

© 2016 Soudeh Ghorbani

SQUEEZING THE MOST BENEFIT FROM NETWORK PARALLELISM IN
DATACENTERS

BY

SOUDEH GHORBANI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Associate Professor Brighten Godfrey, Chair
Professor Jennifer Rexford, Princeton University
Professor Nitin Vaidya
Associate Professor Indranil Gupta

ABSTRACT

One big non-blocking switch is one of the most powerful and pervasive abstractions in datacenter networking. As Moore’s law begins to wane, using parallelism to scale *out* processing units, *vs.* scale them *up*, is becoming exceedingly popular. The one-big-switch abstraction, for example, is typically implemented via leveraging massive degrees of parallelism behind the scene. In particular, in today’s datacenters that exhibit a high degree of *multi-pathing*, each logical path between a communicating pair in the one-big-switch abstraction is mapped to a set of paths that can carry traffic in parallel. Similarly, each one-big-switch abstraction function, such as the firewall functionality, is mapped to a set of distributed hardware and software switches.

Efficiently deploying this pool of networking connectivity and preserving the functional *correctness* of network functions, in spite of the parallelism, are challenging. Efficiently balancing the load among multiple paths is challenging because microbursts, responsible for the majority of packet loss in datacenters today, usually last for only a few microseconds. Even the fastest traffic engineering schemes today have control loops that are several orders of magnitude slower (a few milliseconds [1, 2] to a few seconds [3]), and are therefore ineffective in controlling microbursts. Correctly implementing network functions in the face of parallelism is hard because the distributed set of elements that in parallel implement a one-big-switch abstraction can inevitably have inconsistent states that may cause them to behave differently than one physical switch.

The first part of this thesis (§2) presents DRILL, a datacenter fabric for Clos networks which performs *micro load balancing* to distribute load as evenly as possible on microsecond timescales. To achieve this, DRILL employs packet-level decisions at each switch based on local queue occupancies and randomized algorithms to distribute load. Despite making per-packet forwarding decisions, by enforcing a tight control on queue occupancies, DRILL manages to keep the degree of packet reordering low. DRILL adapts to topological asymmetry (*e.g.*

failures) in Clos networks by decomposing the network into symmetric components. Using a detailed switch hardware model, we simulate DRILL and show it outperforms recent edge-based load balancers particularly in the tail latency under heavy load, *e.g.*, under 80% load, it reduces the 99.99th percentile of flow completion times of Presto and CONGA by 32% and 35%, respectively. Finally, we analyze DRILL’s stability and throughput-efficiency.

In the second part (§3, §4), we focus on the correctness of one-big-switch abstraction’s implementation. We first show that naively using parallelism to scale networking elements can cause incorrect behavior. For example, we show that an IDS system which operates correctly as a single network element can erroneously and permanently block hosts when it is replicated. We then provide a system, COCONUT, for *seamless scale-out* of network forwarding elements; that is, an SDN application programmer can program to what functionally appears to be a single forwarding element, but which may be replicated behind the scenes. To do this, we identify the key property for seamless scale out, weak causality, and guarantee it through a practical and scalable implementation of vector clocks in the data plane. We build a prototype of COCONUT and experimentally demonstrate its correct behavior. We also show that its abstraction enables a more efficient implementation of seamless scale-out compared to a naive baseline (§3). Finally, reasoning about network behavior requires a new model that enables us to distinguish between observable and unobservable events. So in the last part (§4), we present the Input/Output Automaton (IOA) model and formalize networks’ behaviors. Using this framework, we prove that COCONUT enables seamless scale out of networking elements, *i.e.*, the user-perceived behavior of any COCONUT element implemented with a distributed set of concurrent replicas is provably indistinguishable from its singleton implementation.

Overall, our results show that building an observationally correct, efficient parallelized one big switch is surprisingly feasible in best-effort datacenter networks.

To my family.

ACKNOWLEDGMENTS

I am immensely grateful to my advisor, Brighten Godfrey. Having someone with his intellect, positivity, open-mindedness, and profound commitment to excellence in research as my advisor has been a privilege. It takes a special personality to have an awe-inspiring taste and vision in research, and yet let the junior, inexperienced members of your team, and in fact encourage them to, drive and truly lead *all* aspects of their research — from forming their own vision and finding a problem they are passionate about to designing the solution and its execution. I was lucky to have someone like this as my PhD advisor. With Brighten, I had the opportunity to explore bold and controversial research ideas; I enjoyed absolute freedom and autonomy in his team.

Working with Jennifer Rexford, one of my role models, in Princeton during parts of my first two years of PhD was a dream that came true. I will forever be grateful for her mentoring and uniquely generous support, and for helping me feel like an equal member of her research team. I owe some of the most gratifying opportunities during my PhD to Jen and Brighten — from having the option to collaborate with some awesome researchers early in my PhD to discussing my research with some industry partners and practitioners later on. These experiences each shaped and enriched my graduate studies in a different way. Brighten and Jen were the best role models one could ask for, not only for learning how to do high quality research, but for their openness to differing ideas and styles, their trust and confidence in those who work with them and the way they empower them, and for their kindness and respect towards *everyone*.

I am grateful to Nitin Vaidya and Indranil Gupta for their flexibility for scheduling my proposal and thesis defense sessions and for their feedback on the work that appears in this thesis. The second chapter of this thesis builds upon joint work with Brighten Godfrey, Zibin Yang, Yashar Ganjali, and Amin Firoozshahian and draws insights from our discussions with a major network device vendor and Changhoon Kim of Barefoot Networks. The third and fourth chapters are joint

works with Brighten Godfrey. Some of the ideas presented in the third chapter were inspired by my earlier discussions with Cole Schlesinger, Jennifer Rexford, and David Walker. I have benefited from some insightful comments and discussions about formal methods with Sayan Mitra for the material presented in the fourth chapter.

The cultures and values of each of the great universities and institutes where I studied and worked were invaluable during this journey. I came to appreciate the culture of ambition of Sharif University of Technology. Sharif cultivated an atmosphere of dreaming big and working hard for our dreams. If “life begins on the other side of despair¹”, Sharif instilled in us the grit to fight through the despair. I have admired the way University of Toronto encouraged us to know our rights and speak up. U of T holds a special place in my heart because of their commitment to equality, tolerance, diversity, and freedom of expression. Finally, my PhD journey would have been impossible without the flexibility and support from many people at UIUC and Princeton. The academic office of the CS department at UIUC, in particular, went above and beyond their duties to help me overcome some hurdles including some visa issues.

I am grateful to VMware for their generous support of my research via a graduate fellowship, and to many amazing people in the Systems and Networking groups of various places interactions with whom made my graduate life more enriching: Phillipa, Venky, Mohammad, Milad, Soheil, Sajad, Hamed, Amin, Monia, Hossein, Alireza, Cole, Naga, Laurent, Theo, Dushyant, Diego, Mojgan, Xin, Srinivas, Nanxi, Josh, Zhenming, Ahmed, Anduo, Kyle, Ankit, Chi-Yao, Ashish, Virajith, Fred, Jason, Rachit, Qingxi, Mo, Sangeetha, Tong, Rashid, Wenxuan, and Chia-Chi.

I am thankful for the friendship and support of Behrooz, Eelyn, Alex, Hamideh, Azade, Shima, Ali (x2), Atefeh, Chris, Narges, Rafael, Motahhare, Amir, Saba, Mohammad (x2), Hamed, Fatemeh (x3), Ghazale, Kyle, Golnaz, Samaneh, Maryam, and many others. Last but not least, I have been the luckiest person in the world to have Ehteram, Hossein, and Sare in my life. Their infinite love has been the only constant in my life.

¹With apologies to Jean-Paul Sartre.

TABLE OF CONTENTS

LIST OF FIGURES	ix
Chapter 1 INTRODUCTION	1
Chapter 2 MICRO LOAD BALANCING IN DATACENTERS WITH DRILL	11
2.1 Introduction	11
2.2 Background and Motivation	15
2.3 Design and Algorithms	17
2.3.1 Design Overview	17
2.3.2 DRILL Approximates ESF in a Symmetric Clos	20
2.3.3 DRILL Causes Minimal Packet Reordering	25
2.3.4 DRILL Decomposes Asymmetric Networks Into Symmetric Components	26
2.4 Evaluation	29
2.5 Related Work	35
2.6 Conclusion	36
Chapter 3 SEAMLESS SCALE-OUT OF NETWORK ELEMENTS WITH COCONUT	37
3.1 Introduction	37
3.2 Background	41
3.2.1 Basic Abstractions	41
3.2.2 Applications of Replication	42
3.3 What Can Go Wrong?	43
3.3.1 Example 1: SDN-enabled Security	44
3.3.2 Example 2: Logical Firewall	47
3.3.3 Example 3: Logical Load Balancer	48
3.3.4 Shortcomings of Existing Approaches	50
3.4 Design of COCONUT	51
3.4.1 Not All Orderings Are Created Equal	52
3.4.2 COCONUT's High-level Algorithms	54
3.4.3 OpenFlow-compatible Implementation	57
3.5 Evaluation of Prototype	61
3.5.1 Prototype Implementation	62

3.5.2	Experimental Setup	63
3.5.3	Data Plane Performance Impact	65
3.5.4	How Long Are Updates Delayed?	66
3.5.5	How Much Rule Overhead Is Imposed and Where?	67
3.5.6	Can Header Bits Become a Scalability Bottleneck?	68
3.6	Related Work	69
3.7	Conclusion	70
Chapter 4 TOWARDS A RIGOROUS FRAMEWORK FOR REASONING ABOUT NETWORK BEHAVIORS		
4.1	Introducing the IOA Framework	71
4.2	Defining Observational Correctness	73
4.3	Modeling Logical Networks as IOA	74
4.3.1	Modeling Links as IOA	74
4.3.2	Modeling Switches as IOA	82
4.4	Modeling Existing Implementations of Logical Networks as IOA	84
4.5	Modeling COCONUT as IOA	90
4.5.1	More Details on the Modeled IOAs	98
4.6	COCONUT Guarantees Observational Correctness	101
4.6.1	External and Internal Actions, and Hiding	101
4.6.2	Proving that COCONUT Guarantees Observational Correctness	102
4.6.3	Composing One-Big-Switches (and Other IOAs)	105
Chapter 5 CONCLUSION		106
BIBLIOGRAPHY		108
Appendix A MODELING AND PROOFS OF CHAPTER 2		118
Appendix B MODELING AND PROOFS OF CHAPTER 3		127

LIST OF FIGURES

2.1	A simple Clos network.	14
2.2	(a) 80% load. (b) 30% load. Adding a choice and a memory unit improves performance dramatically.	22
2.3	With 48-engine switches & 80% load, too many choices and memory units cause a synchronization effect.	23
2.4	L1-S0 link failure increases FCT in DRILL.	26
2.5	DRILL improves latency in a symmetric Clos.	29
2.6	(a) 30% load (b) 80% load. DRILL's improvement is greater under heavy load.	30
2.7	DRILL keeps FCT short in a VL2 network under (a) 30% and (b) 60% load.	31
2.8	For fewer than 0.006 fraction of flows, DRILL reorders enough packets to reduce TCP's transmission rate (<i>i.e.</i> , $\Pr[\text{num dup ACKs} \geq 3]$) even under high load.	32
2.9	DRILL gracefully handles single link failures.	32
2.10	DRILL gracefully handles 5 link failures.	33
2.11	DRILL cuts the tail latency in incast scenarios.	35
3.1	Composing monitoring and routing.	43
3.2	SDN-enabled security architecture.	44
3.3	Simple replication causes incorrect blocking.	46
3.4	Replicated firewall incorrectly blocks communication.	48
3.5	(a) SC causes significant bandwidth overhead, (b) Replication-aware app increases delay.	64
3.6	Testbed's (a) update initiation & (b) termination delays.	65
3.7	How long does it take to initiate and finish updates? Top: initiation delays for the firewall app; middle and bottom: initiation and termination delays for the IDS app.	66
3.8	How much rule-overhead is imposed and where?	67
3.9	How long does the rule-overhead persist?	69
4.1	Links as IOA.	75
4.2	Switch as IOA: types and variables.	77
4.3	Switch as IOA: packet handling transitions.	78
4.4	Switch as IOA: failure and recovery transitions.	80

4.5	Switch as IOA: time-evolving trajectories and transitions.	80
4.6	Hypervisors as IOA.	81
4.7	Virtualized links as IOA.	83
4.8	COCONUT types.	84
4.9	COCONUT links as IOA.	85
4.10	COCONUT switch as IOA: variables and packet handling transitions.	87
4.11	COCONUT switch as IOA: failure and recovery transitions. . . .	89
4.12	COCONUT switch as IOA: time-evolving trajectories and transitions.	90
4.13	COCONUT switch as IOA: lookup, notifying shell and query- ing controller.	91
4.14	COCONUT hypervisors as IOA.	92
4.15	Shells as IOA.	94
4.16	Controller as IOA.	96

Chapter 1

INTRODUCTION

Programmable modular abstractions are the corner stones of Computer Science [4]. One of such key abstractions in datacenter networking is the one-big-switch abstraction. For practical requirements such as scalability, the one-big-switch abstraction is in reality implemented via massive degrees of parallelism. Network parallelism is everywhere in today's datacenter networking: from edge switches that collectively implement the one-big-switch abstraction functionality [5, 6], to multiple equal cost paths between every source and destination pair that carry the traffic in parallel [7, 8, 2, 9].

To get the utmost degree of scalability and performance, in all these examples, a set of autonomous agents make parallel and independent decisions in order to avoid the fundamental costs of coordination. Specifically, we focus on two areas of network parallelism:

- **Efficient use of the data plane pool of connectivity:** In today's multi-rooted datacenters with a high degree of *multi-pathing*, each source of traffic (switch or host) autonomously selects one of the available paths [7, 8, 2, 9].
- **Fast and efficient implementation of network functions:** the one-big-switch abstractions are exceedingly used to implement network functions such as firewalls, IDS, etc. Once the networking state is replicated to implement a one-big-switch abstraction, each replica processes traffic independently with no coordination with other replicas [5, 6].

While, by avoiding the prohibitive latency and throughput costs of coordination, these approaches enhance the scalability of network elements, they can potentially jeopardize both correctness and in some cases, interestingly, performance if the decisions of non-coordinating agents have interdependencies. In multi-path datacenters, for example, independent sources of traffic can overload parts of the network while leaving the rest underutilized, resulting in load imbalance and degradation of throughput and latency (§2). Indeed, when highly synchronous traffic

patterns such as incast cause short-lived bursts, even the fastest exiting traffic engineering approaches are too slow and therefore ineffective in suppressing such bursts (§2).

In addition to the performance degradation caused by the inefficiency of the existing traffic engineering techniques in exploiting multi-paths, existing techniques for network function parallelisms may cause incorrect application-level behaviors. As an example, if a firewall allows external traffic only after an internal request, two separate firewall replicas might process the internal request and its subsequent external reply; leading to one of the replicas incorrectly blocking the external traffic. We show that incorrect behaviors can happen frequently as a result of using the current techniques of implementing parallel functions such as the existing network virtualization techniques (§3). In other words, such techniques can break the semantics of best-effort networks.

One might think that the tussle between performance/scalability and strong semantics is a fundamental one. Preserving strong consistency in databases and distributed systems, for instance, fundamentally requires a few rounds of communication which adds to the performance costs (latency, throughput) and hence limits the scalability of the system [10]. When it comes to the significant performance costs of guaranteeing strong consistency, best-effort networks are not an exception. Providing strong consistency among the replicas of network functions in a small network, for instance, can add an average latency of $50ms$ to each packet [11]¹.

It is therefore hardly surprising that today’s datacenters, with stringent requirements on latency, availability, and scale [12, 13, 8, 14], typically forfeit strong semantics in favor of maximizing scalability and performance. In load balancers that exploit multi-paths, for example, sources of traffic typically work in an uncoordinated manner, based on potentially inconsistent traffic and topology states [9, 2, 7]. Similarly, the systems that offer the function parallelism usually settle for eventual consistency [11, 5].

Weak unfamiliar semantics, however, makes network programming complex, error prone, and inefficient. We argue that in the datacenter setting, the trade-off between performance and strong semantics is not fundamental, *i.e.*, one can have the best of both worlds. Our position is based on two key observations:

- In highly regular datacenter topologies, near optimally exploiting parallel

¹Note that the *avg* RTT is below $1ms$ in datacenters.

paths can be achieved via local load sensing and a slight degree of intelligence in the network while strictly avoiding coordination (and its costs).

- The native service model that networks provide, best-effort-networking, has a distinct—and more relaxed—semantics compared to some of extensively studied and rigorously formalized service model in other domains such as distributed file systems and databases [10]. We show that it is possible to preserve strong semantics in best-effort networks, *i.e.*, make parallel network functions indistinguishable from their singleton implementation, while incurring relatively low costs.

The two parts of this thesis investigate the above observations in turn. In the first part of this thesis (§2), we introduce DRILL, a system for exploiting multi-paths at microsecond timescales. In the second part, we introduce a logical property, which we call weak causal correctness, that parallelized network functions should retain to be indistinguishable from their singleton counterparts. We present COCONUT, a system we built which guarantees this property with minimal overhead (§3). We then introduce an analytical framework for formalizing and reasoning about the behavior of networks, and formally prove that parallelized networks under COCONUT are observationally indistinguishable from singleton networks (§4). We give a brief overview of these two parts below.

Micro load balancing in datacenters with DRILL. Data centers are overwhelmingly built as topologies that are characterized by large path diversity such as Clos networks (Figure 2.1) [15, 2, 9, 16, 17, 18, 19, 16, 20]. A critical issue in such networks is the design of an efficient algorithm that can evenly balance the load among available paths. While Equal Cost Multi Paths (ECMP) is extensively used in practice [9], it is known to be far from optimal for efficiently exploiting all available paths [9, 2, 21]. Data center measurement studies, for instance, indicate that a significant fraction of core links regularly experience congestion despite the fact that there is enough spare capacity elsewhere [22].

Many proposals have recently tried to address this need [2, 9, 23, 24, 21]. Aligned with the recent trend of moving functionality out of the network fabric [25], these proposals strive to delegate load balancing to centralized controllers [26, 23, 3, 27], to the network edge [2], or even to end-hosts [9, 21]. This recent move of the load balancing functionality is motivated largely by the *perceived necessity of having global congestion or load information* about the potential paths for evenly balancing the load among them [2, 26, 23, 3, 27]. Collecting global

traffic information and routing based on that information could more easily be managed at separate controllers or at the edge. A notable example is CONGA [2], a recent in-network load balancing scheme that gathers and analyzes congestion feedback from the network at the network edge (leaf switches in Clos networks) to make load balancing decisions. CONGA’s central thesis is that *global congestion information is fundamentally necessary for evenly balancing the load*.

We explore a different direction: What can be achieved with decisions that are local to each switch? We refer to this approach as *micro load balancing* because it makes “microscopic” decisions within each switch without global information, and because this in turn allows decisions on microsecond (packet-by-packet) timescales.

Micro load balancing has hope of offering an advantage because load balancing systems based on global traffic information have control loops that are significantly slower than the duration of the majority of congestion incidents in data centers (§2.4). It has been shown that majority of congestion events in data centers are short-lived [22, 28]. The bulk of microbursts that are responsible for over 90% of packet loss, for instance, last for less than 3 microseconds [29]. Systems that attempt to collect and react based on global congestion information typically have orders of magnitude slower control loops than the duration of the majority of congestion events [1, 2]. For example, even though CONGA adds mechanisms to leaf and spine switches to assist in obtaining congestion information, it still typically requires a few RTTs (tens to hundreds of microseconds), by which time the congestion event is likely already over. In addition, we found that amassing macroscopic traffic information can lead to a pitfall: feeding global traffic information to distributed, non-coordinating sources of traffic (input ports of all the leaf switches in CONGA) can cause them to select the same set of least-congested paths in a synchronized manner which in turn leads to bursts of traffic in those paths.

To study whether micro load balancing offers a viable solution, we designed and evaluated DRILL (Distributed Randomized In-network Localized Load-balancing). DRILL is in essence a switch scheduling algorithm that acts only based on local switch queue length information without any coordination among switches or any controllers. Even within a single switch, deciding how to route and schedule packets is nontrivial. DRILL’s scheduling algorithm is inspired by the “power of two choices” paradigm [30]. To make it practical for packet routing within a data center switch, we extend the classic design to accommodate a distributed set of sources (input ports) and show that the key stability result holds in

the distributed version as well (§3.6, §3.3.3, §2.3.2). More concretely, DRILL assumes that a set of candidate next-hops for each destination have been installed in the forwarding table, using well-known mechanisms such as the shortest paths (as in ECMP). Next, upon arrival of each packet at any input port, that input port, independently and with no coordination with other input ports, compares the queue lengths of two randomly-chosen candidate output ports and the port that was least loaded during the previous samplings, and sends the packet to the least loaded of these three candidates. Note that this is unlike ECMP since the decision is based on local load rather than static hashing of the packet header. We show how to optimize DRILL’s parameters—number of choices and amount of memory—so as to avoid damaging synchronization effects where many input ports choose the same output.

In contrast to the works that operate on a global “macroscopic” view of the network, DRILL’s micro load balancing enables it to instantly react to load variations as the queues start building up. DRILL results in dramatically better performance than CONGA in heavily loaded systems (78% improvement in average flow completion time §2.4) and in incast scenarios ($2.5\times$ improvement in average flow completion time §2.4). In addition, DRILL offers a simpler switch implementation than CONGA since DRILL does not need to detect flowlets or send and analyze feedback.

Presto [9], a very recent host based load balancing scheme, offers an interesting comparison point to DRILL. Unlike schemes with global information, Presto is congestion-oblivious. Presto argues that the main culprit of inefficiencies in schemes like ECMP is the coarse granularity: each flow, even a large one, hashes all its packets onto one path. Therefore, Presto partitions flows into equal size chunks of 64KB, called flowcells, and “sprays” them in a round-robin fashion among available paths. This can be executed by the source with a form of source routing, releasing the network from that burden. A key assumption in this design is that *the small size and size-uniformity of data units is sufficient for preserving balanced load in symmetric topologies*. Our simulations confirm that Presto outperforms CONGA in non-incast scenarios, but DRILL in turn performs better than Presto (§2.4). DRILL’s improved performance even for identical small size flows (Presto’s ideal setting) results from (a) the load adaptation of DRILL, in contrast to the load-agnostic nature of Presto, and (b) balancing finer granularity of load: packets vs. flowcells. We also show that DRILL has significantly better flow completion time in an incast scenario ($9.5\times$ better than Presto) because of its

fast reaction to congestion (§2.4).

DRILL’s micro load balancing raises several concerns. First, how can we deal with packet reordering that results from load balancing at sub-flow granularities? Interestingly, we find that in a symmetric Clos data center network, DRILL balances load so well that packets nearly always arrive in order despite traversing different paths. This is because queue lengths have very small variance and hence packets have almost identical queueing delays, even under heavy load (§2.3.3). Regardless, the occasional re-orderings could still adversely affect TCP’s performance. Hence, similar to prior work [24, 9], we deploy a buffer under TCP to restore correct ordering of packets. Practical challenges of deploying such a technique are solidly addressed and solved by Presto [9]. Compared to Presto, DRILL causes significantly less frequent out of order delivery of packets, shorter buffers, and smaller buffering latencies (§2.3.3).

The second concern is that load-based scheduling algorithms within a switch could result in instability and hence low throughput [31]. Therefore, we formally prove DRILL’s switch-level stability and show that it guarantees 100% throughput for admissible traffic (§2.3.2).

Third, is DRILL’s micro load balancing sufficient alone, or is some macroscopic information necessary? For topological changes such as link failures that reduce the number of paths, however, it is likely that more global path planning is necessary.

Since these changes, unlike congestion, happen in slow time scales [32], DRILL leverages existing distributed or centralized topology-information dissemination techniques to detect the new topology, it decomposes the network into symmetric component, and performs micro load balancing inside each component. We prove that for admissible traffic, this approach provides throughput efficiency. In summary, our results in the first part of this thesis strongly indicate that micro load balancing belongs in the data center fabric to achieve the key goal of high performance traffic delivery; and that a significant and interesting question for future research is when and how micro load balancing and macroscopic information should be combined to get the best of both worlds.

Seamless scale-out of network elements with COCONUT. In the second part of this thesis (§3, §4), we turn our attention to the parallelism in network functions which is used as the primary way for scaling out network elements.

An important use of software-defined networking (SDN) is to automate scaling of networks, so that individual network functions or forwarding elements can be

replicated as necessary. Replication of network elements allows capacity to scale gracefully with demand [33], provides high availability [33], assists function mobility [34, 35], and overcomes lack of capacity of physical elements (*e.g.*, when the capacity of one switch is insufficient for a full implementation of the logical abstraction).

Multiple systems use replication of network elements, in different ways. One key use is in network virtualization for software-defined data centers. Each tenant in a virtualized data center might be presented one logical “big switch” abstraction that in reality spans multiple physical hardware or software switches [33, 36, 37]. As another example, Microsoft Azure’s host-based SDN solution leverages VM-Switches to build virtual networks where each host performs all packet-actions for its own VMs² [38]; these VMSwitches act in parallel and independently despite the fact that they might form a single virtual network. Google’s virtualized SDN, Andromeda [39], integrates software network function virtualization (NFV), such as virtual firewall, rate limiting, *etc.* into the data-path, and deploys replication in the data plane to meet its performance and scalability needs. Replication is used outside of virtualization as well. Caching of forwarding rules at multiple locations enhances performance in [40, 41, 42]. Caching at finer granularity—from user-space to kernel-space – is critical for performance of software switches such as Open vSwitch [43]. All these systems deploy replication techniques where one logical network element is implemented using a distributed set of physical replicas, typically by simple duplication of forwarding rules across multiple locations.

Our work begins by asking: *Do these techniques for scaling out network elements preserve the semantics of a single element?* For example, if a developer writes a network function such as a firewall on top of a single virtual “big switch”, is its functional behavior the same as if it were running on an actual single physical switch? We show that such incorrect behavior can in fact occur with common replication techniques; for example, a replicated firewall can erroneously and *permanently* block hosts. In fact, our experiments show there are scenarios that these problems occur frequently (§3.3).

How, then, could an SDN programmer deal with this problem? Living with the risk of incorrect functionality is unappealing, as critical infrastructure elements such as security appliances (firewalls, intrusion detection systems, *etc.*) are increasingly being deployed in order to scale out their capacity. Alternately, the

²This is done to make SDN scale.

programmer could write her application so that it takes into account the distributed implementation of network elements and associated race conditions. But this is inconvenient for the programmer at best, and in many cases is infeasible, because the fact that network elements are replicated may be explicitly hidden from the network programmer/operator—as is the physical infrastructure underneath a tenant’s virtual network. Indeed, one lure of the virtualized cloud for tenants, for instance, is the prospect of migrating their workloads and network applications to the cloud “as-is”, *i.e.*, with no re-designing and re-architecting of their applications, and expect them to perform in a way exactly akin to their non-virtualized networks [44, 6].

Our goal is thus to build a system that provides a **seamless scale-out abstraction** for network forwarding elements: *an SDN application writer (or tenant) can program to the abstraction of a single device, which may be implemented behind the scenes by multiple replicated elements*. Achieving this is not easy. The most generic solution would be to synchronize replicas to provide a strongly consistent logical view, but the required locking would not achieve the performance necessary for the data plane [38, 42]. Recent work [45, 46, 47, 48] provides a form of consistency in the data plane in the sense of ensuring “trace” properties of a single packet’s path, as in Consistent Updates (CU) [46]. But this is essentially orthogonal to our goal; seamless scale-out does not require per-packet path consistency, and systems that provide per-packet path consistency can even *cause* the correctness problems described above (§3.3.3).

The system we present here, COCONUT (“COrrrect COncurrent Networking UTensils”), provides seamless scale-out of network elements with provable correctness, for network elements defined by an OpenFlow-like abstraction. We observe that the culprit of scale-out correctness problems is *weak causality violation*. For example, a simplistic replication technique can cause a replicated firewall to miss the causal dependency between a client’s outbound request and a server’s inbound response, so it sees a seemingly-unsolicited inbound response first and permanently blocks traffic from that server. We design a set of high-level algorithms to avoid such causality violations, drawing on the classical concept of logical clocks [49] to track the state of each forwarding rule at each abstract network element. Although conceptually simple, providing a practical and scalable implementation of these high-level algorithms is challenging; switches do not directly implement logical clocks, and emulating a large vector of logical clocks in packet header fields is impractical. We provide a practical realization of those algorithms

that emulates their behavior using OpenFlow-compatible switches, leveraging the distinguishing characteristics of SDNs and virtualized networks, and is thus suitable for deployment in the context of a modern virtualized data center with software switches in each physical host. Our design uses limited bits in header fields of a physical network to emulate logical clocks in the virtual network, while dealing with concurrent creations and changes of multiple virtual networks that may interleave with each other and compete for use of these bits.

Finally, we implemented a prototype of COCONUT using Floodlight [50], Open vSwitch [43], and OpenVirtex [5]. Our experiments with our prototype on an SDN testbed and Mininet [51] with data center topologies of various scales and with different load and traffic patterns demonstrate that COCONUT provides observational correctness under scenarios in which existing replication techniques often result in incorrect behavior, at reasonable cost in terms of update delay and rule overhead (§3.5). In particular these costs are lower than for CU [46] for large-scale networks (§3.5).³ We also demonstrate that COCONUT enables an SDN application to be conveniently written so that it provides 19-30% lower latency, compared with the natural implementation where the programmer deals with replication manually within the application.

To prove that COCONUT correctly provides seamless scale-out, we need a new analytical framework. In §4, we introduce the Input/Output Automaton (IOA) framework which allows us to model the network, define its behavior, and reason about its correctness. Our goal is that the scaled-out network is indistinguishable from the case of a singleton network element. Thus, we need to take into account the sequence of observations made by the end-points, with potential interdependencies. We formalize this with a definition we call **observational correctness** that requires that any sequence of end-point observations in the scaled-out network is plausible for the singleton version. In tune with what applications expect from best-effort networks, this model is permissive of occasional packet drops and re-ordering, while prohibiting weak causality violations that could jeopardize applications' correctness. We formally prove that COCONUT provides observational correctness (§3).

In summary, the second part of this thesis presents the design, evaluation, and formal analysis of COCONUT and demonstrate that achieving true correct seamless scale-out, in the context of OpenFlow forwarding elements in a virtualized

³As mentioned above, CU provides trace properties which are different than causal consistency, but it provides a useful reference point in terms of performance.

data center, is surprisingly feasible. We believe this lays the foundation for a practical and dependable service model for virtualized network infrastructure, as well as a powerful abstraction for programming SDNs.

Chapter 2

MICRO LOAD BALANCING IN DATACENTERS WITH DRILL

The trend towards simple datacenter network fabric strips most network functionality, including load balancing capabilities, out of the network core and pushes them to the edge. We investigate the opposite direction — incorporating minimal load balancing intelligence into the network fabric — and show that this slightly smarter fabric significantly enhances performance.

In this chapter we present DRILL, a data center fabric for Clos networks which performs *micro load balancing* to distribute load as evenly as possible on microsecond timescales. To achieve this, DRILL employs packet-level decisions at each switch based on local queue occupancies, randomized algorithms to distribute load, and adaptation to asymmetry caused by link or device failures. Using a detailed switch hardware model, we simulate DRILL and show that it outperforms recent edge-based load balancing techniques. Finally, we analyze the switch-level stability and throughput-efficiency of DRILL’s scheduling algorithm.

2.1 Introduction

Datacenters are overwhelmingly built as topologies that are characterized by large path diversity such as Clos networks (Figure 2.1) [15, 2, 9, 16, 17, 18, 19, 16, 20]. A critical issue is the design of an efficient algorithm that can evenly balance the load among available paths. While Equal Cost Multi Path (ECMP) is extensively used in practice [52, 8, 9], it is known to be far from optimal for efficiently exploiting all available paths [9, 2, 21]. Datacenter measurement studies, for instance, indicate that a significant fraction of core links regularly experience congestion despite the fact that there is enough spare capacity elsewhere [22].

Many proposals have recently tried to address this need [2, 9, 23, 24, 21]. Aligned with the recent trend of moving functionality out of the network fabric [25], these proposals strive to delegate load balancing to centralized controllers

[26, 23, 3, 27], to the network edge [2], or even to end-hosts [9, 21]. These entities serve as convenient locations for collecting global or cross-network information about congestion. A notable example is CONGA [2], a recent in-network load balancing scheme that gathers and analyzes congestion feedback from the network at the network edge (leaf switches in Clos networks) to make load balancing decisions. Planck [1], MicroTE [3], Mahout [27] and Hedera [23] also collect global load information to balance load. All these approaches are based on a central thesis that *global congestion information is necessary for evenly balancing the load*.

We explore a different direction: What can be achieved with decisions that are local to each switch? We refer to this approach as *micro load balancing* because it makes “microscopic” decisions within each switch without global information, and because this in turn allows decisions on microsecond (packet-by-packet) timescales.

Micro load balancing has hope of offering an advantage because load balancing systems based on global traffic information have control loops that are significantly slower than the duration of the majority of congestion incidents in data-centers, which are short-lived [22, 28]. The bulk of microbursts responsible for most packet loss, for instance, last for a few microseconds [53, 29]. Systems that attempt to collect and react based on global congestion information typically have orders of magnitude slower control loops than this [1, 2]. For example, even though CONGA adds mechanisms to leaf and spine switches to assist in obtaining congestion information, it still typically requires a few RTTs (tens to hundreds of milliseconds), by which time the congestion event is likely already over.

To understand the problem, we consider a particular fluid version of ECMP, ESF (§2.3.1), which is optimal for Clos networks, and then attempt to design and evaluate a practical approximation of it that we call DRILL (Distributed Randomized In-network Localized Load-balancing). DRILL is in essence a switch scheduling algorithm that acts only based on local switch queue length information without any coordination among switches or any controllers. Even within a single switch with multiple forwarding engines (§2.3.2), deciding how to route and schedule packets is nontrivial. DRILL’s scheduling algorithm for such switches is inspired by the “power of two choices” paradigm [30]. To make it practical for packet routing within a switch, we extend the classic design to accommodate a distributed set of sources (forwarding engines) and show that the key stability result holds in the distributed version as well (§2.3.2).

