raft [102] are distributed consensus protocols that can be used to reach a consensus on input processed by the replicas but they do not address the effects on state external to the replicas. Fault-tolerant journaling file systems [106] and database systems [95] assume that the commands are idempotent and that replicas can replay the log after failure to complete transactions. However, the commands executed on switches are not idempotent. The authors of [60] discuss strategies for ensuring exactly-once semantics in replicated messaging systems. These strategies are similar to our mechanisms for exactly-once event semantics

but they cannot be adopted directly to handle cases where failure of a switch can effect the dataplane state on other switches.

TCP fault-tolerance: Another approach is to provide fault tolerance within the network stack using TCP failover techniques [81, 88, 124]. These techniques have a huge overhead because they involve reliable logging of each packet or low-level TCP segment information, in both directions. In our approach, much fewer (application-level) events are replicated to the slaves.

VM fault-tolerance: Remus [42] and Kemari [18] are techniques that provide fault-tolerant virtualization environments by using live VM migration to maintain availability. These techniques synchronize all in-memory and on-disk state across the VM replicas. The domain-agnostic checkpointing can lead to correctness issues for high-performance controllers. Thus, they impose significant overhead because of the large amount of state being synchronized.

Observational indistinguishability: Lime [52] uses a similar notion of observational indistinguishability, in the context of live switch migration (where multiple switches emulate a single virtual switch) as opposed to multiple controllers.

Statesman [118] takes the approach of allowing incorrect switch state when a master fails. Once the new master comes up, it reads the current switch state and incrementally migrates it to a target switch state determined by the controller application. LegoSDN [40] focuses on application-level fault-tolerance caused by application software bugs, as opposed to complete controller crash failures. Akella et. al. [22] tackle the problem of network availability when the control channel is in-band whereas our approach assumes a separate out-of-band control channel. The approach is also heavy-handed where every element in the network including the switches is involved in a distributed snapshot protocol. Beehive [122] describes a programming abstraction that makes writing distributed control applications for SDN easier. However, while the focus in Beehive is on controller scalability, they do not discuss consistent handling of the switch state.

4.9 Conclusion

Ravana is a distributed protocol for reliable control of software-defined networks. In our future research, we plan to create a formal model of our protocol and use verification tools to prove its correctness. We also want to extend Ravana to support multi-threaded control applications, richer failure models (such as Byzantine failures), and more scalable deployments where each controller manages a smaller subset of switches.

Chapter 5

Conclusion

The primary appeal of Software-Defined Networking is that the network operator can write programs on top of simple high-level abstractions in order to achieve her objectives. However, in practice, the implementation of these abstractions are either (i) inefficient due to shift in burden to the controller or (ii) inflexible due to incapable switch resources or (iii) unreliable due to faulty network components. Subsequently, the operator takes on extra burden in terms of additional low-level mechanisms to handle the various shortcomings. This results in complicating the use of what are otherwise simple abstractions in SDN.

This thesis proposes runtime mechanisms that achieve the goals of efficiency, flexibility and reliability in a completely transparent manner. The operator simply builds the network on top of three simple abstractions — big non-blocking switch, one switch with infinite rule space, and one logically centralized controller. The underlying runtimes provide an architecture where basic routing is done efficiently at dataplane timescales, policy enforcement is done scalably with the help of software data planes and the control plane is fault-tolerant.

5.1 Summary of Contributions

In our HULA work, we provide the abstraction of one big non-blocking switch with capacity equivalent to that of the bisection bandwidth of the network. This is achieved by a dataplane load balancing algorithm that is built on top of recently proposed programmable dataplanes. HULA uses periodic probes that proactively propagate global utilization information to all the switches. The switches use this information to track the *best path* to a destination through a neighboring switch. Using extensive packet level simulations in NS2, we show that HULA outperforms state-of-the-art alternatives by 1.6$\times$ at 50% load and 3$\times$ at 90% load in terms of end-to-end flow completion times. We also show empirically that HULA is robust to a wide range of parameter settings and is stable under perturbation. In addition, we show how to implement HULA in the P4 language that targets a wide variety of programmable dataplanes which in turn decouples HULA from vendor-specific hardware capabilities in the spirit of SDN.

