EFFICIENT AND SAFE MIGRATION OF NETWORK FUNCTIONS USING SOFTWARE-DEFINED NETWORKING

Sheng Liu

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2021

Approved by:

Michael K. Reiter

Jasleen Kaur

Prasun Dewan

Rob Sherwood

Theophilus A. Benson

ABSTRACT

Sheng Liu: Efficient and Safe Migration of Network Functions Using Software-Defined Networking
(Under the direction of Michael K. Reiter)

Network function (NF) migration alongside (and possibly because of) routing policy updates is a delicate task, making it difficult to ensure that all traffic is processed by its required network functions, in order. Achieving traffic redistribution while ensuring correct processing of all packets requires an efficient network forwarding-state update and careful coordination between routing-policy change and NF migration. To achieve consistent network updates, in this dissertation, we propose a new method that is inspired by *causal consistency*, a consistency model for shared-memory systems. We propose and analyze a property called *suffix causal consistency* (SCC) as an interpretation of causal consistency for rule updates in an SDN network. We design an algorithm implementing this property and formally verify the correctness of this algorithm using model checking. Our evaluation results show that SCC provides greater efficiency than competing consistent-update alternatives while offering consistency that is strong enough to ensure high-level routing properties (e.g., black-hole freedom).

To coordinate routing-policy updates with NF migration, we propose a design called Nimble for interleaving these tasks to achieve more efficient completion of both while ensuring complete processing of traffic by the required sequences of NFs. Our technique works with any route-update protocol that implements a property we call *relaxed waypoint correctness*, which includes our SCC algorithm and many consistent-update protocols. We also provide a route-update protocol that is customized to achieve relaxed waypoint correctness without conforming to conventional "consistent update" semantics, as typically defined for such pro-

tocols. We confirm the sufficiency of relaxed waypoint correctness using model checking, and the implementation demonstrates the efficiency and efficacy of Nimble.

To my family.

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude to my advisor, Prof. Michael Reiter, for providing me an opportunity to be his student and for his patience, motivation, and knowledge. I enjoy the time discussing with him, and his guidance helped me all the time at UNC. I thank my other committee members: Prof. Theophilus Benson, Dr. Rob Sherwood, Prof. Jasleen Kaur, and Prof. Prasun Dewan, for their valuable advice, insightful comments, and precious time they committed to serving on my committee.

I am grateful for all my collaborators, including Prof. Vyas Sekar, who helped me a lot during the first two years of my Ph.D. study; Prof. Theophilus Benson, who can always give me valuable suggestions for my research; Dr. Muntasir Rahman, who provided me an opportunity to join Bell Labs as an intern; and Dr. Lalita Jagadeesan and Robert Hanmer who gave me helpful feedback throughout my internship.

Thanks to all my friends, without whom my time at UNC would not be nearly as delightful. In particular, thanks to Andrew Chi, Adam Humphries, Dr. Victor Heorhiadi, Jun Jiang, Marie Nesfield, Ke Coby Wang, Dr. Qiuyu Xiao, and Dr. Ziqiao Zhou. I have been very fortunate to be part of the team.

Finally, I would like to thank my parents, Xianxing Liu and Baoying Yao, who always support and encourage me. I thank my wife Dr. Ziqiao Zhou, who supports me and endures me nonetheless. I could never have done this without them. I love you all.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Network functions (NFs), or middleboxes, are a staple of modern network infrastructures. These middleboxes play a critical role in the network and can perform a variety of functions on packets, such as access control, load balancing, and intrusion detection. For example, network intrusion detection/prevention systems (NIDS/NIPS) enjoy widespread use and are central to supporting secure and efficient operations of the network. Network service providers traditionally deploy these NFs using proprietary physical devices and maintain strict chaining or orders that must be reflected in the position of NFs. As modern networks require ever-increasing flexibility and scalability for network function deployment, the traditional dedicated hardware equipment suffers from significant reconfiguration overhead and lacks the elasticity for service providers to provide dynamic and high-quality services. To meet the required elasticity, network function virtualization (NFV) has emerged to replace traditional dedicated hardware devices with software applications running in virtual machines (e.g., VMware workstation) or containers (e.g., docker).

The main idea of NFV is decoupling of the physical network device from the network services that run on it, such that these services can be realized on commodity hardware using virtualization. This enables a specific chain of services to be decomposed into multiple virtual network functions, which can be packaged as software running on one or more industry-standard servers located in data centers or at the network edge. As a result, service providers can deliver these services on demand without purchasing new hardware. As the progression of server virtualization and network virtualization, NFV is rapidly emerging and creating a brand new market. The global network function virtualization market size is forecasted to grow from 12.9 billion dollars in 2019 to 36.3 billion dollars by 2024 [74]. Many companies have launched their products to provide NFV services (e.g., VMware vCloud NFV [97],

Cisco NFVI [14]) or leverage NFV techniques on their platforms (e.g., Microsoft Azure [67], Amazon AWS [4]).

A significant benefit of NFV is that it can dynamically scale-out a particular virtualized network function by increasing the number of VM/container instances in response to the real-time traffic load. Realizing such elasticity requires moving the network function to a new position with corresponding states and redistributing traffic among multiple NF instances. The speed of network update and NF migration is crucial because it determines the agility of the network control. If the network adapts to traffic volume changes, a slow update can result in a long period during which network functions are overloaded or underutilized. It further leads to degraded network performance or resource waste. Therefore, the effectiveness of NFV depends on how fast the traffic redistribution and NF migration can be achieved.

Efficient traffic reassignment needs rapidly determining new routing policies based on network topology and updating forwarding rules on switches to route traffic through the desired sequence of network functions correctly. However, achieving fast network updates is challenging in the traditional network. First, it is complicated and hard to reconfigure network devices to enforce new routing policies because network operators have to leverage vendor-specific low-level commands manually. This operation is performed separately on each device and thus can be very time-consuming and error-prone. Second, network environments have to endure frequent changes in both traffic volume and topology. Traditional network lacks the elasticity and programmability to adapt to network changes.

Software-defined networking (SDN) provides a promising alternative for deploying network updates with improved network management and network programmability. Specifically, SDN separates the network into an "intelligent" control plane and a "dumb" data plane that implements a standardized interface for the controller to change the network forwarding functions. The control plane, typically a logically centralized controller, stores a global view of network topology information and is responsible for configuring switches in the data plane. Each switch forwards packets based on the instructions sent by the controller

using a standard configuration protocol, e.g., OpenFlow [77]. Such a decoupled architecture provides fine-grained network control and a high degree of network programmability.

Although SDN started as an academic proposal, due to its benefits, it has drawn lots of attention in the industry over the past few years. Many vendors (e.g., Cisco, Barefoot) support OpenFlow on their commercial switches. For example, Pica8 [81] is the first hardware-independent switch to support the OpenFlow standard. Moreover, a lot of large companies have leveraged SDN in their core network. Google, for instance, has deployed an SDN-driven WAN to connect its data centers across the planet. This network has been in production for many years and helps the company serve more traffic and save costs [37]. AT&T announced in 2019 that 75% of the traffic traversing their MPLS tunnels is now under the control of SDN, and the number will hit 100% soon [6]. In conclusion, SDN is promising from the perspective of both industry and academia.

Though SDN offers a cost-effective way to configure network devices, an inconsistent network update will result in packet loss or, worse, violations of network policies. For example, due to the varying delays between the controller and switches, switches cannot be updated atomically, which may cause traffic to sidestep intended NFs. This issue becomes even challenging when packets need to traverse a sequence of NFs, and traffic is redistributed among NF instances. The traditional approach to update consistency [86], on which most other update mechanisms [32, 39, 80, 54, 44, 71] build, is atomic in nature — packets either traverse the old path or the new path, but never both. Some improvements in this vein focus on reducing overheads [75, 44, 93] or congestion [54, 32]; others focus on finding better update orderings [80, 52, 39, 59, 60, 21, 22]. Despite these improvements, enforcing atomicity places a fundamental limit on the speed with which the network can be updated by forcing packets (or flows) to wait until the new path is completely updated before it can be used. Additionally, this requirement forces rules for both the new and old paths to co-exist, costing efficiency. Inefficiency is not the only reason why these approaches is not a good fit for NFV. Most prior works on SDN routing-policy updates that ensure packets traverse intended NFs

assume that NFs remain at fixed locations of the network during the routing-policy update, and thus they cannot be naively used in scenarios where NFs can move from one position to another.

To ensure complete and correct processing of packets by intended NFs, the controller requires more than a consistent and efficient routing-policy update algorithm. For example, suppose the routing-update algorithm guarantees that all packets traverse the old or new position of each NF by ensuring each packet is sent through either its old or new path in its entirety. In that case, some NF packets may still not be processed if they arrive at the new position before NF migration is completed or if they arrive at the old location after NF migration begins. Therefore, ensuring that all packets get processed by their intended NFs needs additional coordination between migrating NFs to their new locations and adjusting traffic routing policies. To our knowledge, all works proposed to do so (e.g., [24, 85, 23, 61]) coordinate these actions by performing network forwarding-state update strictly after NF migration is finished. While these techniques are capable of being used with arbitrary route-update protocols, their generality slows down the process of flow redistribution longer than necessary, possibly delaying rectification of the issue that required the routing-policy update in the first place.