More concretely, DRILL assumes that a set of candidate next-hops for each destination have been installed in the forwarding table, using well-known mechanisms such as the shortest paths (as in ECMP). Next, upon arrival of each packet at any engine, that engine, independently and with no coordination with other engines, compares the queue lengths of two randomly-chosen candidate output ports and the port that was least loaded during the previous samplings, and sends the packet to the least loaded of these three candidates. Note that this is unlike ECMP since the decision is based on local load rather than static hashing of the packet header. We show how to optimize DRILL’s parameters—number of choices and amount of memory—so as to avoid damaging synchronization effects where many engines choose the same output. We further investigate whether DRILL’s load-based scheduling algorithms within a switch could result in instability and hence low throughput [31]. We formally prove DRILL’s stability and show that it guarantees 100% throughput for all admissible independent arrival processes (§2.3.2).

DRILL’s micro load balancing raises several concerns. First, how can we deal with packet reordering that results from load balancing at sub-flow granularities? Interestingly, we find that in Clos datacenter networks, even with failures, DRILL balances load so well that packets nearly always arrive in order despite traversing different paths. This is because queue lengths have very small variance and hence packets have almost identical queueing delays, even under heavy load (§2.4). Regardless, the occasional reorderings could still be undesirable for certain applications. Hence, similar to prior work [24, 9], in virtualized datacenters, we optionally deploy a buffer in hypervisors to restore correct ordering of packets. Practical challenges of deploying such a technique are addressed by Presto [9]. Compared to Presto, DRILL causes significantly less frequent out of order delivery of packets (§2.3.3).

The second challenge is: how does a purely local scheme like DRILL adapt to topological changes, such as failures? To handle asymmetric topologies, DRILL decomposes the network into symmetric partitions and applies micro load balancing inside each partition. We show that this technique results in bandwidth efficiency for admissible traffic (§2.3.4) and short flow completion times even under multiple failures (§2.3.4, §2.4).

Via extensive simulations using a detailed switch hardware model and a variety of topologies and workloads, we find that, in contrast to the works that operate on a global macroscopic view of networks, DRILL’s micro load balancing enables it to instantly react to load variations as the queues start building up. DRILL

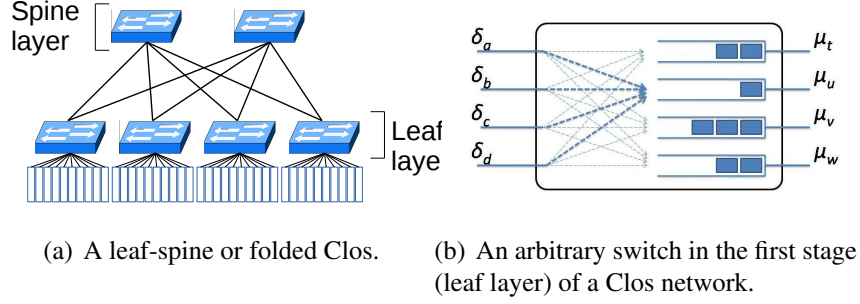


Figure 2.1: A simple Clos network.

results in dramatically shorter tail latencies, especially in incast scenarios ($2.5\times$ reduction in the 99.99th percentile of flow completion times (FCT) compared to CONGA) and under heavy load (32% shorter 99.99th percentile of FCT compared to CONGA under 80% load). Plus, DRILL offers a simpler switch implementation than CONGA since DRILL does not need to detect flowlets or send and analyze feedback. We implement DRILL in Verilog to test its hardware feasibility (§2.4).

Presto [9], another recent host-based scheme, offers an interesting comparison point to DRILL. Unlike schemes with global information, Presto is congestion-oblivious. Presto argues that the main culprit of inefficiencies in schemes like ECMP is the coarse granularity: each flow, even a large one, hashes all its packets onto one path. Therefore, Presto partitions flows into equal size chunks of 64KB, called flowcells, and “sprays” them in a round-robin fashion among available paths. This can be executed by the source with a form of source routing, releasing the network from that burden. A key assumption in this design is that *the small size and size-uniformity of data units is sufficient for preserving balanced load in symmetric topologies*. We find that the nature of workload dynamics, in addition to the flow size distribution, is also key in load balancing. In bursty workloads, for instance, load-sensitive load balancers such as DRILL have better FCT due to their reaction to congestion (*e.g.*, $3.5\times$ improvement in FCT’s tail in an incast scenario; §2.4). DRILL’s improved performance results from (a) the load adaptation of DRILL, in contrast to the load-agnostic nature of Presto, and (b) balancing finer granularity of load: packets *vs.* flowcells.

In summary, our results strongly indicate that micro load balancing belongs in the datacenter fabric to achieve the key goal of high performance traffic delivery.

2.2 Background and Motivation

Clos topologies enable the datacenter providers to build large scale networks out of smaller, and significantly cheaper, commodity switches with fewer ports connected with links of less capacity [54, 15]. Today, most datacenter and enterprise topologies are either built as one two-stage folded Clos, also called leaf-spine topologies (one example is shown in Figure 2.1) [2] or incorporate Clos subgraphs in various layers in their design. Various generations of datacenter at Google, for instance, are built out of different variants of the Clos topology [8]. As another example, the VL2 network [15] is composed of a Clos network between its Aggregation and Intermediate switches. Similarly, in the fat-tree network of [54], Clos networks are used to build pods and the network between pods and core switches.

A key characteristic of Clos networks is having multiple paths between any source and destination hosts. The common practice in datacenters today for balancing load among these paths is ECMP [52]. When more than one “best path”, commonly selected to minimize the number of hops of each path, is available for forwarding a packet towards its destination, each switch selects one via hashing the 5-tuple packet header: source and destination IPs, protocol number, and source and destination port numbers. This path selection mechanism enables ECMP to avoid reordering packets within a TCP flow without per-flow state. All the examples of the Clos networks given above deploy ECMP [8, 15, 54].

ECMP, however, is routinely reported to perform poorly and cause congestion when flow hash collisions occur [9, 2, 8, 29]. Datacenter measurement studies, for instance, show that a significant fraction of core links regularly experience congestion despite the fact that there is enough spare capacity elsewhere to carry the load [22]. Many proposals have tried to enhance ECMP’s performance by balancing finer-grained units of traffic. Aligned with the recent trend of moving functionality out of the network fabric [25], these proposals strive to delegate this task to centralized controllers [23, 1, 26, 3, 27], to the network edge [2], or even to end-hosts [9, 21]. In Presto, for instance, end-hosts split flows into *flowcells*, TSO (TCP Segment Offload) segments of size *64KB*; the network balances flowcells, instead of flows, in a load-oblivious manner [9]. Presto is built on a major premise that per-flow coarse granularity of ECMP combined with the existence of large flows in datacenters is the primary deficiency of ECMP, and in any Clos network with small flows, ECMP is close to optimal [9]. In CONGA, as another example, each edge switch balances flowlets [2] based on global load information.

Its central thesis is that not only the fine granularity of the load to balance, but also that global load information is essential for optimal load balancing and reacting to congestion. Presto and CONGA balance granularities coarser than packets to reduce reordering.

While improving ECMP, these proposals cannot effectively suppress short-lived congestion events that tend to persist for only sub-millisecond intervals [55, 53], sometimes called *microbursts* [22, 53, 56, 57], as even the fastest ones have control loops with 10s of millisecond to a few second delays [2, 1, 3]. However, microbursts are responsible for majority of packet loss in datacenters [22]. In today's datacenters, despite the reportedly low average link utilizations (1% to 20-30% at various stages [14, 8]), the highly bursty nature of traffic [3, 2, 14] makes very short-lived periods of buffer overrun and consequently high loss rates the norm rather than the exception. The buffer utilization statistics at a 10-microsecond granularity from Facebook datacenters for switches connecting web servers and cache nodes, for instance, demonstrate a persistent several orders of magnitude difference between the maximum and the mean queue occupancies [14]. Plus, the maximum buffer occupancy in these Facebook web server racks is reported to approach the configured limit for approximately three quarters of the 24-hour measurement period, even though the mean link utilization for the same rack is only 1% [14]. These ephemeral high buffer occupancies are correlated with high drop rates [14]. The inherent traffic burstiness also results in high congestion drop rates in Google datacenters as utilization approaches 25%; so the utilization is typically kept below that level [8]. Given the pervasiveness of microbursts and their adverse impact on the performance, in terms of low flow completion times and high throughput, *our first goal is to provide high performance especially when microbursts emerge.*

Despite ECMP's suboptimality in handling congestions, its extreme simplicity and scalability effectively has turned it into the de facto load balancing practice in most of the datacenters [8, 14, 15]. Notably, the fact that it is *local* to each switch in the sense that, for forwarding packets, each switch autonomously selects among available paths, irrespective of the load and choices of other switches, makes it easily deployed in conjunction with most routing protocols. Once the global topological information is gathered, each switch makes local forwarding decisions. Networks are therefore relieved of the burden of complex mechanisms for gathering global load information either via distributed algorithms (as in CONGA [2]) or in a centralized manner (as in Planck [1]). Ideally, we would want to share

ECMP’s scalability and simplicity. Hence, in designing DRILL, *our second goal is to make load balancing decisions that are local to each switch.*

2.3 Design and Algorithms

In this section, we provide a high level overview of DRILL’s overall design (§2.3.1), how it achieves micro load balancing in symmetric networks (§2.3.2), how it handles reordered packets (§2.3.3), and how it deals with failures (§2.3.4).

2.3.1 Design Overview

Defining an ideal model for symmetric Clos networks: Equal Split Fluid (ESF). In order to work towards a solution, we define a theoretical ideal that we call *Equal Split Fluid (ESF)*. ESF assumes a fluid model of traffic (rather than a discrete packet-based model). At each switch with n least-cost paths towards a particular destination, ESF sends exactly $1/n$ of the fluid traffic to that destination along each of the n least-cost paths. ESF is switch-local and, in any *symmetric* leaf-spine topology, with a bipartite graph between the leaf and spine switches with identical links, it has *precisely optimal load balance regardless of the traffic pattern*. To briefly explain why: the fact that the first hop traffic, going out of leaves to spines, is balanced across all paths follows immediately from the definition of ESF—each leaf splits its incoming traffic among all available paths. As a result of that, the spines act as an intermediary stage where each spine receives an exactly equal fraction of the traffic destined to each leaf. Hence, the second hop traffic, from spines to the destination leaf, is also balanced, resulting in an overall perfectly balanced load across paths. This intuition extends to more general Clos networks (see Theorem 4 in §A) and is essentially the fluid-model intuition behind why Valiant load balancing (VLB) [58] is an effective load-oblivious routing algorithm.

While optimal, ESF is merely a theoretical fluid-model ideal that the switching fabric needs to approximate in a real discrete world. We can interpret several existing load balancing schemes as attempting to approximate ESF. In ECMP, instead of exactly equally splitting outgoing traffic, (a) decisions are made in very coarse-grained chunks of whole flows, and (b) decisions are pseudorandom, resulting in occasional unlucky load collisions. Presto [9] shrinks the unit of discretization to

the 64 KB flowcell, and randomly spreads these flowcells using end-host source routing; this partially mitigates problem (a). One could imagine going a step further down to what we call *per-packet VLB* which sends each packet through a random intermediate (spine) switch. This design was considered in [9] but was avoided in order to reduce end-host CPU overhead and packet reordering. Even if per-packet VLB could be implemented, it would help problem (a) but not (b), and we will see experimentally that both problems are important.

DRILL as a near-optimal approximation of ESF. The previous discussion did not introduce any truly new material, but helps us frame the problem in a way that provides a direction forward: Can we approximate ESF even more closely? If we could succeed in doing so, the ESF approach could achieve our goals of high performance even at microsecond timescales, and using a switch-local algorithm. But approximating the theoretical ideal is nontrivial. To achieve this, DRILL first chooses the smallest practical unit of discretization, *i.e.*, single packets. This is also a decision unit that is simple for switches to deal with statelessly, and with forwarding in switches, we can avoid the concern mentioned in [9] of overhead of per-packet forwarding decisions at endhosts. Second, DRILL does not forward traffic uniform-randomly. Instead, DRILL leverages switch-local load information, sampling some or all outgoing queues when making each packet’s forwarding decision and placing the packet in the shortest of these queues. Intuitively, this minimizes the “error” between the ideal fluid-based ESF and actual packet forwarding. In particular, we prove in §2.3.2 that DRILL is stable and can deliver 100% throughput for all admissible independent arrival processes.

Together, these mechanisms achieve a significantly better approximation of ESF than past designs. However, two key challenges remain, which we discuss next.

DRILL causes minimal packet reordering. DRILL’s fine grained per-packet load balancing based on potentially rapidly changing local load information raises concern about reordering that could imperil TCP throughput. We show in §2.3.2 and §2.3.3 that under this algorithm, the load is so well balanced that even under heavy load, the probability of reordering is very small—in most cases, well below the degree that damages TCP throughput, and indeed well below the degree that can be resolved by some recent proposals for handling reordering at the end hosts such as Presto [9]. DRILL can employ a shim layer as in [9] to eliminate reordering completely, but in many environments, even without the shim DRILL provides a substantial benefit.

DRILL handles topological asymmetry by decomposing the network into symmetric components. In a symmetric Clos, for perfect load balancing, the *objective* of each source of traffic is clear: equal splitting of its traffic among available paths. In its nature, this is an *oblivious* objective since it does not depend on the load of the other nodes in the network. Different versions of the oblivious routing then deploy different mechanisms for achieving that objective, from local load sensing combined with randomization in DRILL to pure randomization in VLB and ECMP.

If the paths are asymmetric, however, the optimal splitting ratio of traffic at each source may depend on the load from other sources, a potentially rapidly changing, and inherently non-oblivious goal. Naively splitting traffic equally among all paths in this case, as is done in ESF as well as some other variants of oblivious routing such as VLB and Presto, can cause excessive bandwidth loss and packet reordering. Intuitively, in a asymmetric network, multiple paths that a flow can take may have different capacities. Therefore, splitting the load equally among them effectively limits the rate on each path equal to the rate of the path with minimum capacity. This implies that the paths with more capacity will have idle bandwidth even if the flow has a demand for that bandwidth. In addition to bandwidth inefficiency, splitting flows among a set of paths with differential load, and hence different latencies, can potentially cause high degrees of packet reordering.

We observe that both problems rise due to splitting flows among asymmetric paths. Hence, in an asymmetric Clos, DRILL initially decomposes the graph into symmetric components, and then runs the $\text{DRILL}(\mathbf{d}, \mathbf{m})$ inside each component. The rate independence across components implies that the rates in a component can grow unaffected by congestions of other components; thus the bandwidth inefficiency problem is resolved. Plus, since each component is symmetric, splitting flows across its paths does not lead to excessive packet reordering; so the reordering problem is mitigated.

In the spectrum of strictly load oblivious schemes such as VLB, ECMP, and Presto [9] to globally load aware and adaptive ones such as CONGA [2] and Planck [1], DRILL occupies the middle ground: it retains most of the simplicity and scalability of the first class by requiring only local load information and negligible amount of state independent of the number of flows, while improving upon the performance of the state of the art load balancers in both classes (§2.4). We explain how DRILL achieves near optimal load balancing for a wide range of switching hardware in a symmetric Clos (§2.3.2), and how it handles reordering

(§2.3.3) and asymmetry (§2.3.4).

2.3.2 DRILL Approximates ESF in a Symmetric Clos

In a symmetric Clos, our mission is to get as close to ESF as possible. We show that a slight degree of load sensing and intelligence in the switches can get us close to this goal, significantly closer compared to a past approach, VLB, which is considered close to optimal by some load balancing proposals that try to approximate it [9, 23], and ECMP. Before presenting the algorithms, we provide a high level overview of the switching hardware that might affect load balancing.

Switching hardware: Switches have forwarding engines that make forwarding decisions for packets. While many of the simple switches deployed in datacenters have one centralized engine [59], higher-performance switches invariably have multiple forwarding engines [60, 61, 62, 63]. Very high performance switches might have multiple engines on each interface card [63]. These engines make parallel and independent forwarding decisions. Cisco 6700 Series [64], Cisco 6800 Series [64], Cisco 7500 Series [65], Cisco Catalyst 6500 backbone switch series [64], and Juniper MX Series [66] are some examples of switches that support multiple forwarding engines. In Cisco switches, for example, multiple Distributed Forwarding Card (DFC) are installed for line cards. The forwarding logic is then replicated on each DFC-enabled line card, and each card makes forwarding decisions locally and independent of other cards. Some switches have constant access to queue depth, typically as a means for micro-burst monitoring [53, 56, 67, 68, 55]. This feature allows the network provider to monitor traffic on a per-port basis to detect unexpected data bursts within a very small time window of μsec . [53]. Our discussions with [59] indicate that while this information is easily accessible for packet forwarding, it is not always precise: the queue length does not include the packets that are just entering the queue until after they are being fully enqueued. Our simulator models this behavior.

DRILL (\mathbf{d}, \mathbf{m}) scheduling policies: We show that a simple $O(1)$ algorithm achieves near optimal load balancing in a symmetric Clos irrespective of the number of switch engines. We assume that a set of candidate next-hops for each destination has been installed in the forwarding tables of each engine of the switch, using well-known mechanisms such as the shortest paths used by ECMP. DRILL is essentially a switch-local scheduling algorithm inspired by the seminal work on

power of two choices [69] that, whenever more than one next hop is available for the destination of a packet, decides which hop the packet should take.

$\text{DRILL}(d, m)$: Upon each packet arrival, the forwarding engine chooses d random output ports out of possible N next hops, finds the one with the current minimum queue occupancy between these d samples and m least loaded samples from previous time slots, and routes its packet to that port. Finally, the engine updates the contents of its m memory units with the identity of the least loaded output queues.

This algorithm has the complexity of $O(d + m)$. Our experiments with Clos networks with various sizes, switches with diverse number of engines, and different load show that (a) having a few choices and few units of memory is critical to the efficiency of our algorithms, *e.g.*, $\text{DRILL}(2, 1)$ significantly outperforms VLB, (b) increasing d and m beyond 2 and 1 has less of an impact on DRILL’s performance, and in some cases may degrade performance, *i.e.*, while $\text{DRILL}(2, 1)$, with complexity of $O(1)$, consistently outperforms VLB and ECMP, $\text{DRILL}(d, m)$ where $d \gg 2$ and $m \gg 1$ may underperform $\text{DRILL}(2, 1)$ due to a phenomenon we call the *synchronization effect*. We explain each of these points in turn.

Setting the right parameters: the pitfalls of choice and memory: We show in §2.3.2 that for stability, it is necessary to set $m \geq 1$. To set d and m , we evaluate $\text{DRILL}(d, m)$ ’s performance and compare it with ECMP and VLB using the following methodology: We build Clos datacenters of different sizes in a packet level simulator (details in §2.4), draw flow sizes and interarrival times from [14], and scale the interarrival times to emulate various degrees of network load. Given that the dominant source of latency in datacenters is queueing delay [70], in this section, we measure queue lengths as the load balancing evaluation metric. §2.4 measures higher level metrics such as flow completion times and throughput. An ideal load balancer should be able to keep the queues balanced at both leaf and spine layers, *i.e.*, it should balance the load across uplink queues of each leaf switch as well as across the spine layer’s downlink queues connected to the same leaf switch. Hence, as the performance metric, at every $10\mu\text{sec.}$ during the 100 *sec.* experiments, we measure the standard deviation (STDV) of uplink queue lengths for each leaf switch and the STDV of queue lengths of all downlinks of spine switches connected to each leaf switch. ESF keeps this metric constantly zero, and we strive to get close to zero.

Small amounts of choice and memory dramatically improve performance.

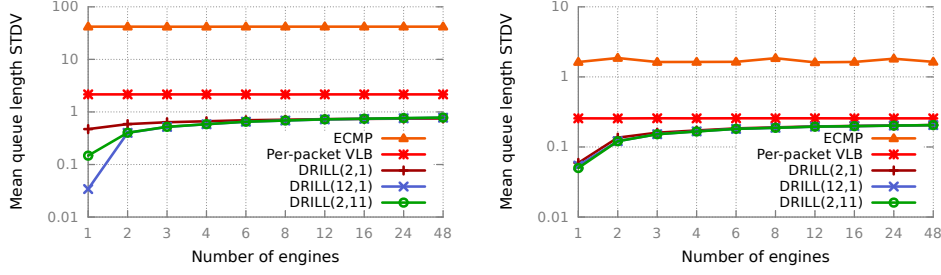


Figure 2.2: (a) 80% load. (b) 30% load. Adding a choice and a memory unit improves performance dramatically.

Our experiments show that in networks with different sizes, deploying switches with different number of engines, and under high and low load, adding a slight amount of choice and memory, *e.g.*, $\text{DRILL}(2, 1)$ instead of VLB, significantly improves the load balancing performance especially under heavy load. In networks with 48 spines and 48 leaves each connected to 48 hosts, for instance, under 80% load, $\text{DRILL}(2, 1)$ reduces the *avg.* STDV of queue lengths by over 65% compared to VLB, irrespective of the number of engines switches have, (Figure 2.2 (a)). DRILL 's improvement upon VLB is more pronounced when the number of engines is small, *e.g.*, DRILL 's mean queue length STDV is approximately 80% smaller than VLB's for single engine switches. VLB, in turn, improves upon ECMP by around 94% as a result of its finer grained, per-packet operations. When the network is less loaded and switches have more engines, however, the improvement is less dramatic. As an example, under 30% load, $\text{DRILL}(2, 1)$ outperforms VLB by around 20% if the network is built out of 48 engine switches, and by over 75% with single-engine ones (Figure 2.2 (b)).

Too much memory and too many choices may degrade performance. While a few choices and units of memory improve performance dramatically, excessive amounts of them degrade the performance for switches with large number of engines (number of engines > 6 in our experiments) under heavy load. Figures 2.3 shows an example for a network with 48-engine switches under 80% load. While the first extra choice, *i.e.*, $\text{DRILL}(1, 2)$ vs. $\text{DRILL}(1, 1)$ reduces the mean queue length STDV by 11%, having 20 choices, *i.e.*, $\text{DRILL}(1, 20)$, increases this metric by 8%. The reason is that the larger number of random samples or memory units makes it more likely for a large number of engines to simultaneously select the same set of output ports which will in turn cause bursts of packets at those ports. We call this phenomenon *synchronization effect*. The resulted load

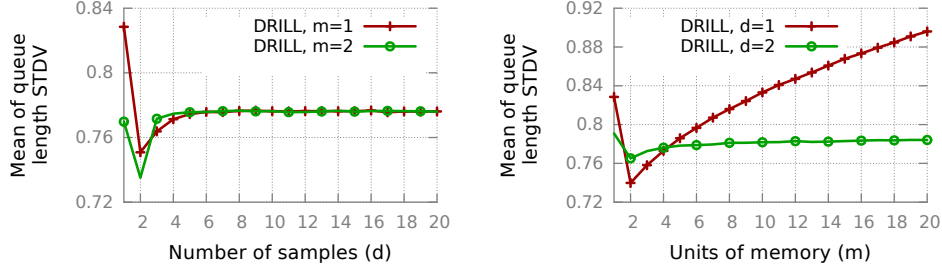


Figure 2.3: With 48-engine switches & 80% load, too many choices and memory units cause a synchronization effect.

imbalance may cause more queueing delays, *e.g.*, while the 99.999th percentile of queue lengths is below 1 under DRILL (1, 2) (*i.e.*, the queues are almost always empty), the 99th percentile of queue lengths under DRILL (1, 20) is slightly larger than 1, *i.e.*, under DRILL (1, 20), in 1% of the cases, packets experience some queueing latency because of the synchronization effect. For other cases (under light load or having fewer engines), setting $d \gg 2$ and $m \gg 1$ results in more balanced load, but the impact on queue lengths is marginal given that the queues are already almost perfectly balanced under DRILL (2, 1). With one engine switches under 80% load, for example, while the mean queue length STDV is considerably lower in DRILL (12, 1) compared to DRILL (2, 1), the 99.999th percentile of queue lengths is under 1 for both, *i.e.*, packets rarely experience any queueing delays.

DRILL guarantees stability

A system is *stable* if the expected length of no queue grows without bound [31]. We consider an $M \times N$ combined input output queued switch with FIFO queues in which the arrivals are independent and packets could be forwarded to any of the N output ports. We assume traffic admissible, *i.e.*, $\sum_{i=1}^M \delta_i \leq \sum_{i=1}^N \mu_i$, where δ_i is the arrival rate to input port i and μ_j is the service rate of output queue j . We place *no restriction on the heterogeneity of arrival rates or service rates*. These rates can be different and could dynamically change over time. Particularly, we focus on the more interesting and more challenging case where service rates could vary over time because of various reasons such as failures and recoveries that are common in data centers [32]. We first prove that purely randomized algorithms without memory, *e.g.*, DRILL (d, 0), are unstable then prove the stability

of $\text{DRILL}(d, m)$ for $m > 0$.

Pure random sampling is unstable. First, we consider $\text{DRILL}(d, 0)$, i.e., the algorithm in which every forwarding engine chooses d random outputs out of possible N queues, finds the queue with minimum occupancy between them and routes its packet to it. Theorem 1 proves that such algorithm cannot guarantee stability.

Theorem 1. *For admissible independent arrival processes, $\text{DRILL}(d, 0)$ cannot guarantee stability for any arbitrary number of samples $d < N$.*

Proof. Let δ_i be the arrival rate to engine i , and μ_j be the service rate of output queue j . Now consider output queue I . For any forwarding engine, the probability that it chooses I as a sample is $\frac{d}{N}$. So, maximum arrival rate to I is $\frac{d}{N} \times \sum_i^M \delta_i$. Thus, the minimum arrival rate to the remaining $N-1$ output queues is $\zeta = \sum_i^M \delta_i - \frac{d}{N} \times \sum_i^M \delta_i = (1 - \frac{d}{N}) \times \sum_i^M \delta_i$. Clearly, if ζ is larger than the sum of the service rates of these $N-1$ queues, the system is unstable. \square

It should be noted that the argument does not hold: (a) when there are some restrictions regarding arrival or service rates, e.g., when the service rates are equal, or (b) when $d=N$. These special cases, however, are of little interest, since the former opts out some admissible traffic patterns, and the latter nullifies the benefit of randomization and may cause a synchronization effect (§2.3.2). The results of our experiments suggest that the system performs well with $d \ll N$.

Random sampling with memory is stable. We showed above that randomized policy cannot guarantee stability without using unit of memory. Similar to [31] and using the results of Kumar and Meyn [71], we prove that DRILL 's scheduling algorithms are stable for all uniform and nonuniform independent arrival processes up to a maximum throughput of 100%.

Theorem 2. *For all admissible independent arrivals, $\text{DRILL}(1, 1)$ is stable and achieves 100% throughput.*

To prove that the algorithm is stable, we show that for an $M \times N$ switch scheduled using $\text{DRILL}(1, 1)$, there is a negative expected single-step drift in a Lyapunov function, V . In other words,

$$E[V(n+1) - V(n) | V(n)] \leq \epsilon V(n) + k,$$

where $k, \epsilon > 0$ are some constants. We do so by defining $V(n) = V_1(n) + V_2(n)$, $V_1(n) = \sum_{i=1}^N V_{1,i}(n)$, $V_{1,i}(n) = (\tilde{q}_i(n) - q^*(n))^2$, $V_2(n) = \sum_{i=1}^N q_i^2(n)$. $q_k(n)$, $\tilde{q}_i(n)$, and $q^*(n)$, respectively, represent the lengths of the k -th output queue, the output queue chosen by the engine i , and the shortest output queue under DRILL (1, 1) at time instance n . Details of the proof are included in §A.

2.3.3 DRILL Causes Minimal Packet Reordering

DRILL makes forwarding decisions for each packet, independent of other packets of the same flow, based on the local and potentially volatile switch load information. One might expect this approach to cause excessive degrees of packet reordering that could degrade TCP performance. Reordering may degrade TCP’s performance by triggering its duplicate ACK mechanism, one of the primary means of TCP for detecting packet loss. As explained in RFC2581 [72], when a segment arrives out of order, the receiver immediately sends a “duplicate ACK” to the sender. The sender uses the TCP *retransmission threshold*, the arrival of *three* duplicate ACKs, as an indication of packet loss and infer that the network is congested. It then reacts by retransmitting the packet that is perceived lost and reducing its transmission rate. Wary of this rate reduction, the majority of load balancing schemes, from ECMP to CONGA [2] to Presto [9], balance coarser units of traffic in an effort to mitigate the risk of packet reordering.

Although DRILL splits flows into packets, the finest practical unit, and forwards them independently, it causes minimal packet reordering. This may be somewhat surprising, but using multiple paths only causes reordering if the delays along those paths differ by more than the time between packets in a flow. Queueing delay is famously the dominant source of network delay in datacenters [70], and DRILL’s well balanced load and extremely low variances among queue lengths (as demonstrated in §2.3.2) imply that packets experience almost identical queueing delays irrespective of the paths they take. Hence, even though flows’ packets take divergent paths at very fine granularity, they should not be reordered frequently. Our experiments, using the actual TCP implementations taken from Linux 2.6, confirm this hypothesis and show that TCP performance is not significantly impacted (§2.4).

However, for certain legacy or specialized applications it may be desirable to eliminate all reordering. In a modern virtualized data center, can be accomplished

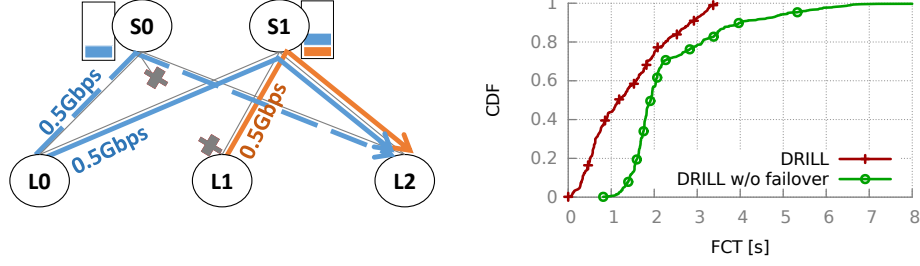


Figure 2.4: L1-S0 link failure increases FCT in DRILL.

with an end-host “shim” layer transparent to the guest OS as developed in [9]. In §2.4 we evaluate both variants of DRILL, with the shim and without.

2.3.4 DRILL Decomposes Asymmetric Networks Into Symmetric Components

For the majority of failures—failures of hosts, switches at any stage, and the links between switches and hosts—ESF (and $\text{DRILL}(\mathbf{d}, \mathbf{m})$ as its approximation) is self-healing because such changes turn the topology into a smaller, but still symmetric, Clos. For such failures, the underlying topology dissemination mechanisms, such as those deployed by ECMP, purge the failed parts from forwarding tables and DRILL continues to distribute traffic among the remaining paths as before.

If a link between a leaf and a spine fails or leaf-spine links have different capacities, however, local load balancers that try to split the load equally among available paths may waste bandwidth because of their interactions with TCP’s control loop, as noted in [2]. This happens because the asymmetric paths available to a flow may have different and varying capacities for it (depending on the load of other flows that use those paths). Flow rates on each path are controlled by TCP to avoid congestion. So splitting the load of the flow equally among asymmetric paths effectively limits its rates on each path to the rate of the most congested path. This implies that the paths with more capacity will be left underutilized even if the flow has a demand for their bandwidth.

As a simple example, consider Figure 2.4 (left) where hosts under leaf switches L_0 and L_1 have infinite TCP traffic demands to send to those under L_2 . Assume that the link between L_1 and S_0 fails and that all links have 1Gbps capacity. Under local schemes such as ESF, this link failure can cause collateral damage to the

flows going through other links. This happens because the flows from L_0 and L_1 that are sent to S_1 share the bottleneck link $S_1 \rightarrow L_2$. Assuming that the number of these flows are equal and they are all in the steady state, TCP does not allow the rate of the flows from L_0 that take the path Q , $L_0 \rightarrow S_1 \rightarrow L_2$, to increase beyond $0.5Gbps$ to avoid congestion on $S_1 \rightarrow L_2$. Now if the load balancer tries to keep the load on path P ($L_0 \rightarrow S_0 \rightarrow L_2$) and Q equal, it keeps the rate on P also equal to $0.5Gbps$, in spite of the fact that P can serve traffic at $1Gbps$. In other words, 50% of the bandwidth of P will be lost. A similar experiment, discussed later in this section, shows that, without its failover mechanism, DRILL(2, 1) also wastes around 50% of the capacity of this path. Note that some other local load balancers also suffer from this problem. Presto’s failover mechanism [9], for example, prunes the spanning trees affected by the failure and uses a static weighted scheduling algorithm, similar to WCMP [7], over the remaining paths. In this example, since P and Q have static capacity of $1Gbps$ each, their associated weights will be equal and Presto continues to equally spread $L_0 \rightarrow L_2$ ’s load across them.

Note that changing weights in a load oblivious manner does not solve this problem since the appropriate weight values depend on the load from other sources—a potentially rapidly evolving parameter. In the above example, for instance, optimal weight assignments would be $w(P)=1$ and $w(Q)=0$, but if the $L_1 \rightarrow L_2$ demand was 0, then the optimal weights would be $w(P)=w(Q)=1$ as the previous weight assignment leaves Q idle.

Also note that in addition to this *bandwidth inefficiency* in the asymmetric case, local schemes such as Presto and DRILL that split flows across asymmetric paths may cause an extra problem of *excessive packet reordering*. In the example above, packets traversing Q experience higher queueing delay compared to those traversing P given that S_1 is more congested than S_0 . Splitting flows between P and Q , therefore, may result in an exceeding degree of reordering under heavy load.