In CacheFlow, we provide the abstraction of a switch with logically infinite rule space. We show how to give applications the illusion of high-speed forwarding, large rule tables, and fast updates by combining the best of hardware and software processing. CacheFlow "caches" the most popular rules in the small TCAM, while relying on software to handle the small amount of "cache miss" traffic. However, we cannot blindly apply existing cache-replacement algorithms, because of dependencies between rules with overlapping patterns. Rather than cache large chains of dependent rules, we "splice" long dependency chains to cache smaller groups of rules while preserving the semantics of the policy. We discuss how CacheFlow preserves the semantics of the OpenFlow interface so that our abstraction works on top of existing controllers and switches without any changes . Experiments with our CacheFlow prototype—on both real and synthetic workloads and policies—demonstrate that CacheFlow achieves a cache-hit rate of 90% of the traffic by caching less than 5% of

the rules. This proves that rule splicing makes effective use of limited TCAM space, while adapting quickly to changes in the policy and the traffic demands.

In Ravana, we provide the abstraction of a logically centralized controller. We design a fault-tolerant SDN controller platform that processes the control messages transactionally and exactly once (at both the controllers and the switches). Ravana maintains these guarantees in the face of both controller and switch crashes. The key insight in Ravana is that replicated state machines can be extended with lightweight switch-side mechanisms to guarantee correctness, without involving the switches in an elaborate consensus protocol. The Ravana fault-tolerant control protocol ensures *observational equivalence* between an ideal centralized controller and a physically replicated controller platform. In addition, our prototype implementation of Ravana enables *unmodified* controller applications to execute in a fault-tolerant fashion. Experiments show that Ravana achieves good performance with a reasonable 31% throughput overhead, compared to a single controller. We also study how each of our design goals add to the overhead of the system. Our failover time remains under 100ms which is acceptable in most modern distributed system settings.

5.2 Future Work

In this section, we discuss where our effort in making SDN more efficient and reliable is headed. We discuss how to extend the three main ideas proposed in this thesis so that the underlying abstractions are either more efficient or more flexible or both.

5.2.1 Heterogenous and Incremental HULA

While HULA shows how to design a scalable load balancing in the dataplane, it needs to be made flexible enough to accomodate a wide variety of devices and objectives in order to be flexible and incrementally deployable.

Heterogenous objectives. The current version of HULA is designed to achieve an important network-wide objective — to minimize the bottleneck link utilization in the net-

work. For this, each switch in HULA maintains local link utilization (based on an exponential moving average of bytes sent) and tracks the minimum among the global path utilization values. However, if the objective of the system were to minimize flow latency or jitter, or to maximize throughput of certain paths, then HULA will have to use a different sets of local and global metrics and different load balancing schemes.

Heterogenous networks. Currently, HULA assumes that all the switches are in one administrative domain and that the switches in the network support all the primitive capabilities in a P4 [33] supported target. However, in cloud scale networks, network traffic typically passes through multiple domains where each domain may have different switch capabilities and may have different routing policies (like path preferences etc.). For example, while intra-domain routing simply may pick shortest paths, inter-domain routing could depend on route preferences. HULA at that scale will have to handle the heterogeneity of networks towards the destination in the presence of multiple routing domains.

Incremental deployemnt. When network operators running traditional networks migrate to P4-capable switches, they need a strategy to incrementally deploy HULA with minimal disruption to ongoing traffic. This means HULA will have to work with some set of programmable dataplanes and some which run traditional routing protocols. Another possibility is that some vendors may just support network monitoring functions like INT [78] that simply export link utilization and not the other capabilities of P4 like stateful packet processing. In this case, HULA will have to modify its current distance vector type load balancing scheme to accomodate such heterogenous capabilities in the 'underlay'.

Heterogenous applications. Today, HULA does not distinguish between multiple applications running on top of the underlying network. It treats all of them equally. However, in large-scale deployments, operators typically require quality-of-service guarantees expressed in terms of bandwidth or latency guarantees for certain set of applications. Operators may also express prioritization of applications when the network is congested. While HULA currently deals only with unicast traffic, multicast applications are in vogue in to-

days networks. HULA should be extended to accomodate these wide range of quality of service requirements.