This dissertation aims at providing solutions to these challenges and contains three components.

- To achieve efficient network update, in Chapter 3 we propose an alternative consistent update abstraction in which packets are allowed to traverse a combination of both old and new path, thus relaxing the consistency model and speeding up the update times. Our approach is inspired by *causal consistency* [1], a consistency model for shared-memory systems that guarantees that processes (in our case, packets) observe operations (in our case, rules) in a causal order. Applied to SDNs, causal consistency would imply that once a packet is matched to ("reads") a forwarding rule in a switch, it can be matched in downstream switches only to rules that are equally or more

up-to-date. We propose and analyze a relaxed version of this property called *suffix causal consistency* (SCC). We present an algorithm that implements this property without updating switches unnecessarily, and show that SCC provides greater efficiency than competing consistent-update alternatives while offering consistency that is strong enough to ensure high-level routing properties (black-hole freedom, bounded looping, waypoint enforcement).

- To ensure correct processing of packets, in Chapter 4, we propose a design called Nimble for interleaving routing-policy update and NF migration using software-defined networking (SDN), in a way that significantly reduces the latency to achieve both and without permitting packets to evade processing by NFs. Our technique works with any route-update protocol that implements a property we call *relaxed waypoint correctness*, which includes consistent-update protocols like CU [86] and our proposed algorithm SCC. However, we provide a route-update protocol that is customized to achieve relaxed waypoint correctness without conforming to conventional "consistent update" semantics, as typically defined for such protocols. The benefits of Nimble are myriad, including lower latency for completion of both tasks and, depending on the routing-update protocol with which NF migration is being deployed and the circumstances requiring their update, reduced packet loss and/or reduced rule overhead in switches.

- It is difficult to verify the correctness of our algorithm considering the complex network setting with all possible switch states and varying delays between the controller and switches. In Chapter 5, we construct a model using Z3 solver [72] and verify the correctness of both SCC and Nimble. Specifically, we verify the enforcement of suffix causal consistency, as well as other high-level routing properties (black-hole freedom, etc.) for the SCC algorithm by exploring all possible switch configurations and arbitrary latencies for switch updates to occur. Also, we verify the correctness of Nimble by

exploring all possible delays between the controller and switches with any route-update protocol satisfying the property of relaxed waypoint correctness.

Together, SCC and Nimble support the following thesis statement: *Consistent network-forwarding state update can be achieved efficiently through an appropriate adaptation of causal consistency for the network setting. Interleaving this network-update method and NF migration can significantly speed up the completion of both tasks while ensuring correct processing of traffic.*

CHAPTER 2: BACKGROUND AND RELATED WORK

In this chapter, we describe background on software-defined networking and network function migration – two fields included by this dissertation. We also summarize closely related works on consistent network updates, causal consistency, and model checking.

2.1 Software-Defined Networking

Software-defined networking emerged to facilitate network management by centralizing and simplifying network control. It provides a new network paradigm by decoupling the network's control plane and the data plane. Specifically, SDN uses a logically centralized controller to maintain the network information base and manage underlying routers and switches. Routers and switches provide the controller a well-defined programming interface (e.g. OpenFlow [77]) to manipulate packet-forwarding logic. Each switch maintains one or more flow-entry tables which store packet-handling rules matching specific traffic and performing certain actions (e.g., dropping, forwarding). These rules can be updated by the controller such that switches can perform certain network functions (e.g., routing, load balancing, filtering traffic). The separation between the specification of network policies and their implementation in hardware devices offers flexibility and programmability for simplifying network management. Therefore, SDN significantly saves time and effort for network operators to reconfigure network equipment. Besides, it also shortens the development cycle to implement novel network management protocols.

2.1.1 SDN Applications

SDN has been widely used to implement network applications and optimize resource utilization. The flexibility of SDN facilitates the deployment of a variety of applications for traffic routing [96, 98], network measurement [56, 99] and middlebox deployment [84, 5]. Fibbing [96] offers central control over distributed routing protocols, such as OSPF, by in-

troducing fake links and nodes. Google uses Espresso [98], an Internet peering edge routing infrastructure using SDN, to achieve fine-grained BGP-compliant bandwidth management. Besides the routing control, SDN can also be used to facilitate network measurement. OpenSketch [99] separates the control plane from the measurement data plane to efficiently support a wide variety of measurement tasks. UnivMon [56] offers both high accuracy and generality for flow monitoring. Moreover, SDN provides an alternative way to enforce middlebox deployment. Simple [84] uses SDN to steer traffic through a sequence of middleboxes while balancing the load across middleboxes. Slick [5] handles both the placement of network functions and traffic routing through the elements.

SDN can also be leveraged for network resource management since it simplifies the network control. Google uses SDN to deploy its backbone WAN named B4 [37] so that it can leverage centralized traffic engineering to maximize link utilization. ElasticTree [90] utilizes a centralized power manager to handle traffic loads and save up energy cost dynamically. SWAN [32] achieves high bandwidth by applying frequent congestion-free network updates. A more comprehensive survey on SDN can be found in this paper [50].

2.1.2 Programmable Switches

In recent years, a new high-level packet-processing language called P4 [9] has been developed to offer flexibility for programmable switches in the SDN data plane. Specifically, programmers can customize the forwarding logic of switch hardware by extracting specific packet headers, defining match and action formats for tables, and specifying the order in which these tables process packets. Programmable switches offer high programmability as well as high packet processing rates. A wide interest has arisen, and much effort has been made in both industries (e.g., Barefoot Networks' Tofino [94]) and academia (e.g., PISCES [89]). Barefoot Networks' hardware switch Tofino supports P4 language and can achieve up to 12.8 Tb/s throughput. PISCES is a software switch derived from OpenvSwitch [78], that is customized for the P4 language.

The flexibility and programmability of P4 significantly speed up the implementation and

deployment of new protocols and algorithms. Hula [43] uses programmable switches to implement a load balancing mechanism. Contra [34] achieves performance-aware routing based on real-time link conditions and allows network operators to specify user-defined performance objectives. NetCache [38] leverages switches to cache frequently-accessed items and balance loads across key-value stores. Since it is convenient to define new packet headers with P4 language, in this dissertation, we use P4 language to implement our SCC algorithm in Chapter 3.

2.2 Consistent Network Updates

Networks require fast and efficient updates for various reasons, ranging from planned maintenance to unexpected failures. Meanwhile, network operators need to establish certain correctness conditions, such as black hole freedom or enforcing complex security policies, to achieve successful and safe network operation. As modern networks are changing continually, it is critical to maintaining these conditions even during transitions. Therefore, network systems require consistent updates that preserve certain properties during transitions between two operator-specified configurations. Consistent network update in SDN networks has recently received considerable attention (e.g., [86, 39, 59, 66, 70, 71, 44, 75, 62, 103]). Most approaches provide either strong consistency in the sense that packets traverse either the old path or the new path (but not a mix) [86, 71, 44] or ensure specific properties (e.g., loop freedom, congestion freedom) via weaker, transient consistency [59, 60, 39, 21, 93, 22, 32, 54].

The earliest work of the former class is Consistent Update (CU) [86], which uses a two-phase commit to apply rule updates atomically across the network and requires each switch to temporally maintain both old rules and new rules during the update. In addition, a new rule configuration cannot be applied to packets until it is confirmed as having reached all switches. Tal *et al.* [71] leverage synchronized clocks among the controller and switches to implement time-triggered Consistent Update and thus speeding up the process. These approaches can guarantee that traffic is processed by either an old or a new configuration, but not a mix of the two. However, though CU is capable of being used with any configuration, it

has significant rule-space overhead by storing both old and new rules at switches. To reduce memory overhead, Naga *et al.* [44] propose performing a two-phase commit in multiple steps to update the network incrementally.

Another class of solutions performs network updates incrementally and focuses on specific properties via weaker consistency. An example is Transiently Secure Network Updates (TSU) [59], which provides loop freedom and waypoint enforcement properties, implemented by scheduling updates to the network in multiple steps. Only a subset of switches are updated in each step, and the next step must wait for the completion of the previous one. Specifically, TSU computes the optimal update schedules using mixed-integer programming. Klaus-Tycho *et al.* [21] propose an algorithm to perform network updates in minimal steps and meanwhile achieving loop-freedom property. zUpdate [54] and SWAN [32] perform congestion-free network updates to maximize link utilization. Klaus-Tycho *et al.* [22] discuss the tradeoff between various consistency properties and the speed to update the network. Also, some works [39, 103] allow network operators to customize correctness properties that should be satisfied during the transitions. Dionysus [39] proposes an algorithm to generate a dependency graph among updates for a specific property and compute an update schedule to increase the update speed. Similar to Dionysus, CCG [103] supports general network properties and can be applied to applications with wildcarded rules.