We observe that both problems are rooted in imposing rate dependencies across asymmetric paths, *e.g.*, keeping P and Q rates equal in the example above. Intuitively, to solve these problems, DRILL needs to break the rate dependencies between asymmetric paths. To achieve this, DRILL decomposes the network into components with symmetric paths (defined below), assigns each flow to one component, and balances it among the paths inside it. The *utilization factor* [73] of the path $L_{src} \rightarrow S_i \rightarrow L_{dst}$ is defined as $u(src, i, dst) =$

$\frac{\text{capacity}(S_i \rightarrow L_{dst})}{\text{capacity}(L_{src} \rightarrow S_i)}$. At leaf L_{src} , for any two paths $L_{src} \rightarrow S_i \rightarrow L_{dst}$ and $L_{src} \rightarrow S_j \rightarrow L_{dst}$ towards leaf L_{dst} , to be *symmetric*, not

only their utilization factors through S_i and S_j should be equal, but also the utilization factors of any other leaf switch, L_k , that use those 2 spines towards L_{dst} should be equal as well, *i.e.*, $u(k, i, dst) = u(k, j, dst)$. The reason to impose this condition on utilization factors is to keep each component's queues, in both its leaf and spine layers, balanced. Theorem 3 in §A shows that, for admissible independent traffic, this condition is sufficient to guarantee DRILL's stability and 100% throughput. In the example above, $u(1, 0, 2) = 0$ whereas $u(1, 1, 2) = 1$. This implies that the load towards L_2 may be different at S_0 and S_1 . Thus, L_0 puts P and Q in different components and avoids splitting flows across them.

DRILL's failover algorithm: If the topology is asymmetric, DRILL follows 3 steps: **Step 1: Network decomposition.** For each destination leaf L_{dst} , each leaf L_{src} first detects the group of all available spines $\cup_i S_i$ connected to both L_{src} and L_{dst} . L_{src} then annotates each spine S_i in this group with a set of pairs, where each pair shows the ID of the leaf that can send traffic to L_{dst} via S_i and its utilization factor, *i.e.*, $A_{S_i, L_{dst}} = \cup_j (L_j, u(j, i, dst))$. In the example above, L_0 's annotation for S_1 towards L_2 is $A_{S_1, L_2} = \{(L_0, 1), (L_1, 1)\}$. Components are then the largest sets of spines with identical annotations, *i.e.*, any two spines S_i and S_j are in the same component *iff* $A_{S_i, L_{dst}} = A_{S_j, L_{dst}}$. Once each leaf decomposes the set of available paths to each destination into symmetric components, each DRILL's source assigns a weight to each component which is proportional to the aggregate utilization factor of its paths *from that source*. In the example above, L_0 detects that P and Q have different annotations but equal utilization factors. So it puts them in different components with equal weights. This weight assignment to components is similar to the path weight assignments in [9, 7] and can be implemented in switches with the techniques discussed in [74]. **Step 2: Flow classification.** By hashing the 5-tuple header of each packet, DRILL assigns it to a component considering the weights set in the previous step. **Step 3: Intra-component micro load balancing.** Inside each component, DRILL uses $\text{DRILL}(d, m)$ to balance the load across its symmetric paths.

The algorithm above avoids bandwidth loss and reduces reordering by restricting application of per-packet load balancing to symmetric paths. As an example, consider the topology depicted in Figure 2.4, with a workload similar to the Interpod Incoming workload [54] where hosts send traffic to those under L_0 and L_2 . In this experiment, each leaf is connected to 40 hosts each sending TCP flows to a random host under L_0 or L_2 with equal probability. Assume further that flows sizes are drawn from a normal distribution with the mean flow size equal to 20MB,

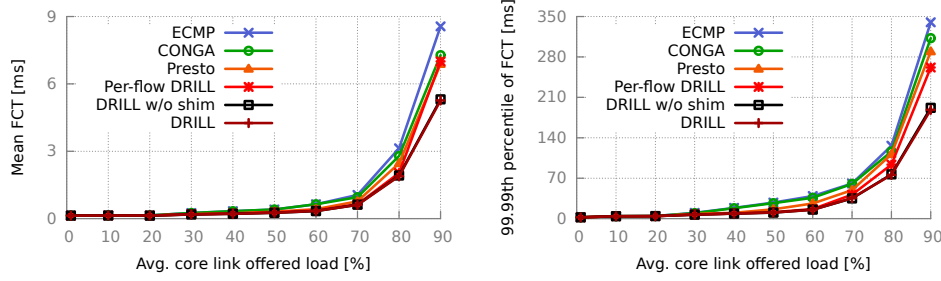


Figure 2.5: DRILL improves latency in a symmetric Clos.

and flow interarrival times are generated by a poisson process (similar to [2]) in such a way to keep the offered core load equal to 80%. DRILL (2, 1), without its failover algorithm, keeps the utilization of P to 43%, and causes 13% packet re-ordering (the rate of packet reordering with Presto’s failover mechanism is 37%). These two factors can increase flow completion times (FCT) as Figure 2.4 (right) shows. DRILL’s failover algorithm mitigates this problem by avoiding reordering and increasing the utilization of P to 74%.

While, for ease of exposition, we focus on a two-tier Clos here, our design and results are recursively applicable to a Clos with arbitrary depth.

2.4 Evaluation

We evaluate DRILL in detailed simulations. We find that DRILL achieves high performance, *e.g.*, it has $0.77\times$, $0.68\times$, and $0.6\times$ lower mean FCT than ECMP, CONGA, and Presto, respectively, under 80% load. Both our fine granularity and load-awareness are important factors in that performance, with the second becoming more important in highly bursty traffic patterns such as incast, and with link failures. DRILL is especially effective in handling incast as it is the most agile load balancer to react to spontaneous load bursts; it results in $2.5\times$ and $3.5\times$ lower 99.99th percentile of FCT compared to CONGA and Presto, respectively. We also show DRILL has minimal packet reordering, and explore the effect of failures, synthetic traffic patterns, and scaling out. Finally, we implemented DRILL in Verilog to evaluate deployability. Details of these evaluations follow.

Performance evaluation methodology: To test DRILL’s performance at scale, we measure flow completion times (FCT) and throughput under DRILL, and compare it with CONGA, Presto, and ECMP via simulation. We use the OMNET++

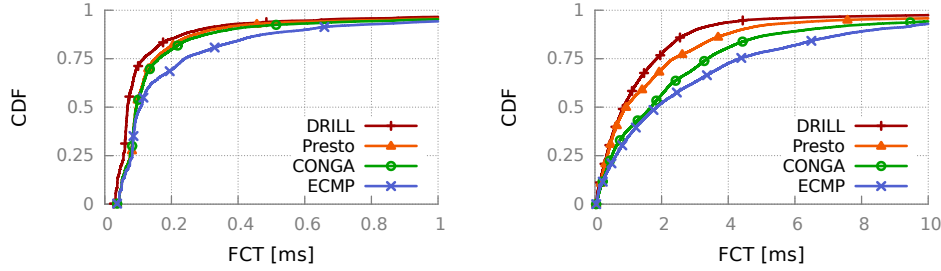


Figure 2.6: (a) 30% load (b) 80% load. DRILL’s improvement is greater under heavy load.

simulator [75] and the INET framework [76], with standard Ethernet switches’ and hosts’ networking stacks. We port the real-world TCP implementations taken from Linux 2.6 via the Network Simulation Cradle library [77]. For DRILL, unless stated otherwise, we use single engine switches under $\text{DRILL}(2, 1)$. We use 2 and 3 stage Clos networks with various sizes, without failures and with multiple link failures, under a set of realistic and synthetic workloads, and an incast application.

In a symmetric Clos, DRILL reduces mean and tail latencies. We use trace-driven workloads from real datacenter traffic for flow sizes, flow interarrival times, and traffic pattern from [14], and use a Clos with 8 spine and 10 leaf switches, where each leaf is connected to 40 hosts; all links are 1Gbps. To emulate various degrees of the core offered load, we scale flow interarrival times. Under this setting, we find the load balancing granularity to be a key player in the effectiveness of the load balancer. DRILL achieves lower FCT compared to Presto which in turn has lower FCT than CONGA. The difference is larger under heavy load and in the tail, *e.g.*, under 80% load, DRILL reduces the 99.99th percentile of FCT of Presto and CONGA by 32% and 35%, respectively (Figure 2.5). Figure 2.6 shows the FCT CDFs for 30% and 80% load. Datacenters today experience high congestion drops as utilization approaches 25% [8]. Thus, the average load is kept around 25% to control the latency [8, 14]. We note that compared to ECMP, DRILL allows the providers to use 10% more of their bandwidth capacity while keeping the 99.99th percentile of FCT lower than ECMP’s under 25% load. That is, DRILL supports $1.4\times$ higher load with the same tail FCT performance compared with ECMP, $1.32\times$ higher than CONGA and $1.25\times$ higher than Presto.

Note that despite the importance of the load balancing granularity, load-awareness is important too even in the symmetric case. We show a strawman

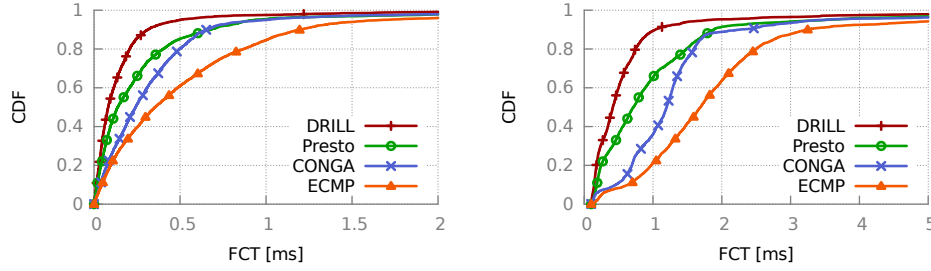


Figure 2.7: DRILL keeps FCT short in a VL2 network under (a) 30% and (b) 60% load.

“per-flow DRILL” which makes load-aware decisions for the first packet of a flow and then pins the flow; this marginally improves the tail latency of Presto and CONGA while being coarser grained than both.

DRILL has minimal packet reordering. The previous figures show that FCT is low despite reordering, but next we dig deeper to see why. Figure 2.8 shows amount of reordering measured in terms of the number of TCP duplicate ACKs, under 30% and 80% load. ECMP and CONGA do not cause reordering, but as a strawman comparison, we also show the amount of reordering in per-packet VLB (i.e., random forwarding of each packet with no load-awareness). We note two important conclusions. First, per-packet VLB and DRILL have the same granularity of load balancing but DRILL has dramatically lower packet reordering. This demonstrates how local load awareness keeps queues extremely well-balanced across paths.

Second, the degree of reordering under DRILL rarely reaches the TCP retransmission threshold. Under 30% load only 0.9% of flows have any duplicate ACKs, and only 0.08% have more than the typical TCP retransmission threshold of 3. Even under 80% load, these numbers are 5.8% and 0.6%, respectively – more than $3.2\times$ lower than per-packet VLB and $2.6\times$ lower than Presto without its shim layer. This minimal degree of reordering shows why DRILL with and without the shim layer have very similar performance.

DRILL gracefully handles failures. Even though high scale datacenters show high reliability [32], with the majority of links having higher than four 9’s of reliability [32], there is still a high probability of at least one failure at each point in time [78, 7]. Therefore, handling failures gracefully is imperative for any load balancer. We test the performance of DRILL under 2 failure scenarios: (a) one single leaf-spine link failure, as single failures are the most common failure cases

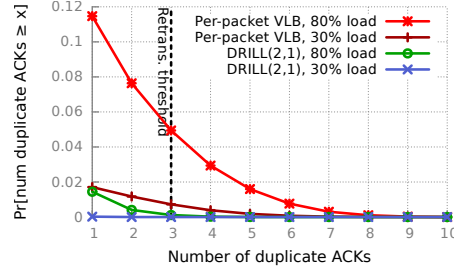


Figure 2.8: For fewer than 0.006 fraction of flows, DRILL reorders enough packets to reduce TCP’s transmission rate (*i.e.*, $\Pr[\text{num dup ACKs} \geq 3]$) even under high load.

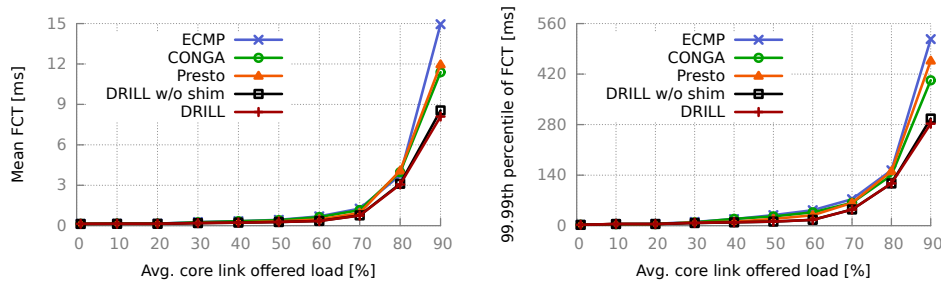


Figure 2.9: DRILL gracefully handles single link failures.

in datacenters [32], and (b) 5 randomly selected leaf-spine link failures; this scenario presents a rare, but still possible, case. Even in large scale datacenters, big groups of correlated link failures are rare with only 10% of failure groups (failures with downtimes either simultaneous or close together in time) containing more than four failures [32]. As before, we load the system up to 90% of the available core capacity. We observe that DRILL and CONGA are more effective in handling multiple failures (Figures 2.9 and 2.10). This is because CONGA shifts the load towards the parts of the topology with more capacity, and DRILL breaks the rate interdependencies between asymmetric paths, effectively allowing flows to grab the available bandwidth, increase their rates, and finish faster. Note that in all these cases, DRILL’s performance with and without the shim layer that reorders the out-of-order packets (from [9]) are almost identical, since its degree of reordering is so low that it rarely reaches TCP’s retransmission threshold (§2.3.3).

DRILL reduces the tail latency in incast scenarios. A common and vexing traffic pattern in datacenters is incast [8, 70]. It is one of the key factors in causing excessive congestion and packet loss [8]. With the exception of a recent study from Google that reports incast-induced packet drops at various layers [8], most

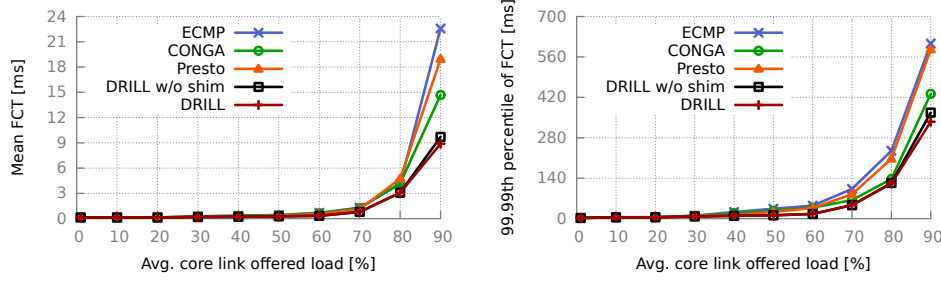


Figure 2.10: DRILL gracefully handles 5 link failures.

of the works on incast study the problem within a cluster (hosts connected via one switch or a tree topology), and naturally focus exclusively on overrun of the last hop buffer (connected to the receiver) [79, 80, 81, 82, 83, 84, 85, 86, 87, 88]. Consistent with the observation in [8], our experiments show that in multi-rooted datacenter topologies, the incast traffic pattern triggers buffer overruns at other layers too. Furthermore, our results underscore the fact that this problem is interwoven with load balancing and can be mitigated by an agile load balancer capable of reacting to microbursts. Figure 2.11 shows an example for a network under the typical load of 20%; hosts run an incast application similar to [79], and 10% of them send simultaneous requests for 10KB flows to 10% of the other hosts (all randomly selected). The background traffic and interarrival times are drawn from [14] as before. DRILL significantly reduces the tail latency; it has $2.5\times$ and $3.5\times$ lower 99.99th percentile of FCT compared to CONGA and Presto, respectively. As the load increases, the gap widens, *e.g.*, the reduction is $3.1\times$ and $4.7\times$ under 40% load (not shown). This happens because this highly bursty traffic pattern causes microbursts not just at the last hop, but at other layers of the topology too. DRILL can swiftly divert the load and reduce the loss rate; its loss rate is, respectively, 32% and 40% of that of Presto and CONGA. Plus, only 11% of the packet loss with DRILL happens at layers other than the last hop; the corresponding number is 41% and 29% under Presto and CONGA, respectively.

Synthetic workloads: In addition to the trace driven workload, similar to previous works [54, 23, 1, 9], we use a set of synthetic workloads, known to either appear frequently in datacenters or to be challenging for load balancing designs [23]: *Stride(x)* in which server[i] sends flows to server[(i+x) mod number of servers], *Random* where each server communicates with a random destination not under the same leaf as itself. We use *Stride(8)*, and *Shuffle* in which each server sends flows to all other servers in a random order. Similar to [9], we use 1GB “elephant”

flows, and in addition we send 50 KB “mice flows” every 100 ms. We use a Clos with 4 leaf and 4 spine switches with each leaf connected to 8 hosts where all links have 1Gbps capacity. Table 2.1 reports the mean and 99.99th percentile of FCT for mice and mean flow throughput for elephants, all normalized by ECMP. For the Random and Stride workloads, DRILL significantly reduces mice latencies particularly in the tail and achieves higher throughput for the elephant flows. None of the tested schemes improve ECMP much for the shuffle workload since it is mainly bottlenecked at the last hop.

Effect of scale: We also test DRILL’s ability to balance load in Clos topologies with more than 2 stages such as VL2 [15] and fat-tree [54]. Figure 2.7 shows the result of an experiment with a VL2 network with 4 ToR switches, each connected to 40 hosts, 4 Aggregate switches, and 2 Intermediate switches. All links are 1Gbps. We put 30% and 60% load on the network. Figure 2.7 shows that DRILL is effective in keeping the FCT short in such networks.

We also tested the effect of scale in terms of number of forwarding engines in each switch. We find the impact of the number of engines to be negligible on FCT for $\text{DRILL}(2, 1)$, *e.g.*, we find less than 0.9% difference in the mean FCT between 1- and 48-engine switches under 80% load (no plot).

Hardware and deployability considerations: We implemented DRILL in Verilog in less than 400 lines of code. We estimate DRILL’s area overhead by using Xilinx tools from ISE9.1i and the area estimation from [89, 90]. DRILL is estimated to require $0.04mm^2$ of chip area. Using the minimum chip area estimate of $200mm^2$ in [91], similar to [92], we estimate this to be about 0.2% of the area of a typical switch chip. This demonstrates the feasibility and ease of implementing DRILL in hardware.

DRILL involves two additional components. In the case of topological asymmetry, switches need to calculate the weights of traffic for each symmetric component; this can be done in control software local to the switch (if topology information is available via the routing algorithm) or through a central controller. Optionally, DRILL can employ a shim layer, deployed in a hypervisor as in [9]. As we have shown, this is not always necessary, and [9] showed it is feasible for modern virtualized datacenters.

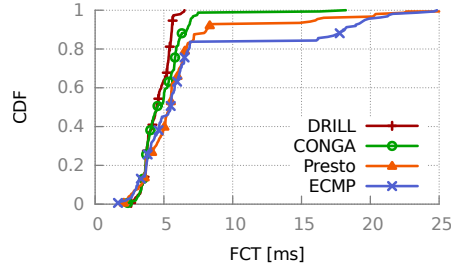


Figure 2.11: DRILL cuts the tail latency in incast scenarios.

2.5 Related Work

Recent works attribute the poor performance of ECMP to (a) its lack of global congestion information, or (b) hash collision when there are large flows. In the first group, Planck presents a fast network measurement architecture that enables rerouting congested flows in milliseconds [1]. Fastpass [26] posits that each sender should delegate control to a centralized arbiter to dictate when and via which path each packet should be transmitted. Hedera [23], MicroTE [3], and Mahout [27] re-route large flows to compensate for the inefficiency of ECMP hashing them onto the same path.

In the second category, Presto argues that in a symmetric Clos where all flows are small, ECMP provides near optimal load balancing [9]. Presto divides flows into “flowcells” which are source-routed so they are striped across all paths; a centralized controller helps respond to failures. Other efforts in this category include dividing flows into “flowlets” [93, 2] and balancing flowlets instead of flows, or per-packet spreading of traffic in a round robin fashion [21, 24]. Presto’s choice of flowcells is motivated by the fact that flowlets are coarse grained and dictated by the practical challenges of performing per-packet load balancing *in hosts*. The assumption among the work in this category is that ECMP’s inefficiency is mainly caused by large flows and can be addressed by splitting flows into small units and routing them separately in a proactive manner, with no need for load information.

CONGA takes a hybrid approach by both splitting traffic into flowlets and using in-network congestion feedback mechanisms to estimate load and allocate flowlets to paths based on the congestion feedback. Their main thesis is that efficient load balancing requires global load information. Our experiments indicate that DRILL’s micro load balancing outperforms these proposals.

DRILL’s queueing algorithm is inspired by the seminal “power of two choices”

	Stride			Bijection			Shuffle		
	CONGA	Presto	DRILL	CONGA	Presto	DRILL	CONGA	Presto	DRILL
Eleph. trput	1.55	1.71	1.8	1.46	1.62	1.78	1	1.1	1.1
Mean FCT	0.51	0.41	0.21	0.71	0.63	0.45	0.95	0.91	0.86
99.99-p FCT	0.2	0.15	0.04	0.22	0.18	0.08	0.86	0.79	0.68

Table 2.1: Mean elephant flow throughput and mice FCT normalized to ECMP for the synthetic workloads.

work on using randomized load-sensitive algorithms for load balancing [30]. Mitzenmacher showed that in the *supermarket model*, with a single input queue and many output queues, load balance greatly improves with $d \geq 2$ choices [30]. [94] and [95] study the impact of using memory on the performance and stability of randomized load balancing. Central to these theoretical models is having *one* arbiter responsible for balancing the load among multiple servers. Our setting, however, may have multiple arbiters (i.e. forwarding engines), which produces distinctly different behavior; in particular, as d increases, performance can worsen (Figure 2.3). This has led us to experimentally optimize parameter choice, but a theoretical analysis of our model may be valuable work in the future.

2.6 Conclusion

Contrary to the pervasive approach of load balancing based on macroscopic view of traffic, we explore *micro load balancing*: enabling the fabric to make decisions at $\mu sec.$ timescales based on traffic information local to each switch. Our experiments show that our simple provably-stable switch scheduling algorithm, DRILL, outperforms the state-of-the-art load balancers in Clos networks, particularly under load. To achieve this, DRILL makes per-packet decisions at each switch based on local queue depth and randomization. DRILL adapts to asymmetry by decomposing the network into symmetric parts. We implement DRILL in Verilog to show its switch implementation feasibility. We leave the study of micro load balancers in other topologies to future work.

Chapter 3

SEAMLESS SCALE-OUT OF NETWORK ELEMENTS WITH COCONUT

A key use of software-defined networking is to enable scale-out of network data plane elements. Naively scaling networking elements, however, can cause incorrect behavior. For example, we show that an IDS system which operates correctly as a single network element can erroneously and permanently block hosts when it is replicated.

In this chapter, we provide a system, COCONUT, for *seamless scale-out* of network forwarding elements; that is, an SDN application programmer can program to what functionally appears to be a single forwarding element, but which may be replicated behind the scenes. To do this, we identify the key property for seamless scale out, weak causality, and guarantee it through a practical and scalable implementation of vector clocks in the data plane. We prove that COCONUT enables seamless scale out of networking elements, i.e., the user-perceived behavior of any COCONUT element implemented with a distributed set of concurrent replicas is provably indistinguishable from its singleton implementation. Finally, we build a prototype of COCONUT and experimentally demonstrate its correct behavior. We also show that its abstraction enables a more efficient implementation of seamless scale-out compared to a naive baseline.

3.1 Introduction

An important use of software-defined networking (SDN) is to automate scaling of networks, so that individual network functions or forwarding elements can be replicated as necessary. Replication of network elements allows capacity to scale gracefully with demand [33], provides high availability [33], and assists function mobility [34, 35]. Multiple SDN systems replicate network elements, in different ways. Each tenant in a virtualized data center might be presented one logical “big switch” abstraction that in reality spans multiple physical hardware or software

switches [33, 36, 37]. As another example, Microsoft Azure’s host-based SDN solution leverages VMSwitches to build virtual networks where each host performs all packet-actions for its own VMs [38]; these VMSwitches act in parallel and independently despite the fact that they might form a single virtual network. Outside of virtualization, caching of forwarding rules is a form of replication; for example, [40, 41, 42] cache rules at multiple locations in the network, and Open vSwitch [43] caches rules from user-space into kernel-space, which is critical to improve performance.

All these systems replicate logical network elements by duplicating forwarding rules across multiple locations, without coordination between them, which we call *simple replication*. Our work begins by asking: *Does simple replication for scaling out network elements preserve the semantics of a single element?* If the network elements are stateless, the simple replication approach taken by existing systems is enough (§3.3). But if a developer writes a network function or application such as a stateful firewall on top of a single virtual “big switch”, is its functional behavior the same as if it were running on an actual single physical switch? We show that simple replication does indeed change the network’s semantics: for example, a replicated firewall can erroneously and *permanently* block hosts. In fact, our experiments show there are scenarios that these problems occur frequently (§3.3).

How, then, could an SDN programmer deal with this problem? Living with the risk of incorrect functionality is unappealing, as critical infrastructure elements such as security appliances (firewalls, intrusion detection systems, *etc.*) are increasingly deployed in a scale-out manner. Alternately, the programmer could write her application so that it takes into account the distributed implementation of network elements and associated race conditions. But this is inconvenient for the programmer at best, and is infeasible at worst, when replication is explicitly hidden in the physical infrastructure underneath a tenant’s virtual network. Indeed, one lure of the virtualized cloud for tenants is the prospect of migrating their workloads and network applications to the cloud “as-is”, *i.e.*, with no re-designing and re-architecting of their applications, with the expectation that they perform exactly akin to their non-virtualized networks [44, 6].

Our goal is thus to achieve **seamless scale-out** for network forwarding elements: *a system which guarantees that an SDN application writer can program to the semantics of a single device, but which utilizes multiple replicated elements behind the scenes.*

Achieving seamless scale-out is not easy. The most generic solution would be to synchronize replicas to provide a strongly consistent logical view, but the required locking would not achieve the performance necessary for the data plane [38, 42]. Recent work [45, 46, 47, 48] provides a form of consistency in the data plane in the sense of ensuring “trace” properties of a single packet’s path, as in Consistent Updates (CU) [46]. But this is essentially orthogonal to our goal; seamless scale-out does not require per-packet path consistency, and systems that provide per-packet path consistency can even *cause* the correctness problems described above (§3.3.3). Also, the mechanisms used to implement CU assume a single atomic update point (the ingress switch of a packet’s path). No such atomic update point exists in our setting, because we need to preserve the single-device semantics across a large number of flows across the whole network with potentially unspoken dependencies.

The system we present here, COCONUT (“COrrrect COncurrent Network-ing UTensils”), provides seamless scale-out for network elements defined by a dynamically-updatable OpenFlow-like abstraction. To work towards a solution, we observe that the culprit of scale-out correctness problems is violation of what we call *weak causality*. For example, simple replication can cause a replicated firewall to miss the weak causal dependency between a client’s outbound request and a server’s inbound response, so it sees a seemingly-unsolicited inbound response first and permanently blocks traffic from that server.¹ We design a set of high-level algorithms to avoid weak causality violations, drawing on the classical concept of logical clocks [49] to track the state of each forwarding rule at each abstract network element. But providing a practical and scalable implementation of these high-level algorithms is challenging; switches do not directly implement logical clocks, and emulating a large vector of logical clocks in packet header fields is impractical. We provide a practical realization of those algorithms using OpenFlow-compatible switches, leveraging the distinguishing characteristics of SDNs and virtualized networks, and is thus suitable for deployment in the context of a modern virtualized data center with software switches in each physical host. Our design uses limited bits in header fields of a physical network to emulate logical clocks in the virtual network, while dealing with concurrent creations and changes of multiple virtual networks that may interleave with each other and

¹Note that even this simple example involves multiple flows entering the network at different points, illustrating the aforementioned insufficiency of using a single atomic update point as in CU [46].

compete for use of these bits.

To prove that COCONUT correctly provides seamless scale-out, we need a new analytical framework. To show that the scaled-out network is indistinguishable from a singleton network element, we need to take into account the sequence of observations made by the end-points, with potential interdependencies. We formalize this with a definition we call **observational correctness** that requires that any sequence of end-point observations in the scaled-out network is plausible for the singleton version. In tune with what applications expect from best-effort networks, this model is permissive of occasional packet drops and re-ordering, while prohibiting weak causality violations (breaking “happened before” relations [49] adopted for best-effort networks, §3.4.1) that could jeopardize applications’ correctness. We formally prove that COCONUT provides observational correctness.

We implemented a prototype of COCONUT integrated with Floodlight [50], Open vSwitch [43], and OpenVirtex [5]. We evaluated COCONUT and several alternative schemes on a hardware SDN testbed arranged to emulate a 20-switch fat-tree topology and in Mininet [51] emulations up to 180 switches, with multiple topologies, load patterns, and SDN application scenarios. Our findings are as follows: **(a)** A strawman solution, providing strong consistency (SC) similar to [34] by routing all data traffic through a controller during updates, would come at too high a cost: about 12 Gbps bandwidth overhead and a $20\times$ increase in user traffic latency even in a modest-sized network. COCONUT incurs no measurable data plane performance overhead, and has significantly lower overhead in terms of forwarding rule update delay ($3.5\times$ faster in a network with 128 hosts and 80 switches) and number of forwarding rules ($2\times$ lower).

(b) Compared with baseline simple replication (which lacks seamless scale-out), COCONUT correctly achieves seamless scale-out with modest overhead. For a 180-switch network, for example, the mean forwarding rule updates is only $1.2\times$ slower than simple replication, and the mean number of forwarding rules increases by only $1.6\times$, with just 0.7% of that overhead persisting for longer than 100ms.

(c) We also compare with a natural implementation where the programmer avoids replication-related race conditions “manually” within the SDN application. COCONUT enables an implementation that is both more convenient for the programmer and provides $2.8\times$ lower mean latency for user data flow initiation due to its efficient logical clock-based approach.

In summary, our key contributions are: (1) we observe that simple replication

breaks the semantics of a single network element and show experimentally that it causes application-level incorrect behavior; (2) we present COCONUT , a system for seamless scale-out in the context of OpenFlow forwarding elements in a virtualized data center, and prove it correctly preserves a single-element abstraction; (3) we demonstrate experimentally that COCONUT achieves its goals with modest performance overhead. We believe this lays the foundation for a practical and dependable service model for virtualized network infrastructure, as well as a powerful abstraction for programming SDNs.

3.2 Background

In this section, we discuss the logical abstractions we provide and some of the applications of replication.

3.2.1 Basic Abstractions

COCONUT provides seamless scale-out for network elements. The abstraction of a *network element* that we work with here is essentially an SDN device such as an OpenFlow switch. Each element or switch has a table of *rules*, each rule containing a *priority*, a *match* on packet headers, and a list of *actions*. Although each individual rule may be stateless, the system is not: the controller can dynamically update rules based on dataplane events, *e.g.*, failures. Upon receiving a packet the switch executes the actions for the highest priority rule that matches the packet. These actions could result in changes in the packet, dropping it, or forwarding it.

(How) are networking elements scaled-out today? Scaling out can be realized via *simple replication* or one-to-many mapping, where a logical rule is mapped to a distributed set of physical rules, each individually capable of fully implementing the logical rule. In this technique, before installing a rule in multiple physical flow tables, an entity such as the network hypervisor [6] typically rewrites the rule. For example, a rule that matches on virtual ports will need to be rewritten to refer to physical ports [33, 34]; virtual addresses may be translated to physical addresses or packets may be placed into tunnels [6]; and rules that match in part with wildcards may be “cloned” into multiple entries in which wildcarded fields are replaced by exact-match values [96, 6]. The latter mechanism is used in

software switches, where wildcard entries in userspace are cached as exact match ones in the kernel to enhance performance [6].

Prior to COCONUT, a number of systems have provided simple replication, mostly for scaling out the *static stateless* network elements, *i.e.*, those whose actions or presence in the network do not depend on the history of previous matching packets or previous actions [97]. Simple replication of stateless elements preserves the semantics of applications [97].

Modern programmable networks, however, are exceedingly more dynamic. This can come in the form of controllers adding, removing, or modifying forwarding rules dynamically in response to application traffic. The question is, can these stateful elements be replicated via simple replication technique? In §3.2.2, we list a few key existing applications of simple replication, before showing in §3.3 that this technique may cause incorrect application behavior when used for implementing dynamic stateful network functions; We also show that the existing works on correctness in network not only do not solve this problem but can exacerbate it (§3.3).

3.2.2 Applications of Replication

Network virtualization: Simple replication is a key technique for building distributed virtual switches. Nicira’s NVP [6] and OpenVirtex [5], for example, provide a one-big-switch abstraction that can connect VMs on the same virtual network even though they are located in different physical hosts or regions of the physical network, and whose locations may change due to spinning up VMs or VM mobility. This is implemented with simple replication from a single virtual switch onto a distributed set of software switches. **Composition** of multiple virtual switches can also result in replication. Under existing composition techniques, multiple logical rules are jointly mapped to a set of physical rules where each physical rule is individually capable of implementing multiple logical rules [98, 37, 99]. For example, Figure 3.1 shows (a) a monitoring module that performs monitoring based on source address, (b) a destination-based routing module, and (c) a Monitoring+Routing application resulting from parallel composition of the previous two modules (rules ordered from highest to lowest priorities) [37]. The first rule of the monitoring, module, for example, is implemented with 2 rules in the composed application.

Network Function Virtualization (NFV): Performance is a critical consider-

Monitoring	Routing	Monitoring + Routing
srcip=127.1.*.* count	dstip=127.2.*.* fwd(1)	srcip= 127.1.*.* ,dstip=127.2.*.* count,fwd(1)
* drop	* drop	srcip= 127.1.*.* count
		dstip=127.2.*.* fwd(1)
		* drop

Figure 3.1: Composing monitoring and routing.

ation in NFV where software is used to implement network functions or applications such as firewalls, load balancers, *etc.* Simple replication, used in caching, is a key technique to enhance forwarding performance in software switches and NFVs [6, 43, 42, 41, 40].

Implementing higher level abstractions: In the context of network programming languages, Frenetic [98] provides high-level primitives such set difference, not directly supported by the hardware, by mapping those primitives to multiple OpenFlow rules, *e.g.*, a rule with the match field `src-IP=186.206.176.* OR src-IP=62.205.112.38`, is implemented via two rules.