5.2.2 Cooperative Caching for Efficient Resource Utilization

In CacheFlow, rule caching is performed independently for each switch, considering the user given switch policy and the local traffic distribution. However, neighboring switches often have similar rule tables. The similarity arises from the network operator typically using, say, the same firewall or access control policy on all the switches in the network (albeit with some small differences to account for the switch's placement in the topology). If the rules that are same across switches actually carry a substantial amount of traffic, then caching the rule in one switch and then asking the neighboring switches to send the matching traffic to this switch would save the cost of installing the said rule and its dependents on all the switches.

In addition, different switches might see different traffic distributions depending on the location or the role of the switch in a given topology. However, recent measurements [80] indicate large intersection between the traffic working sets in multiple switches of the same network. In this context, it is worth asking if a switch can borrow an adjacent switch's TCAM for completing some of its packet classification. Recent work in Difane [123] describes one such approach where a set of authoritative switches are used as a backup for classifying cache misses from the edge switches. This shows that keeping packets in fast hardware datapath typically offers better performance than the slow software datapath, even though some traffic follows a slightly longer path and imposes additional load on the links. However, Difane only lets the authoritative switches help the edge switches but not the other way round. As part of future work for CacheFlow, one can explore the possibility of switches mutually *co-operating* with each other for the purpose of rule caching.

Cooperative Caching [44] is a well-studied approach to improve the performance of a distributed file systems by coordinating the cache policies among clients. To increase the

total cache hit rate, the cache of one client can serve requests of other clients and design of the caching policy takes into account the space availability on each cache. Various schemes requiring different levels of cooperation have been described [109]: Starting from locally independent caching policies to the ability to serve cache misses of the other cache. In the context of CacheFlow, It will be intersting to study *the applicability of cooperative caching for rule caching* in software-defined networks. Can we allow packets to be forwarded to other switches for completing the classification process? This way, we increase the ratio of traffic classified in the data plane while using the switch TCAM resources more efficiently.

5.2.3 Ravana with Runtime Knobs for Consistency Requirements

The Ravana protocol described in this thesis is oblivious to the nature of control application state or the various types of control messages processed by the application. This is what led to the design of a truly transparent runtime that works with unmodified control applications. However, given the breakdown in terms of throughput and latency overheads for various correctness guarantees, it is natural to ask if there are control applications that can benefit from relaxed consistency requirements.

For example, a Valiant load balancing application that processes flow requests (PacketIn events) from switches and assigns paths to flows randomly is essentially a stateless application. So the constraint on total event ordering can be relaxed entirely for this application. But if this application is run in conjunction with a module that also reacts to topology changes (PortStatus events), then it makes sense to enable the constraint just for the topology events and disable it for the rest of the event types. This way, both the throughput and latency of the application can be improved significantly.

A complete study of which applications benefit from relaxing which correctness constraints and how to enable programmatic control of runtime knobs is an interesting extension to Ravana. However, from a preliminary analysis as shown in chapter 4.7, many

applications seem to benefit from either completely disabling certain correctness mechanisms or only partially disabling them for certain kinds of OpenFlow messages.

5.3 Concluding Remarks

The systems described in this thesis attempt to solve three pertinent and timely issues with the practice of software-defined networking. Taken together, these systems portend a new network architecture where basic routing is done efficiently at dataplane timescales, policy enforcement is done scalably with the help of software data planes and the control plane is fault-tolerant. Thus the new architecture has the properties of fast routing and fault-tolerance of traditional networks while delivering the promise of efficient enforcement of fine-grained control policies.

This also signals a movement from both of the extremes of system design to a middle approach where there is a thin software layer on the switches that helps with processing cache misses and the controller's burden of load balancing at smaller time scales is lowered.

Bibliography

[1] Private communication with the authors of CONGA.

[2] Cavium and XPliant introduce a fully programmable switch silicon family scaling to 3.2 terabits per second. http://tinyurl.com/nzbqtr3.

[3] High Capacity StrataXGS®Trident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[4] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[5] Members of the p4 consortium. http://p4.org/join-us/.

[6] OSPF Version 2. https://tools.ietf.org/html/rfc2178.

[7] P4 Specification. http://p4.org/wp-content/uploads/2015/11/p4-v1.1rc-Nov-17.pdf.