We compare our proposed network-update algorithm SCC empirically to CU and TSU in Sec. 3.5, but the lessons we draw from that comparison, we believe, apply more broadly to the classes of solutions they represent: approaches (like CU) that ensure that packets traverse either the old path or the new path (but not a mix) come at a significantly greater network update delay and transient rule-storage overhead than our approach, and those that ensure specific network properties via weaker transient consistency (like TSU) tend to scope their targeted properties narrowly and still may incur significantly greater network update delay than our approach, due to their multi-stage strategies. As we will show, our approach incurs low delay by avoiding multi-step deployment strategies and implements a

property, namely *suffix causal consistency*, that implies a broad range of useful properties during routing changes.

2.3 Causal Consistency

Suffix causal consistency is inspired by causal consistency [1], a consistency model for shared-memory systems. Informally, a system is causally consistent if reads return values consistent with any reads and writes that could have influenced them (in the sense of Lamport's *potential causality* relation [51]). Causal consistency is widely used to ensure consistency and high availability of data objects with minimal delay across a wide-area network [57, 58, 7]. For example, COPS [57] implements a scalable distributed key-value wide-area system that performs read and write operations in a local data center in a linearizable way and replicates data among datacenters satisfying causal consistency. Specifically, COPS explicitly tracks the causal dependencies among keys and checks whether these dependencies are satisfied before executing the write operation to a local datacenter. Similar to COPS, Eiger [58] also uses causal consistency to provide a scalable, consistent geo-replicated storage system. However, different from COPS that tracks the dependencies on versions of keys, Eiger maintains dependencies on operations. Peter *et al.* [7] develop a bolt-on framework to achieve causal consistency based on eventually consistent stores such that safety and liveness concerns can be separated. Our work adapts the causal consistency property to network routing, introducing improvements to reduce the extent and hence delays associated with network updates.

Due to its shared basis in causal consistency, in Sec. 3.5 we will also empirically compare our SCC algorithm to COCONUT [25], which seeks to enable seamless scaling of logical network elements to multiple physical replicas. The core technical problem that COCONUT tackles are that naive replication can result in incorrect behavior by a logical component during routing updates if one physical replica applies an old policy to packets that depend causally on packets to which another replica applied a new policy. COCONUT thus leverages (compressed) vector timestamps [20, 64] in packets, with one component per logical rule

undergoing an update, to signal to physical replicas the rule version that other replicas previously applied to flows on which this packet causally depends, enabling them to apply an equally current version. COCONUT thus ensures that each replicated logical element respects causal relationships between flows (though not across distinct logical elements). In contrast, our SCC algorithm does not focus on causal relationships between flows, but instead ensures that each flow "reads" (is matched to) rules in causal order across the network elements, seamlessly transitioning it from old routing policy to new. Despite the somewhat different goals of COCONUT, we will coerce it to implement our goals in Sec. 3.5 and compare SCC to it empirically, as another point in the design space.

2.4 Network Functions Migration

Stateful network functions are widely used in modern networks for network monitoring (e.g., PRADS [82]), intrusion detection (e.g., Snort [91]) and load balancing (e.g., HAProxy [30]). These NFs maintain state for ongoing connections, e.g., TCP connection state or number of transmitted bytes per host, and update the state when processing packets. Network function virtualization (NFV) [73] enables a network controller to spin up middleboxes in virtual machines/containers in response to real-time traffic volume and place these VMs/containers at arbitrary positions in the network. Due to its high flexibility and elasticity, many companies have embraced network function virtualization. Microsoft Azure [67] and Amazon AWS [4] support NFV on their platform to minimize operational complexity. VMware vCloud NFV [97] and Cisco NFVI [14] provide NFV service to customers and help them deploy new services faster.

Though NFV offers elasticity for middlebox deployment, ensuring the correct state for NFs collectively is difficult. Specifically, it requires identifying the states that should be migrated, migrating network function to a new location along with related states, and redistributing affected traffic.

A lot of works [40, 61, 53] have been done to identify states using program analysis automatically. stateAlyzr [40] leverages data and control-flow analysis to identify states

that need to be migrated to ensure consistency when scaling network functions. vNIDS [53] uses static program analysis for identification of shared states to achieve safe virtualization of Network Intrusion Detection Systems. SwingState [61] analyzes the P4 program that implements specific network functions in the SDN data plane and figures out the states that require migration. It then modifies the P4 code to enable the live migration of these states. In this dissertation, we focus on how to migrate network functions and how to coordinate NF migration with traffic redistribution.

Network function (NF) migration using software-defined networking (SDN) requires careful coordination between efficient routing-policy updates and NF migration. To ensure that all packets are processed correctly during NF migration, all existing works [61, 24, 85, 23] update network forwarding state strictly after NF migration is done. While some approaches [85, 24] use a centralized controller to reroute affected traffic from the old to new NF position, other works [61, 23] tunnel traffic directly to the new NF position to reduce latency. Examples of the former class are OpenNF [24] and Split/Merge [85]. OpenNF uses a centralized SDN controller to coordinate NF migration with packet redistribution. The affected incoming packets arriving at old NF positions are buffered at the controller during NF migration to avoid packet loss. Split/Merge leverages SDN to split per-flow states among VM replicas and redistribute traffic among them. An example of the latter class is SwingState [61]. SwingState creates a tunnel between the old position and the new position of each NF. NF states are prepended to packets arriving at the old location and are then forwarded to the new location. Asron *et al.* [23] leverage virtual Ethernet interfaces to bridge old NF instance and new instance for packet redistribution. To reduce service downtime, they do not halt the operation of an old instance during migration. Old instance mirrors packets to the new instance such that the new instance maintains up-to-date states. All of these works change network routing policy after NF migration is finished, which slows down traffic redistribution.

Numerous works focus on virtual machine (VM) migration [15, 26, 19, 65]. Some works [15,

19] clone VM instances in their entirety. Live migration [15] copies a snapshot of VM memory to the new location. To minimize the VM downtime, write operations at the old position are intercepted and copied to the new location. Remus [19] clones memory and disk states across multiple VM replicas and uses checkpointing to provide fault tolerance. These works do not coordinate VM migration with the change of routing policies. Other works [26, 65] propose to migrate VMs along with the underlying virtual network. XenFlow [65] migrates Xen VMs and the virtual network using OpenFlow switches to minimize performance disruption. LIME [26] migrates a collection of VMs and virtual switches as an ensemble to provide transparency to the applications, though with the risk of packet loss while VMs or switches are temporarily frozen.

In this dissertation, NFs move along with their states. Some works [47, 41, 90], which are orthogonal to ours, maintain an external data store such that states do not need to migrate during NF migration. Some papers [41] propose systems that keep all NF states in a standalone centralized store, while other works [47, 90] maintain states both locally and remotely for performance. StatelessNF [41] breaks the tight coupling between the processing of network functions and state management by placing all states in a remote data store. However, customized techniques, e.g., RAMcloud [76] and Infiniband [36], need to be used to improve database and network performance. StreamNF [47] provides state management among chain-wide NFs by classifying states into per-flow state and cross-flow state. A per-flow state can only be updated by a single instance, and cross-flow state objects are shared among multiple instances. Cross-flow state objects can be cached locally for performance improvement but require synchronization with remote data store upon local write operations. S6 [90] leverages distributed shared object (DSO) to distribute and share states among all NFs.

2.5 Correctness Checking

A lot of works [45, 46, 69, 63, 102, 48] utilize automatic techniques, such as model checking, to find bugs in network applications, detect network anomaly or verify the correctness of

network property. These works can be classified into two kinds, online and offline checking. The difference is that online checking trouble-shoots the network system at runtime by observing the change of network state when packets traverse the network. Examples of offline checking SDN are SDNRacer [69] and HSA [46]. SDNRacer detects concurrency issues in the data plane by defining the happens-before relation for network events. HSA developes a framework, called *header space analysis*, to statically check the correctness of network configurations in the data plane. These works either use a simplified switch model for efficiency or can only be used for small-scale network settings. Examples of online checking SDN are NetPlumber [45], VeriFlow [48] and ATPG [102]. NetPlumber and VeriFlow verify the network-wide invariants in SDN at runtime by observing OpenFlow messages. ATPG injects test packets into the data plane to monitor the router configurations. None of these verification tools can be directly used to verify the correctness of our algorithm since modeling our algorithm needs to take into account additional factors such as unknown delays for switch updates to occur and diverse behavior of network protocols. In this dissertation, we use model checking to model these factors and verify the correctness of our algorithms.

Model checking [16] emerged in the early 1980s to prove concurrent system correctness. A model-checking tool allows users to describe a system (called model) and a property (called specification) this system is expected to satisfy. The tool automatically verifies if this property is satisfied by searching for a counterexample that violates the property. To reduce state space that the model checking tools need to explore, techniques, such as symbolic execution [49], are used. Recently, model checking has been utilized to verify network-wide properties for SDN. NICE [63] uses model checking in combination with symbolic execution to find bugs in OpenFlow controller applications. FLOVER [92] model checks if the rules generated by an SDN app satisfy specific security policies. FlowChecker [3] checks misconfigurations for scenarios where multiple users share the same network and deploy their rules separately. In Chapter 5, we elaborate on how to model our system and demonstrate the correctness of our algorithms using model checking.