In all of the above techniques, each physical instance or replica is *functionally equivalent* to a faithful implementation of one (or more) logical rules, *i.e.*, the replica performs identical actions as the logical rule. In a fully static network, packets traversing the physical network result in the same end-to-end fate as if they were processed directly by rules in the ideal, non-replicated implementation of the logical network. However, as network state changes over time, there may be inconsistent state among the multiple replicas that implement one logical element. Furthermore, this problem may become more serious if the changes are interdependent with application-level behavior (rather than simple route changes). We next see how this may cause application-level incorrect behavior.

3.3 What Can Go Wrong?

We show with a few examples that the simple replication can break the semantics of a single element and lead to incorrect application behavior. Per-packet consis-

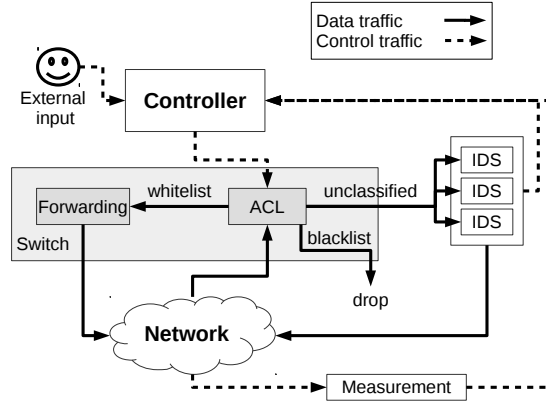


Figure 3.2: SDN-enabled security architecture.

tency [46] does not fix the problem, and interestingly, can even trigger the problem in an otherwise-correct network.

3.3.1 Example 1: SDN-enabled Security

Network Intrusion Detection systems (IDS) and stateful firewalls perform complex traffic processing and analysis that are CPU intensive and hard to implement at high speed. Performance can be improved significantly by programming faster devices like SDN switches to act as an initial triage filter [100]. As depicted in Figure 3.2, the switch *whitelists* traffic known to be benign, forwarding it directly to its destination; *blacklists* traffic known to be malicious, dropping it immediately; and sends only the remaining *unclassified* traffic to the IDS device for more expensive analysis (e.g., DPI). The controller uses external input, traffic measurement tools, and notices from the IDS cluster to craft whitelists and blacklists in the ACL table of the switch.

This concept is the crux of several security and DoS protection systems such as Radware’s SDN-enabled DefenseFlow [101], and SciPass used in the TransPAC network and Indiana University [102]. At Lawrence Berkeley National Laboratory (LBNL) and NCSA, a similar system that whitelists GridFTP traffic, which is uninteresting from a security standpoint in such scientific environments, reportedly reduces the total traffic volume to their security appliance cluster by about 32-37% on a typical day [103].

This architecture results in frequent ACL changes on switches. Using custom

setups that interface with the Bro and Snort IDS, for instance, LBNL and Indiana University block an average of 6,000-7,000 and 500-600 IPs per day, respectively [103], and systems that whitelist GridFTP traffic at LBNL and NCSA result in a few hundred to several tens of thousands ACL operations per day [104].

The traffic which is unclassified is sent to a cluster of security appliances. Such devices usually ship with analyzers for many protocols and applications to detect protocol and application specific attacks. The `weird.bro` and `scan.bro` scripts in Bro, for instance, give notices when Bro observes data being transferred in a session without seeing the SYN ACK packet of that session, data being transferred without observing ACK, repeated SYN ACK packets for the same session, and failed connection attempts to multiple hosts over a time interval. The notices from the IDS are then sent to the controller application which might in turn install rules on the ACL to block IPs. In some systems, such as SciPass, this blocking is by default *permanent* [105]. Erroneous IP blocking is notoriously hard to debug; in most cases it requires the owner of the IP to call the network operator who then manually inspects the IDS logs [106].

However, this system can encounter a problem if the triage switch is replicated. Consider the following setup. The IDS cluster is set to analyze some protocols including TCP port 80, *i.e.*, if it receives a reply, it checks if the reply is solicited or not. If it is, it forwards the packet to its destination. Otherwise, it sends a notice to the controller to block the source of the traffic. A popular web service on the internal network receives a continual stream of incoming requests from clients on port 80.² Let $P1$ refer to the initial policy that TCP port 80 on the switch is unclassified. Next, the network operator chooses to update the policy from policy $P1$ to $P2$ where TCP port 80 is whitelisted. The only affected module is the ACL that should add a rule to forward TCP port 80 to the forwarding table instead of the IDS cluster.

Without replication, at any point during the update, if a server receives a request, it is allowed to reply: its solicited reply either traverses the forwarding table and reaches its destination, or through the IDS that already knows about the request—the request can only be forwarded to its destination by the IDS after the IDS observes the request.

With simple replication, however, the switch might be implemented using rules across multiple physical devices. For example, in a one-big-switch setup

²Similar problems arise if the service is external and the client is internal.

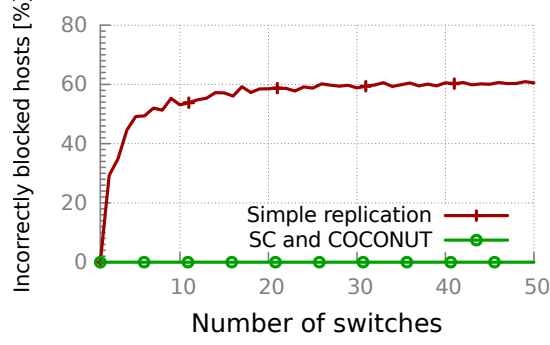


Figure 3.3: Simple replication causes incorrect blocking.

with OVX [5], the single rule that sends TCP port 80 traffic to the forwarding table is now translated into multiple rules, one residing on each physical edge switch that acts as part of the one-big-switch. These rules cannot be installed all at once.

Hence, the following race condition can happen: The new rule for $P2$ is installed at the edge switch connected to host A . Then, A sends a request to host B . The request is directly forwarded to B ; therefore, the IDS does not observe the request. B replies, and its reply hits a *different* edge switch which still uses policy $P1$. Thus, B 's reply is forwarded to the IDS. Since the IDS never saw the request, the IDS sends (false) notices to the controller informing it that the server is sending a stream of unsolicited replies. This will eventually cause the controller to block the server even though the traffic it is sending is already whitelisted and it is legitimately replying to requests it receives. In other words, the hosts observe the following invalid sequence of events: A sends a request, B receives it and replies, B is blacklisted.

This problem is troublesome to resolve. Even though the controller knows a certain type of traffic was whitelisted, it is difficult for the controller to realize the mistake, because a host with some valid traffic might still have sent malicious traffic as well. If the server owner realizes a mistake and phones the network operator, the problem would be hard to resolve as the IDS logs indicate a suspicious server activity (sending unsolicited replies).

To determine how frequent this error can be, we implemented a tree topology with up to 50 leaf switches acting as the logical ACL. Each leaf switch is connected to 5 hosts in Mininet. Each host sends requests to randomly selected hosts with flow interarrival times and sizes drawn from the web-server workload information in [14]. Control delays are drawn from the measurements of HP Procurve

switches in [107]. Figure 3.3 shows the percentage of hosts incorrectly blocked following a single $P1 \rightarrow P2$ policy change, averaged over 100 trials. The percentage of incorrectly blocked hosts rapidly increases with scale, *e.g.*, with a medium-sized network of 20 switches, it approaches 60%. An alternative approach of using symmetric paths for all related flows in that example imposes great overhead for some applications such as GridFTP, used in both NCSA and LBNL [103], that depend on many flows.

3.3.2 Example 2: Logical Firewall

Imagine that an enterprise network has a firewall at the periphery of its network that permits an external server to talk to an internal client if and only if the client has sent a request to the server. This policy could be achieved as follows, using a single switch and a firewall application *FW* running on the controller (Figure 3.4). Initially, *FW* installs in the switch a low priority flow table entry that matches all client and server traffic and sends the packet to the controller. When *FW* receives a packet from a client, it instructs the switch to do three things: (1) install rules to allow bidirectional communication between the client and the server, bypassing the controller, (2) wait for these rules to take effect, via a `BarrierRequest` message, and (3) process the original packet again using the new rules. When *FW* receives a packet from a server, it must have been unsolicited, so it blacklists the server by installing a permanent high priority rule that drops packets from the server. This rule provides the desired property of safeguarding clients from connecting to malicious servers, even if the client tries to connect.

With simple replication, *i.e.*, if that logical switch is in reality mapped to more than one physical switch, the client-to-server traffic could traverse one physical switch, s_1 , and the resulting server-to-client traffic traverses a different physical switch s_2 . In this case, the response traffic may reach s_2 before the rules for bidirectional communication are installed on it, intuitively because the `BarrierRequest` now waits for rules to take effect at only one switch, rather than all. The packet, therefore, will be handled by the default rule which sends it to the firewall application, which proceeds to install a high priority rule *D* to block all traffic for that flow—an undesirable outcome and something that would not happen without replication. Note that even when the rules that allow client-server communication are installed on s_2 , the switch continues dropping traffic

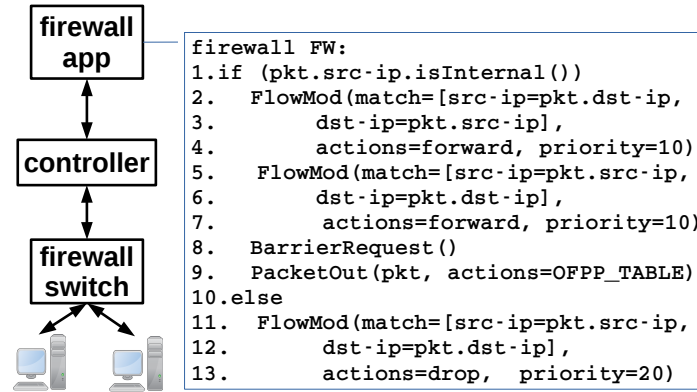


Figure 3.4: Replicated firewall incorrectly blocks communication.

due to rule D , since it has a higher priority. In an experiment with similar setup as §3.3.1, when the client and server are connected to two separate replicas, we found the communication is incorrectly blocked 21% of the time. This example is similar to the previous IDS example in its effect, but here it is triggered by normal client-server traffic rather than an administrator’s policy change.

3.3.3 Example 3: Logical Load Balancer

Server load balancers (SLBs), that distribute incoming traffic among available servers, are fundamental to create scale-out web services in public clouds; they serve almost the entire inter- and half of intra-datacenter traffic [12]. Web services have uptime SLA as high as 99.9 to 99.999 [12, 13]; therefore load balancers’ uptime has to be at least as high but often significantly higher to account for failures in other parts of the infrastructure [12].

Cloud services put huge pressure on SLBs: inbound traffic, where every packet needs to hit the SLB, could be greater than 100 Gbps for each single service [12]. Plus, by enabling convenient deployment, scaling, deleting, and migration of service, the pay-as-you-go model imposes a high rate of configuration changes on SLBs: an average of 12,000 changes per day peaking at one per second for a cluster of 1,000 servers [12].

Given the high and rapidly changing load that SLBs handle, it is perhaps no

surprise that they dominate in terms of failure occurrences in datacenters [32] and could cause high rates of SLA violations and failures accounting for 37% of all live site incidents [12]. While modern SLBs such as Microsoft’s Ananta [12] or Google Maglev [13] offer significant improvement over traditional SLBs, scaling out and failure handling remain challenging and cause connection drops even in these modern designs.

We give a brief overview of such systems focusing on Ananta, and explain how inconsistent replica state could cause connection drops (as also reported in [12]). In §3.5, we show how COCONUT could reduce the rate of such drops in those systems and enable their seamless scale-out—a challenging task today that can cause excessive connection drops—while strictly avoiding the performance penalties, such as increased latency, of an alternative approach.

Anatomy of a modern SLB: In a SLB, one or more high performance routers spread the incoming traffic to the datacenter among a set of SLB replicas³, using the stateless and fast Equal Cost Multi-path (ECMP) algorithm. These replicas spread traffic among the set of currently available servers by hashing packet headers, similar to ECMP⁴. Unlike ECMP, however, they have to save connection state. Keeping per-connection state in replicas is essential for maintaining high uptime due to the dynamic nature of the cloud (ever changing set of servers that a service deploys). Once a replica selects a server for an incoming connection, it remembers that decision in a flow table. Every non-SYN TCP packet is first matched against this table, and if no match is found the packet is treated as the first packet of the connection; a server will be selected and the decision will be remembered in the table⁵. These operations help preserve connection affinity and consequently reduce the rate of connection drops when the set of servers changes *if the set of SLB replicas is static*.

In reality, however, this set changes due to replica failures; in fact, SLBs are among the most failure prone devices in datacenters [12, 32]. Plus, in environments as dynamic as public clouds with rapidly varying demands, elasticity of different resources including the SLBs is indeed a desirable property. It enables the providers to scale them out whenever they are overloaded. Overloading SLBs is conceivably one of the main culprits of low availability with 50% of the low

³Called Mux pool in Ananta [12].

⁴In addition to load balancing, these systems also offer capabilities such as NAT and direct server return. For simplicity, we focus exclusively on the load balancing functionality in this example.

⁵The notion of pseudo connections is used for protocols other than TCP such as UDP [12].

availability conditions in the measured time period (January 21 to 26) being reportedly caused by SLB overload [12].

Alas, changes in the SLB set today can cause connection drops, *even when all other parts of the system, including the server, links, and all replicas serving the connection, are all up*: when the set of SLBs change, *e.g.*, due to failures, ongoing connections will be redistributed among the live replicas. This happens because high-end routers, with standard (stateless) implementation of ECMP, are used to efficiently spread the incoming traffic among available SLB replicas [12]. Therefore, any change in the set of replicas causes rehashing and redistribution for incoming traffic, and may result in an ongoing connection to be assigned to a different replica even when the original replica is functional. Different replicas may have inconsistent state since the set of servers constantly changes and in an asynchronous distributed systems, different replicas learn and react to these changes at different times. This means that the connections that relied on the flow state on another replica could now be directed to a different server if there has been a change in the mapping entry since the connection started [12]. Such connections will be dropped.

3.3.4 Shortcomings of Existing Approaches

Per-packet Consistency Is Not Enough. A line of work has preserved properties of a single packet’s journey, even during network updates — for example, avoiding loops and black holes [45, 47] or preserving *per-packet (or per-flow) consistency*, wherein every packet (or flow) traversing the network is processed by exactly one global network configuration and never by a mix of multiple configurations [46]. These properties do not achieve seamless scale-out, because they do not preserve dependencies *across* different packets or flows. In the IDS example, each source-to-destination flow was processed by only a single policy; the problems are only visible *across* flows, violating the request/reply orderings that the IDS policy depends on. This is a critical distinction, because Consistent Updates (CU) [46] implements per-packet consistency by relying on each packet’s entry point as a single point of atomic update. Seamless scale-out involves behavior of packets across many flows from multiple entry points and potentially flowing through endhosts; no atomic update is possible.

In fact, application-level incorrectness can occur *because* of deploying CU [46] to guarantee per-packet consistency. This is because the two-phase update algo-

rithm of CU itself causes replication. In the example above, in the non-replicated setting, if the network uses CU to update the policies from TCP port 80 traffic to whitelisted, the update will no longer be a single step, because the flows using rule $R1$ on the ACL need to be updated one by one. Suppose that the flow from A to B is updated, but other flows (including the one from B to A) are not still updated. In this case, the ACL will have 2 rules corresponding to $R1$ on the ACL in Figure 3.2 (not shown): an old rule to match traffic using old tags (old policy traffic) and the new rule with new tags (new policy traffic). Now, host A sends TCP port 80 traffic with the new tag, which is forwarded to B (new policy). B receives the packet, and replies. Its reply to A , however, is delivered to the IDS since it has the old tag. The IDS consequently blocks B given that it has not seen the request, something that would not happen if CU was not being used. The underlying problem in this case is that CU maps a single logical rule to multiple physical rules with different tags.

Strong Consistency Is Cost Prohibitive. SDN switches do not directly provide primitives to preserve strong consistency, but one can implement it using the controller [11, 34, 108]: when a rule needs to be updated, direct all related flows to the controller, which temporarily emulates the switches' behavior; perform an atomic rule update at the controller; update the switches; and finally shift traffic back to the switches. This technique would correctly achieve seamless scale-out. But we show in §3.5 that it has dramatic performance penalties, *e.g.*, $20\times$ increase in delay for the IDS example. Shifting traffic to the controller is thus appropriate for relatively rare virtual network migration events supported by [11, 34, 108] but not for the ongoing process which we hope to support.

3.4 Design of COCONUT

The previous examples demonstrate that simple replication does not provide seamless scale-out. In this section, we begin by presenting the intuition of what logical property the network requires to achieve seamless scale out. We call this property weak causal correctness, formalize it (§3.4.1), demonstrate the intuition behind our design with simple (but impractical) algorithms to preserve weak causal correctness (§3.4.2), and finally present a practical realization of the design (§3.4.3).

3.4.1 Not All Orderings Are Created Equal

Causality violations in §3.3, *e.g.*, receiving a response before or without the request that caused the response, are caused by inconsistent state among replicas of one single logical rule—a packet is handled by a new instance of a logical rule and another packet that “comes after” it is handled by an old instance of the same logical rule. In the IDS example in §3.3.1, for instance, the request packet is handled by a new instance of the ACL rule, but the reply that it triggers is handled by the old instance of the same logical rule. As a result, the IDS receives the reply packet first, missing its dependency (the request).

On the surface, it might seem counter-intuitive that the ordering between those packets is a problem that could compromise application correctness, since even in non-replicated best-effort networks, packets can be reordered or dropped. The subtlety here is that even in best-effort networks some orderings, that we call weak causality, are always preserved. For example, no amount of reordering or packet loss will change the fact that with a standard TCP implementation, receiving a SYN packet always happens before sending the first SYN ACK.

We use this intuition to formally define weak causality and observational correctness. We first formalize network events and define networks’ behaviors.

The endpoints interact with the network with *send* and *receive events*. These are the only events we are ultimately interested in because they are the only *externally visible* events, *i.e.*, while the network could have multiple internal events such as rule lookup, packet rewrite, *etc.*, those internal events are not visible to the endpoints. The distinction between internal and external events is a common technique for defining correct behavior of state machines [109]. The notation $r_{h,i}(pkt)$ and $s_{h,i}(pkt)$ are used to respectively refer to the event of receiving and sending packet pkt by endpoint p_h where this event is the i th event happening at p_h . Each sequence of external events is a *trace*. The *behavior* of a system is the set of all plausible traces in that system [109]. In a system with n endpoints, $p_i \in \{0, \dots, n-1\}$, a *local history* of endpoint p_i , denoted by L_i , is a sequence of $e_{i,j}$ s, the external events that happen at p_i , *i.e.*, the system’s behavior observable by p_i . A *history* $H = \langle L_0, L_2, \dots, L_{n-1} \rangle$ is a collection of local histories, one for each endpoint.

Observational correctness: For a physical network, P , to be an observationally-correct implementation of a logical or abstract network, L , any trace in P ’s history should be a plausible trace in the history of an ideal, non-replicated implementation of L . That is, the possible behavior of P is a subset

of the possible behavior of a non-replicated implementation of L . We see in §3.3 that this condition does not hold under simple replication, *e.g.*, the following trace that happens in the replicated network in the example in §3.3.1 is not plausible in the non-replicated networks: *A sends a request, B receives the request, B sends a reply, IDS receives the reply.* (*i.e.*, the trace misses the event of the IDS receiving the request that triggers the reply).

Weak causality: Event $e_{k,l}$ has *weak causal dependency* on event $e_{i,j}$, shown by $e_{i,j} \rightarrow e_{k,l}$, if one of the following cases hold:

R1: local dependencies. This applies when $i=k$ (*i.e.*, both events happen in the same endpoint), $j < l$ (*i.e.*, $e_{i,j}$ comes before $e_{k,l}$), and $e_{k,l}$ is a *send* event. Note that we replace the traditional “program order” [110] with local dependencies in rule $R1$. This is done to account for the fact that a best-effort network can reorder packets. The above condition on $e_{k,l}$ is what distinguishes our notion of weak causality from the original definition of causality in [110].

R2: sends-to. $e_{i,j}$ and $e_{k,l}$ are respectively the events of sending and receiving the same packet.

R3: transitivity. there is some other $e_{r,t}$ event such that $e_{i,j} \rightarrow e_{r,t} \rightarrow e_{k,l}$.

If an event $e_{i,j}(p)$ involving packet p has weak causal dependency on an event $e_{k,l}(q)$ involving packet q , we say that p has weak causal dependency on q , denoted by $p \rightarrow q$. Events and packets with no weak causal dependencies are called *concurrent*.

While best-effort networks can drop packets and reorder concurrent packets, they preserve weak causality. For instance, if concurrent packets $pkt1$ and $pkt2$ are sent to endpoint p_i , receiving them with any order or not receiving one or both of them are permissible, *e.g.*, \emptyset , $\langle r_{i,j}(pkt1) \rangle$, and $\langle r_{i,j}(pkt2), r_{i,j+1}(pkt1) \rangle$ are plausible traces. However, a host always receives a SYN ACK packet after sending a SYN packet (its weak causal dependency). Receiving a SYN ACK without sending a SYN packet, or receiving it before sending a SYN packet, therefore, are not plausible traces.

Unlike non-replicated networks, replicated ones can violate weak causality, *e.g.*, the IDS in §3.3.1 receives a reply while missing its dependency. This implies that replicated networks can have traces (those that violate weak causality) that are not plausible in logical networks that they intend to implement, and consequently are not correct.

Root cause of weak causality violation: It is not hard to see that if no rule changes, then any trace in the replicated network is a plausible trace of the logi-

cal network (§B). The fact that a replicated network can have implausible traces, therefore, results from handling packets with inconsistent instances of rules. Intuitively, handling concurrent packets with inconsistent instances does not result in a implausible trace. Even in non-replicated networks, it is permissible to handle two concurrent packets with inconsistent state while the network state is changing. The problem happens when orderings of packets are known, *e.g.*, $p \rightarrow q$. In non-replicated networks, p cannot be handled by a newer state compare to q . Under simple replication, in contrast, this property does not automatically hold because the instances handling p and q could be different. Therefore, p might be handled by a newer state compared to q . In the IDS example, for instance, the event of the IDS receiving the reply (e_3) happens after the event of B sending the reply (e_2), which in turn happens after the event of B receiving the request (e_1). Yet, even though $e_1 \rightarrow e_3$, the packet associated with e_1 (request) is handled by a newer instance compared to the instance that handles the packet associated with e_3 (reply). We provide algorithms to ensure that with COCONUT's replication, for any two packets p and q where $p \rightarrow q$, applying a logical rule on q implies that no newer version of the same logical rule is applied on p . We show in §B that preserving this property is sufficient for observational correctness:

Theorem 1: Any behavior of COCONUT's implementation of replicated networks could have happened in the logical network.

The intuition behind the proof is to show that COCONUT is weak causality-aware⁶ (Lemma 2 in §B) and this is sufficient for observational correctness (Theorem 1 in §B).

3.4.2 COCONUT's High-level Algorithms

In an implementation of a logical network with m logical rules, $LR_{0 \leq i < m}$, one single logical rule, LR_i , is mapped to multiple physical instances, $PR_{i,j}$, where j is the ID of the switch hosting the $PR_{i,j}$ instance.

Changes to a logical rule should be replicated across all the physical rules that implement it. Without enduring the prohibitive cost of synchronization for atomically updating all the physical rules at once and in unreliable networks where elements can fail, inevitably, there exist instances when different physical replicas are in different and inconsistent states. Fortunately, this different network state

⁶A network is *weak causality aware* iff for any two packets p and q and for any logical rule R , $p \rightarrow q$ implies that the version of R that handles q is at least as large as the one that handles p .

usually does not cause anomalous application behaviors — unless endpoints’ applications receive packets from the network, they are unaware of the network state. The problem happens when the packet is handled by a new version of the rule and then triggers a causal sequence of events leading to some packet (perhaps the same or newly generated packet) being handled by an old version of the rule.

We leverage this observation and the classical concepts of logical and vector clocks to prevent such weak causality violations. We use *logical clocks for tracking network state changes and restricting the space of executions to those that are weakly causally consistent*. Endpoints affix vectors of logical clocks to packets that show their latest observed network state. These clocks prohibit switches from applying outdated rules that might violate weak causal correctness, and prompt them to update their rules before applying them to packets.

More specifically, in a network with m logical rules, each packet pkt carries an m -dimensional vector of logical clocks, VC_{pkt} , in which $VC_{pkt}[j]$ shows the latest version number of logical rule LR_j that pkt has “observed”—that is, the latest version known at the sender of pkt when it was sent, or the version applied to pkt along its path (whichever is more recent). As an example, the switch that handles a packet p with the second version of the logical rule LR_j sets its $VC_p[j]=2$, and the endpoint that receives p sets $VC_q[j]=2$ for a packet q that it sends after receiving p . We assume that switches are preloaded with all versions of rules, similar to the way that OpenFlow switches can be preloaded with failover rules.

Algorithm 1 Ideal Switch sw

```

1: procedure UPDATE(rule  $PR_{i,sw}$ )
2:    $VC_{sw}[i]++$ 
3:   regular-update( $PR_{i,sw}$ )
4: end procedure
5: procedure RECEIVE(packet  $pkt$ , port  $ip$ )
6:   rule  $PR_{i,sw} = \text{lookup}(pkt, ip)$ 
7:   while ( $VC_{sw}[i] < VC_{pkt}[i]$ ) do
8:     update( $PR_{i,sw}$ )
9:   end while
10:   $VC_{pkt}[i] := \max(VC_{sw}[i], VC_{pkt}[i])$ 
11:  regular-apply( $PR_{i,sw}, pkt, ip$ )
12: end procedure

```

The reader will have already realized that in large-scale multi-tenant datacenters hosting 10Ks of virtual networks [111, 112], storing a clock value for every rule in every packet, performing operations on these VC s, and preloading switches

with all rules are infeasible. Our goal in this section is to convey the intuition behind our design and reason about its correctness. Later, §3.4.3 presents a scalable and OpenFlow-compatible, but slightly more complex, emulation of these algorithms. Three types of entities—switches, shells, and the controller—work with the vector clocks carried by packets. We describe the role of each next.

Switch operations: Each physical switch sw has a logical clock $VC_{sw}[j]$ for each logical rule $PR_{j,sw}$ hosted at the switch. This clock stores the current version number of the rule that the switch will apply to matched packets. Note that one logical rule can be hosted at multiple physical switches, and these may have different clock values while the rule is being updated. When a switch needs to update a rule, it also increments its corresponding logical clock (procedure `UPDATE`; Algorithm 1; `regular-update` is the regular rule update operation without COCONUT).

When receiving a packet pkt on input port ip (procedure `RECEIVE`; Algorithm 1), the switch sw looks up the rule that needs to be applied on the packet, $PR_{i,sw}$. If $VC_{pkt}[i] > VC_{sw}[i]$, the packet or a packet that *happened before* was already handled by a newer version of LR_i than the one currently active on sw . Hence, applying the outdated version risks weak causality violations once pkt is received by any endpoints. So at this point, sw is required to update the rule before handling pkt . The `update($PR_{i,sw}$)` function has the switch update $PR_{i,sw}$ using the preloaded rules, its clock for this rule, and the packet’s clock for this rule, $VC_{pkt}[i]$, to show the latest version number. Finally the switch acts on pkt by applying the rule (line 11; Algorithm 1).

Deleting a rule $PR_{j,sw}$ is a special case of updating it: the logical clock of the deleted rule, $VC_{sw}[j]$, is incremented and its value is set to \emptyset (a special value) dictating sw to apply other rules for matching packets.

Controller’s operations: The *controller* sits between the network hypervisor and the network, and is tasked with installing the physical rules, such as those sent by the network hypervisor to it, on switches.

Shell’s operations: A *shell* is a shim layer sitting between each endpoint and the network, which can run in the hypervisor. Shells hide VC s from the endpoints by performing the necessary logical clock operations on their behalf. For each endpoint p_i , its shell $shell_i$ keeps an m -dimensional vector VC_i of logical clocks. $VC_i[j]$ contains the *max* version number of logical rule j observed in the logical clock of any packet p_i has received.

For each incoming packet, pkt , $shell_i$ updates VC_i if the packet carries any

Algorithm 2 $Shell_i$

```
1: procedure RECEIVE(packet  $pkt$ )
2:   for  $j \in VC_i$  do
3:     if  $VC_{pkt}[j] > VC_i[j]$  then
4:        $VC_i[j] := VC_{pkt}[j]$ 
5:     end if
6:   end for
7:   remove-VC( $pkt$ )
8:   regular-fwd-to-host( $pkt$ )
9: end procedure
10: procedure SEND(packet  $pkt$ )
11:   add-VC( $pkt, VC_i$ )
12:   regular-send-to-net( $pkt$ )
13: end procedure
```

newer information, *i.e.*, $\forall j, VC_i[j] = \max(VC_i[j], VC_{pkt}[j])$. It then removes VC_{pkt} from the packet before passing it to the endpoint (procedure RECEIVE in Algorithm 2). For any outgoing packet pkt , $shell_i$ appends its local VC , VC_i , to the packet before sending pkt (procedure SEND in Algorithm 2). This VC prevents switches from handling pkt with outdated rules that could violate weak causality.

3.4.3 OpenFlow-compatible Implementation

Having a scalable implementation of the simple algorithms in §3.4.2 is challenging. A major scalability bottleneck is the size of the time vectors. In general, in a distributed computation with N processes, causality can only be characterized by vector timestamps of size N , *i.e.*, the causal order has in general dimension N [113]. For implementing weak-causally consistent SDNs, where the vector timestamp tracks the version of every forwarding rule in the network, it would be overly burdensome (in terms of bandwidth and CPU) for packets to carry such large vectors and endpoints, switches, and controllers to operate on them. Another scalability challenge is preloading switches with all versions of rules. In addition to these *scalability* challenges, there is a *feasibility* challenge: vector clocks and their related operations cannot be readily implemented with the match/action operations on commodity switches today.

To overcome the feasibility challenge, we note that the weak causality problem that VCs solve only arises when a logical rule is *in flux*: there are both old and new

physical instances of the rule in the network. Vector operations are not needed for *stable* rules that are not in flux (*i.e.*, before or after updates). Even when rules are in flux, their exact version numbers are not necessary for preserving weak causality. As long as the old versions of a rule are eliminated from the network, it is sufficient to know that the rule is being updated which can be sufficiently characterized by one single bit, which we call a *tag bit* (TB), to identify the current and new versions. Switches and endpoints then need to “mark” the TBs of the packets that are handled by such rules or any packet after them (by a tagging operation which can be implemented in existing switch hardware), and for in-flux rules, switches need to apply their updated versions to the tagged packets (*e.g.*, by having the updated rules as higher priority rules that match on the tag). These simple tricks enable us to emulate vector operations for updating a logical rule by reserving a TB for it and deploying regular match-action operations, thus solving the feasibility challenge. Concurrent updates could use separate update TBs.

The fact that only the in-flux rules require tags for correct operations, along with coordination at the SDN controller, also aids us to sidestep the scalability challenge: once an update operation terminates, *i.e.*, once the controller learns that all the physical instances of a logical rule LR are updated, it can re-use its TB for updating other rules. We can thus concurrently update as many logical rules as the number of bits dedicated to TBs. While this is likely to be sufficient for a single virtual network, it will still be a scalability bottleneck for cloud providers that host $10K$ s of virtual networks and should support millions of concurrent updates of all of these networks per day [111, 112]. We resolve this by capitalizing on the fact that virtually all network virtualization platforms [6, 5, 114, 43] isolate traffic within each virtual network, so that traffic cannot leak between two virtual networks. Packets carrying extra bits disjoint from the bits used by the hypervisor and rules matching on them do not violate this property. Hence, multiple virtual networks can concurrently use the same TBs. Furthermore, the controller can preload switches with only the necessary rules.

We describe the practical implementation of COCONUT’s algorithms as well as its failover operations after explaining the notations and requirements.

Requirements: In addition to requiring traffic isolation between virtual networks, COCONUT requires that the TB bits are dedicated to COCONUT’s operations, *i.e.*, no other entity (such as the tenants or the network hypervisor) is allowed to use these bits. For simpler presentation, we further assume that arbitrary bitmask (supported since OpenFlow 1.1, early 2011) is supported for the

header-field used for TBs. Note that this is not a fundamental requirement; algorithms that emulate the §3.4.2 algorithms using only longest prefix match rules are presented in [115]. COCONUT requires that the network hypervisor should not cause *ambiguity*, *i.e.*, it should not install multiple rules with overlapping match fields and identical priority on a switch. Moreover, assuming that by default rule priorities are integer values between 0 and max-priority, COCONUT requires the priorities of the physical rules that the network hypervisor sends to the controller to be integers between 0 and $\lfloor (\text{max-priority})/2 \rfloor$, *i.e.*, COCONUT uses half the priority-space to “pre-install” rules to accelerate the update process without causing ambiguity. As we will see, for any rule P with priority x , the priority of the stable rule that COCONUT eventually installs is $2x$ and the priority of the pre-installed rules for P is $2x + 1$. This implies that for any two rules P and L , where $x = P.\text{priority}$ and $y = L.\text{priority}$, if $y \geq x + 1$, then L ’s priorities ($2y$ and $2y + 1$) will be strictly larger than P ’s priorities ($2x$ and $2x + 1$) throughout.

Algorithms: For updating a set of physical rules corresponding to a logical rule of a virtual network $v\text{-net}$, the network hypervisor sends a set called the *rule-batch*, the identifier of $v\text{-net}$, and the identifiers of the $v\text{-net}$ ’s shells to the controller (arguments of the `UPDATE` procedure in Algorithm 3). Each element of the rule-batch set, b , is a tuple that includes the new rule that needs to be installed $b.\text{new-rule}$, and the old rule that is being replaced, $b.\text{old-rule}$. Also, $\text{rule-batch}.\text{new-rules}$ and $\text{rule-batch}.\text{old-rules}$ show, respectively, the set of all new and old rules in the *rule-batch*. For any given physical rule, R , we denote the match, action, priority, and the switch hosting R by, respectively, $R.\text{match}$, $R.\text{action}$, $R.\text{priority}$, and $R.\text{sw}$. We show the action of installing a set of rules SR by `install(SR)`, the action of updating SR by overwriting value val on the var header field by `update(SR, var, val)`. For instance, updating the priority values of all rules in SR to 10 is shown by `update(SR, priority, 10)`.