[8] P4's action-execution semantics and conditional operators. https://github.com/anirudhSK/p4-semantics/raw/master/p4-semantics.pdf.

[9] Protocol-independent switch architecture. http://schd.ws/hosted_files/p4workshop2015/c9/NickM-P4-Workshop-June-04-2015.pdf.

[10] REANZZ. http://reannz.co.nz/.

[11] Routing Information Protocol. https://tools.ietf.org/html/rfc1058.

[12] Stanford backbone router forwarding configuration. http://tinyurl.com/oaezlha.

[13] The rise of soft switching. See http://networkheresy.com/category/open-vswitch/, 2011.

[14] Intel DPDK overview. See http://www.intel.com/content/dam/www/public/us/en/documents/presentation/dpdk-packet-processing-ia-overview-presentation.pdf, 2012.

[15] SDN system performance. See http://pica8.org/blogs/?p=201, 2012.

[16] TCAMs and OpenFlow: What every SDN practitioner must know. See http://www.sdncentral.com/products-technologies/sdn-openflow-tcam-need-to-know/2012/07/, 2012.

[17] Cbench - scalable cluster benchmarking. See http://sourceforge.net/projects/cbench/, 2014.

[18] Kemari. See http://wiki.qemu.org/Features/FaultTolerance, 2014.

[19] Ryu software-defined networking framework. See http://osrg.github.io/ryu/, 2014.

[20] Ryubook 1.0 documentation. See http://osrg.github.io/ryu-book/en/html/, 2014.

[21] Cisco's massively scalable data center. http://www.cisco.com/c/dam/en/us/td/docs/solutions/Enterprise/Data_Center/MSDC/1-0/MSDC_AAG_1.pdf, Sept 2015.

[22] Aditya Akella and Arvind Krishnamurthy. A Highly Available Software Defined Fabric. In *HotNets*, August 2014.

[23] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. NSDI 2010, pages 19–19, Berkeley, CA, USA. USENIX Association.

[24] M. Alizadeh and T. Edsall. On the data path performance of leaf-spine datacenter fabrics. In *HotInterconnects 2013*, pages 71–74.

[25] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. *SIGCOMM Comput. Commun. Rev.*, 44(4):503–514, August 2014.

[26] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). SIGCOMM 2010, pages 63–74. ACM.

[27] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. SIGCOMM 2013, pages 435–446, New York, NY, USA. ACM.

[28] Eleftheria Athanasopoulou, Loc X. Bui, Tianxiong Ji, R. Srikant, and Alexander Stolyar. Back-pressure-based packet-by-packet adaptive routing in communication networks. *IEEE/ACM Trans. Netw.*, 21(1):244–257, February 2013.

[29] Baruch Awerbuch and Tom Leighton. A simple local-control approximation algorithm for multicommodity flow. pages 459–468, 1993.

[30] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. NSDI 2015, pages 455–468. USENIX Association.

[31] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. CoNEXT 2011, pages 8:1–8:12. ACM.

[32] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *HotSDN*, August 2014.

[33] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[34] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM*, 2013.

[35] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[36] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *OSDI*, November 2006.

[37] Matthew Caesar, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *NSDI*, May 2005.

[38] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet load-balanced, low-latency routing for clos-based data center networks. CoNEXT 2013, pages 49–60. ACM.

[39] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM*, August 2007.

[40] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating SDN Application Failures with LegoSDN. In *HotNets*, August 2014.

[41] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

[42] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, April 2008.

[43] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *ACM SIGCOMM*, 2011.

[44] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *USENIX OSDI*, 1994.

[45] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *SOSP*, pages 15–28, New York, NY, USA, 2009. ACM.

[46] Qunfeng Dong, Suman Banerjee, Jia Wang, and Dheeraj Agrawal. Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough. In *ACM SIGMETRICS*, pages 253–264, New York, NY, USA, 2007. ACM.

[47] A. Elwalid, Cheng Jin, S. Low, and I. Widjaja. Mate: Mpls adaptive traffic engineering. In *IEEE INFOCOM 2001*, pages 1300–1309 vol.3.