CHAPTER 3: EFFICIENT AND SAFE NETWORK UPDATES WITH SUFFIX CAUSAL CONSISTENCY[1]

In this chapter, we propose a new method to achieve efficient and safe network forwarding-state update using an alternative consistent update abstraction that we call *suffix causal consistency*. Unlike existing mechanisms [32, 39, 80, 54, 44, 71] which enforce atomicity, this approach allows packets to traverse a combination of both old and new paths, thus relaxing the consistency model and speeding up update times. Our work builds on the following insights: (1) for most network policies, the network paths are designed to control routes to a destination (or a suffix), and (2) a packet (or a flow) traversing a mixture of old and new paths can retain correctness provided it traverses the old policy and then the new. These insights are a natural fit for causal consistency [1], a shared memory consistency model that guarantees that processes (in our case packets) observe operations (in our case rules) in a causal order. To this end, we propose *suffix causal consistency* (SCC), a practical and efficient networking domain-specific realization of causal consistency.

There are several challenges in practically realizing causal consistency within the network of distributed devices. The first is designing update algorithms that provide causal consistency while simultaneously preserving a broad range of network invariants, e.g., black-hole freedom. We tackle this challenge by tagging each packet with a Lamport timestamp [51]; each switch then updates this timestamp to reflect the rule matched to the packet. Naively, this approach would then require that downstream switches match this packet only to a rule with a timestamp at least as large, but doing so requires that any network update affect *all* of these downstream switches (to increase their rule timestamps, even if their rules need

[1]This chapter is excerpted from previously published work [55].

not change). We thus propose a novel method of managing these timestamps to limit the number of switches that each network update must involve, thereby accelerating the update process. A second challenge is developing network primitives to efficiently support causal consistency on commodity switches in the face of practical switch constraints and dynamic switch behavior. Despite the development of highly programmable switches [9] that unlock flexible functionality, support for causal consistency poses several challenges. Specifically, supporting causal consistency requires switches to detour packets and temporarily buffer packets.

To demonstrate the effectiveness and efficiency of suffix causal consistency, we developed prototypes for both Open vSwitch (OVS) and P4 [9]. We evaluate our prototypes against realistic workloads and topologies. Our analyses show that SCC deploys updates faster than state-of-the-art alternatives (COCONUT [25], TSU [59] and CU [86]) while simultaneously providing for less packet loss and less rule overhead during updates. We also show that our rule-generation algorithm scales well to topologies of considerable size.

Results of our evaluation show that: 1. SCC outperforms COCONUT [25] and the original, uncoordinated approach to rule updates in terms of the packets dropped during an update, and outperforms COCONUT, CU [86], and TSU [59] in terms of the packets dropped due to link failures. 2. SCC deploys rules more efficiently than CU, COCONUT, or TSU and imposes less rule storage overhead in amount and/or duration than these alternatives. 3. The rule generation time of our algorithm scales across a range of both fat-tree and ISP topologies.

3.1 Network Model and Goals

In this section, we detail our model of the network, which is general enough to include SDN setups and some others (Sec. 3.1.1). We then motivate and define our main goal in this dissertation, a property that we call *suffix causal consistency* (Sec. 3.1.2).

3.1.1 Network Model

Controller The network has a logically centralized controller that is responsible for configuring the switches. To do so, the controller stores network topology information, the *rules*

deployed on each switch (i.e., flow table snapshot) as discussed below, and switch configurations. It makes network information available to one or more *applications* that make routing decisions. The controller produces a new routing policy as needed, based on input from applications. We refer to the emission of a new policy as a new *epoch*. We assume that the routing policy of each new epoch is deployed to the network (in the form of *rules* described below) before the next epoch begins. Epochs are thus totally ordered by time, and we use an epoch counter $epochCtr = 1, 2, \ldots$ to index a particular epoch.

Routing policies A routing policy specifies how to route *flows* through the network. A flow consists of packets with the same addressing information (IP 5-tuple). We assume that the packets of any flow enter the network at a single ingress point that remains constant across epochs (as is commonly assumed, e.g., [86]).

Rules The instructions for how a switch should treat certain packets are specified by *rules*. Each rule R includes (at least) the following fields, all of which are immutable:

- R.cover specifies the set of flows to which this rule pertains (i.e., that can be matched to this rule);
- R.priority specifies the priority of this rule, with higher priorities indicated by larger numbers and with a special priority ∞ to represent the maximum priority, which can be used only by our algorithm;
- R.sendTo specifies the switch identifier (in practice, an outbound port) to which packets matched to this rule should be forwarded, or *drop* if the packets should be dropped;
- R.switch specifies the unique switch S into which R can be installed; and
- R.epochCtr records the index *epochCtr* of the epoch that produced this rule.

Each epoch yields a collection of rules for switches in the network to implement the routing policy for this epoch. That said, not all such rules will necessarily be installed at

(i.e., deployed to) switches, since they may be redundant with rules already installed in some of the switches.

Switches Each switch maintains a flow entry table which stores a set of rules for flow management. We denote the set of rules in the flow table of switch S as $S.\mathsf{ruleSet}$; e.g., $S.\mathsf{ruleSet} = \{R_1, R_6, R_{10}\}$ means that switch S includes rules R_1, R_6 and R_{10}. The controller modifies this set by invoking the following interface, which is similar to that provided by OpenFlow:

- $S.\mathsf{flowadd}(R_j)$ inserts rule R_j into $S.\mathsf{ruleSet}$. This command fails with no effect if $R_j.\mathsf{switch} \neq S$ or if $S.\mathsf{ruleSet}$ already contains a rule $R_{j'}$ such that $R_{j'}.\mathsf{priority} = R_j.\mathsf{priority}$ and $R_j.\mathsf{cover} \cap R_{j'}.\mathsf{cover} \neq \emptyset$.
- $S.\mathsf{flowdel}(R_j)$ removes rule R_j from $S.\mathsf{ruleSet}$.

Due to the communication delay between the controller and each switch, invoking these switch commands on multiple switches simultaneously cannot ensure that the switches reflect these changes at the same time, which may cause inconsistent states across the switches. For example, if one switch has already deleted a stale rule, but its upstream switch still keeps sending packets to it, this switch may not find a matching rule or leverage the default rule (which may drop the packets and create a black hole).

An example Consider a packet *pkt* that traverses a sequence of switches $S_j \rightarrow \ldots \rightarrow S_{j'}$, as directed by the rules on these switches, written $R_j \rightarrow \ldots \rightarrow R_{j'}$. For example, in Fig. 3.1, the rules applied to packet *pkt* on path $S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5$ are $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4$, where rule R_1 directs switch S_1 to send the packet *pkt* to switch S_2, and so on. If the application wants to change the path of packet *pkt* from $S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5$ (the dashed line in Fig. 3.1, denoted $path_{pkt}^{\mathsf{old}}$) to $S_1 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5$ (the solid line, denoted $path_{pkt}^{\mathsf{new}}$), then the application conveys this to the controller, and the controller generates several commands to update the switch states (resulting in a new epoch). The commands include:

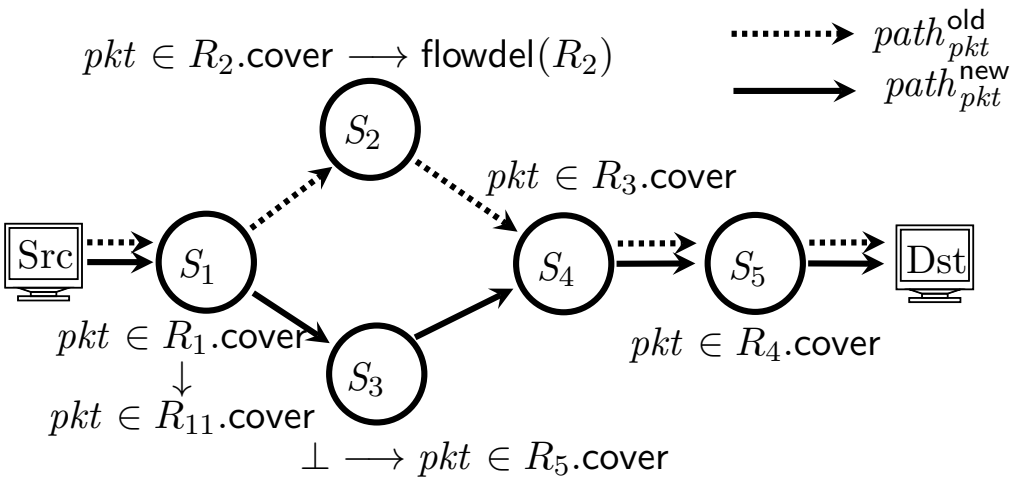


Figure 3.1: Example of route change.