Algorithm 3 starts by installing a set of temporary rules T that are identical to the new rules, except: (1) they have higher priorities; (2) they match on an unused TB, $\text{tag}=1$, in addition to the rules’ existing match requirements; and (3) the action sets $\text{tag}=1$ in addition to the rules’ existing actions (line 10, Algorithm 3). Note that a single tag bit is used for all rules in the batch. The temporary rules T will gradually be updated and eventually turn into the new rules. Initially, these rules are invisible because no transmitted packets have $\text{tag}=1$. But once packets do start using the new tag (*i.e.*, the rules’ increased virtual clock value), the switches are prepared and thus will not have to pay the expensive [116] cost

of relaying packets to the controller while the new rule is “paged in”. Specifically, since the rules have higher priority than the old rules, if a packet matches both a T and an old rule, the action of the new rule will be applied on it.

Once confirmations are received, the T rules are updated not to need the $TB=1$ for matching packets. This makes the update visible as endpoints now can receive packets matched and handled by these rules. After receiving the confirmations (`wait-conf(T)`), every instance of the rule is ready to handle packets with or without TBs. So packets do not need to be marked any longer and the old rules can be deleted, since higher priority rules are already installed (line 12).

After receiving confirmation that the old rules are deleted, the priorities of T rules are converted into the stable value (line 16). Note that this operation turns the T rules into the stable new rules. Finally, once the controller receives the confirmations from the shells that they no longer tag packets with the TB and switches have installed the new non-tagging rules, it can release the tag for v -net after waiting for the *flush time*, the time for in-flight packets and the buffered packets (that might be tagged) to be delivered or expired and dropped (line 18). Algorithm 3 is

Algorithm 3 Controller Update Algorithm

```

1: procedure UPDATE(set rule-batch, set shells, id v-net)
2:    $TB\ tag := \text{get-tag}(v\text{-net})$ 
3:   map  $T$ 
4:   for  $b \in \text{batch}$  do
5:      $T[b] := b.\text{new-rule}$ 
6:      $T[b].\text{match} := (T[b].\text{mtach}) \& (tag = 1)$ 
7:      $T[b].\text{priority} := 2 \times T[b].\text{priority} + 1$ 
8:      $T[b].\text{action} := (tag = 1) \& (T[b].\text{action})$ 
9:   end for
10:  install( $T$ )
11:  wait-conf( $T$ ); update( $T$ , match,  $T.\text{match} \& (tag = *)$ )
12:  wait-conf( $T$ ); update( $T$ , action,  $T.\text{action} \& (tag = *)$ )
13:  delete(rule-batch.old-rule)
14:  wait-conf( $T$ ); stop-tagging(shells, tag)
15:  wait-conf(rule-batch.old-rule)
16:  update( $T$ , priority,  $T.\text{priority}-1$ )
17:  wait-conf(shells); wait-conf( $T$ )
18:  release-tag(v-net, tag)
19: end procedure

```

for updating rules. Algorithms for deleting and adding new rules are similar: for **deleting** a set of rules DR , we set *rule-batch.new-rule* and *rule-batch.old-rule*,

respectively, to the set of rules that should match packets after DR 's deletion, and DR . The deletion procedure is identical to the update procedure except for line 16, where instead of updating the priorities of T , T is deleted since switches already host the rules that should match packets after DR is removed with their correct priorities. For **adding** a set of new rules, $rule\text{-}batch.old\text{-}rule=\emptyset$, and the deletion of old rules (in line 13) and waiting for its confirmation should be skipped (see details in [115]).

Shell operations are identical to the operations explained in §3.4.2 except that each shell i keeps a VC_i for the TB bits tag (and not all the logical rules), shown with $VC_i[tag]$, a timer associated with each TB bit, shown with $timer(tag)$. If the shell receives a `stop-tagging(TB tag)` command from the controller, it sets $VC_i[tag]=0$, resets tag 's timer, *i.e.*, $timer(tag)=0$, and sends a confirmation to the controller. Shells honor the `stop-tagging(TB tag)` commands for the flushtime. If shell i receives a packet with $tag=1$ after the flush timer for tag has elapsed, it assumes it to be related to a different update batch and sets $VC_i[tag]=1$.

Handling Failures: We assume that different components of the system might experience **crash failure**, but not Byzantine failure. We further assume that each endpoint and its shell share fate, *i.e.*, they fail together. Switches and the controller are assumed to have reliable channels between them, similar to the main control channel in OpenFlow. Updates related to failed links are carried out similar to regular updates. Non-responsive switches (those not reacting to controller commands within a threshold) are assumed to have failed. When a switch fails, other switches and endpoints connected to it are populated with *detour* rules to reroute the traffic originally sent to the failed switch, and drop traffic they receive from it (*failover operations*). Dropping this traffic is essential for preserving safety; if the controller loses control over a switch, the switch's behavior, *e.g.*, its tagging operations, will be unknown. When a failed switch recovers, it communicates with the controller which populates it with correct version of rules (including the possible transient rules) before undoing the failover operations, *i.e.*, removing the rules that drop the traffic received from the failed switch from the network and endpoints as well as deleting the detour rules.

3.5 Evaluation of Prototype

We implemented a prototype of COCONUT (§3.5.1) and evaluated it in both a hardware SDN testbed and a Mininet emulation with multiple SDN applications and workloads (§3.5.2). We compared COCONUT’s performance with a number of baselines: simple replication (SR), a strawman solution which provides strong consistency (SC), and CU. In summary, we found SC to be cost prohibitive, *e.g.*, even in modest-sized networks, it causes 12 Gbps bandwidth overhead and a $20\times$ increase in user traffic latency, COCONUT, CU, and SR⁷ incur no measurable data plane performance overhead (§3.5.3). In terms of forwarding rule update delay and rule overhead, COCONUT has significantly lower overhead compared to SC ($3.5\times$ and $2\times$ times lower respectively in a 80-switch network), and CU ($1.5\times$ and $245\times$, respectively). This overhead is only $1.2\times$ and $1.3\times$, respectively, higher than SR (§3.5.4). Moreover, COCONUT’s extra temporary rules are likely to be evacuated from the network faster (§3.5.5). This result should be expected: switch update time is known to vary significantly [117, 118], with the 99th percentile 10 times larger than the median in some cases [117]. Thus, by updating much fewer switches, COCONUT runs a lower risk of encountering stragglers. In some cases, the application developer can prevent replication-related race conditions by rewriting her applications to take the network replication into account. We show that in addition to offering programming simplicity, *i.e.*, enabling developers to use their applications “as-is”, COCONUT’s efficient logical clock-based approach of tracking causality results in $2.8\times$ lower mean latency for user data flow initiation compared to this approach (§3.5.3). We give more details about each of these conclusions next.

3.5.1 Prototype Implementation

Our COCONUT prototype consists of approximately 4K lines of Java and python code and integrates a number of third party libraries and tools. In our prototype, the controller is implemented using the Floodlight platform [50]. Floodlight runs a series of modules (*e.g.*, user applications), and each module is supplied with mechanisms to control and query an SDN network. The COCONUT controller is implemented as a layer (which is itself a module) residing between the Floodlight Virtual Switch, a simple network virtualization developed as a Floodlight application, and the controller platform. Our prototype exposes much the same interface

⁷Note that CU and SR do not guarantee observational correctness.

as the Floodlight platform. Hence, modules such as the virtualization applications that wish to be Floodlight clients simply use its interface instead. The COCONUT controller instruments the rules received from client modules and coordinates with shells to maintain correctness. We use OVS [43] to implement shells at the hosts with a bridge through which passes all traffic between the network and hosts.

3.5.2 Experimental Setup

Environment: For the physical network, we use a hardware testbed which includes 13 Pica8 SDN Pronto 3290 switches, having a total of 676 switch ports. We “sliced” these ports to emulate fat-tree topologies with various sizes (up to 20 switches). To test COCONUT at scale, we also use the `Mininet` emulator [51] and implement fat-tree [119] and VL2 [15] topologies with a few hundred switches in it. Switches’ delays to apply and confirm application of updates (hereafter called *control delay*) are drawn from [107] in which the authors measure the performance of several commercial switches (HP Procurve, Fulcrum, and Quanta). We emulate the behavior of the HP Procurve switches in our Mininet experiments. We draw job allocation, flow interarrival times, and flow sizes from [14, 96].

Controller & Applications: We used two network virtualization platforms, OpenVirtex [5] and Pyretic [37] to create one-big-switch abstractions over physical fat-tree [119] and VL2 [15] networks of various sizes. Tenants of the network use several canonical applications to insert and update rules on their virtual one-big-switches. For OpenVirtex, the tenant runs the Floodlight controller [50] and its existing applications such as the learning switch and firewall, as well as the applications explained in §3.3. When these applications install, remove, or update a rule on the one-big-switch, OpenVirtex translates that to possibly multiple `FlowMod` messages and sends them to the physical network. For Pyretic, we use the parallel composition of the firewall and MAC learning implementations provided in [37]. The graphs in this section, unless stated otherwise, show the results for the ACL application running over an OpenVirtex’s one-big-switches over fat-tree networks with parameter $k=\{2,\dots,10\}$, *i.e.*, networks with (2 hosts, 5 switches), (16 hosts, 20 switches) ..., and (432 hosts, 180 switches), and the workload from [14]. Over these one-big-switches, the tenant’s applications redirect a stream of traffic to a different host. These logical rules are then mapped to many

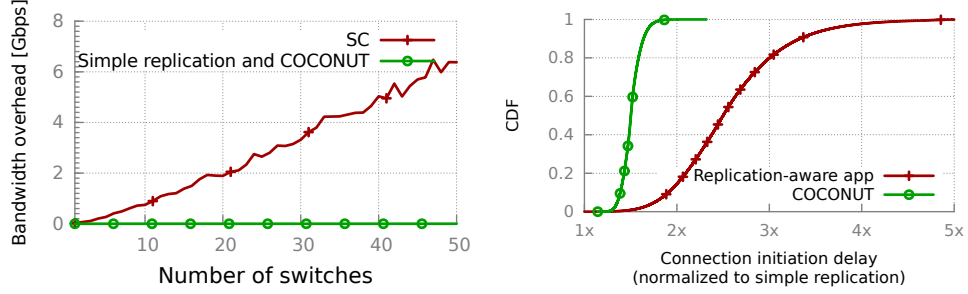


Figure 3.5: (a) SC causes significant bandwidth overhead, (b) Replication-aware app increases delay.

physical rules: one rule for each port that connects to a host. We then update all those rules concurrently. Unless stated otherwise, we observe similar trends for other explained settings.

Scale-out schemes evaluated: COCONUT, SR, SC, CU. In addition to simple replication (SR) as a baseline, we use an implementation of Strong Consistency (SC) in SDNs [34]. For updating a rule, SC first installs temporary tunneling rules to direct all traffic that would be affected by the change to the controller (where it is handled by a single, strongly consistent, version of the logical rule), and from the controller to its destination. It then updates the rule at the controller; next it updates switches with the new rule and tears down the tunnels.

As another comparison point, we implemented a version of consistent updates (CU) which provides per-packet or per-flow consistency (§3.3.3). Of course, CU and COCONUT provide different correctness properties. The goal of this comparison is to evaluate whether COCONUT is expensive relative to the most powerful previously-studied notions of correctness⁸. Note that CU fundamentally operates at the granularity of a *flow*: (a subset of) traffic between ingress and egress switches [117]; it installs rules that are tagged to be specific to that flow. However, the abstraction we work with here operates on *forwarding rules* in a virtual network, and a single rule may apply to multiple flows. To translate CU to this rule-based abstraction, we implemented a module that duplicates rules so each flow using the rule has its own copy. It then runs CU to update each of those flows in parallel.

⁸Recent works on optimizing CU require special rule-formats [120, 117, 47], *e.g.*, each rule is exact-match on a single flow [117]. Such assumptions are more likely to hold in the network core as those rules that violate those constraints are being moved to the network edge in virtualized datacenters [117, 121]. Thus, CU remains the most appropriate comparison for our setting.

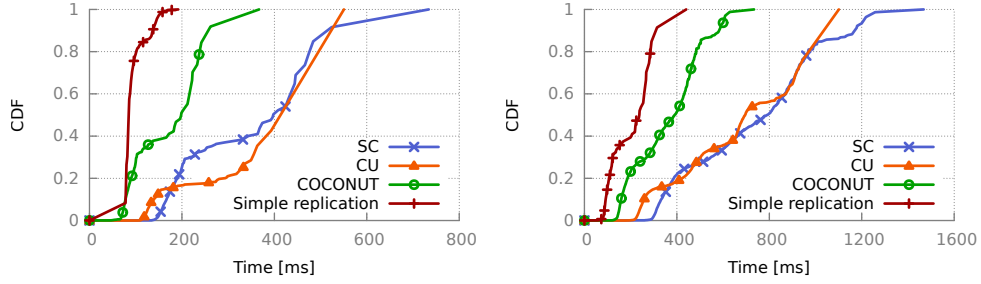


Figure 3.6: Testbed's (a) update initiation & (b) termination delays.

3.5.3 Data Plane Performance Impact

While the correctness problems discussed in §3.3 can be avoided via preserving strong consistency instead of weak causal consistency, doing so comes at a great data plane performance cost. Figure 3.5(a) shows the aggregate bandwidth overhead imposed by SC on the controller, already a bottleneck in SDNs [96, 122, 116, 118, 6], for the IDS example of §3.3.1. In addition to bandwidth overhead, this practice imposes added latency to flows. The overhead is prohibitive and rapidly increases with scale, *e.g.*, for networks of size 100, over 5,000 flows experience an average of $20\times$ increase in latency due to an ACL rule update (not shown). This approach, therefore, is not practical and scalable.

In some cases, the application developer can rewrite her applications to take the network replication into account. In the example of §3.3.2, for instance, if the firewall application developer is aware of the underlying replication, she could ensure correctness by preserving the orderings of installed rules on not just one switch but across *all* replicas, *e.g.*, after receiving a packet from a client, the application could send the rules for allowing bidirectional communication to all replicas, followed by `BarrierRequests` (line 8 in Figure 3.4) and *wait* to receive the `BarrierReplies` from all replicas before sending the packet out. This approach, however, increases the delay of communication. With the previous experimental setup, for instance, more than half of the sessions experience an increase of $2.8\times$ or higher in their connection initiation latency compared to COCONUT. Figure 3.5(b) shows the CDF of connection initiations' delays caused by this approach over 100 runs. In addition to the performance penalty, in this approach, the programmer needs to be aware of the underlying replication and rewrite her applications to account for it. Note that while COCONUT's delay is slightly higher than SR, unlike SR, it prevents incorrect blocking.

3.5.4 How Long Are Updates Delayed?

When network state changes, SC installs tunneling rules to and from the controller; COCONUT and CU start with a phase that installs some initially-invisible rules. These operations cause delay before the change starts to become visible to data traffic (*update initiation delay*), and before all switches have informed the controller their update is complete (what we call *update termination delay*).

For a given “target” rule R being updated, COCONUT and SC only install rules that are co-located with R (here, the edge rules produced by OpenVirtex). CU, in its standard implementation, updates all rules along the paths of flows passing through R . In our evaluation, as an optimization for CU, we limit this to flows that have active traffic.

First, we measure update delays on the testbed sliced to emulate a 20-switch fat-tree topology. Figure 3.6 shows that while compared to SR, COCONUT increases the delay (e.g., $1.4\times$ increase in the median update termination time), it reduces the delay of SC and CU ($2\times$ and $1.8\times$ lower median update termination delay, respectively). We use Mininet to measure this metric at scale and observe similar trends. Figure 3.7 shows mean values; error bars show 10th and 90th percentile over 100 runs. We observe similar trends for the IDS and

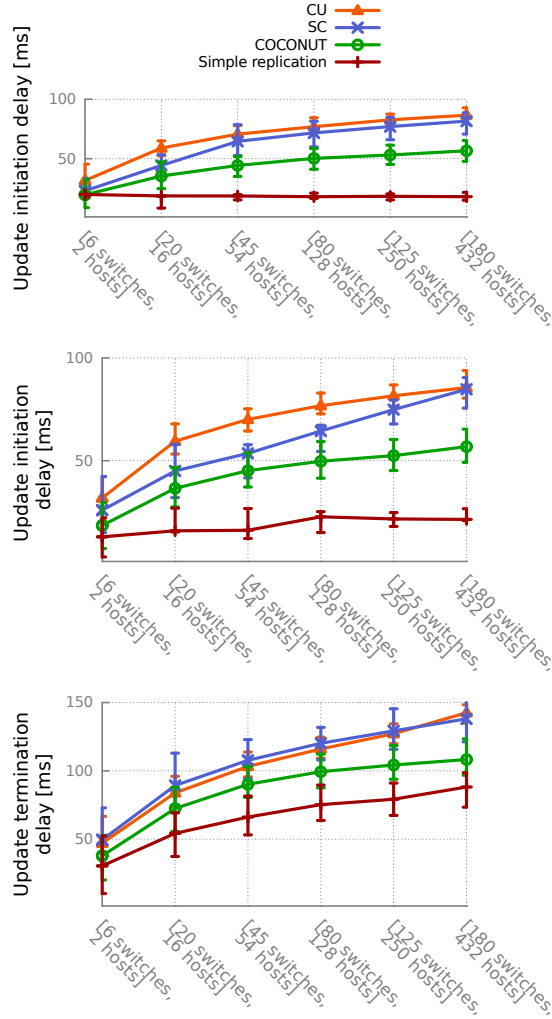


Figure 3.7: How long does it take to initiate and finish updates? Top: initiation delays for the firewall app; middle and bottom: initiation and termination delays for the IDS app.

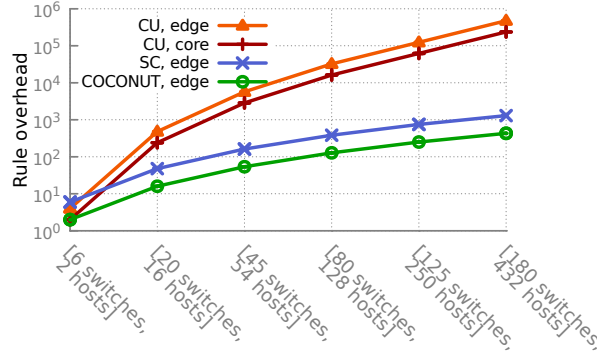


Figure 3.8: How much rule-overhead is imposed and where?

firewall applications. Note that SC’s costs rapidly increases with scale.

The impact of the topology: In addition to the fat-tree networks, we experimented with the VL2 network [15] in Mininet. We found the number of edge switches, which are the switches that need to be updated, is the key player in COCONUT’s speed, with little variation across these topology types. For example, the mean initiation delay for the IDS application was **47.8 ms** on a VL2 network with 25 edge switches (35 switches total and 500 hosts), which is very close to the delay on fat-tree networks of similar size: **45.1 ms** with 18 edge switches (45 switches total and 54 hosts) and **49.8 ms** with 32 edge switches (80 switches total and 128 hosts). Similarly, COCONUT’s mean termination delays were 95.4 ms, 90.2 ms, and 99.3 ms on those three networks. SR and SC were similarly unaffected by the topology change, and CU worsened; we omit the results for brevity.

3.5.5 How Much Rule Overhead Is Imposed and Where?

COCONUT’s, SC’s, and CU’s operations all require installing some temporary extra rules. Since the number of rules switches can support is limited [41], it is important to keep this cost low. We measure the amount, locations, and lifespans of this overhead.

By installing only one set of temporary rules, T s, and morphing them into the final desired rules, COCONUT keeps the number of extra rules minimal.

Plus, similar to SR and SC, COCONUT imposes this rule overhead only on the switches directly hosting the rules in the update batch. This implies that if COCONUT is used in conjunction with the common systems that place virtualized

rules at the edge of the network [6, 36, 38], then only edge switches need to tolerate this overhead. In contrast, CU imposes this overhead on all the switches hosting the rules of the associated flows, possibly including core switches. Figure 3.8 shows the rule overhead (number of extra rules) and its location. Unlike CU, COCONUT and SC only have overhead at edge switches. Even for edge switches, COCONUT’s overhead is significantly lower than SC’s and CU’s, *e.g.*, in a 80-switch network, respectively $2\times$ and $245\times$ lower.⁹

How Long Does Rule-overhead Persist? The extra rules installed by SC, CU, and COCONUT are supposed to be short-lived and all techniques remove those rules in their clean-up operations. Figure 3.9 shows that only 0.7% of COCONUT’s rule overhead persists in the network for more than 100ms compared to 80.6% for SC and 60.7% for CU. This can again be explained by the fact that CU and SC update a significantly larger number of rules and impose a greater load on switches and controllers.

3.5.6 Can Header Bits Become a Scalability Bottleneck?

COCONUT’s ability to handle concurrent updates is limited by the number of header bits available to it; if there are too many concurrent updates, COCONUT will have to queue the requests. With this in mind, can COCONUT handle modern network dynamics? A campus network may experience up to 18K updates per month [123], but the rate is significantly larger and more bursty in cloud environments where customers continuously deploy, delete, and migrate services, with an average of 12K updates per day in a typical cluster, peaking at one update per second [12].

To test COCONUT’s rate of applying updates, we reserve 12 header bits (the number of bits of the VLAN tag, the header field reserved for the update operations in CU [46]), 19 header bits (the number of bits in one MPLS label), and 4 header bytes (the smallest possible option length in Geneve [124]) for COCONUT and modify the IDS application to send to COCONUT 12K update requests, equivalent to the average number of updates in *one day* in a cloud environment of [12]. We run this experiment on a fat-tree with 180 switches and measure the time COCONUT consumes to apply all the updates. Over 20 runs of this experiment, COCONUT applies these updates in respectively 2.4, 1.3, and

⁹Note that we measure only CU’s overhead *on top of* the rules we duplicated to move from a flow-based to a rule-based abstraction (3.5.2).

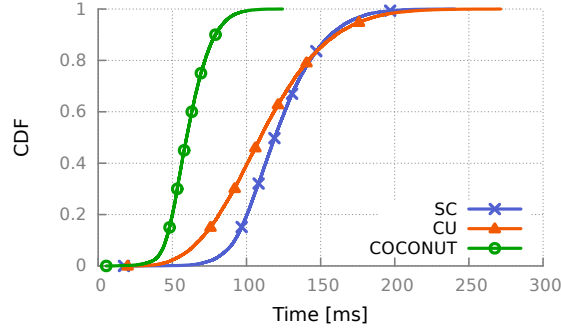


Figure 3.9: How long does the rule-overhead persist?

0.9 minutes on average. Its 90th percentile update time is respectively, 2.6, 1.8, and 1.4 minutes, *i.e.*, more than 90% of the time its rate is $76\times$, $112\times$, and $144\times$ faster than the peak update rate cited in [12]. Thus, we believe the existing header fields for carrying meta-data are more than sufficient for COCONUT’s operations [12].

3.6 Related Work

Sequence planning techniques synthesize an ordering of updates to preserve certain invariants (verified by verification tools [125, 126]) during updates [127]. Finding such orderings is NP-complete [127] and there does not always exist a sequence that preserves invariants such as loop freedom and congestion freedom [46, 128]. Thus, CU proposes an alternative approach for updating the network that guarantee to preserve trace properties [46]. CU formalized *trace properties* characterizing the paths individual packets take through the network, introduced per-packet consistency, and used a 2-phase update algorithm to implement it. As discussed previously, per-packet consistency does not preserve the weak causal correctness that is of interest to us (but also, COCONUT does not attempt to preserve per-packet consistency; to achieve these, the network provider could choose to run CU). While there are a few aspects of technical similarity in mechanism between CU and COCONUT (*e.g.*, version numbers and preloading initially invisible rules), COCONUT also has quite different mechanisms, in particular a vector of virtual clocks, each implemented as a single bit.

A few recent studies try to improve CU’s efficiency, with various restrictions — either preserving narrower properties such as loopfreedom that are *subsets* of per-

packet consistency or with constraints on forwarding rules [129, 117, 120, 48, 47]. None of those works provide guarantees that are stronger than CU’s per-packet consistency.

The recently added atomic update operation of OVS[130] and the bundle capability of OpenFlow [131] enable atomic update of a *single* switch and cannot be extended to multiple replicas at different locations. For the short-lived replication caused by migrations of middleboxes or virtual networks, OpenNF, LIME, and Split/Merge strive to retain strong consistency (SC) by heavy-weight operations¹⁰ such as dropping packets or redirecting them to the controller [11, 34, 132]. In COCONUT, the dataplane continues processing packets during the update, *i.e.*, packets are not buffered (unlike [132, 11]), not redirected to the controller (unlike [34, 108, 11]) which is already a scalability bottleneck in SDNs [96, 122, 116, 118, 6], and not dropped (unlike [34, 132]).

3.7 Conclusion

We demonstrated that current network scale-out techniques do not preserve the semantics of the native network, leading to application-level incorrectness, and presented COCONUT, a system that solves this problem by preserving weak causal correctness. Some practical challenges remain, *e.g.*, requiring modification at endhost hypervisors. However, COCONUT appears to be surprisingly feasible, and represents a promising first step in an area that we believe will become increasingly important with roll-out of network virtualization and NFV.

¹⁰OpenNF’s implementation of SC, for instance, adds 10s of *ms* latency to each packet (*avg.* RTT < 1ms in datacenters). The added latency rapidly increases with traffic rate and number of flows [11].

Chapter 4

TOWARDS A RIGOROUS FRAMEWORK FOR REASONING ABOUT NETWORK BEHAVIORS

In the previous chapter, we argue that focusing on end-points' *observations* is key for providing simple network abstractions with well understood semantics. In this chapter, we introduce a mathematical framework, Input Output Automata (IOA), that allows us to rigorously define and reason about observational correctness and explains its prominent features that make it a suitable choice for us. We then model COCONUT in this framework and prove its observational correctness.

4.1 Introducing the IOA Framework

Input/Output automaton (IOA) is a mathematical framework used for analyzing concurrent and distributed discrete systems that exhibit a combination of discrete and continuous behavior. It models each system component as a nondeterministic, I/O state machine which is essentially an automaton with an action labeling each transition [109]. One key feature of this model is its rigorously defined notion of *external behavior* which captures the visible behavior and interactions of systems with their environments [109]. COCONUT's core idea of focusing on *observations* discernible to external entities makes this framework a natural fit for us. An IOA is a state machine, with a set of states, where transitions are associated with *external* and *internal* actions. External actions consist of *input* and *output* actions and are used for communicating with the environment of the automaton whereas internal actions are visible only to the automaton. More precisely, an IOA $A = \langle X, Q, T, D, \tau \rangle$ has the following components [133, 109]:

- A set of *states*, X , which describes the collection of A 's state variables. Each variable could be external (visible to A 's environment) or internal (visible only to A).

- A set of *start* or *initial states*, Q , which is a non-empty subset of states of A .
- A *signature* that lists the disjoint sets of input, output, and internal actions of A , A .

An action x is said to be *enabled* in a state s if there is another state s' such that (s, x, s') is a transition in the automaton. While output and internal actions that are under the control of the automaton, *i.e.*, it can decide which internal and output actions to perform, input actions are not assumed to be under its control. In other words, input actions are enabled in every state and the IOA is not able to “block” them [133]. Transition relations are usually expressed in *precondition-effect* style in which a precondition is a predicate on the state indicating the conditions under which the action is permitted and the effect describes the change in the IOA’s state that results from performing the action [133].

- A set of *state transition* relations, D , which contains tuples, usually referred to as *transitions* or *steps*, of the form (state, action, state). $D \subset \text{val}(X) \times T \times \text{val}(X)$. We write $(u, a, u') \in D$ in short as $u \rightarrow^a u'$.
- A Set of trajectories of X , τ , that is closed under prefix, suffix, and concatenation. If t is a trajectory, then $t.fstate$ shows the first valuation of t and, if t is closed, $t.lstate$ is the last valuation of t .

An *execution segment* of an IOA is a, finite or infinite, sequence $s_0, x_1, s_1, x_2, \dots$ of alternating states s_i and actions x_i such that $\forall i \geq 0, (s_i, x_{i+1}, s_{i+1})$ is a transition in the IOA. An *execution* is an execution segment that starts with an initial state [133]. The *trace*, *external behavior*, or *behavior* for short, of an execution is the sequence of external actions in that execution. The behavior of an IOA is the set of all plausible traces of that IOA. One important operation of IOA is *composition* which enables this framework to model large complex systems by composing their individual components [133].

Moreover, this framework defines what it means for one IOA A to be an “abstraction” of another IOA B , based on an inclusion relationship between their external behavior sets, defines a notion of simulation which provides a sufficient condition for demonstrating abstraction relationships, and includes a composition operation for IOAs which respects the abstraction relation (§4.2).

We show how the system can be modeled in this framework before formalizing observational correctness and proving that COCONUT is observationally correct.

4.2 Defining Observational Correctness

We may express the correctness of an implementation of an abstraction by showing that its IOA implements an abstract specification automaton. An IOA A implements an IOA B (equivalently, B is an abstraction of A) if there is a *forward simulation* relation from A to B [109]. Forward simulation guarantees that any trace (external behavior) of A is a plausible trace of B . At a high level, a forward simulation relation is a binary relation from the state variables of A to B that satisfies three types of conditions (relating start states, discrete transitions, and trajectories of A and B). After modeling the system, we show that such a binary relation exists from COCONUT’s implementation of one-big-switch to the one-big-switch abstraction. Consequently, any behavior of the physical implementation of the one-big-switch abstraction, implemented by COCONUT, is guaranteed to be a plausible behavior of the ideal, non-virtualized implementation of it:

Theorem 1: Any external behavior of COCONUT’s implementation of one-big-switch could have happened in the logical one-big-switch.

COCONUT’s IOAs are composable: Composition of IOAs requires certain “compatibility” conditions, *i.e.*, (a) *no interference*, internal variables and actions of one automaton cannot be shared by any other automaton and (b) *exclusive control*, each output variable and output action be controlled by at most one automaton [109]. We show later in this section that the output actions, as well as the internal variables and actions of the COCONUT’s IOAs are disjoint, *e.g.*, only `Hypervisors` send packets to the end-points (exclusive control), or at most one switch can modify the vector clock of a packet stored on a switch (no interference). Moreover, it is proven that if an IOA $Abs1$ is an abstraction of IOA $Impl1$, and another IOA $Abs2$ is an abstraction of IOA $Impl2$, then composition of $Abs1$ and $Abs2$ is an abstraction of $Impl1$ and $Impl2$. Hence, multiple COCONUT’s switch IOAs can be composed to form different topologies while guaranteeing observational correctness. Later in this section, we prove:

Theorem 2: COCONUT’s one-big-switch is composable.

We now model each component of the system, before composing them and

proving the above theorems about them.

4.3 Modeling Logical Networks as IOA

Non-virtualized ideal implementation of virtual network abstractions is composed of switches and links. We model links and switches below.

4.3.1 Modeling Links as IOA

We assume that links connect ports. Following the common convention in modeling networks and to keep the model simple, we also assume that ports have unique identifiers, `Port`¹. The type `Port` has subtypes `Ext_port` and `Int_port` for, respectively, ports connected to applications or end-points. Such ports are visible to the users of the network and are therefore classified as *external* in the network IOA. Other ports are connected to switches that are not visible to users (end-points or applications) and are therefore classified as *internal* in the network IOA. Links in the network are assumed to be best-effort, *i.e.*, connecting ports of type `be_port`. These ports can fail to deliver packets and they can reorder them.

Internal variable `to_sendi` is a map that for each port keeps the, possibly empty, sequence of packets that need to be sent on that port. To represent the best-effort nature of networks, packets will be sent without any predetermined order; this is automatically achieved by the internal non-determinism of IOA and the way we model the sending process. The internal variable `status` maps each port to the type of connectivity it provides: reliable FIFO², best-effort, or failed (if the port has failed and not yet recovered). A failed link is modeled as a link connected to at least one failed port.

End-host and controller applications interface with links via two external `app_sendi` and `app_receivei` actions where *i* is the ID of the external port connected to applications or end-points³. **These two sets of actions are the only visible part of the network automaton to applications and end-points.**

¹It is straightforward to replace this choice for port IDs with (switch ID, port ID) and (host ID, port ID).

²This is similar to the main control channel in OpenFlow. We use this type of connectivity for the communication between the controller and shells and switches.