[48] D.C. Feldmeier. Improving gateway performance with a routing-table cache. In *IEEE INFOCOM*, pages 298–307, 1988.

[49] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of ICFP '11*.

[50] R.G. Gallager. A minimum delay routing algorithm using distributed computation. *Communications, IEEE Transactions on*, 25(1):73–85, Jan 1977.

[51] Soudeh Ghorbani, Cole Schlesinger, Matthew Monaco, Eric Keller, Matthew Caesar, Jennifer Rexford, and David Walker. Transparent, live migration of a software-defined network. SOCC '14, pages 3:1–3:14, New York, NY, USA. ACM.

[52] Soudeh Ghorbani, Cole Schlesinger, Matthew Monaco, Eric Keller, Matthew Caesar, Jennifer Rexford, and David Walker. Transparent, Live Migration of a Software-Defined Network. In *SOCC*, November 2014.

[53] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev.*, 39(4):51–62, August 2009.

[54] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: A high performance, server-centric network architecture for modular data centers. SIGCOMM 2009, pages 63–74. ACM.

[55] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-accelerated software router. In *ACM SIGCOMM*, pages 195–206, New York, NY, USA, 2010. ACM.

[56] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *SIGCOMM*, 2015.

[57] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. SIGCOMM 2013, pages 15–26. ACM.

[58] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. NSDI 2015, pages 15–28. USENIX Association.

[59] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, August 2013.

[60] Yongqiang Huang and Hector Garcia-Molina. Exactly-once Semantics in a Replicated Messaging System. In *ICDE*, April 2001.

[61] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, June 2010.

[62] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[63] 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. B4: Experience with a globally-deployed software defined wan. SIGCOMM 2013, pages 3–14. ACM.

[64] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. Eyeq: Practical network performance isolation at the edge. NSDI 2013, pages 297–312, Berkeley, CA, USA. USENIX Association.

[65] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *NSDI*, 2015.

[66] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, 2014.

[67] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. SIGCOMM 2005, pages 253–264. ACM.

[68] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, March 2007.

[69] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "one big switch" abstraction in software-defined networks. CoNEXT '13, New York, NY, USA. ACM.

[70] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the 'one big switch' abstraction in Software Defined Networks. In *ACM SIGCOMM CoNext*, December 2013.

[71] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *IEEE INFOCOM Mini-conference*, April 2013.

[72] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Infinite Cacheflow in software-defined networks. In *HotSDN Workshop*, 2014.

[73] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 6:1–6:12, New York, NY, USA, 2016. ACM.

[74] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 10:1–10:12, New York, NY, USA, 2016. ACM.

[75] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 4:1–4:12, New York, NY, USA, 2015. ACM.

[76] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[77] Samir Khuller, Anna Moss, and Joseph (Seffi) Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1):39–45, April 1999.

[78] Changhoon Kim, , Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J. Wobker. In-band network telemetry via programmable dataplanes. Demo paper at SIGCOMM '15.

[79] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting route caching: The world should be flat. In *Passive and Active Measurement*, pages 3–12, Berlin, Heidelberg, 2009. Springer-Verlag.

[80] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting route caching: The world should be flat. In *Passive and Active Network Measurement (PAM)*, 2009.

[81] R. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith. Transparent TCP Connection Failover. In *DSN*, June 2003.

[82] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, October 2010.

[83] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, May 1998.

[84] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Trans. Netw.*, 18(2):490–500, April 2010.

[85] Huan Liu. Routing prefix caching in network processor design. In *International Conference on Computer Communications and Networks*, pages 18–23, 2001.

[86] Yaoqing Liu, Syed Obaid Amin, and Lan Wang. Efficient FIB caching using minimal non-overlapping prefixes. *SIGCOMM Comput. Commun. Rev.*, January 2013.

[87] Guohan Lu, Rui Miao, Yongqiang Xiong, and Chuanxiong Guo. Using cpu as a traffic co-processing unit in commodity switches. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 31–36, New York, NY, USA, 2012. ACM.

[88] Manish Marwah and Shivakant Mishra. TCP Server Fault Tolerance Using Connection Migration to a Backup Server. In *DSN*, June 2003.

[89] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, April 2008.