- $S_1.\mathsf{flowdel}(R_1)$ and $S_1.\mathsf{flowadd}(R_{11})$ where $R_{11}.\mathsf{sendTo} = S_3$ and $pkt \in R_{11}.\mathsf{cover}$;
- $S_2.\mathsf{flowdel}(R_2)$ to delete rule R_2 from S_2; and
- $S_3.\mathsf{flowadd}(R_5)$ to instruct S_3 to send pkt to S_4 (i.e., $R_5.\mathsf{sendTo} = S_4$ and $pkt \in R_5.\mathsf{cover}$).

Rule R_3 instructing S_4 to send pkt to S_5 need not be changed (assuming $R_3.\mathsf{cover}$ does not specify the inbound port on S_4). Nor does rule R_4 instructing S_5 to send pkt to the destination node.

3.1.2 Goals

In the example of Fig. 3.1, if S_1 is updated prior to the addition of R_5 to S_3, then pkt might be directed to S_3 before S_3 has a rule to handle it. Similarly, if switch S_1 is not yet updated but switch S_2 has already deleted the rule R_2, then switch S_2 would not have a rule to match pkt upon its arrival. Our central goal in this dissertation is to develop a rule-update framework that avoids such inconsistencies.

More specifically, the property we seek to implement in this dissertation, namely *suffix causal consistency* (SCC), prevents such inconsistencies from occurring. At a high level, SCC ensures that a packet traverses a suffix of the most recent path specified for it and for which

it encounters rules. If the path routes the packet to a desirable egress point from the network, then any suffix of that path also delivers the packet to that egress point.

> *Suffix causal consistency (SCC)*: Let $pkt.\mathsf{epochCtr}$ denote the largest value of $R.\mathsf{epochCtr}$ among all rules R to which a packet pkt is matched between its entry to and departure from the network. Let $path$ be the path specified for pkt in epoch $pkt.\mathsf{epochCtr}$. Then, the sequence of switches traversed by pkt ends in a suffix of $path$.

It is instructive to put this definition to work using the example in Fig. 3.1, discussed above. In the first source of inconsistency considered there, S_1 is updated with rule R_{11} before R_5 is added to S_3, causing pkt to be directed to S_3 before S_3 has a rule to handle it. Since $pkt.\mathsf{epochCtr} = R_{11}.\mathsf{epochCtr}$ (i.e., assuming pkt does not encounter a more recent update later), SCC ensures that pkt is routed along the new path; in this case, the suffix along which pkt is routed is all of $path_{pkt}^{\mathsf{new}}$. To do so, our framework ensures that S_3 can detect that it needs to buffer pkt and await the arrival of R_5.

The second potential source of inconsistency in the discussion above was that S_2 already deleted R_2 but S_1, having not yet been updated, still forwards pkt to S_2. Since R_2 is gone from S_2, there is no hope of forwarding pkt further along the old path, $path_{pkt}^{\mathsf{old}}$. So, if the system deleted R_2, SCC also obligates the system to match pkt to some rule, say R_6 (not shown in Fig. 3.1), with $R_6.\mathsf{epochCtr}$ reflecting the existence of a new path $path_{pkt}^{\mathsf{new}}$ and, in fact, that forwards pkt in the direction of that new path. (Additional rules will need to ensure it gets there.) As we will see, in our framework, R_6 is deployed to S_2 alongside the deletion of R_2, expressly for the purpose of forwarding pkt back toward the switch at which $path_{pkt}^{\mathsf{new}}$ departed from $path_{pkt}^{\mathsf{old}}$ (which is S_1 in this example). The pkt will then pick up the new path at that departure point, traveling the suffix of $path_{pkt}^{\mathsf{new}}$ beginning there. R_6 will need to remain in S_2 only temporarily.

As we will discuss in Sec. 3.3.2, SCC is a strong property, in that it facilitates a number of other, more familiar properties such as black-hole freedom and bounded looping during updates, as well as various forms of waypoint enforcement.

Of course, we seek to implement SCC as efficiently as possible. Our primary efficiency concerns include the speed of an epoch taking effect, the update of as few switches as is necessary to do so, and minimizing the additional rules that switches must (even temporarily) maintain during an update.

3.2 Components

To support the SCC primitive, we augment both the controllers and the switches with modules to support operations for managing and maintaining the timestamps and epochs. We summarize these components here.

Controller Module Operations In our system, the SDN applications (SDNApps) remain unmodified. Instead, the SDN controller intercepts rules and introduces the timestamps and epoch counters into the rules. To do this, the controller module maintains all information required to efficiently manage the different epochs and timestamps. Additionally, the controller coordinates with the network edge to ensure that appropriate timestamps are added into the different packets.

Switch Module Operations We modify the switches to provide operations required to maintain and support timestamps. Specifically, upon the arrival of the packet pkt, the switch will first search for the highest-priority rule R covering the packet. If $R.\mathsf{epochCtr} \geq pkt.\mathsf{tstamp}$, then the switch tags the packet with the rule's *tagging timestamp* $R.\mathsf{tstamp}$ (i.e., $pkt.\mathsf{tstamp} \leftarrow R.\mathsf{tstamp}$; see Sec. 3.3) and forwards pkt to $R.\mathsf{sendTo}$. Otherwise, the packet is buffered by the switch and until its highest-priority rule R covering the packet satisfies $R.\mathsf{epochCtr} \geq pkt.\mathsf{tstamp}$.

The initial insertion and the final removal of the packet timestamp $pkt.\mathsf{tstamp}$ can be accomplished by the source and destination endhosts themselves, by appliances between the endhosts and switches, or by the ingress and egress switches. If switches are in charge of inserting and removing timestamps, the ingress switch should insert $R.\mathsf{tstamp}$ for the rule R to which it matches the packet. For example, when packet pkt_1 first arrives at switch S_1

in Fig. 3.1, the switch tags the packet according to the rule R_1 to which it is matched (i.e., $pkt_1.\mathsf{tstamp} \leftarrow R_1.\mathsf{tstamp}$). Then switch S_5 can remove the tag from the packets by setting the corresponding header field to a default value (e.g., $pkt_1.\mathsf{tstamp} \leftarrow \perp$).

We require switches to support bundled operations. A bundle is a sequence of multiple flow table modifications from the controller (i.e., $S.\mathsf{flowadd}$ and $S.\mathsf{flowdel}$ operations) to the same switch that the switch should apply atomically. (Bundled operations are supported in the OpenFlow specification starting with version 1.4.) In each epoch, the controller submits all changes to each switch in one bundle.

3.3 Algorithm Description

In this section we provide an algorithm for preventing inconsistencies during path updates such as those described in Sec. 3.1, and specifically to implement SCC. In our framework, we add to each rule a *tagging timestamp* $R.\mathsf{tstamp}$ (an integer) with which a switch tags packets matched to that rule before forwarding them. Each packet thus includes a new field $pkt.\mathsf{tstamp}$ to hold this timestamp. This timestamp plays a role similar to a Lamport timestamp [51], in that it indicates to the switch at which a packet arrives the recency of the previous rules applied to that packet. The switch is then required to match this packet to a rule at least this recent. However, in the event that the controller recognizes that a rule already deployed to a switch is just as good for a packet pkt as the most recent rule R for that packet, then it can forego installing R at that switch. Instead, it *backdates* the timestamp on the packet, by deploying a rule R_j to the immediately upstream switch (if it had to do so anyway) with a tagging timestamp $R_j.\mathsf{tstamp}$ that, when carried forward by the packet (in $pkt.\mathsf{tstamp}$), will not induce downstream switches to await a new rule from the controller. In the remainder of this section we detail this algorithm.

3.3.1 Controller Operation

Upon computing a new routing policy, the controller computes the rules currently deployed that must be changed in this epoch. The controller does this by first computing forwarding rules based on the new epoch's routing policy; here we simply borrow an existing

algorithm (in our implementation, we use the algorithm of Kang et al. [42]). This algorithm outputs a rule set $\mathcal{R}^{\mathsf{new}}$, and let $\mathcal{R}^{\mathsf{old}}$ denote the rules already deployed to the network.

To define the controller's algorithm for generating the rules $\mathcal{R}^{\mathsf{add}} \subseteq \mathcal{R}^{\mathsf{new}}$ to add to the network and the rules $\mathcal{R}^{\mathsf{del}} \subseteq \mathcal{R}^{\mathsf{old}}$ to delete, we first define some additional notation. First, we say that $R_1 \in \mathcal{R}^{\mathsf{new}}$ and $R_2 \in \mathcal{R}^{\mathsf{old}}$ are *copies* of one another if R_1 and R_2 are identical except for their $\mathsf{epochCtr}$ and tstamp fields. Second, for any set $\mathcal{R}$ of rules, any $R_1 \in \mathcal{R}$, and any packet pkt, the predicate $match(\mathcal{R},\ R_1,\ pkt)$ is true if and only if $pkt \in R_1.\mathsf{cover}$ and there is no higher priority rule $R_2 \in \mathcal{R}$ such that $pkt \in R_2.\mathsf{cover}$ and $R_2.\mathsf{switch} = R_1.\mathsf{switch}$.