³Following the common convention of modeling IOAs, the **pre** keyword shows the condition that should be met for the IOA to make the transition, *i.e.*, it shows how that action is “enabled”, and the statement after the **pre** keyword shows how the state changes in that transition.

automaton Links

types

<i>Bit</i>	:	$\{0, 1\}$
<i>Packet</i>	:	Bit^+
<i>Port</i>	:	$\{1, \dots, k\}$
<i>Connectivity</i>	:	$\{\text{RELIABLE_FIFO}, \text{BEST_EFFORT}, \text{FAILED}\}$
<i>Ext_port</i>	\subset	<i>Port</i>
<i>Int_port</i>	\subset	<i>Port</i>
<i>rf_port</i>	\subset	<i>Port</i>
<i>be_port</i>	\subset	<i>Port</i>

internal variables

<i>to_send_l</i>	:	$[\text{Port} \rightarrow \text{Packet}^*]$
<i>status_l</i>	:	$[\text{Port} \rightarrow \text{Connectivity}]$

signature

input app_send_i(*p:Packet*)
output app_receive_i(*p:Packet*)
input switch_send_i(*p:Packet*)
output switch_receive_i(*p:Packet*)
internal link_drop(*i:be_port, p:Packet*)
input port_fail_l(*i:bf_port*)
input port_recover_l(*i:bf_port*)

transition

$\forall i \in \text{Ext_port} :$

input	app_send _i (<i>p: Packet</i>)
pre	true
eff	<i>to_send_l</i> [<i>i</i>].append(<i>p</i>)
output	app_receive _i (<i>p: Packet</i>)
pre	<i>status_l</i> [<i>i</i>] = RELIABLE_FIFO $\wedge (p = \text{to_send}_l[i].\text{head})$
eff	<i>to_send_l</i> [<i>i</i>].remove(<i>p</i>)
output	app_receive _i (<i>p: Packet</i>)
pre	<i>status_l</i> [<i>i</i>] = BEST_EFFORT $\wedge (p \in \text{to_send}_l[i])$
eff	<i>to_send_l</i> [<i>i</i>].remove(<i>p</i>)

Figure 4.1: Links as IOA.

automaton Links (cont.)

transition

```

 $\forall i \in \text{Int\_port} :$ 
  input switch_sendi(p: Packet)
  pre true
  eff to_sendl[i].append(p)
  output switch_receivei(p: Packet)
  pre p ∈ to_sendl[i]
  eff to_sendl[i].remove(p)
internal link_deliver(i,j: be_port, p: Packet)
  pre p ∈ to_sendl[i]
  eff to_sendl[i].remove(p)
  to_sendl[j].append(p)
internal link_deliver(i,j: rf_port, p: Packet)
  pre p = to_sendl[i].head
  eff to_sendl[i].remove(p)
  to_sendl[j].append(p)
internal link_drop(i: be_port, p: Packet)
  pre statusl[i]=FAILED
  eff to_sendl[i].remove(p)
input port_faill(i: be_port)
  pre true
  eff statusl[i] := FAILED
input port_recoverl(i: be_port)
  pre true
  eff statusl[i] := BEST_EFFORT

```

Figure 4.1 (cont.)

automaton Switch(Switch ID: ID)

types

<i>Priority</i>	: Int
<i>Time</i>	: Real
<i>Idle-Timer</i>	: Time \cup null
<i>Idle-Timeout-Cst</i>	: Time \cup null
<i>Hard-Timer</i>	: Time \cup null
<i>Hard-Timeout-Cst</i>	: Time \cup null
<i>Counter</i>	: Int
<i>Match</i>	: $\{1, 0, *\}^+$
<i>R_action</i>	: {MOD, SEND, DROP, UPDATE_RULES, UPDATE_TSTAT}
<i>MOD</i>	: Match
<i>UPDATE_RULES</i>	: (Priority, Idle-Timer, Idle-Timeout-Cst, Hard-Timer, Hard-Timeout-Cst, Counter, Match, R_action*)
<i>Rule</i>	: (Priority, Idle-Timer, Idle-Timeout-Cst, Hard-Timer, Hard-Timeout-Cst, Counter, Match, R_action ⁺)

internal variables

<i>Rules</i> _{Switch ID}	: set(Rule)
<i>switch_failed</i> _{Switch ID}	: Boolean
<i>status</i> _{Switch ID}	: [Int_port \rightarrow Connectivity]
<i>to_send</i> _{Switch ID}	: [Int_port \rightarrow Packet*]
<i>received</i> _{Switch ID}	: [Int_port \rightarrow (Packet, R_action ⁺)*]
<i>Backup_rules</i> _{Switch ID}	: [Int_port \rightarrow set(Rule)]

Figure 4.2: Switch as IOA: types and variables.

automaton Switch(Switch ID: ID) (cont.)
transition

```

 $\forall i \in \text{Int\_port connected to Switch ID:}$ 
input switch_receivei(p:Packet)
  pre true
  eff receivedSwitch ID[i].add((p, lookup(p, i)))

 $\forall i \in \text{Int\_port connected to Switch ID:}$ 
output switch_sendi(p:Packet)
  pre statusSwitch ID[i] = RELIABLE_FIFO
     $\wedge (p = \text{to\_send}_{\text{Switch ID}[i]}.head)$ 
  eff to_sendSwitch ID[i].remove(p)

output switch_sendi(p:Packet)
  pre statusSwitch ID[i] = BEST_EFFORT
     $\wedge (p \in \text{to\_send}_{\text{Switch ID}[i]})$ 
  eff to_sendSwitch ID[i].remove(p)

internal switch_int_deliverSwitch ID(i, j: Int_port, p:Packet)
  pre (p  $\in$  receivedSwitch ID[i])
     $\wedge (!\text{switch\_failed}_{\text{Switch ID}})$ 
     $\wedge (\text{status}_{\text{Switch ID}[i]} \neq \text{FAILED})$ 
     $\wedge (\text{status}_{\text{Switch ID}[j]} \neq \text{FAILED})$ 
     $\wedge (\text{lookup\_action}(p, i).type = \text{SEND})$ 
  eff receivedSwitch ID[i].remove(p, i)
    to_sendSwitch ID[lookup_outPort(p, i)]
      .append(p)

internal switch_rewriteSwitch ID(i: Int_port, p:Packet)
  pre (p  $\in$  receivedSwitch ID[i])
     $\wedge (!\text{switch\_failed}_{\text{Switch ID}})$ 
     $\wedge (\text{status}_{\text{Switch ID}[i]} \neq \text{FAILED})$ 
     $\wedge (\text{lookup\_action}(p, i).type = \text{MOD})$ 
  eff receivedSwitch ID[i].remove(p)
    receivedSwitch ID[i].replace(
      rewrite(p, lookup_pattern(p, i)))

```

Figure 4.3: Switch as IOA: packet handling transitions.

automaton Switch(Switch ID: ID) (cont.)

transition

```

internal rules_updateSwitch ID(i: Int_port, p:Packet)
  pre (p ∈ receivedSwitch ID[i])
    ∧ (!switch_failedSwitch ID)
    ∧ (statusSwitch ID[i] ≠ FAILED)
    ∧ (lookup_action(p).type = UPDATE_RULES)
  eff receivedSwitch ID[i].remove(p)
      RulesSwitch ID.update(
        lookup_update(p, i)
      )

internal stats_updateSwitch ID(i: Int_port, p:Packet)
  pre (p ∈ receivedSwitch ID[i])
    ∧ (!switch_failedSwitch ID)
    ∧ (statusSwitch ID[i] ≠ FAILED)
    ∧ (lookup_action(p).type = UPDATE_TSTAT)
  eff receivedSwitch ID[i].remove(p)
      RulesSwitch ID.updateStat(p, i)

internal switch_dropSwitch ID(i: Int_port, p:Packet)
  pre (p ∈ receivedSwitch ID[i]) ∧
    ((switch_failedSwitch ID)
    ∨ (statusSwitch ID[i] ≠ FAILED)
    ∨ (lookup_action(p).type = DROP))
  eff receivedSwitch ID[i].remove(p)

internal switch_dropSwitch ID(i: Int_port, p:Packet)
  pre (p ∈ to_sendSwitch ID[i]) ∧
    ((switch_failedSwitch ID)
    ∨ (statusSwitch ID[i] ≠ FAILED))
  eff to_sendSwitch ID[i].remove(p)

```

Figure 4.3 (cont.)

```

automaton Switch(Switch ID: ID)
  transition

    input local_failoverSwitch ID(i: be_port)
    pre true
    eff RulesSwitch ID.update(backup_rules[i]))

    input port_failSwitch ID(i: be_port)
    pre true
    eff statusSwitch ID[i] := FAILED

    input switch_failSwitch ID
    pre true
    eff switch_failedSwitch ID := true

    input port_recoverSwitch ID(i: be_port)
    pre true
    eff statusSwitch ID[i] := BEST_EFFORT

    input switch_recoverSwitch ID
    pre true
    eff switch_failedSwitch ID := false

```

Figure 4.4: Switch as IOA: failure and recovery transitions.

```

automaton Switch(Switch ID: ID)
  transition

    internal rule_expireSwitch ID(rule: ∈ Rule)
    pre (rule ∈ Rules) ∧
      (rule.Idle-Timer ≥ rule.Idle-Timeout-Cst ∨
       rule.Hard-Timer ≥ rule.Hard-Timeout-Cst)
    eff RulesSwitch ID.remove(rule)

  trajectories
  ∀rule ∈ RulesSwitch ID such that rule.Idle-Timer ≠ null:
    evolve d(rule.Idle-Timer) = 1

  ∀rule ∈ RulesSwitch ID such that rule.Hard-Timer ≠ null:
    evolve d(rule.Hard-Timer) = 1

```

Figure 4.5: Switch as IOA: time-evolving trajectories and transitions.

automaton Hypervisors

internal variables

$to_send_h : [\text{Port} \rightarrow \text{Packet}^*]$
 $status_h : [\text{Port} \rightarrow \text{Connectivity}]$

transition

$\forall i \in \text{Ext_port} :$

input $\text{app_send}_i(p: \text{Packet})$
pre **true**
eff $to_send_h[i].\text{append}(\text{devirtualize}(p))$

output $\text{app_receive}_i(p: \text{Packet})$
pre $status_h[i] = \text{RELIABLE_FIFO}$
 $\wedge (p = to_send_h[i].\text{head})$
eff $to_send_h[i].\text{remove}(\text{virtualize}(p))$

output $\text{app_receive}_i(p: \text{Packet})$
pre $status_h[i] = \text{BEST_EFFORT}$
 $\wedge (p \in to_send_h[i])$
eff $to_send_h[i].\text{remove}(\text{virtualize}(p))$

$\forall i \in \text{Int_port} :$

input $\text{link_send}_i(p: \text{Packet})$
pre **true**
eff $to_send_h[i].\text{append}(p)$

output $\text{link_receive}_i(p: \text{Packet})$
pre $p \in to_send_h[i]$
eff $to_send_h[i].\text{remove}(p)$

internal $\text{hypervisor_deliver}(i, j: \text{be_port}, p: \text{Packet})$
pre $p \in to_send_h[i]$
eff $to_send_h[i].\text{remove}(p)$
 $to_send_h[j].\text{append}(p)$

internal $\text{hypervisor_deliver}(i, j: \text{rf_port}, p: \text{Packet})$
pre $p = to_send_h[i].\text{head}$
eff $to_send_h[i].\text{remove}(p)$
 $to_send_h[j].\text{append}(p)$

Figure 4.6: Hypervisors as IOA.

automaton Hypervisors (cont.)

transition

```
internal hypervisor_drop(i: be_port, p: Packet)  
  pre statusl[i]=FAILED  
  eff to_sendh[i].remove(p)  
input port_failh(i: be_port)  
  pre true  
  eff statush[i] := FAILED  
  
input port_recoverh(i: be_port)  
  pre true  
  eff statush[i] := BEST_EFFORT
```

Figure 4.6 (cont.)

switch_send_{*i*}, switch_receive_{*i*}, link_deliver, and link_drop are, respectively, responsible for receiving the packets a switch sends and sending a packet to a switch on an internal port *i* connected to the switch, moving a packet from one side of a link to another, and dropping packets in case a best-effort link fails. port_fail_{*l*} and port_recover_{*l*} represent failure and recovery of ports.

To enable switches and links to communicate, we initially define switch_send_{*i*} and switch_receive_{*i*} actions as external actions in Links and Switch IOAs. In the next section, we explain why and how, in the composition of Links and Switches, we “hide” these two actions such that applications and end-points do not observe them.

4.3.2 Modeling Switches as IOA

Similar to regular SDN switches, the virtual switch is assumed to have a set of **rules** each having a “priority”, a “match field”, and a partially ordered set of “actions”. Match fields could match packets based on their headers and some state local to the switch such as counters.

For modeling, we make the following realistic assumptions about switches:

- Switches are distributed systems and consist of several chassis. In general, **switches might internally re-order events**, *e.g.*, they might reorder the packets they receive, look them up with different orders, apply actions on them with arbitrary orders, etc. The applications cannot make assumptions

automaton Virtualized Links**internal variables**

$to_send_l : [\text{Port} \rightarrow \text{Packet}^*]$
 $status_l : [\text{Port} \rightarrow \text{Connectivity}]$

transition

$\forall i \in \text{Port connected to Hypervisors:}$

input $\text{hypervisor_send}_i(p: \text{Packet})$
pre **true**
eff $to_send_l[i].append(p)$

output $\text{hypervisor_receive}_i(p: \text{Packet})$
pre $status_l[i] = \text{RELIABLE_FIFO}$
 $\wedge (p = to_send_l[i].head)$
eff $to_send_l[i].remove(p)$

output $\text{hypervisor_receive}_i(p: \text{Packet})$
pre $status_l[i] = \text{BEST_EFFORT}$
 $\wedge (p \in to_send_l[i])$
eff $to_send_l[i].remove(p)$

$\forall i \in \text{Ext_port} :$

input $\text{switch_send}_i(p: \text{Packet})$
pre **true**
eff $to_send_l[i].append(p)$

output $\text{switch_receive}_i(p: \text{Packet})$
pre $p \in to_send_l[i]$
eff $to_send_l[i].remove(p)$

internal $\text{link_deliver}(i, j: \text{be_port}, p: \text{Packet})$

pre $p \in to_send_l[i]$
eff $to_send_l[i].remove(p)$
 $to_send_l[j].append(p)$

internal $\text{link_deliver}(i, j: \text{rf_port}, p: \text{Packet})$

pre $p = to_send_l[i].head$
eff $to_send_l[i].remove(p)$
 $to_send_l[j].append(p)$

internal $\text{link_drop}(i: \text{be_port}, p: \text{Packet})$

pre $status_l[i] = \text{FAILED}$
eff $to_send_l[i].remove(p)$

Figure 4.7: Virtualized links as IOA.

automaton Virtualized Links (cont.)

internal variables

$to_send_l : [\text{Port} \rightarrow \text{Packet}^*]$
 $status_l : [\text{Port} \rightarrow \text{Connectivity}]$

transition

input $port_fail_l(i: be_port)$
pre **true**
eff $status_l[i] := \text{FAILED}$

input $port_recover_l(i: be_port)$
pre **true**
eff $status_l[i] := \text{BEST_EFFORT}$

Figure 4.7 (cont.)

LC : Int
 VC : vector(LC)
 $COCONUTPacket$: [Packet, VC]

Figure 4.8: COCONUT types.

about the internal orderings of events in the switch without receiving packets from the switch. If the applications require switches to perform certain actions with specific orderings, then they should use the existing mechanisms such as “barrier” or confirmations from switches to enforce those orderings.

We use the nondeterminism of IOAs to model this: Multiple actions may be enabled from the same state, and there may be multiple post states from the same action.

- We assume that switches might experience **crash failures**, but not Byzantine failures.

4.4 Modeling Existing Implementations of Logical Networks as IOA

For providing address space virtualization, isolation, and decoupling logical and physical topologies, current network virtualization platforms act as a proxy between the tenants and the actual network, such that they can rewrite data packets,

automaton COCONUT Links

internal variables

to_send_l : [Port \rightarrow COCONUTPacket*]
 $status_l$: [Port \rightarrow Connectivity]

transition

$\forall i \in$ Port connected to Shells:

input shell_send_i(p : COCONUT Packet)

pre true

eff $to_send_l[i].append(p)$

output shell_receive_i(p : COCONUT Packet)

pre $status_l[i] = \text{RELIABLE_FIFO}$

$\wedge (p = to_send_l[i].head)$

eff $to_send_l[i].remove(p)$

output shell_receive_i(p : COCONUT Packet)

pre $status_l[i] = \text{BEST_EFFORT}$

$\wedge (p \in to_send_l[i])$

eff $to_send_l[i].remove(p)$

$\forall i \in$ Ports connected to COCONUT Switches :

input switch_send_i(p : COCONUT Packet)

pre true

eff $to_send_l[i].append(p)$

output switch_receive_i(p : COCONUT Packet)

pre $p \in to_send_l[i]$

eff $to_send_l[i].remove(p)$

internal link_deliver(i, j : be_port, p : COCONUT Packet)

pre $p \in to_send_l[i]$

eff $to_send_l[i].remove(p)$

$to_send_l[j].append(p)$

internal link_deliver(i, j : rf_port, p : COCONUT Packet)

pre $p = to_send_l[i].head$

eff $to_send_l[i].remove(p)$

$to_send_l[j].append(p)$

internal link_drop(i : be_port, p : COCONUT Packet)

pre $status_l[i] = \text{FAILED}$

eff $to_send_l[i].remove(p)$

Figure 4.9: COCONUT links as IOA.

automaton COCONUT Links (cont.)

internal variables

$to_send_l : [\text{Port} \rightarrow \text{COCONUTPacket}^*]$
 $status_l : [\text{Port} \rightarrow \text{Connectivity}]$

transition

input $port_fail_l(i: be_port)$
pre **true**
eff $status_l[i] := \text{FAILED}$

input $port_recover_l(i: be_port)$
pre **true**
eff $status_l[i] := \text{BEST_EFFORT}$

Figure 4.9 (cont.)

control messages, and packet handling rules. Similar to the terminology used in prior work [5], we call the functions that translate tenants' packets, control messages, and rules to the physical packets that will travel the physical network, the messages that will be sent to the physical network and the rules that will be installed in the physical network `devirtualize`. `devirtualize` function takes as input one or more virtual entities and translates them to the corresponding physical entities, *e.g.*, they translate one logical `FlowMod` from a tenant's application for installing a rule on her one big switch abstraction to the corresponding set of `FlowMods` that will be sent to physical switches used for implementing that one-big-switch. Similarly, `virtualize` functions are responsible for translating physical entities to their corresponding logical ones, *e.g.*, they translate a `Packet-In` message from received from the physical network to the corresponding virtual message that should be sent to a tenant.

Virtualization and devirtualization actions are carried out by the network hypervisors that are placed between the tenants' end-host and controller applications and the physical network. Hence, in a virtualized physical networks, applications interface with hypervisors (and not directly with links). Therefore, for modeling such networks, we add hypervisor IOAs that, similar to `Links` IOA in the non-virtualized networks, have external `app-send` and `app-receive` actions to interact with application (Figure 4.6). Virtualized `Links` IOA in virtualized networks are similar to non-virtualized ones except that instead of interfacing with apps directly, they interface with `Hypervisors` (Figure 4.7). `Switch` IOAs are the same; one should note that the rules installed on and packets traversing the virtualized network, however, are devirtualized by the hypervisors before

automaton COCONUT Switch(Switch ID: ID)

internal variables

$Rules_{\text{Switch ID}}$:	set([Rule, VC])
$switch_failed_{\text{Switch ID}}$:	Boolean
$status_{\text{Switch ID}}$:	[Int_port \rightarrow Connectivity]
$to_send_{\text{Switch ID}}$:	[Int_port \rightarrow COCONUTPacket*]
$buffer_{\text{Switch ID}}$:	[Int_port \rightarrow COCONUTPacket*]
$received_{\text{Switch ID}}$:	[Int_port \rightarrow (COCONUTPacket, R_action ⁺)*]
$Backup_rules_{\text{Switch ID}}$:	[Int_port \rightarrow set([Rule, VC])]
$RF_packets_{\text{Switch ID}}$:	[COCONUTPacket, Int_port, R_action ⁺]*
$pending_update_{\text{Switch ID}}$:	[Rule \rightarrow Boolean]

transition

$\forall i \in \text{Int_port}$ connected to Switch ID:

input switch_receive_i(p :COCONUT Packet)

pre true

eff $buffer_{\text{Switch ID}}[i].append((p, lookup(p, i)))$

$\forall i \in \text{Int_port}$ connected to Switch ID:

output switch_send_i(p :COCONUT Packet)

pre $status_{\text{Switch ID}}[i] = \text{RELIABLE_FIFO}$

$\wedge (p = to_send_{\text{Switch ID}}[i].head)$

eff $to_send_{\text{Switch ID}}[i].remove(p)$

output switch_send_i(p :COCONUT Packet)

pre $status_{\text{Switch ID}}[i] = \text{BEST_EFFORT}$

$\wedge (p \in to_send_{\text{Switch ID}}[i])$

eff $to_send_{\text{Switch ID}}[i].remove(p)$

internal switch_int_deliver_{Switch ID}(i, j : Int_port, p :COCONUT Packet)

pre ($p \in received_{\text{Switch ID}}[i].HeadKey$)

$\wedge (!switch_failed_{\text{Switch ID}})$

$\wedge (status_{\text{Switch ID}}[i] \neq \text{FAILED})$

$\wedge (status_{\text{Switch ID}}[j] = \text{BEST_EFFORT})$

$\wedge (lookup_action(p, i).type = \text{SEND})$

$\wedge (lookup_outPort(p, i) = j)$

eff $received_{\text{Switch ID}}[i].remove(p, i)$

$to_send_{\text{Switch ID}}[j].append(p)$

Figure 4.10: COCONUT switch as IOA: variables and packet handling transitions.

automaton COCONUT Switch(Switch ID: ID) (cont.)
transition

```

internal switch_int_deliverSwitch ID(i, j: Int_port, p: COCONUT Packet)
  pre (p ∈ receivedSwitch ID[i].HeadKey)
    ∧ (!switch_failedSwitch ID)
    ∧ (statusSwitch ID[i] ≠ FAILED)
    ∧ (statusSwitch ID[j] = RELIABLE_FIFO)
    ∧ (lookup_action_num(p, i) ≠ 1)
    ∧ (lookup_action(p, i).type = SEND)
    ∧ (lookup_outPort(p, i) = j)
  eff tuple = pop(p, i)

internal switch_rewriteSwitch ID(i: Int_port, p: COCONUT Packet)
  pre (p ∈ receivedSwitch ID[i])
    ∧ (!switch_failedSwitch ID)
    ∧ (statusSwitch ID[i] ≠ FAILED)
    ∧ (lookup_action(p, i).type = MOD)
  eff receivedSwitch ID[i].remove(p)
    receivedSwitch ID[i].replace(
      rewrite(p, lookup_pattern(p, i)))

internal rules_updateSwitch ID(i: Int_port, p: COCONUT Packet)
  pre (p ∈ receivedSwitch ID[i])
    ∧ (!switch_failedSwitch ID)
    ∧ (statusSwitch ID[i] ≠ FAILED)
    ∧ (lookup_action(p).type = UPDATE_RULES)
  eff receivedSwitch ID[i].remove(p)
    RulesSwitch ID.updatewVC(
      lookup_update(p, i))

internal stats_updateSwitch ID(i: Int_port, p: COCONUT Packet)
  pre (p ∈ receivedSwitch ID[i])
    ∧ (!switch_failedSwitch ID)
    ∧ (statusSwitch ID[i] ≠ FAILED)
    ∧ (lookup_action(p).type = UPDATE_TSTAT)
  eff receivedSwitch ID[i].remove(p)
    RulesSwitch ID.updateStat(p, i)

```

Figure 4.10 (cont.)

automaton COCONUT Switch(Switch ID: ID)
transition

```

internal switch_dropSwitch ID(i: Int_port, p: COCONUT Packet)
  pre ( $p \in received_{Switch\ ID}[i]$ )  $\wedge$ 
    ( $(switch\_failed_{Switch\ ID})$ 
      $\vee (status_{Switch\ ID}[i] \neq FAILED)$ 
      $\vee (lookup\_action(p).type = DROP)$ )
  eff  $received_{Switch\ ID}[i].remove(p)$ 

internal switch_dropSwitch ID(i: Int_port, p: COCONUT Packet)
  pre ( $p \in to\_send_{Switch\ ID}[i]$ )  $\wedge$ 
    ( $(switch\_failed_{Switch\ ID})$ 
      $\vee (status_{Switch\ ID}[i] \neq FAILED)$ )
  eff  $to\_send_{Switch\ ID}[i].remove(p)$ 
input local_failoverSwitch ID(i: be_port)
  pre true
  eff  $Rules_{Switch\ ID}.update(backup\_rules[i])$ 

input port_failSwitch ID(i: be_port)
  pre true
  eff  $status_{Switch\ ID}[i] := FAILED$ 

input switch_failSwitch ID
  pre true
  eff  $switch\_failed_{Switch\ ID} := true$ 

input port_recoverSwitch ID(i: be_port)
  pre true
  eff  $status_{Switch\ ID}[i] := BEST\_EFFORT$ 

input switch_recoverSwitch ID
  pre true
  eff  $switch\_failed_{Switch\ ID} := false$ 

```

Figure 4.11: COCONUT switch as IOA: failure and recovery transitions.

```

automaton COCONUT Switch(Switch ID: ID)
transition

    internal rule_expireSwitch ID(rule:  $\in Rule$ )
        pre (rule  $\in Rules$ )  $\wedge$ 
            (rule.Idle-Timer  $\geq$  rule.Idle-Timeout-Cst  $\vee$ 
             rule.Hard-Timer  $\geq$  rule.Hard-Timeout-Cst)
        eff RulesSwitch ID.rewriteAction(rule, null)

    trajectories
     $\forall rule \in Rules_{Switch\ ID}$  such that rule.Idle-Timer  $\neq$  null:
        evolve d(rule.Idle-Timer) = 1

     $\forall rule \in Rules_{Switch\ ID}$  such that rule.Hard-Timer  $\neq$  null:
        evolve d(rule.Hard-Timer) = 1

```

Figure 4.12: COCONUT switch as IOA: time-evolving trajectories and transitions.

entering the network, and the packets sent from the network to the applications are virtualized by hypervisors before reaching them.

We assume that hypervisors can experience crash failure on any ports connected to them. For simplicity of the model, the ID of the ports connected to them are assumed to be identical to the ID of the ports on physical network. That is, if an application and a link are connected with port X in the non-virtualized network, then in the virtualized network, the hypervisor receives and sends packets between the application and the network on the same port X.

The virtualized network is the composition of Hypervisors, Virtualized Links, and Switches.

4.5 Modeling COCONUT as IOA

COCONUT system is a composition of COCONUT links, COCONUT Switches, Shells, Controller, and COCONUT Hypervisors.

COCONUT types: To model the COCONUT system, we define a few new types: LC for logical clocks, VC for the vector of logical clocks that packets carry, and shells and COCONUT switches keep, COCONUT Packet that is similar to the original Packet type except that it has a vector of logical clocks of type VC.

```

automaton COCONUT Switch(Switch ID: ID)
  transition
    output notify_shell(ID: Switch ID, [p: COCONUT Packet, in: Int_port,
                                     actions: R_action+])
      pre (RF_packets.headSwitch ID = [p, in, actions])
      eff RF_packets.remove([p, in, actions])

    internal shell_notificationSwitch ID([p: COCONUT Packet, in: Int_port,
                                           actions: R_action+])
      pre true
      eff received[in].append([p, actions])

    output request_update(ID: Switch ID, r: Rule, vc: VC)
      pre pending_updateSwitch ID[r]
      eff pending_updateSwitch ID[r] = false
    internal checkVC(r: Rule, vc: VC p: COCONUT Packet, i: Int_port)
      pre match(p, r) ∧ [r, vc] ∈ RulesSwitch ID ∧
        (p = bufferSwitch ID[i].head) ∧ (vc ≥ p.VC)
      eff p.VC = max(p.VC, vc)
        received[i]Switch ID.append([p, lookup(p, i)])
        received[i]Switch ID.buffer[i]Switch ID.removehead
    internal checkVC(r: Rule, vc: VC p: COCONUT Packet, i: Int_port)
      pre match(p, r) ∧ [r, vc] ∈ RulesSwitch ID ∧
        (p = bufferSwitch ID[i].head) ∧ NOT(vc ≥ p.VC)
      eff pending_updateSwitch ID[r] = true

```

Figure 4.13: COCONUT switch as IOA: lookup, notifying shell and querying controller.

automaton COCONUT Hypervisors

internal variables

to_send_h : [Port \rightarrow Packet*]
 $status_h$: [Port \rightarrow Connectivity]

transition

$\forall i \in \text{Ext_port} :$

input $app_send_i(p: \text{Packet})$
pre true
eff $to_send_h[i].append(devirtualize(p))$

output $app_receive_i(p: \text{Packet})$
pre $status_h[i] = \text{RELIABLE_FIFO}$
 $\wedge (p = to_send_h[i].head)$
eff $to_send_h[i].remove(virtualize(p))$

output $app_receive_i(p: \text{Packet})$
pre $status_h[i] = \text{BEST_EFFORT}$
 $\wedge (p \in to_send_h[i])$
eff $to_send_h[i].remove(virtualize(p))$

$\forall i \in \text{Int_port} :$

input $shell_send_i(p: \text{Packet})$
pre true
eff $to_send_h[i].append(p)$

output $shell_receive_i(p: \text{Packet})$
pre $p \in to_send_h[i]$
eff $to_send_h[i].remove(p)$

internal $hypervisor_deliver(i, j: be_port, p: \text{Packet})$
pre $p \in to_send_h[i]$
eff $to_send_h[i].remove(p)$
 $to_send_h[j].append(p)$

internal $hypervisor_deliver(i, j: rf_port, p: \text{Packet})$
pre $p = to_send_h[i].head$
eff $to_send_h[i].remove(p)$
 $to_send_h[j].append(p)$

Figure 4.14: COCONUT hypervisors as IOA.

automaton COCONUT Hypervisors (cont.)

internal variables

to_send_h : [Port \rightarrow Packet*]
 $status_h$: [Port \rightarrow Connectivity]

transition

internal hypervisor_drop(i : be_port , p : Packet)

pre $status_l[i]=FAILED$
eff $to_send_h[i].remove(p)$

input port_fail $_h$ (i : be_port)

pre **true**
eff $status_h[i] := FAILED$

input port_recover $_h$ (i : be_port)

pre **true**
eff $status_h[i] := BEST_EFFORT$

Figure 4.14 (cont.)

COCONUT links are modeled similar to Virtualized Links except that (a) they interface with Shells instead of Hypervisors, *i.e.*, hypervisor_send and hypervisor_receive actions are replaced by shell_send and shell_receive, and (b) they send, receive, save (in their internal state), deliver, and drop COCONUT Packets and not Packets (4.9).

COCONUT switches (COCONUT Switch) are similar to regular switches except that (a) they process COCONUT Packets instead of Packets, (b) for their internal state, they also keep vectors of logical clocks for each rule residing on them, (c) their lookup functions in the switch_receive actions check vectors that COCONUT Packets carry, in addition to the normal lookup operation, (d) the actions that modify the forwarding rules on switches, *i.e.*, rules_update, local_failover, and rule_expire, also update the vector of logical clocks of the switch, in addition to their normal operations.

Also, backup rules of COCONUT switches are assumed to have higher VCs than current active rules. Unlike regular switches, when an COCONUT switch receives an COCONUT packet, the packet is first buffered (in `buffer`) where its VC is checked and the switch makes sure that the related rule that matches the packet is up to date for handling that packet, *i.e.*, its VC is equal or larger than that of the packet, before removing the packet from the buffer and putting it (as regular switches) into `received`. If the rule is not updated enough, its key in the `pending_update`_{Switch ID} is set to `true` (and the packet is not moved to the

automaton Shells

internal variables

to_send_s : $[Int_port \rightarrow COCONUTPacket^*]$
 $buffer_s$: $[Int_port \rightarrow COCONUTPacket^*]$
 $status_s$: $[Port \rightarrow Connectivity]$
 vc_s : VC
 RF_s : $[SwitchID, COCONUTPacket, Int_port, R_actions^+]^*$

transition

$\forall i \in Port$ connected to Hypervisors:

input shell_receive_i(p : *Packet*)

pre true

eff $to_send_s[i].append(addVC(p))$

output shell_send_i(p : *Packet*)

pre $status_s[i] = RELIABLE_FIFO$

$\wedge (p = to_send_s[i].head.PACKET)$

eff $to_send_s[i].remove(removeVC(to_send_s[i].head))$

output shell_send_i(p : *Packet*)

pre $status_s[i] = BEST_EFFORT$

$\wedge (\exists q : COCONUTPacket \in to_send_s[i]$

such that $q.PACKET = p)$

eff $to_send_s[i].remove(removeVC(q))$

$\forall i \in Int_port$:

input link_send_i(p : *COCONUT Packet*)

pre true

eff $buffer_s[i].append(p)$

output link_receive_i(p : *COCONUT Packet*)

pre $p \in to_send_s[i]$

eff $to_send_s[i].remove(p)$

internal shell_deliver(i, j : *be_port*, p : *COCONUT Packet*)

pre $p \in to_send_s[i]$

eff $to_send_s[i].remove(p)$

$to_send_s[j].append(p)$

internal shell_deliver(i, j : *rf_port*, p : *COCONUT Packet*)

pre $p = to_send_s[i].head$

eff $to_send_s[i].remove(p)$

$to_send_s[j].append(p)$

Figure 4.15: Shells as IOA.

automaton Shells (cont.)

transition

```

internal shell_drop(i: be_port, p: COCONUT Packet)
    pre  $status_s[i] = \text{FAILED}$ 
    eff  $to\_send_s[i].remove(p)$ 

input port_fail_s(i: be_port)
    pre true
    eff  $status_s[i] := \text{FAILED}$ 

input port_recover_s(i: be_port)
    pre true
    eff  $status_s[i] := \text{BEST\_EFFORT}$ 

internal notify_shell(ID: Switch ID, [p:COCONUT Packet, in: Int_port,]
    actions:R_action+)
    pre true
    eff  $RF_s.append([ID, p, in, actions])$ 

internal updateVC(i: Int_port, p: COCONUT Packet)
    pre  $buffer[i].head.VC > vc_s$ 
    eff  $vc_s = buffer[i].head.VC$ 
     $to - send_s.append(buffer[i].head)$ 
     $buffer[i].removehead$ 

internal updateVC(i: Int_port, p: COCONUT Packet)
    pre  $\text{NOT}(buffer[i].head.VC > vc_s)$ 
    eff  $to - send_s.append(buffer[i].head)$ 
     $buffer[i].removehead$ 

 $\forall \text{Switch IDs connected to Shells:}$ 
output shell_notificationSwitchID([p:COCONUT Packet, in: Int_port,]
    actions:R_action+)
    pre  $RF_s.head = \text{SwitchID}$ 
    eff  $RF_s.removeHead$ 

```

Figure 4.15 (cont.)

automaton Controller**internal variables**

to_send_{co} : $[SwitchID \rightarrow Packet^*]$
 $sw_requests$: $[sw \rightarrow [Rule, VC]^*]$
 $batch$: $set([SwitchID, Rule])$

transition

input request(sw : *Switch ID*, r : *Rule*)

pre true
eff $to_send_{co}[sw].append(FlowMod(r))$

output shell_send_i(p : *COCONUT Packet*)

pre $\exists sw \in Switch\ ID\ such\ that$
 $(connect(i, sw)) \wedge (to_send_{co}[sw].head = p)$
eff $to_send_{co}[sw].removehead$

input request_update(sw : *Switch ID*, r : *Rule*, vc : *VC*)

pre true
eff $switch_requests[sw].append([r, vc])$

internal process_switch_request(sw : *Switch ID*, $rule$: *Rule*, vc : *VC*)

pre $(switch_requests[sw] \neq null) \wedge$
 $(to_send_{co}[sw].includeRule(sw_requests[sw].headRule)$
 $\wedge (sw_requests[sw].headVC = VC)$
eff $switch_requests[sw].removehead$

internal process_switch_request(sw : *Switch ID*, $rule$: *Rule*, vc : *VC*)

pre $(switch_requests[sw] \neq null) \wedge$
 $(!to_send_{co}[sw].includeRule(sw_requests[sw].headRule)$
 $\wedge (sw_requests[sw].headVC = VC)$
 $\wedge (maxVC(queryBatch(sw, rule)) > VC)$
eff $to_send_{co}[sw].append($
 $FlowMod(maxRule(queryBatch(sw, rule))))$
 $switch_requests[sw].removehead$

internal process_switch_request(sw : *Switch ID*, $rule$: *Rule*, vc : *VC*)

pre $(switch_requests[sw] \neq null) \wedge$
 $(!to_send_{co}[sw].includeRule(sw_requests[sw].headRule) \wedge$
 $(sw_requests[sw].headVC = VC)$
 $\wedge (maxVC(queryBatch(sw, rule)) \leq VC)$
eff $to_send_{co}[sw].append($
 $FlowMod(modifyAction(rule, DROP)))$
 $switch_requests[sw].removehead$

Figure 4.16: Controller as IOA.

received). Rules pending to be updated are not applied on packets before they are updated.