[90] Chad R. Meiners, Alex X. Liu, and Eric Torng. Topological transformation approaches to tcam-based packet classification. volume 19, February 2011.

[91] Chad R. Meiners, Alex X. Liu, and Eric Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.*, 20(2), April 2012.

[92] N. Michael and A. Tang. Halo: Hop-by-hop adaptive link-state optimal routing. *Networking, IEEE/ACM Transactions on*, PP(99):1–1, 2014.

[93] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[94] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM*, pages 537–550. ACM, 2015.

[95] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, March 1992.

[96] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *NSDI*, 2013.

[97] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *NSDI 2013*.

[98] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. SIGCOMM 2009, pages 39–50. ACM.

[99] The CAIDA anonymized Internet traces 2014 dataset. http://www.caida.org/data/passive/passive_2014_dataset.xml.

[100] Noviflow. http://noviflow.com/.

[101] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *PODC*, August 1988.

[102] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*, June 2014.

[103] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized ”zero-queue” datacenter network. SIGCOMM, 2014, pages 307–318, New York, NY, USA. ACM.

[104] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of Open vSwitch. In *NSDI*, 2015.

[105] Lucian Popa, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: Sharing the network in cloud computing. HotNets-X, pages 22:1–22:6, New York, NY, USA, 2011. ACM.

[106] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *USENIX ATC*, April 2005.

[107] Sivasankar Radhakrishnan, Malveeka Tewari, Rishi Kapoor, George Porter, and Amin Vahdat. Dahu: Commodity switches for direct connect data center networks. ANCS 2013, pages 59–70. IEEE Press.

[108] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. SIGCOMM 2011, pages 266–277. ACM.

[109] Prasenjit Sarkar and John H. Hartman. Efficient cooperative caching using hints. In *USENIX OSDI*, 1996.

[110] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging Zipf's law for traffic offloading. *SIGCOMM Comput. Commun. Rev.*, 42(1):16–22, January 2012.

[111] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *SIGCOMM*, 2014.

[112] Siddhartha Sen, David Shue, Sunghwan Ihm, and Michael J. Freedman. Scalable, optimal flow routing in datacenters via local link balancing. CoNEXT 2013, pages 151–162. ACM.

[113] Anirudh Sivaraman, Mihai Budiu, Alvin Cheung, Changhoon Kim, Steve Licking, George Varghese, Hari Balakrishnan, Mohammad Alizadeh, and Nick McKeown. Packet transactions: A programming model for data-plane algorithms at hardware speed. *CoRR*, abs/1512.05023, 2015.

[114] Joseph G. Slember and Priya Narasimhan. Static Analysis Meets Distributed Fault-tolerance: Enabling State-machine Replication with Nondeterminism. In *HotDep*, November 2006.

[115] Haoyu Song and Jonathan Turner. Nxg05-2: Fast filter updates for packet classification using tcam. In *GLOBECOM'06. IEEE*, pages 1–5.

[116] Ed Spitznagel, David Taylor, and Jonathan Turner. Packet classification using extended TCAMs. In *IEEE ICNP*, Washington, DC, USA, 2003. IEEE Computer Society.

[117] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. PAST: Scalable Ethernet for data centers. In *ACM SIGCOMM CoNext*, December 2012.

[118] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A Network-state Management Service. In *SIGCOMM*, August 2014.

[119] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *INM/WREN*, April 2010.

[120] Balajee Vamanan and T. N. Vijaykumar. Treecam: Decoupling updates and lookups in packet classification. CoNEXT '11, pages 27:1–27:12, New York, NY, USA. ACM.

[121] Patrick Verkaik, Dan Pei, Tom Scholl, Aman Shaikh, Alex Snoeren, and Jacobus van der Merwe. Wresting Control from BGP: Scalable Fine-grained Route Control. In *USENIX ATC*, June 2007.

[122] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking. In *HotNets*, August 2014.

[123] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. In *ACM SIGCOMM*, pages 351–362, New York, NY, USA, 2010. ACM.

[124] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C. Bressoud. Practical and Low-overhead Masking of Failures of TCP-based Servers. *ACM Trans. Comput. Syst.*, May 2009.

[125] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: Reducing the flow completion time tail in datacenter networks. SIGCOMM 2012, pages 139–150. ACM.