Then, the controller computes $\mathcal{R}^{\mathsf{add}}$ and $\mathcal{R}^{\mathsf{del}}$ using an algorithm consisting of five steps, executed in order:

1. Initialization
2. Backward closure
3. Forward closure
4. Set tagging timestamps
5. Send-back rules

We first describe the goals of these steps and then elaborate on them in detail below. The "initialization" step simply sets $\mathcal{R}^{\mathsf{add}}$, $\mathcal{R}^{\mathsf{del}}$, and $\mathcal{R}^{\mathsf{keep}}$ to initial values. The "backward closure" step updates $\mathcal{R}^{\mathsf{add}}$ to include rules from $\mathcal{R}^{\mathsf{new}}$ that precede (on routing paths) those already in $\mathcal{R}^{\mathsf{add}}$ in certain circumstances, thereby propagating the installation of new rules "backward" along routing paths. The "forward closure" step then updates $\mathcal{R}^{\mathsf{add}}$ to include rules from $\mathcal{R}^{\mathsf{new}}$ that follow (on routing paths) those already in $\mathcal{R}^{\mathsf{add}}$ in other circumstances, thus propagating the installation of new rules "forward" along routing paths. The "set tagging timestamps" step sets the $R.\mathsf{tstamp}$ field of rules $R \in \mathcal{R}^{\mathsf{add}}$ that have not been set in the preceding steps. Finally, the "send-back rules" step adds new rules to $\mathcal{R}^{\mathsf{add}}$ to account for the possibility that a packet traveling its old path encounters a switch at which the rule

it would have matched in the old configuration has already been deleted and no new rule has been added to instead match this packet (e.g., since the new path does not traverse this switch). To avoid dropping the packet, this step adds a temporary rule on the switch to send the packet back to the upstream switch from which it came, eventually allowing the packet to pick up the new path toward its destination.

Initialization The controller initializes sets $\mathcal{R}^{\mathsf{keep}}$, $\mathcal{R}^{\mathsf{add}}$, and $\mathcal{R}^{\mathsf{del}}$ that it will then update throughout the remainder of the algorithm. By the end of the algorithm, $\mathcal{R}^{\mathsf{add}}$ will contain those rules that the controller will install via $S.\mathsf{flowadd}$ invocations, and $\mathcal{R}^{\mathsf{del}}$ rules will contain those rules that controller will remove via $S.\mathsf{flowdel}$ invocations. (As we will discuss below, some of the added rules will also be deleted afterwards.) The rules in $\mathcal{R}^{\mathsf{keep}}$ at the end of the algorithm will be those that the controller leaves in place.

Initialization: Initialize sets $\mathcal{R}^{\mathsf{keep}}$, $\mathcal{R}^{\mathsf{add}}$, and $\mathcal{R}^{\mathsf{del}}$ as follows. Initialize $\mathcal{R}^{\mathsf{keep}}$ to contain each $R \in \mathcal{R}^{\mathsf{old}}$ for which there is a copy in $\mathcal{R}^{\mathsf{new}}$. Initialize $\mathcal{R}^{\mathsf{del}}$ to $\mathcal{R}^{\mathsf{old}} \setminus \mathcal{R}^{\mathsf{keep}}$, and initialize $\mathcal{R}^{\mathsf{add}}$ to include any $R \in \mathcal{R}^{\mathsf{new}}$ for which there is no copy in $\mathcal{R}^{\mathsf{keep}}$. Note that each $R \in \mathcal{R}^{\mathsf{add}}$ has $R.\mathsf{epochCtr} = \mathit{epochCtr}$ and $R.\mathsf{tstamp} = \perp$ (undefined) since $\mathcal{R}^{\mathsf{add}} \subseteq \mathcal{R}^{\mathsf{new}}$.

Taking the backward closure of $\mathcal{R}^{\mathsf{add}}$ The next stage of the algorithm is motivated by situations like that shown in Fig. 3.2, where $\mathit{path}_{\mathit{pkt}}^{\mathsf{old}} = (S_1 \to S_2 \to S_5)$ is the path taken by *pkt* under the previous routing policy and the new path $\mathit{path}_{\mathit{pkt}}^{\mathsf{new}} = (S_1 \to S_3 \to S_4 \to S_5)$ is the path it will take under the new routing policy. At this stage of the algorithm, $R_1 \in \mathcal{R}^{\mathsf{keep}}$ and $R_2 \in \mathcal{R}^{\mathsf{add}}$. If we keep R_1 unchanged and if the packet *pkt* arrives at S_3 on the new path, it will be tagged using the old timestamp (e.g., $\mathit{pkt}.\mathsf{tstamp} \leftarrow R_1.\mathsf{tstamp} = 8$). Therefore, upon the arrival of the packet at S_4, if S_4 has not been updated, the stale rule R_0 will be applied on the packet. This rule may send the packet to a switch that belongs to neither the old path nor the new path (e.g., S_6) or to a switch (e.g., S_1) that the packet has already passed, potentially creating a black-hole or loop. So we need to add a copy R_3 of R_1 to $\mathcal{R}^{\mathsf{add}}$,

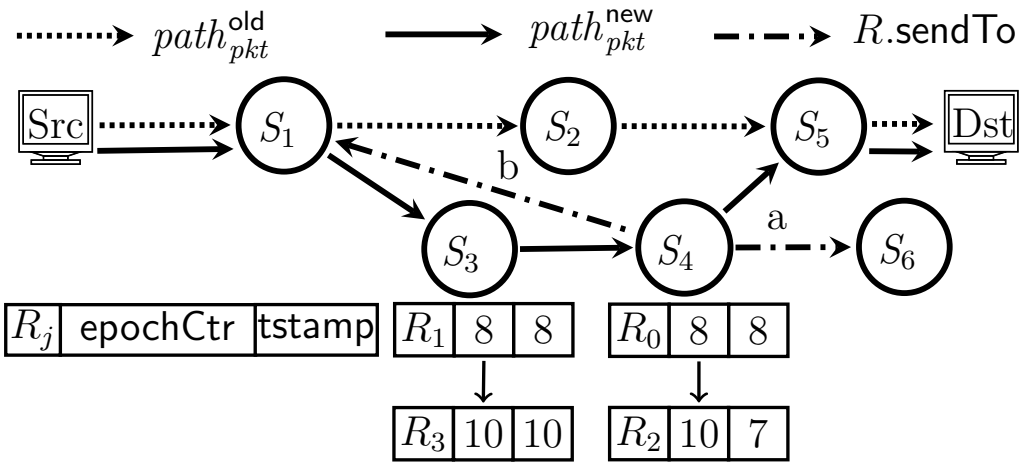


Figure 3.2: Example motivating backward closure.

so that upon passing S_3, the packet will carry a new timestamp $pkt.\mathsf{tstamp} \leftarrow R_3.\mathsf{tstamp} = 10$, ensuring that new rule R_2 will be applied to it at switch S_4.

Backward closure: Repeat this step until no more rules can be added to $\mathcal{R}^{\mathsf{add}}$. If for any $R_1 \in \mathcal{R}^{\mathsf{keep}}$, there is a pkt where

- $match(\mathcal{R}^{\mathsf{add}} \cup \mathcal{R}^{\mathsf{keep}}, R_1, pkt)$,
- there is a $R_2 \in \mathcal{R}^{\mathsf{add}}$ with $R_2.\mathsf{switch} = R_1.\mathsf{sendTo}$ such that $match(\mathcal{R}^{\mathsf{add}} \cup \mathcal{R}^{\mathsf{keep}}, R_2, pkt)$,
- if $path^{\mathsf{new}}_{pkt}$ is the path that pkt would travel if routed by $\mathcal{R}^{\mathsf{new}}$, and if $path^{\mathsf{old}}_{pkt}$ is the path that pkt would travel if routed with $\mathcal{R}^{\mathsf{old}}$, then either $R_1.\mathsf{switch} \notin path^{\mathsf{old}}_{pkt}$ or the prefixes of $path^{\mathsf{new}}_{pkt}$ and $path^{\mathsf{old}}_{pkt}$ ending at $R_1.\mathsf{switch}$ are not the same,

then add to $\mathcal{R}^{\mathsf{add}}$ the rule $R_3 \in \mathcal{R}^{\mathsf{new}}$ that is a copy of R_1, and move R_1 from $\mathcal{R}^{\mathsf{keep}}$ to $\mathcal{R}^{\mathsf{del}}$. Set $R_3.\mathsf{tstamp} \leftarrow epochCtr$.

As such, the next step of the algorithm identifies any rule R_2 to be added but for which some packet that will be matched to it could be matched at the immediately upstream switch to a rule R_1 that is currently slated to be kept. If R_1 is not replaced by its copy R_3 that will timestamp the packet to force it to await the arrival of R_2, then the packet could be routed incorrectly or routed along the old path indefinitely. The latter case cannot be allowed (and

achieve SCC) if the packet could have been previously routed differently by new rules, i.e., if the prefixes of the old and new paths ending at R_1.switch differ, or if R_1.switch isn't even on the old path.