COCONUT hypervisors are similar to regular hypervisors except that they interface with `Shells` instead of `Virtualized Links`.

COCONUT's controller is modeled as an IOA with the internal state `rule-state` that is a set of `[rule: Rule, switch: Switch ID, vc: VC]` tuples. Each tuple shows a rule, the hosting switch, and the VC of that rule that the controller is aware of⁴. The internal action `request` is used for modeling the batch requests that users of COCONUT, such as the network virtualization systems, send to it.

OpenFlow Implementation of COCONUT: In addition to the high-level algorithms and design of COCONUT, we also provide its OpenFlow implementation. This OpenFlow implementation is a system composed of COCONUT links, COCONUT ISwitches, IShells, IController, Edge controller, and COCONUT Hypervisors. These IOAs are very similar to their high-level counterparts explained above. We discuss the differences below:

COCONUT OpenFlow implementation shells IShells are similar to Shells except that they get information about (de)activation of tags from the edge controller.

COCONUT OpenFlow implementation switches (COCONUT ISwitch) are similar to COCONUT CSwitches except for their actions of updating rules: `rules_update` only update the forwarding state if the command to do so is sent by the controller and otherwise sends an update request to the controller. Similarly, `rule_expire` and `local_failover` send update requests to the controller instead of updating the switch rules.

COCONUT OpenFlow implementation controller (IController) is similar to COCONUT's controller in the high-level design, except that it has actions for handling rule updates that switches autonomously do in the high-level version. IController controller keeps track of tags and informs the edge controllers about them.

COCONUT edge controller (Edge Controller) gets tag information from the IController. IShells consult it for knowing whether they should tag a packet with a tag or not.

⁴Note that switches are allowed to locally update their rules without informing the controller. In this case, the VCs might be outdated. This does not cause any COCONUT correctness issue.

4.5.1 More Details on the Modeled IOAs

Initial (start) states: we assume that in the initial state all buffers (buffers, to_send, and received variables) as well as the sw_requests, and batch variables are all initially empty; the vc variables are set to 0; no port or switches has failed (hence, failed variables are **false** and status variables show the connectivity type: best effort or reliable FIFO). It is also assumed that the Rules and Backup_rules variables of the COCONUT switches are populated by translating Rules and Backup_rules variables of the logical switch by the virtualization system, *i.e.*, by applying the devirtualize function on them.

In addition to the actions, each IOA also uses internal functions (that do not alter its state). The functions are listed below:

- `lookup(p: Packet, i:Port)`, `lookup(p: COCONUT Packet, i:Port)`: look up packet (or COCONUT packet) p from input port i in the rules and returns a list of actions that should be applied on p .
- `lookup_action(p: Packet, i:Port)`, `lookup_action(p: COCONUT Packet, i:Port)`: return the head action of the packet p which is received on port i .
- `remove(i: Port)`: remove the head action of the [packet, action sequence] at the head of the `received[i]`, *i.e.*, the first action that needs to be applied on the packet buffered at port i that is being processed. If the actions sequence becomes null, it also removes the tuple from the `received[i]` buffer.
- `lookup_outPort(packet, i: Port)`: return the output port of the head action of `received(i)`.
- `rewrite(p: Packet, pattern: Match)`, `rewrite(p: COCONUT Packet, pattern: Match)`: return a packet that is similar to p except for the bits in pattern.
- `replace(p': Packet)`, `replace(p': COCONUT Packet)`: (called on `received[i]`) replace the head packet of `received[i]` with p' .

- `lookup_update(p: Packet, i: Port),`
`lookup_update(p: COCONUT Packet, i: Port):` **re-**
turn the update action (with UPDATE type) of packet p in `received[i]`
that should be applied to rules.
- `updateStat(p: Packet, i: Port), updateStat(p:`
`COCONUT Packet, i: Port):` **(called on Rules) replace the rules**
that match packet p received on port i in *Rules* with those with updated
stats, e.g., with rules with incremented counters.
- `lookup_outPort(p: Packet, i: Port),`
`lookup_outPort(p: COCONUT Packet, i: Port):` **re-**
turn the output port of the head action of `received[i]` for p .
- `lookup_action(p: Packet, i: Port),`
`lookup_action(p: COCONUT Packet, i: Port):` **return the**
head action of `received[i]` for p .
- `lookup_action_num(p: Packet, i: Port),`
`lookup_action_num(p: COCONUT Packet, i: Port):`
return the size of the action-list of `received[i]`'s head.
- `pop(p: Packet, i: Port), pop(p: COCONUT Packet,`
`i: Port):` **return the $[q: \text{Packet}, i: \text{Port}, \text{Ractions}^+]$**
tuple where $[q: \text{Packet}, \text{Ractions}^+]$ is the head of
`received[i]` and removes it from `received[i]`. This is used
for sending packet q with Ractions^+ actions on `receive[i]` again.
- `replace(p: Packet), replace(p: COCONUT Packet):`
(called on `received`) replaces the head packet of `received[i]` with p .
- `updatewVC(a: UPDATE):` **(called on Rules inside `rule_update`) incre-**
ments VC of the rules that a updates.
- `rewriteAction(rule: Rule, a: Action):` **rewrites rule by**
replacing its action with a . Dropping is explicitly modeled as an action.
So, passing null for a means that the rule was removed. In this case, any
matching packet needs to just check its VC against the VC of this rule (no
action will be performed on it on behalf of this rule), and the actions of

other rules matching on it will be performed on the packet. Normal VC operations are performed — both for updating the rule and for matching packets against it.

Put differently, lookup function returns a rule whose action is not null, but triggers `request_update` if the rule with the null action matches a packet and has a higher VC than it.

- `addVC(p: Packet)`: takes a packet p as input, adds a VC to it, and returns an COCONUT Packet as output.
- `PACKET`: (called on COCONUT packets) discards the VC of the COCONUT Packet and returns its packet.
- `rmVC(p: COCONUT Packet)`: removes VC from the packet (in Shells).
- `FlowMod(r: Rule)`: creates and returns a packet out of the r rule. This packet can then be sent to the switch for applying r .
- `connect(i: Port, sw: Switch ID)`: returns true iff sw is connected to port i of shell.
- `includeRule(r: Rule)`: (called on `to_send[sw]`) returns true iff the entry for switch sw has the packet for the rule r .
- `queryBatch(sw: Switch ID, rule: Rule)`: queries all switches in the same batch with $(sw, rule)$ pair for the rule r . It returns a set of rules and their VCs, *i.e.*, `set([rule: Rule, vc: VC])`.
- `maxVC(set([rule: Rule, vc: VC]))`: returns maximum value of VC of all rules in the set.
- `maxRule(set([rule: Rule, vc: VC]))`: returns the rule with the maximum VC value in the set.
- `modifyAction(rule: Rule, a: Action)`: modifies rule r to set its action to a .

4.6 COCONUT Guarantees Observational Correctness

IOA A is an *observationally correct* implementation of IOA B iff any behavior of A is a plausible behavior of B . Informally, A is an observationally correct implementation of B if anything that happens in A could have happened in B . This is essentially akin to the way that correctness of abstractions and simulations are defined in the IOA framework [109, 133]. To prove the observational correctness of COCONUT, we first “hide” the external actions that are not visible to the end-points and applications, then prove that the logical one-big-switch (composition of `Links` and `Switch`) is an abstraction of the COCONUT implementation of it that uses replication. Finally, we show that COCONUT IOAs are composable. Hence, any composition of multiple COCONUT switches to form arbitrary topologies is observationally correct.

4.6.1 External and Internal Actions, and Hiding

To compose IOAs and enable communication between them, we initially define the actions that they use for interacting with each other (such as the actions for sending and receiving COCONUT packets between the COCONUT links and shells) as external input and output actions in the model.

In the next phase and to prove correctness, we “hide” those extra external actions that we defined for easier composition of IOAs. In other words, in the composition of various IOAs such as `Links` and `Switch` IOAs, we hide all external actions except sending and receiving from applications using the *hiding operation* [109]. This operation simply reclassifies external actions as internal and hide them from the external world, *i.e.*, the applications.

Action hiding respect the implementation relationship *i.e.*, if IOA $A < B$ (B is an abstraction of A), and E is a subset of actions, then $\text{ActHide}(E, A) < \text{ActHide}(E, B)$, where $\text{ActHide}(X, Y)$ represents hiding the action set X in IOA Y [109].

By providing a forward simulation relation from the high-level algorithms of COCONUT to the abstract virtual network, and from the OpenFlow implementation of COCONUT to its high-level version, and by using the theorem that forward simulation is transitive [109], we prove that both our high-level and OpenFlow im-

plementation designs are correct implementation of virtual network.

4.6.2 Proving that COCONUT Guarantees Observational Correctness

The IOAs outlined above and their hiding and composition operations enable us to prove the main theorem of this section:

Theorem 1: COCONUT is observationally correct, *i.e.*, any external behavior of COCONUT's implementation of one-big-switch could have happened in the logical one-big-switch.

Proof: We prove that the COCONUT algorithms (§3.3.3) are correct by proving that the *Abstraction* IOA is an abstraction of the *Implementation* IOA, where the Abstraction IOA is the composition of one logical switch and links IOAs, and the Implementation IOA is the composition of (a) multiple physical COCONUT switches that together implement the one big logical switch, *i.e.*, each physical switch is the output of `devirtualize(Abstraction)`, (b) COCONUT links, (c) shells, (d) hypervisors, and (e) the controller.

This is achieved by showing that there always exists a binary relation $R \subset \text{val}(\text{Implementation})^5 \times \text{val}(\text{Abstraction})$. Equivalently, for every $\theta_{Imp} \in \text{val}(\text{Implementation})$, we have $\theta_{Imp} R \theta_{Abs}$ where $\theta_{Abs} \in \text{val}(\text{Abstraction})$, and:

- $\theta_{Abs}.\text{Rules}_L = \text{virtualize}(\text{rulesWithOldestVCs}(\theta_{Imp}.\text{Rules}_P))$,
where P represents the set of IDs of the physical switches, *i.e.*, those contained in the *Implementation* IOA and L shows the ID of the logical switch, *i.e.*, the switch in the *Abstraction* IOA.
- $\theta_{Abs}.\text{Backup_rules}_L = \text{virtualize}(\text{rulesWithOldestVCs}(\theta_{Imp}.\text{Backup_rules}_P))$
- $\forall i \in \text{Ext_port } \theta_{Abs}.\text{to_send}_l[i] = [\theta_{Imp}.\text{to_send}_h[i], \theta_{Imp}.\text{to_send}_s[j], \theta_{Imp}.\text{to_send}_l[k]]$ where similar to the logical network, the end-point is connected to the network at port i in the COCONUT network. Unlike the logical network in which i directly connects an end-point and a link, in the

⁵ $\text{val}(A)$ represents the valuation of IOA A 's state.

COCONUT network, port i connects the end-point to the hypervisor. Hypervisor's port i in turn is connected to a shell's port (let's call this port j). j is then connected to a link's port k .

- $\forall i \in \text{Int_port } \theta_{Abs}.\text{to_send}_L[i] =$
 $[\theta_{Imp}.\text{buffer}_{P_1}[i_1], \theta_{Imp}.\text{to_send}_{P_1}[i_1],$
 $\theta_{Imp}.\text{buffer}_{P_2}[i_2], \theta_{Imp}.\text{to_send}_{P_2}[i_2], \dots,$
 $\theta_{Imp}.\text{buffer}_{P_N}[i_N], \theta_{Imp}.\text{to_send}_{P_N}[i_N]]_{(P_1, i_1), (P_2, i_2), \dots, (P_N, i_N) \in \text{devirtualize}(L, i)}$

where similar to the logical network in which switch L is connected to the Links IOA at port i , the replicated physical switches P_1, P_2, \dots, P_N are connected to links at ports i_1, \dots, i_N .

- $\theta_{Abs}.\text{status}_l = \theta_{Imp}.\text{status}_l$
- $\theta_{Abs}.\text{switch_failed}_L = \bigwedge_{P \in \{\text{devirtualize}(L)\}} (\theta_{Imp}.\text{switch_failed}_P)$
- $\theta_{Abs}.\text{status}_L = \text{virtualize}(\theta_{Imp}.\text{status}_P)$
- $\theta_{Abs}.\text{to_send}_L = \bigcup_{P \in \{\text{devirtualize}(L)\}} (\theta_{Imp}.\text{to_send}_P.\text{PACKET})$
- $\theta_{Abs}.\text{received}_L = \bigcup_{P \in \{\text{devirtualize}(L)\}} (\theta_{Imp}.\text{received}_P.\text{PACKET})$

Note that R does not place any restrictions on values of other variables of θ_{Imp} except those outlined above, *i.e.*, $\forall \mathbf{c}$ variables can assume any values. This allows a high degree of flexibility—we do not need to keep the state in the physical replicas consistent unless that violates observational correctness.

R provides a simulation relation from *Implementation* to *Abstraction* because:

- For every initial state θ_{Imp} of *Implementation*, that is for every state of *Implementation* that meets the conditions in §4.5.1, R maps θ_{Imp} to an initial state θ_{Abs} of *Abstraction* that meets the conditions in §4.5.1, *i.e.*, where buffers are empty and $\theta_{Abs}.\text{Rules}$ and Backup_rules are rules on which virtualize function of the virtualization systems are applied.
- $\forall x_1, x'_1, x_1 \rightarrow_\alpha x'_1 \in \text{val}(\text{Implementation})$ and $x_2 \in \text{val}(\text{Abstraction})$ with $x_1 R x_2$, $\exists x'_2$ such that (a) $x_2 \rightarrow_\beta x'_2$, (b) $x'_1 R x'_2$, and (c) $\text{trace}(\beta) = \text{trace}(\alpha)$, where $\text{trace}(x)$ shows the external (input and output) action of x .

- For every external action a of *Implementation* (after hiding) these conditions hold: $a = \text{app_send}$ or app_receive actions of

the COCONUT hypervisor. In this case, $\beta = \text{app_send}$ or app_receive actions of the *Links*, and the change in variables for both automata is that the packet that is sent by the end-point is, respectively, added or removed from the to_send_h in the *Implementation* and from the to_send_l in the *Abstraction*. Thus, if we had $\theta_{Abs}.\text{to_send}_l = \theta_{Imp}.\text{to_send}_h$ before these actions, we will have the same equation, after that too (other variables are not changed). Hence $x'_1 R x'_2$. Also, $\text{trace}(\beta) = \text{app_send}$ or app_receive which is similar to α .

- For action a being the failure and recovery of physical ports and switches, β is the failure and recovery of the related virtual elements.
- For rule updates, (a) if the physical switch is the only physical switch with the old rule among the physical replicas that the logical switch is mapped to, then updating the rule r has an equivalent in *Abstraction*: updating the $\text{virtualized}(r)$, (b) if it's not, then $\beta = \text{null}$.
- When action a is the internal actions of delivering a packet from one port of links, switch, shells hypervisors to the other port, *e.g.*, for actions `shell_send`, `shell_receive`, `shell_deliver`, `link_send`, `link_receive`, `hypervisor_deliver`, `switch_send`, `switch_receive`, the trace is empty (hence $\beta = \text{null}$ and $\text{trace}(a) = \text{trace}(\beta) = \text{null}$). The change in the state of the physical network is moving a packet from the head of a buffer $i-1$ to the back of buffer i where by contactation of buffers $i-1$ and i have a binary relation to a buffer in the logical network by R . Since before a that relation was hold, it is guaranteed to hold after it too.

As an example, for $a = \text{switch_receive}_P$ in *Implementation*, $\beta = \text{switch_receive}_{\text{virtualize}(P)}$ in *Abstraction*. The change in variables for *Implementation* is adding the packet to `buffer`, and the change in variables for *Abstraction* is adding the packet to `to_send` and `buffer` in *Implementation* is mapped by R to `to_send` in *Abstraction*. So, $\theta_{Abs}.\text{buffer} = \cup_{s \in sw} \theta_{Imp}.\text{buffers}[s]$, and consequently, $x'_1 R x'_2$ will continue to hold. $\text{trace}(\beta) = \text{receive}$ which is similar to α .

- For $a \in \{\text{switch_internal_deliver}, \text{switch_rewrite}, \text{stats_update}, \text{switch_drop}\}$ actions in the physical network,

related actions β with identical name exist in the *Abstraction*

- $\forall t \in \tau_{Implementation}, x_2 \in val(Abstraction)$ with $t.fstateRx_2, \exists \beta \in \tau_{Abstraction}$ such that (a) $\beta.fstate = x_2$ (because the initial values of the idle- and hard-timers are identical), (b) $t.lstateR\beta.lstate$ since timer of an expired physical rule has a corresponding value in the logical network, and (c) $trace(\beta) = trace(t)=null$.

4.6.3 Composing One-Big-Switches (and Other IOAs)

In this section, we show that COCONUT's one-big-switch abstractions are composable, *i.e.*, any arbitrary topology built out of COCONUT's one-big-switch abstractions is guaranteed to be observationally correct,

Theorem 2: COCONUT's one-big-switch is composable.

Proof: In the IOA framework, abstraction and implementation relations are preserved under composition [109]. That is, if an IOA $Abs1$ is an abstraction of IOA $Impl1$, shown as $Impl1 < Abs1$, and a different IOA $Abs2$ is an abstraction of IOA $Impl2$, *i.e.*, $Impl2 < Abs2$, then composition of $Abs1$ and $Abs2$ is an abstraction of $Impl1$ and $Impl2$, *i.e.*, (composition of $Impl1$ and $Impl2$) $<$ (composition of $Abs1$ and $Abs2$). Hence, multiple COCONUT Switch IOAs can be composed to form different topologies while guaranteeing correctness [109].

Composition in the IOA framework, however, requires several conditions [109]:

- *No interference*, internal variables and actions of one automaton cannot be shared by any other automaton.
- *Exclusive control*, each output variable and output action be controlled by at most one automaton.

These two conditions hold in COCONUT, because the action and variable names of each IOA in each system (logical network and COCONUT) are chosen to be distinct, *i.e.*, there is no overlap between actions and variables of any two IOA in one system. This enforces exclusive control and avoids interference. Note that this is not solely a syntactical consideration, as an example the mechanisms for managing logical clocks is designed in a way to ensure multiple logical rules composed into a single physical one do not share control over the same logical clocks, *e.g.*, they each have their own separate dimension on the vector clocks.

Chapter 5

CONCLUSION

This thesis studies two areas of parallelism in networks: multi-pathing or the path parallelism in the data plane, and the network function parallelism. When having multi-paths, in contrast to the currently pervasive approach of balancing the load based on global and macroscopic view of traffic, we explore an alternative approach of *micro load balancing* (§2). We present a datacenter micro load balancer, DRILL, which enables the network fabric to make load balancing decisions at microsecond time scales based on traffic information local to each switch. Our experiments show that DRILL’s simple provably-stable switch scheduling algorithm outperforms the state-of-the-art load balancing schemes in Clos networks, particularly under heavy load. We leave the investigation of micro load balancing in other topologies to future work.

In addition to multi-pathing, parallelism is used extensively today for implementing network functions. Notably, modern virtualized data centers provide a simple virtual abstraction of network. The implementations of these virtual networks, such as a “big switch” abstraction, commonly use nontrivial mappings from one virtual element to multiple physical elements. A key question is, do these abstractions faithfully preserve their native semantics? In §3, we show that the answer to that question is “no” for existing network virtualization methods: behavior can differ between the virtual network abstractions and their physical implementations, resulting in incorrect application-level behavior, even when the common correctness condition of per-packet consistency is preserved throughout. This indicates that a new understanding of correctness and new techniques to guarantee it are needed. We develop the COCONUT framework for seamless scale-out of composable one-big-switch abstractions, so that any virtual network composed in COCONUT is guaranteed to have a plausible behavior of its ideal implementation. Surprisingly, we show that this strong correctness condition is feasible: our experiments demonstrate that COCONUT does not impose greater overhead compared with existing systems. Finally in §4, we present, IOA, an analytical

framework to describe network behavior observable by end-points. We formally prove that COCONUT preserves observational correctness, *i.e.*, any external behavior of COCONUT's implementation of one-big-switch could have happened in the logical one-big-switch. Furthermore, we show that COCONUT's one-big-switch is composable.

BIBLIOGRAPHY

- [1] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, “Planck: millisecond-scale monitoring and control for commodity networks,” in *SIGCOMM*, 2014.
- [2] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese et al., “CONGA: Distributed congestion-aware load balancing for datacenters,” in *SIGCOMM*, 2014.
- [3] T. Benson, A. Anand, A. Akella, and M. Zhang, “MicroTE: Fine grained traffic engineering for data centers,” in *CoNEXT*, 2011.
- [4] B. Liskov, “The power of abstraction,” *Distributed Computing*, pp. 3–3, 2010.
- [5] A. Al-Shabibi, M. D. Leenheer, M. Gerola, A. Koshibe, E. Salvadori, G. Parulkar, and B. Snow, “OpenVirteX: Make Your Virtual SDNs Programmable,” in *HotSDN*, 2014.
- [6] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram et al., “Network virtualization in multi-tenant datacenters,” in *NSDI*, 2014.
- [7] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, “WCMP: Weighted cost multipathing for improved fairness in data centers,” in *EuroSys*, 2014.
- [8] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano et al., “Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network,” in *SIGCOMM*, 2015.
- [9] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, “Presto: Edge-based load balancing for fast datacenter networks,” in *SIGCOMM*, 2015.
- [10] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, “Understanding replication in databases and distributed systems,” in *ICDCS*.

- [11] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, “OpenNF: Enabling innovation in network function control,” in *SIGCOMM*, 2014.
- [12] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu et al., “Ananta: Cloud scale load balancing,” in *CCR*, vol. 43, no. 4, 2013.
- [13] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingeroglu, B. Cheyney, W. Shang, and J. D. Hosein, “Magle: A fast and reliable software network load balancer,” in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.
- [14] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *SIGCOMM*, 2015.
- [15] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, “VL2: a scalable and flexible data center network,” *Commun. ACM*, vol. 54, no. 3, 2011.
- [16] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, “PortLand: a scalable fault-tolerant layer 2 data center network fabric,” *CCR*, 2009.
- [17] X. Li and M. J. Freedman, “Scaling IP Multicast on Datacenter Topologies,” *CoNEXT*, 2013.
- [18] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *CCR*, 2008.
- [19] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, “F10: A fault-tolerant engineered network,” in *NSDI*, 2013.
- [20] “ONS 2015 Keynote: A. Vahdat, Google,” 2015, www.youtube.com/watch?v=FaAZAI2x0w.
- [21] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, “Per-packet load-balanced, low-latency routing for clos-based data center networks,” in *CoNEXT*. ACM, 2013.
- [22] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *IMC*. ACM, 2010.
- [23] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” in *NSDI*, 2010.

- [24] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *INFOCOM*, 2013.
- [25] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: a retrospective on evolving SDN," in *HotSDN*, 2012.
- [26] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," in *SIGCOMM*, 2014.
- [27] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM*, 2011.
- [28] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *SIGCOMM*, 2009.
- [29] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *CCR*, 2010.
- [30] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, 2001.
- [31] A. Mekittikul and N. McKeown, "A practical scheduling algorithm to achieve 100% throughput in input-queued switches," in *INFOCOM*, 1998.
- [32] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *CCR*, 2011.
- [33] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *PRESTO*, 2010.
- [34] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker, "Transparent, Live Migration of a Software-Defined Network," in *SoCC*, 2014.
- [35] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: substrate support for path splitting and migration," *CCR*, 2008.
- [36] M. Ciosi et al., "Network functions virtualization," ETSI, Tech. Rep., 2013, <http://goo.gl/Q84Bxi>.
- [37] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker et al., "Composing software defined networks," in *NSDI*, 2013.
- [38] "ONS 2014 Keynote: A. Greenberg, Microsoft Azure," <http://www.youtube.com/watch?v=8Kyoj3bKepY>, 2014.

- [39] “ONS 2014 Keynote: A. Vahdat, Google,” 2014. [Online]. Available: <https://www.youtube.com/watch?v=n4gOZrUwWmc>
- [40] N. Shelly, E. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, “Flow caching for high entropy packet fields,” in *HotSDN*, 2014.
- [41] N. P. Katta, O. Alipourfard, J. Rexford, and D. Walker, “Infinite cache flow in software-defined networks,” in *HotSDN*, 2014.
- [42] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, “Scalable flow-based networking with DIFANE,” in *SIGCOMM*, 2011.
- [43] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar et al., “The design and implementation of Open vSwitch,” in *NSDI*, 2015.
- [44] “SDN and NFV: Now for the Enterprise Community: Mark Russinovich, Microsoft Azure,” 2015. [Online]. Available: <https://www.youtube.com/watch?v=NVGeYDvoHQ8&feature=youtu.be>
- [45] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani, “Consensus routing: The Internet as a distributed system,” in *NSDI*, 2008.
- [46] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, “Abstractions for network update,” in *SIGCOMM*, 2012.
- [47] R. Mahajan and R. Wattenhofer, “On Consistent Updates in Software-Defined Networks,” in *HotNets*, 2013.
- [48] N. P. Katta, J. Rexford, and D. Walker, “Incremental consistent updates,” in *HotSDN*, 2013.
- [49] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [50] “Project Floodlight,” www.projectfloodlight.org/floodlight/.
- [51] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, “Reproducible network experiments using container-based emulation,” in *CONEXT*, 2012.
- [52] P. Lapukhov and A. Premji, “RFC 7938: Use of BGP for Routing in Large-Scale Data Centers,” 2016.
- [53] “Microburst Monitoring,” http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus6000/sw/qos/7x/b_6k-QoS_Config_7x/micro_burst_monitoring.pdf.

- [54] C. Leiserson, "Fat-trees: Universal networks for hardware efficient super-computing," *IEEE Transactions on Computer*, vol. C-34, no. 10, 1985.
- [55] "WSS Monitoring - Handling Microbursts," 2011, <http://www.vssmonitoring.com/resources/feature-brief/Microburst.pdf>.
- [56] "Monitor Microbursts on Cisco Nexus 5600 Platform and Cisco Nexus 6000 Series Switches," <http://www.cisco.com/c/en/us/products/collateral/switches/nexus-5000-series-switches/white-paper-c11-733020.html>.
- [57] "WSS Monitoring - Handling Microbursts," 2014, <http://www.cisco.com/c/en/us/products/collateral/switches/nexus-5000-series-switches/white-paper-c11-733020.pdf>.
- [58] L. G. Valiant, "A scheme for fast parallel communication," *SIAM journal on computing*, vol. 11, no. 2, 1982.
- [59] "Private discussions with a major switch vendor."
- [60] V. Bollapragada, C. Murphy, and R. White, *Inside Cisco IOS software architecture*. Cisco Press, 2000.
- [61] T. M. Thomas and D. E. Pavlichek, *Juniper Networks reference guide: JUNOS routing, configuration, and architecture*, 2003.
- [62] "Cisco Catalyst 4500 Series Line Cards Data Sheet," 2016, http://www.cisco.com/c/en/us/products/collateral/interfaces-modules/catalyst-4500-series-line-cards/product_data_sheet0900aecd802109ea.html.
- [63] K. Chudgar and S. Sathe, "Packet forwarding system and method using patricia trie configured hardware," July 1 2014, uS Patent 8,767,757. [Online]. Available: <http://www.google.com/patents/US8767757>
- [64] "6800 Series 10 Gigabit and Gigabit Ethernet Interface Modules for Cisco 6500 Series Switches Data Sheet," 2016, http://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/data_sheet_c78-451794.html.
- [65] "High-density, highly available aggregation and intelligent distributed network services at the edge for service providers and enterprises," 2016, http://www.cisco.com/c/en/us/products/collateral/routers/7500-series-routers/product_data_sheet0900aecd800f5542.html.
- [66] "Understanding MX Fabric," 2016, <http://kb.juniper.net/InfoCenter/index?page=content&id=KB23065&actp=search>.
- [67] "Arista Visibility," <https://www.arista.com/en/products/eos/visibility>.

- [68] “LANZ - A New Dimension in Network Visibility,” <https://www.arista.com/assets/data/pdf/TechBulletins/Lanz.pdf>.
- [69] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, “Balanced allocations,” *SIAM journal on computing*, vol. 29, no. 1, 1999.
- [70] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (DCTCP),” *CCR*, vol. 41, no. 4, 2011.
- [71] P. Kumar and S. Meyn, “Stability of queueing networks and scheduling policies,” in *Decision and Control*, 1993.
- [72] M. Allman, V. Paxson, and W. Stevens, “Rfc 2581: Tcp congestion control,” 1999.
- [73] D. P. Bertsekas, R. G. Gallager, and P. Humblet, *Data networks*. Prentice-Hall International New Jersey, 1992, vol. 2.
- [74] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, “Efficient traffic splitting on commodity switches,” in *CoNEXT*, 2015.
- [75] “OMNeT++ Discrete Event Simulator,” <https://omnetpp.org/>.
- [76] “INET Framework,” <https://inet.omnetpp.org/>.
- [77] “Network Simulation Cradle Integration,” https://www.nsnam.org/wiki/Network_Simulation_Cradle_Integration.
- [78] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, “Evolve or die: High-availability design principles drawn from googles network infrastructure,” in *SIGCOMM*, 2016.
- [79] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, “Safe and effective fine-grained TCP retransmissions for datacenter communication,” in *SIGCOMM*, 2009.
- [80] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, “Understanding TCP incast throughput collapse in datacenter networks,” in *WREN*, 2009.
- [81] D. Nagle, D. Serenyi, and A. Matthews, “The panasas activescale storage cluster: Delivering scalable high bandwidth storage,” in *SC*, 2004.
- [82] H. Wu, Z. Feng, C. Guo, and Y. Zhang, “ICTCP: Incast congestion control for TCP in data-center networks,” *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 2, 2013.

- [83] P. Devkota and A. N. Reddy, "Performance of quantized congestion notification in TCP incast scenarios of data centers," in *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.
- [84] J. Zhang, F. Ren, and C. Lin, "Modeling and understanding tcp incast in data center networks," in *INFOCOM*, 2011.
- [85] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems," in *FAST*, 2008.
- [86] J. Hwang, J. Yoo, and N. Choi, "IA-TCP: a rate based incast-avoidance algorithm for TCP in data center networks," in *ICC*, 2012.
- [87] K. Chen, H. Zheng, Y. Zhao, and Y. Guo, "Improved solution to tcp incast problem in data center networks," in *CyberC*, 2012.
- [88] Y. Zhang and N. Ansari, "On mitigating tcp incast in data center networks," in *INFOCOM*, 2011.
- [89] K. Padalia, R. Fung, M. Bourgeault, A. Egier, and J. Rose, "Automatic transistor and physical design of fpga tiles from an architectural specification," in *ACM/SIGDA eleventh international symposium on Field Programmable Gate Arrays*, 2003.
- [90] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the netfpga platform," in *ANCS*, 2008.
- [91] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *ANCS*, 2013.
- [92] A. Sivaraman, S. Subramanian, S. Alizadeh, Mohammad Alizadehand Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. K. Katti, and M. Nick, "Programmable packet scheduling at line rate," in *SIGCOMM*, 2016.
- [93] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *CCR*, 2007.
- [94] M. Mitzenmacher, B. Prabhakar, and D. Shah, "Load balancing with memory," in *FOCS*, 2002.
- [95] D. Shah and B. Prabhakar, "The use of memory in randomized load balancing," in *ISIT*, 2002.

- [96] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, “Devoflow: Scaling flow management for high-performance networks,” in *CCR*, 2011.
- [97] S. Ghorbani and B. Godfrey, “Towards correct network virtualization,” in *HotSDN*, 2014.
- [98] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: A network programming language,” in *ACM SIGPLAN Notices*, vol. 46, no. 9. ACM, 2011, pp. 279–291.
- [99] X. Jin, J. Gossels, J. Rexford, and D. Walker, “Covisor: A compositional hypervisor for software-defined networks,” in *NSDI 15*, 2015.
- [100] I. Pepelnjak, “Real-life sdn/openflow applications,” <http://blog.ipSPACE.net/2013/06/real-life-sdnopenflow-applications.html>.
- [101] I. Pepelnjak, “Defenseflow netflow and sdn based ddos attack defense,” <http://www.radware.com/Products/DefenseFlow>.
- [102] J. Amann and R. Sommer, “SDN based DDoS detection using SciPass and Bro,” in *TNC*, 2015.
- [103] J. Amann and R. Sommer, “Providing dynamic control to passive network security monitoring,” in *RAID*, 2015.
- [104] S. Campbell and J. Lee, “Intrusion detection at 100G,” in *State of the Practice Reports*. ACM, 2011.
- [105] “SciPass: IDS Load Balancer and Science DMZ,” <https://github.com/GlobalNOC/SciPass/releases/tag/1.0.4>.
- [106] A. Sharma, “Bro: Actively defending so that you can do other stuff,” in *BroCon*, 2014.
- [107] D. Y. Huang, K. Yocum, and A. C. Snoeren, “High-fidelity switch models for software-defined network emulation,” in *HotSDN*, 2013.
- [108] W. Liu, R. B. Bobba, S. Mohan, and R. H. Campbell, “Inter-flow consistency: Novel SDN update abstraction for supporting inter-flow constraints,” in *SENT*, 2015.
- [109] N. Lynch, R. Segala, and F. Vaandrager, “Hybrid I/O automata,” *Information and Computation*, vol. 185, no. 1, 2003.
- [110] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal memory: Definitions, implementation, and programming,” *Distributed Computing*, vol. 9, no. 1, 1995.