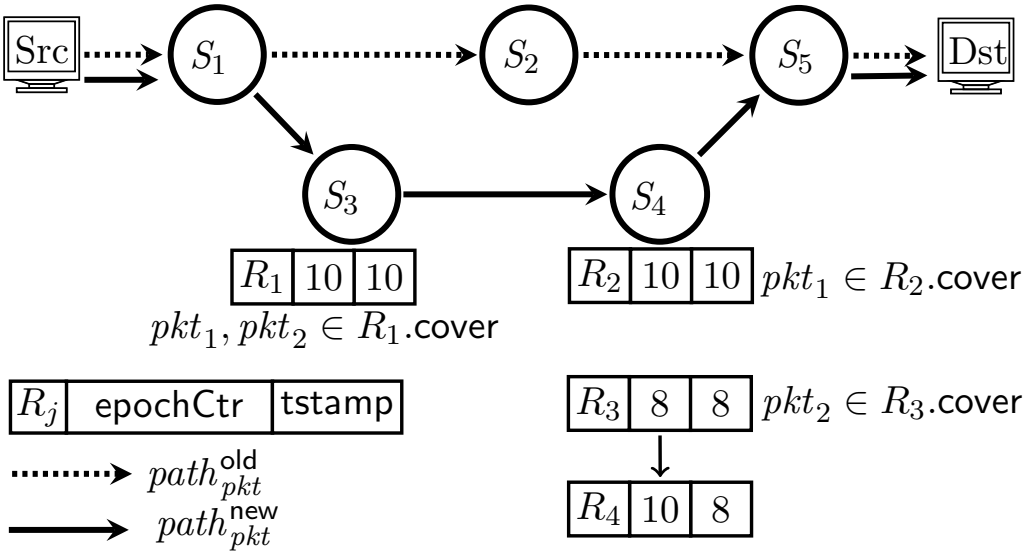


Figure 3.3: Example motivating forward closure.

Taking the forward closure of $\mathcal{R}^{\mathsf{add}}$ To understand the need for the next stage of our algorithm, consider Fig. 3.3, where there is a rule $R_1 \in \mathcal{R}^{\mathsf{add}}$ at S_3 matching two packets pkt_1 and pkt_2, another rule $R_2 \in \mathcal{R}^{\mathsf{add}}$ at S_4 matching pkt_1, and another rule $R_3 \in \mathcal{R}^{\mathsf{keep}}$ at S_4 matching packet pkt_2. Because R_2.epochCtr $= 10$, the pkt_1.tstamp should equal 10 to ensure that R_2 matches pkt_1. To ensure this, R_1.tstamp $= 10$, meaning that pkt_2.tstamp will also be assigned 10. So, the old R_3 with R_3.epochCtr $= 8$ must be replaced, to ensure that pkt_2 will be not buffered indefinitely at S_4.

So, the next step of the algorithm identifies cases in which some packets handled by a rule $R_1 \in \mathcal{R}^{\mathsf{add}}$ will be handled at the downstream switch by another rule $R_2 \in \mathcal{R}^{\mathsf{add}}$, while others handled by R_1 will be handled at the downstream switch by a rule $R_3 \in \mathcal{R}^{\mathsf{keep}}$. In this case, packets of the first type handled by R_1 must be timestamped to force their handling by R_2 (see the "Tagging timestamps" step below), but then packets of the second type will

be stuck waiting indefinitely for a new rule to replace R_3 that will never arrive. As such, we schedule R_3 to be replaced, as well.

Forward closure: Repeat this step until no more rules can be added to $\mathcal{R}^{\mathsf{add}}$. If for any $R_1 \in \mathcal{R}^{\mathsf{add}}$, there are

- a rule $R_2 \in \mathcal{R}^{\mathsf{add}}$ where $R_2.\mathsf{switch} = R_1.\mathsf{sendTo}$,
- a rule $R_3 \in \mathcal{R}^{\mathsf{keep}}$ where $R_3.\mathsf{switch} = R_1.\mathsf{sendTo}$,
- a packet pkt_2 such that $match(\mathcal{R}^{\mathsf{add}} \cup \mathcal{R}^{\mathsf{keep}},\ R_1,\ pkt_2)$ and $match(\mathcal{R}^{\mathsf{add}} \cup \mathcal{R}^{\mathsf{keep}},\ R_2,\ pkt_2)$, and
- a packet pkt_3 such that $match(\mathcal{R}^{\mathsf{add}} \cup \mathcal{R}^{\mathsf{keep}},\ R_1,\ pkt_3)$ and $match(\mathcal{R}^{\mathsf{add}} \cup \mathcal{R}^{\mathsf{keep}},\ R_3,\ pkt_3)$,

then add to $\mathcal{R}^{\mathsf{add}}$ the rule $R_4 \in \mathcal{R}^{\mathsf{new}}$ that is a copy of R_3, and move R_3 from $\mathcal{R}^{\mathsf{keep}}$ to $\mathcal{R}^{\mathsf{del}}$. Set $R_4.\mathsf{tstamp} \leftarrow R_3.\mathsf{tstamp}$.

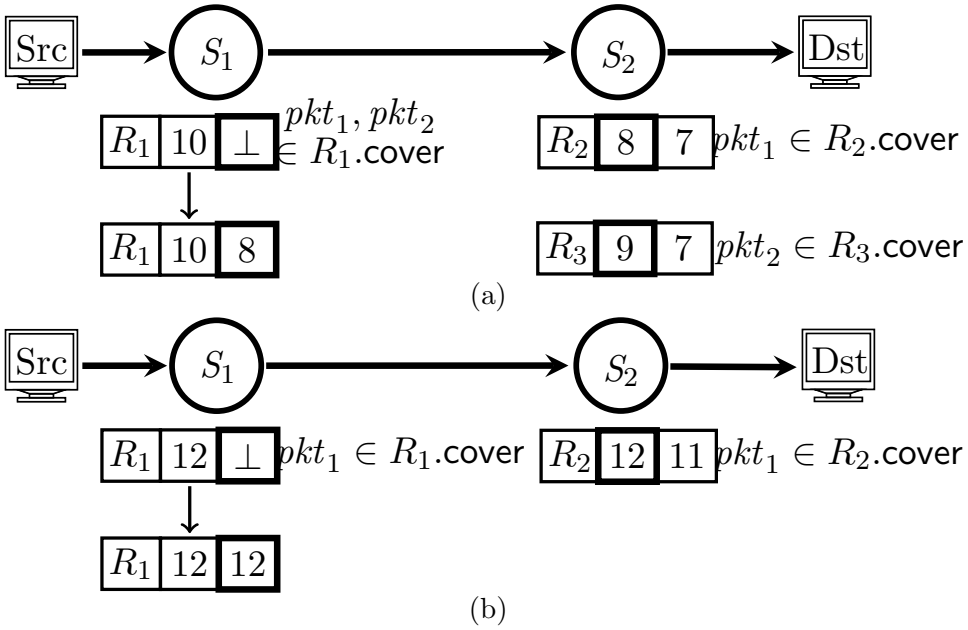


Figure 3.4: Example of tagging timestamps.

Setting tagging timestamps So far, only rules added to $\mathcal{R}^{\mathsf{add}}$ in the preceding "closure" steps had their tstamp fields initialized. The purpose of this step is to set the tstamp fields for the other rules in $\mathcal{R}^{\mathsf{add}}$. In brief, the tstamp field for a rule R in $\mathcal{R}^{\mathsf{add}}$ needs to be set to the minimum of the epochCtr field values for the rules at the immediately downstream switch that will handle the same packets. In this way, none of the packets forwarded by R will be needlessly buffered at the downstream switch.

Examples can be found in Fig. 3.4. In Fig. 3.4(a), assume we have $R_1 \in \mathcal{R}^{\mathsf{add}}$ at S_1 matching two packets, pkt_1 and pkt_2; $R_2 \in \mathcal{R}^{\mathsf{keep}}$ at S_2 matching pkt_1; and $R_3 \in \mathcal{R}^{\mathsf{keep}}$ at S_2 matching packet pkt_2. To ensure that pkt_1 and pkt_2 are matched to R_2 and R_3, respectively, and not needlessly buffered, these packets need to carry timestamps that are at most $R_2.\mathsf{epochCtr}$ and $R_3.\mathsf{epochCtr}$, respectively. Therefore, we set $R_1.\mathsf{tstamp} \leftarrow 8$.

In Fig. 3.4(b), assume $R_1 \in \mathcal{R}^{\mathsf{add}}$ at S_1 matches a packet pkt_1 and $R_2 \in \mathcal{R}^{\mathsf{add}}$ at S_2 matches pkt_1. According to forward closure, any R_3 at S_2 matching packet pkt_2 which is also matched by R_1 should have $R_3.\mathsf{epochCtr} = epochCtr$ where $epochCtr$ is the latest epoch counter. Therefore, we set the tagging timestamp of $R_1.\mathsf{tstamp} \leftarrow epochCtr$, i.e., $R_1.\mathsf{tstamp} \leftarrow 12$.