- [111] Facebook Networking @Scale, “Synchronous Geo-Replication over Azure Tables: A. Greenberg, Microsoft Azure,” <https://code.facebook.com/posts/1421954598097990/networking-scale-recap/>, 2015.
- [112] SDN for the Cloud, “SIGCOMM 2015 Keynote: A. Greenberg, Microsoft Azure,” 2015.
- [113] R. Schwarz and F. Mattern, “Detecting causal relationships in distributed computations: In search of the holy grail,” *Distributed computing*, vol. 7, no. 3, 1994.
- [114] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKown, and G. Parulkar, “Can the production network be the testbed?” in *OSDI*, 2010.
- [115] “COCONUT: Seamless Replication of Network Elements,” Tech. Rep., <http://coconut-project.wikidot.com/>.
- [116] Rob Sherwood, “Modern OpenFlow and SDN,” <http://bigswitch.com/blog/2014/06/02/modern-openflow-and-sdn-part-ii>, 2015.
- [117] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, “Dynamic scheduling of network updates,” in *SIGCOMM*, 2014.
- [118] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu et al., “B4: Experience with a globally-deployed software defined WAN,” in *SIGCOMM*, 2013.
- [119] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” *CCR*, vol. 38, no. 4, pp. 63–74, 2008.
- [120] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, “Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies,” ser. HotNets, 2014.
- [121] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker, “Software-defined internet architecture: decoupling architecture from infrastructure,” in *HotSDN*. ACM, 2012.
- [122] K. He, J. Khalid, S. Das, A. Akella, E. L. Li, and M. Thottan, “Mazu: Taming latency in software defined networks,” Tech. Rep., 2014, <http://minds.wisconsin.edu/handle/1793/68830>.
- [123] H. Kim, T. Benson, A. Akella, and N. Feamster, “The evolution of network configuration: a tale of two campuses,” in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 2011, pp. 499–514.

- [124] J. Gross, T. Sridhar, P. Garg, C. Wright, I. Ganga, P. Agarwal, K. Duda, D. Dutt, and J. Hudson, “Geneve: Generic network virtualization encapsulation,” *IETF draft*, 2014.
- [125] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real time network policy checking using header space analysis.” in *NSDI*, 2013.
- [126] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Godfrey, “VeriFlow: Verifying Network-Wide Invariants in Real Time,” in *NSDI*, 2013.
- [127] B. G. Józsa and M. Makai, “On the solution of reroute sequence planning problem in mpls networks,” *Computer Networks*, vol. 42, no. 2, pp. 199–210, 2003.
- [128] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven WAN,” in *SIGCOMM*, 2013.
- [129] J. McClurg, H. Hojjat, P. Cerny, and N. Foster, “Efficient Synthesis of Network Updates,” in *PLDI*, 2015.
- [130] “Open vSwitch, Set release dates for 2.4.0.” 2015. [Online]. Available: <http://openvswitch.org/pipermail/dev/2015-August/059018.html>
- [131] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
- [132] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, “Split/merge: System support for elastic execution in virtual middleboxes.” in *NSDI*, 2013.
- [133] S. J. Garland, N. A. Lynch, J. A. Tauber, and M. Vaziri, “IOA user guide and reference manual1,” 2003.
- [134] C. Clos, “A study of non-blocking switching networks,” *Bell System Technical Journal*, vol. 32, no. 2, 1953.

Appendix A

MODELING AND PROOFS OF CHAPTER 2

In this section, we prove that in Clos networks DRILL provides network-wide stability and can deliver 100% throughput. We assume that arrival processes are independent and the traffic is *admissible*, i.e., for each leaf switch, its overall arrival rate is less than its overall departure rate (formalized in the assumption in Theorem 2), and for each component, its overall arrival rate for each destination is less than its overall departure rate to that destination (formalized in the assumption in Theorem 3). We prove DRILL's stability and 100% throughput by first proving that every leaf switch is stable and delivers 100% throughput (Theorem 2) and that the spine layer is stable and delivers 100% throughput (Theorem 3). Intuitively, if the traffic is not admissible, the traffic sent to the network has a greater volume than it can transmit and no load balancer is stable. We decouple the overall rate adjustment, typically done by higher layer protocols such as TCP, and load balancing inside the network. DRILL only addresses the second problem and should be used along with a rate control mechanism such as TCP.

We also prove that ESF is optimal for load balancing in any Clos (Theorem 4).

Theorem 2. *DRILL $(1, 1)$ is stable and provides 100% throughput for all admissible independent arrival processes.*

Proof. We first prove the leaf-level switch stability and 100% throughput before proving these properties for the spine layer (Theorem 3). Consider discrete instances of time when there is either an arrival or a departure, since these are the only times that the state of the system changes. We assume the speedup is K . We further assume at each time instance up to M packets arrive at the system according to M independent Bernoulli processes and the arrival rate to the input port i is δ_i ($1 \leq i \leq M$). We denote by μ_i ($1 \leq i \leq N$) the service rate of output port i ($1 \leq i \leq N$). We assume that at most one packet can enter or leave each queue at any given time instance (this assumption can be easily relaxed). Hence, the probability of having an arrival at any given time instant at input i is $\frac{\delta_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i}$.

Similarly, the probability that a departure occurs from output port i at any instant of time is $\frac{\mu_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i}$.

Let $q_k(n)$, $\tilde{q}_i(n)$ and $q^*(n)$ represent the length of the k -th output queue, the length of the output queue chosen by the input i and length of the shortest output queue in the system under policy $(1, 1)$ at time instance n , respectively.

If n is an arrival instant, then the probability that under $(1, 1)$ policy input i chooses the shortest output queue, i.e., $\tilde{q}_i(n) = q^*(n)$, is at least $1/N$. At each time instant, from up to M input ports that are contending for the same output port, at most K of them will be granted permission to direct their packets to that output. If by λ_i we refer to the arrival rate at output port i , then λ_i will be the summation of the arrival rate of these K input ports, and $\sum_{i=1}^N \lambda_i \leq \sum_{i=1}^M \delta_i$. The probability of having a packet forwarded to output port i is $\frac{\lambda_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i}$. Moreover, by Lemma 1, proved in the appendix, we will show that

$$\sum_{i=1}^N \lambda_i \times q_i(n) \leq \sum_{i=1}^M \delta_i \times \tilde{q}_i(n).$$

To prove that the algorithm is stable, using the result of Kumar and Meyn [71], we show that for an $M \times N$ switch scheduled using the $(1, 1)$ policy, there is a negative expected single-step drift in a Lyapunov function, V . In other words,

$$E[V(n+1) - V(n) | V(n)] \leq \epsilon V(n) + k,$$

where $k > 0$ and $\epsilon > 0$. Let $V(n)$ be:

$$V(n) = V_1(n) + V_2(n),$$

where

$$\begin{aligned} V_1(n) &= \sum_{i=1}^M V_{1,i}(n), \\ V_{1,i}(n) &= (\tilde{q}_i(n) - q^*(n))^2, \\ V_2(n) &= \sum_{i=1}^N q_i^2(n). \end{aligned}$$

Since at most K packets can be enqueued at time instance $n+1$ in \tilde{q}_i when the speedup is K ,

$$\tilde{q}_i(n+1) - \tilde{q}_i(n) \leq K.$$

And at most one packet can leave q^* at time instant n . So,

$$-q^*(n+1) + q^*(n) \leq 1.$$

Therefore,

$$\tilde{q}_i(n+1) - q^*(n+1) \leq \tilde{q}_i(n) - q^*(n) + K + 1.$$

Now consider:

$$\begin{aligned} E[V_1(n+1) - V_1(n)|V_1(n)] &= \\ &= \frac{1}{M} \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \\ &\sum_{i=1}^M \delta_i \times ((\tilde{q}_i(n+1) - q^*(n+1))^2 - (\tilde{q}_i(n) - q^*(n))^2) + \\ &\sum_{i=1}^N (1 - \frac{1}{M} \frac{\delta_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i}) \times \\ &((\tilde{q}_i(n+1) - q^*(n+1))^2 - (\tilde{q}_i(n) - q^*(n))^2) \leq \\ &= -\frac{1}{M} \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \sum_{i=1}^M \delta_i (\tilde{q}_i(n) - q^*(n))^2 + \\ &\sum_{i=1}^M (2(\tilde{q}_i(n) - q^*(n)) + K + 1) \leq \\ &= -\frac{1}{M} \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \sum_{i=1}^M \delta_i V_{1,i}(n) + \\ &\sum_{i=1}^M (2\sqrt{V_{1,i}(n)} + K + 1) \end{aligned}$$

So,

$$\begin{aligned} E[V_1(n+1) - V_1(n)|V_1(n)] &\leq \\ &= -\frac{1}{M} \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \sum_{i=1}^M \delta_i V_{1,i}(n) + \\ &\sum_{i=1}^M 2\sqrt{V_{1,i}(n)} + M + MK. \end{aligned}$$

And for V_2 ,

$$\begin{aligned}
E[V_2(n+1) - V_2(n)|V_2(n)] = & \\
& \sum_{i=1}^N \frac{\lambda_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \\
& (q_i(n+1) + q_i(n))(q_i(n+1) - q_i(n)) + \\
& \sum_{i=1}^N \frac{\mu_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \\
& (q_i(n+1) + q_i(n))(q_i(n+1) - q_i(n)) \leq
\end{aligned}$$

At time $n+1$, if we have a packet arrival at input j , *i.e.*, $\delta_j > 0$, and it selects q_i , then $q_i(n+1) = q_i(n) + 1 = \tilde{q}_j(n) + 1$. Hence, $q_i(n+1) - q_i(n) = 1$ and $q_i(n+1) + q_i(n) = 2\tilde{q}_j(n) + 1$. Otherwise, $q_i(n+1) - q_i(n) = 0$. So $(q_i(n+1) + q_i(n))(q_i(n+1) - q_i(n)) = (2\tilde{q}_i(n) + 1)$. Similarly, if a packet leaves queue i , then $q_i(n+1) - q_i(n) = -1$ and $q_i(n+1) + q_i(n) = 2q_i(n) - 1$. Otherwise, $q_i(n+1) - q_i(n) = 0$. Therefore

$$\begin{aligned}
E[V_2(n+1) - V_2(n)|V_2(n)] \leq & \\
& \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \sum_{i=1}^M \delta_i \times (2\tilde{q}_i(n) + 1) + \\
& \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \sum_{i=1}^N \mu_i \times (-2q_i(n) + 1) = \\
& \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \\
& (\sum_{i=1}^M \delta_i \times (1 + 2\sqrt{V_{1,i}} + 2q^*(n)) + \sum_{i=1}^N \mu_i - 2 \sum_{i=1}^N \mu_i q_i(n)) = \\
& \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \\
& [\sum_{i=1}^M \delta_i \times (1 + 2\sqrt{V_{1,i}}) + \\
& 2q^*(n)(\sum_{i=1}^N \delta_i - \sum_{i=1}^N \mu_i) +
\end{aligned}$$

$$2 \sum_{i=1}^N \mu_i (q^*(n) - q_i(n)).$$

So,

$$\begin{aligned} E[V_2(n+1) - V_2(n)|V_2(n)] &\leq \\ &\frac{\sum_{i=1}^N \delta_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} + \\ &\frac{2 \sum_{i=1}^N \delta_i \sqrt{V_{1,i}(n)}}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} + \\ &\frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times 2q^*(n) \left(\sum_{i=1}^N \delta_i - \sum_{i=1}^N \mu_i \right) + \\ &\frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times 2 \sum_{i=1}^N \mu_i (q^*(n) - q_i(n)). \end{aligned}$$

Thus,

$$\begin{aligned} E[V(n+1) - V(n)|V(n)] &\leq \\ &-\frac{1}{M} \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times \sum_{i=1}^M \delta_i V_{1,i}(n) + \\ &2 \sum_{i=1}^M \sqrt{V_{1,i}(n)} \left(\frac{\delta_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} + 1 \right) + \\ &\frac{\sum_{i=1}^N \delta_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} + M + MK + \\ &\frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times 2q^*(n) \left(\sum_{i=1}^N \delta_i - \sum_{i=1}^N \mu_i \right) + \\ &\frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times 2 \sum_{i=1}^N \mu_i (q^*(n) - q_i(n)). \end{aligned}$$

Hence,

$$\begin{aligned} E[V(n+1) - V(n)|V(n)] &\leq \\ &\sum_{i=1}^N \frac{-N(\sum_{i=1}^N \delta_i + \sum_{i=1}^N \mu_i)}{\delta_i} \times \\ &\left(\frac{\delta_i \sqrt{(V_{1,i})}}{N(\sum_{i=1}^N \delta_i + \sum_{i=1}^N \mu_i)} - \left(\frac{\delta_i}{\sum_{i=1}^N \delta_i + \sum_{i=1}^N \mu_i} + 1 \right) \right)^2 \end{aligned}$$

$$\begin{aligned}
& +(M+1) \frac{\sum_{i=1}^M \delta_i}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} + M \frac{\sum_{i=1}^M \delta_i \sum_{i=1}^N \mu_i}{\sum_{i=1}^M \delta_i} + 3M + \\
& MK + \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times 2q^*(n) \left(\sum_{i=1}^N \delta_i - \sum_{i=1}^N \mu_i \right) \\
& + \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times 2 \sum_{i=1}^N \mu_i (q^*(n) - q_i(n)).
\end{aligned}$$

So if we define

$$\begin{aligned}
A_i &= \frac{\delta_i}{N(\sum_{i=1}^N \delta_i + \sum_{i=1}^N \mu_i)} \\
B_i &= \frac{\delta_i}{\sum_{i=1}^N \delta_i + \sum_{i=1}^N \mu_i} + 1 \\
C &= (M+1) \frac{\sum_{i=1}^M \delta_i}{\sum_{i=1}^M \delta_i \sum_{i=1}^N \mu_i} + \\
& M \frac{\sum_{i=1}^M \delta_i \sum_{i=1}^N \mu_i}{\sum_{i=1}^M \delta_i} + 3M + MK
\end{aligned}$$

Then, $A_i \geq 0$ and $B_i \geq 0$ and $C \geq 0$, and

$$\begin{aligned}
& E[V(n+1) - V(n)|V(n)] \leq \\
& \sum_{i=1}^M -\left(\frac{\sqrt{V_{1,i}}}{A_i} - \frac{B_i}{A_i^2}\right)^2 + C \quad (I) \\
& + \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times 2q^*(n) \left(\sum_{i=1}^N \delta_i - \sum_{i=1}^N \mu_i \right) \quad (II) \\
& + \frac{1}{\sum_{i=1}^M \delta_i + \sum_{i=1}^N \mu_i} \times 2 \sum_{i=1}^N \mu_i (q^*(n) - q_i(n)) \quad (III)
\end{aligned}$$

The following upper bounds are easily obtained: $(I) \leq C$ $(II) \leq 0$, since the traffic is admissible. $(III) \leq 0$, by definition of $q^*(n)$. Suppose that $V(n)$ is very large. If $V_1(n)$ is very large, (I) will be negative, from which $E[V(n+1) - V(n)|V(n)] < -\epsilon_1$ for $V_1(n) > L_1$ follows. Otherwise, if $V_1(n)$ is not very large but $V(n)$ is, then $V_2(n)$ should be very large which implies that length of some output queue, $q_i(n)$, is very large. If $q^*(n)$ is not very large, then (III) will be less than $-C$ which is a bounded constant. If $q^*(n)$ is also large, then (II) will be less than $-C$. In both cases, it fol-

lows that $E[V(n+1) - V(n)|V(n)] < -\epsilon_2$ for $V_2(n) > L_2$. Hence, there exist L and ϵ such that $E[V(n+1) - V(n)|V(n) \text{ is very large}] < -\epsilon$ for $V(n) > L$.

The steps above prove the stability of DRILL's scheduling algorithm in the leaf layer. [31] shows that a switch can achieve 100% throughput if it is stable for all independent and admissible arrivals. Hence, leaf switches under DRILL can achieve 100% throughput. Theorem 3 proves the same properties for the spine layer as long as the traffic is admissible inside the component. Together, these two results prove the network-wide stability and the ability to achieve 100% throughput under DRILL for independent admissible arrival processes. \square

Theorem 3. *For every arbitrary source and destination pair, let δ_i and μ_i be, respectively, the arrival and departure rates to the spine switch S_i from the source to the destination. For admissible and independent arrival processes, if the traffic inside a component is admissible, i.e., $\sum_{S_i} \delta_i \leq \sum_{S_i} \mu_i$, then DRILL's failover algorithm is stable and provides 100% throughput inside that component.*

Proof. We prove that for any arbitrary spine S_r in the component, $\delta_r \leq \mu_r$. Hence, each queue is stable and delivers 100% throughput.

For any two spines S_i and S_r to be in the same component, their utilization factors should be equal via S_i and S_r , i.e., $\frac{\mu_i}{\delta_i} = \frac{\mu_r}{\delta_r}$. Therefore, if S_i and S_r are in the same component, $\frac{\delta_i}{\delta_r} = \frac{\mu_i}{\mu_r}$ (note that we can infer from the way components are constructed in DRILL that for any spine S_j in the component, $\delta_j \neq 0$ and $\mu_j \neq 0$, since the leaves should be able to communicate via S_j). Hence, $\sum_{S_i} \delta_i = \sum_{S_i} X_{i,r} \times \delta_r$, and $\sum_{S_i} \mu_i = \sum_{S_i} T_{i,r} \times \mu_r$, where $X_{i,r}$ and $T_{i,r}$ are defined as $X_{i,r} = \frac{\delta_i}{\delta_r}$ and $T_{i,r} = \frac{\mu_i}{\mu_r}$. It derives from the equality of utilization factors that $X_{i,r} = T_{i,r}$ for all S_i s. Plus, since rates are all positive, $X_{i,r} > 0$.

We have $\sum_{S_i} \delta_i - \sum_{S_i} \mu_i = (\delta_r - \mu_r) \times \sum_{S_i} X_{i,r}$. By the condition on the admissibility of traffic, $\sum_{S_i} \delta_i - \sum_{S_i} \mu_i \leq 0$, and we have $X_{i,r} > 0$. Hence, $(\delta_r - \mu_r) \leq 0$. Therefore, each queue at the spine layer of the component is stable and, using the result of [31], can deliver 100% throughput. Note that the length of the queue q_i is proportional to its input minus output rate, $\delta_i - \mu_i$. Hence the result above also shows that the lengths of the spine layer queues are bounded. \square

Theorem 4. *In any Clos network [134], ESF is optimal, i.e., it achieves exactly equal spreading of load across all available shortest paths between any source and destination pairs.*

Proof. We prove this by an induction on the number of intermediary stages.

The base case: In the base 3-tier Clos network [134] with input, output, and one intermediary stages, among a set of paths between a source input switch and destination output switch, each first hop link (from the input switch to the intermediary stage) carries the same load as other paths since the input switch splits the load equally among all available shortest paths, *i.e.*, if there are N switches in the intermediary stage, each link carries $1/N$ of the load from the source switch. Second hop links (from each intermediary switch to the output switch) also all carry equal loads because all input switches split their load equally among all intermediary switches. Hence, each intermediary switch receives $1/N$ of the load destined to the destination output switch. Therefore, all the links from the intermediary stage switches to the destination output switch carry equal load. Thus, overall, all available shortest paths between the source input switch and the destination output switch carry equal load.

The inductive step: ESF is optimal in any T -stage Clos network if we assume that it is optimal for any R -stage Clos, where $R < T$. This statement is true because (a) each input switch splits the load exactly equally among all intermediary stages (definition of ESF). This implies that the load that each intermediary “level” [134]¹ receives is exactly equal. So the first hop load on all paths are equal. (b) Each of these levels is a smaller Clos [134]. Thus, by the hypothesis of induction, ESF is optimal inside each of these smaller Clos, *i.e.*, the first stage switches in each level, balances the load exactly equal among all paths inside that level. So the load on all hops except the first and last hops are equal. (c) Since each level receives exactly equal share of traffic to each destination output switch (part (a)), the last stage of all levels receive exactly equal traffic for each output stage switch. So the last hop load on all paths are equal. Therefore, the overall load on the paths are exactly equal. \square

Lemma 1. $\sum_{i=1}^N \lambda_i \times q_i(n) \leq \sum_{i=1}^M \delta_i \times \tilde{q}_i(n)$.

Proof. Let us define $\rho_{i,j}$

$$\rho_{i,j} = \begin{cases} \delta_j & \text{input } j \text{ chooses output } i \\ 0 & \text{otherwise} \end{cases}$$

¹In a 5 stage Clos, each level consists of three intermediary stages. In a 7 stage Clos, each level consists of five intermediary stages, etc. [134].

It immediately follows that

$$\lambda_i \times q_i(n) \leq \sum_{j=1}^M \rho_{i,j} \times \tilde{q}_j(n).$$

So, $\sum_{i=1}^N \lambda_i \times q_i(n) \leq \sum_{i=1}^M \sum_{j=1}^N \rho_{i,j} \times \tilde{q}_j(n)$. But since the input ports can compete for only a single output port at a time, the term $\rho_{i,j}$ can be non-zero only for at most N pairs of (i, j) . It follows that

$$\sum_{i=1}^N \sum_{j=1}^M \rho_{i,j} \times \tilde{q}_j(n) = \sum_{i=1}^M \delta_i \times \tilde{q}_i(n).$$

So,

$$\sum_{i=1}^N \lambda_i \times q_i(n) \leq \sum_{i=1}^M \delta_i \times \tilde{q}_i(n).$$

□

Appendix B

MODELING AND PROOFS OF CHAPTER 3

The goal of this section is to prove observational correctness of COCONUT . We state the assumptions, and provide a few definitions and lemmas that assist us with proving the theorem.

Assumptions about replication: As stated earlier, we assume that each physical *instance* is individually capable of fully implementing the virtual rule and a packet that is supposed to be handled by a virtual rule will be handled by at most one instance.

Mapping logical rules to multiple physical rules: In simple replication virtual networks, each logical rule, LR , can have multiple physical instances, $PR_i, i \geq 0$, in the network. We assume that there is a total ordering between rule versions (version numbers shown by integers; higher values indicating newer versions), and different entities, such as the controller, can update rules. It is possible for instances of a single logical rule to be inconsistent, *i.e.*, have different versions. This can happen, for example, when the controller is in process of updating the instances of a logical rule. We denote the version of instance of the logical rule LR that handled packet pkt by $v(LR, pkt)$. If no instance of LR is applied on pkt then $v(LR, pkt)$ is not defined. Virtualization systems use a combination of rule placement and packet directing techniques to make sure that if a packet is supposed to be handled by a logical rule, at most one instance of that rule is applied on it. In NVP, for instance, logical datapaths for communications between all pairs of VMs are computed and are implemented on the software switch where the originating VM resides.

While in non-virtual networks, for packets p and q , $p \rightarrow q$ implies that the rules applied on q are at least as updated as those applied on p , this property does not automatically hold under simple replication because the instances handling p and q could be different. We define *causality awareness* to formalize this concept.

We define a *causality chain*, $e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_n$, to be chain of 0 or more events or packets iff for any two consecutive packets or events e_i and e_{i+1} , $0 <$

$i < n$ in it, the following two conditions hold: (a) $e_i \rightarrow e_{i+1}$ and (b) there does not exist any packet or event f such that $e_i \rightarrow f \rightarrow e_{i+1}$. Trivially, $p \rightarrow q$ iff there is a causality chain starting with p and ending at q .

A network is called *causality aware* iff for any two packets p and q and for any logical rule LR , if $v(LR, p)$ and $v(LR, q)$ are defined, then $p \rightarrow q$ implies that $v(LR, p) \leq v(LR, q)$.

Lemma 1.1: In COCONUT's high-level algorithms, for every two packets p and q , if $p \rightarrow q$, then $VC_p \leq VC_q$, i.e., each dimension of the VC of q is at least as large as that of p .

Proof is a simple induction on the length of the causality chain, n :

- **Basis:** The statement trivially holds for $n = 0$, i.e., when the length of the causality chain is zero.
- **Inductive step:** We show that if the statement holds for any causality chain with length n , then it holds for any causality chain of length $n + 1$. This can be done as follows:

For a given causality chain $e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_n \rightarrow e_{n+1}$, we want to prove that $VC_{e_j} \leq VC_{e_{n+1}}, \forall j \leq n$. We claim that $VC_{e_n} \leq VC_{e_{n+1}}$, because by definition of \rightarrow , $e_n \rightarrow e_{n+1}$ either they happen at the same end-point and e_n comes before e_{n+1} , or e_n and e_{n+1} are the same packet (sent at one end-point and received at another). In both cases, $VC_{e_n} \leq VC_{e_{n+1}}$ (I). By the assumption of the inductive step we know that $VC_j \leq VC_{e_n}, j < n$ (II). I and II imply that $VC_{e_j} \leq VC_{e_{n+1}}, j \leq n$.

Lemma 1.2: In COCONUT's OpenFlow algorithms, after receiving a packet p with a TB tagged (e.g., processed by the newer version of an in flux rule), all packets $q, p \rightarrow q$ will be tagged at TB as long as the rule is in flux.

Proof is a straightforward induction on the causality chains' lengths.

- **Base case** (empty causality chains) trivially holds.
- **Inductive step:** We assume that the statement holds for any causality chain of length n , and prove it for an arbitrary causality chain of length $n + 1$, $p_1 \rightarrow p_2 \rightarrow p_3 \dots \rightarrow p_n \rightarrow p_{n+1}$, where the first packet (p_1) is a tagged received packet. Let $s_{i,j}(m)$ be the send-to event that results in p_{n+1} being received. Assumption of the inductive step implies that packet p_n is

tagged at TB. By definition of \rightarrow , p_n and p_{n+1} either (a) happen at the same end-point or (b) they are the same packet (send and received by potentially different end-points). In both cases p_n being tagged at TB implies that p_{n+1} will also be tagged: in case (a) since the rule is in flux, no shell (including the shell where the tagged p_n happened) stops tagging. In case (b), the network does not use the TBs; during the time that the rule LR is in flux, COCONUT does not use that TB for other rules, and the LR instances do not remove the TBs, *i.e.*, they do not set $TB = 0$. Hence, being sent as a tagged packet guarantees that the packet will be received tagged.

Given Lemmas 1.1 and 1.2, it is easy to see that COCONUT is causality aware:

Theorem 2: COCONUT is causality aware.

Proof is by contradiction; assume that COCONUT does not provide causality awareness, *i.e.*, there are packets p and q and a logical rule LR such that $v(LR, p)$ and $v(LR, q)$ are defined, and $p \rightarrow q$ but $v(LR, p) > v(LR, q)$. We call the events associated with p and q , respectively, e_1 and e_2 . The following two cases are possible:

- At least one of the two events e_1 or e_2 is a *send* event. Let's call the packet associated with this *send* event a ($a \in \{p, q\}$). a is not handled by any rule yet (given the assumption that each packet is unique). Therefore, $v(LR, a)$ is undefined. But this contradicts with the assumption that both $v(LR, p)$ and $v(LR, q)$ are defined.
- Both e_1 or e_2 are *receive* events.
 - High-level algorithms: By Lemma 1, if $p \rightarrow q$, then $VC_p \leq VC_q$ which implies that $VC_p[i] \leq VC_q[i]$ where i is in the index of LR . $VC_p[i] \leq VC_q[i]$ causes the same or newer versions of LR to be applied on q compared to p , *i.e.*, $v(LR, p) \leq v(LR, q)$ which contradicts with the assumption.
 - OpenFlow algorithms: By the assumption, the rule LR is in flux (it has different versions applied on packets) and the newer version is applied on p . In the OpenFlow algorithms, these imply that p is tagged by a TB used for LR . By Lemma 2, q will also be tagged. A tagged packet is not handled by old instances. Therefore, either q is not handled by an instance of LR , *i.e.*, $v(LR, q)$ is not defined, or it is handled by an

instance at least as updated as the one applied on p , *i.e.*, $v(LR, q) \geq v(LR, p)$. Both cases contradict our assumption.

It is easy to see that if no rule changes, then simple replication provides a correct virtualization, *i.e.*, any trace in it is a plausible trace in the logical network.

Lemma 3: Any static simple replication is correct. When no rule changes, simple replication is a correct virtualization.

The proof is by induction on the traces' lengths and follows almost immediately from simple replication definition.

- **Base case:** The empty trace (length = 0) is a plausible trace in any network, including simple replication and logical network. Hence, the simple replication network trace's with length 0 is a plausible trace in the logical network.
- **Inductive step:** We prove that if any trace of simple replication with length $m \leq n$ is a plausible trace in the logical network with no rule change, then any trace of length $n + 1$ of simple replication is also a plausible trace in the logical network assuming that the rules do not change.

For any given trace of length $n + 1$, $\langle e_1, e_2, \dots, e_n, e_{n+1} \rangle$, either (a) e_{n+1} is a *send* event, or (b) it is a *receive* event. If it is a *send* event, then the trace is plausible, because $\langle e_1, e_2, \dots, e_n \rangle$ is a plausible trace (assumption) and end-points are allowed to send any packets at any time. If it is a *receive* event and if it does not have any dependencies, it can happen at any time. Therefore, $\langle e_1, e_2, \dots, e_n, e_{n+1} \rangle$ is plausible. If it is a *receive* event and it has dependencies, let e_i be that last event in the trace such that $e_i \rightarrow e_{n+1}$, *i.e.*, $\forall j > i, e_j$ and e_{n+1} are concurrent. In that case, by the assumption of the inductive step, $\langle e_1, \dots, e_i \rangle$ is a plausible trace. Moreover, by definition of \rightarrow , e_i must be a *send* event. Given the assumptions about simple replication (the traffic is directed to at most one instance of the appropriate logical rules) and the fact that rules are not changing, the exact same rules that would handle the packet sent by e_i in the logical network will handle the packet in simple replication. Therefore, $\langle e_1, \dots, e_i, e_{n+1} \rangle$ is a plausible trace. Thus, given that e_{n+1} is concurrent with all $e_j, j > i$ and by Lemma 3.2, $\langle e_1, \dots, e_i, e_{n+1}, e_{i+1}, \dots, e_n \rangle$ is also plausible, and by Lemma 3.1., $\langle e_1, \dots, e_i, e_{i+1}, \dots, e_n, e_{n+1} \rangle = \langle e_1, e_2, \dots, e_n, e_{n+1} \rangle$ is also plausible.

The following lemmas follow almost immediately from the best-effort networks' properties — the fact that they can drop packet, arbitrary delay delivery of packets, and reorder the packets that have no dependencies.

Lemma 3.1: In best effort networks, if e_1, e_2, \dots, e_n is a plausible trace and for two events/packets, e_i and $e_j, i < j$ (i.e., $e_1, e_2, \dots, e_i, \dots, e_j, \dots, e_n$), e_i is concurrent with all $e_k, i < k \leq j$, then the trace resulting from shuffling e_i and e_j (i.e., $e_1, e_2, \dots, e_j, \dots, e_i, \dots, e_n$) is also plausible.

This holds because reordering of the packets and events that are not dependent in best-effort networks is permissible.

Lemma 3.2: In best effort networks, if $E_1, r_{i,j}(pkt)$ and E_1, E_2 are plausible traces, and $r_{i,j}(pkt)$ is concurrent with all the events in E_2 then $E_1, r_{i,j}(pkt), E_2$ is also a plausible trace.

This holds because best-effort networks can delay delivery of independent packets, e.g., delivery of pkt of event $r_{i,j}(pkt)$ and the E_2 postfix events.

Lemma 3.3: In any best effort network, if E is a plausible trace, and e is a *send* event, then E, e is also a plausible trace.

End-points can send packets at any time. It is the *receive* events that might make a trace non-plausible, and E, e does not include e 's corresponding *receive* (if any), given that e is the last event.

Theorem 1: COCONUT is correct, i.e., any trace in COCONUT is a plausible trace in the logical network that it implements.

By Lemma 3, we know that the incorrect behavior might happen only if rules change. We prove that even when rules change, COCONUT is correct, i.e., any trace of it a plausible trace of the logical network. Proof is by induction on the length of traces.

- **Base case:** Empty trace (length=0) is a plausible trace in any network, including the simple replication network with COCONUT and the logical network. Hence, this trace is a plausible trace in the logical network.
- **Inductive step:** We prove that if all traces of length $m \leq n$ in COCONUT are plausible traces in the logical network, then any trace of length $n + 1$ in COCONUT will also be a plausible trace in the logical network.

For any given COCONUT trace of length $n + 1$, $\langle e_1, e_2, \dots, e_n, e_{n+1} \rangle$, $\langle e_1, e_2, \dots, e_n \rangle$ is a plausible trace (assumption). Thus, if e_{n+1} is a *send* event or a *receive* event without dependency, then the trace is plausible because end-points can send any packet at any time or receive packets that do

not depend on other packets or events such as packets informing them of link failures. If e_{n+1} is a *receive* event with dependency, then assume the last event happening before e_{n+1} in the trace is $e_i, i \leq n$, i.e., $e_i \rightarrow e_{n+1}$. By definition of \rightarrow , e_i must be a *send* event. By the assumption of the inductive step, $\langle e_1, \dots, e_i \rangle$ is a plausible trace, i.e., it could have happened in the logical network. simple replication directs the packet sent by e_i only to the instances of the logical rules that this packet would be forwarded to if it was being sent in a non-virtual network. By Theorem 2 we know that these instances are at least as uptodate as the instances applied on prior packets. $\langle e_1, \dots, e_i, e_{n+1} \rangle$ will, therefore, be plausible. Given that e_{n+1} and $e_j, j > i$ events are concurrent, and by Lemma 3.2, $\langle e_1, \dots, e_i, e_{n+1}, e_{i+1}, \dots, e_n \rangle$ is plausible, and by Lemma 3.1, $\langle e_1, \dots, e_i, e_{i+1}, \dots, e_n, e_{n+1} \rangle$ is plausible.