Tagging timestamps: For each $R_1 \in \mathcal{R}^{\mathsf{add}}$ with $R_1.\mathsf{tstamp} = \perp$ and each packet pkt such that $match(\mathcal{R}^{\mathsf{add}} \cup \mathcal{R}^{\mathsf{keep}}, R_1, pkt)$, let R_{pkt} be the rule with $R_{pkt}.\mathsf{switch} = R_1.\mathsf{sendTo}$ such that $match(\mathcal{R}^{\mathsf{add}} \cup \mathcal{R}^{\mathsf{keep}}, R_{pkt}, pkt)$. Then, set $R_1.\mathsf{tstamp} \leftarrow \min_{pkt}\{R_{pkt}.\mathsf{epochCtr}\}$, where the min is taken over all such packets pkt.

Creating send-back rules The last step of the controller's algorithm is to create rules that cause a packet to backtrack if, while traveling its old path, it encounters a switch S_2 at which the rule it would have matched in the old configuration has already been deleted and no new rule has been added to instead match this packet (e.g., since the new path doesn't traverse this switch). Rather than just drop the packet, the switch will send the packet back to the switch S_1 from which it came. This time, however, the send-back rule R at S_2 will tag the packet with $pkt.\mathsf{tstamp} = R.\mathsf{tstamp} = epochCtr$, causing the packet to be buffered at

S_1 until a new rule arrives. This rule might, in fact, be another send-back rule. The detail of this step is shown in the next page.

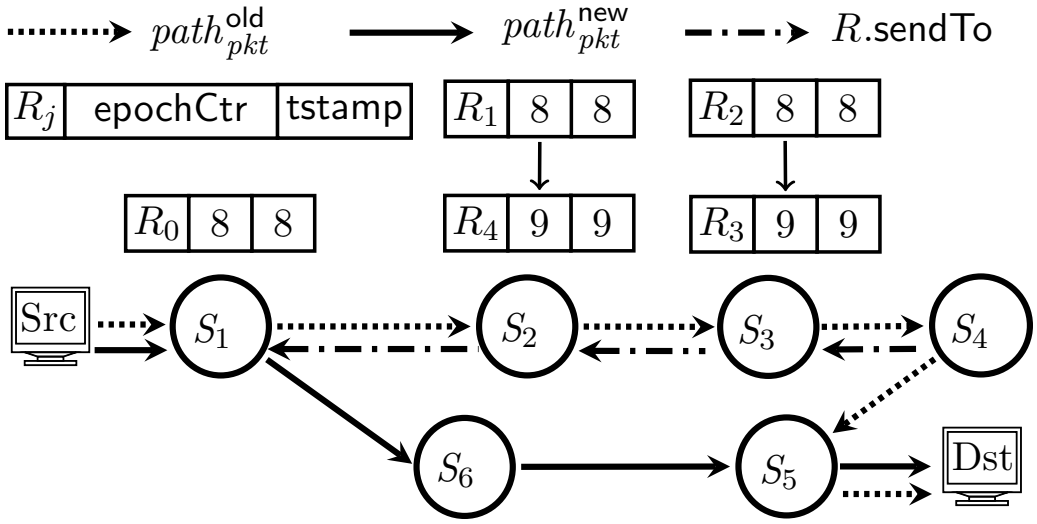


Figure 3.5: Example of send-back rules.

An example is shown in Fig. 3.5, where a packet *pkt* travels along an old path until it reaches switch S_3, which is not on *pkt*'s new path. Rather than drop the packet, the send-back rule R_3 is added to S_3 when the old rule R_2 matching *pkt* in the old configuration is deleted, to direct *pkt* back to S_2. Because $R_3.\mathsf{tstamp} = epochCtr = 9$, $pkt.\mathsf{tstamp} = 9$ when it arrives back at S_2, where it is buffered until the rule R_4 with $R_4.\mathsf{epochCtr} = 9$ is installed. R_4 is also a send-back rule, causing *pkt* to be forwarded back to S_1, where it will await the installation of a rule that will forward the packet on its new path.

Send-back rules: Repeat this step until no more rules can be added to $\mathcal{R}^{\mathsf{add}}$. If for any rule $R_1 \in \mathcal{R}^{\mathsf{old}}$, there is a rule $R_2 \in \mathcal{R}^{\mathsf{old}}$ with $R_2.\mathsf{switch} = R_1.\mathsf{sendTo}$ such that for some packet pkt,

- $match(\mathcal{R}^{\mathsf{old}}, R_1, pkt)$,
- $match(\mathcal{R}^{\mathsf{old}}, R_2, pkt)$, and
- if $path^{\mathsf{old}}_{pkt}$ is the path that pkt would travel if routed with $\mathcal{R}^{\mathsf{old}}$ and if $path^{\mathsf{new}}_{pkt}$ is the path that pkt would travel if routed with $\mathcal{R}^{\mathsf{new}}$, then $R_1.\mathsf{switch} \in path^{\mathsf{old}}_{pkt}$ and $R_2.\mathsf{switch} \notin path^{\mathsf{new}}_{pkt}$,

then add a new rule R_3 to $\mathcal{R}^{\mathsf{add}}$ where

$$\begin{aligned} R_3.\mathsf{switch} &\leftarrow R_2.\mathsf{switch} \\ R_3.\mathsf{sendTo} &\leftarrow R_1.\mathsf{switch} \\ R_3.\mathsf{priority} &\leftarrow \infty \\ R_3.\mathsf{cover} &\leftarrow \bigcup \{pkt\} \\ R_3.\mathsf{epochCtr} &\leftarrow epochCtr \\ R_3.\mathsf{tstamp} &\leftarrow epochCtr \end{aligned} \tag{3.1}$$

where the union in (3.1) is taken over all such packets pkt. The rule R_3 is called a "send-back rule."

Deployment At this point, the controller deploys $\mathcal{R}^{\mathsf{add}}$ using flowadd commands and removes $\mathcal{R}^{\mathsf{del}}$ using flowdel commands on the switches. Recall that all such commands are applied atomically at a single switch, but different switches can execute these command bundles at different times (e.g., due to differing delays between the controller and those switches). Our management of timestamps on packets and rules ensures that SCC is nevertheless achieved. Once the controller receives confirmation that all rules have been deployed

and sufficient time has passed since that confirmation that packets should no longer encounter send-back rules[2], the controller can delete the send-back rules from switches.

Algorithm efficiency Several stages of our algorithm are described inductively, which breaks the algorithm down into (hopefully) understandable steps but somewhat clouds the overall efficiency of the algorithm. Consider for each epoch the equivalence classes of flows that the rules for that epoch route identically. So, the "old" equivalence classes each consists of all flows that each match to the same sequence of rules in $\mathcal{R}^{\mathsf{old}}$, and similarly the "new" equivalence classes each consists of all flows that each match to the same sequence of rules in $\mathcal{R}^{\mathsf{new}}$. Of the five stages of our algorithm, only "Backward closure" and "Send-back rules" are a function of the *paths* taken by flows (i.e., refer to $path_{pkt}^{\mathsf{old}}$ and $path_{pkt}^{\mathsf{new}}$), and so only these stages need consider flows at the granularity of the old and new equivalence classes. Letting c^{old} and c^{new} be the number of old and new equivalence classes, respectively, each iteration of "Backward closure" examines pairs of rules on adjacent switches ($R_1 \in \mathcal{R}^{\mathsf{old}}$ and $R_2 \in \mathcal{R}^{\mathsf{new}}$ where $R_2.\mathsf{switch} = R_1.\mathsf{sendTo}$) and old and new paths of flows they cover (at most $c^{\mathsf{old}} \times c^{\mathsf{new}}$ pairs), and so very coarsely, "Backward closure" costs $O\left(c^{\mathsf{old}}c^{\mathsf{new}} \times \left|\mathcal{R}^{\mathsf{old}}\right| \times |\mathcal{R}^{\mathsf{new}}|\right)$ time. Similarly, "Send-back rules" considers pairs of adjacent, old rules ($R_1 \in \mathcal{R}^{\mathsf{old}}$ and $R_2 \in \mathcal{R}^{\mathsf{old}}$ such that $R_2.\mathsf{switch} = R_1.\mathsf{sendTo}$) and so incurs cost of $O\left(c^{\mathsf{old}}c^{\mathsf{new}} \times \left|\mathcal{R}^{\mathsf{old}}\right|^2\right)$ time. So, the running time of our algorithm is theoretically dominated by steps of $O\left(c^{\mathsf{old}}c^{\mathsf{new}} \times \left(\left|\mathcal{R}^{\mathsf{old}}\right| \times |\mathcal{R}^{\mathsf{new}}| + \left|\mathcal{R}^{\mathsf{old}}\right|^2\right)\right)$ cost.

In practice, we believe this estimate to be wildly pessimistic, since considering rules on *adjacent* switches dramatically reduces the number of switch pairs to consider, and because presumably only a small fraction of the network traffic (rules and flow equivalence classes) changes from one epoch to the next. As such, in Sec. 3.5 we will empirically demonstrate

[2]A delay of *linkLatency* × *diameter* should suffice, where *diameter* is the length of the longest routing path in the network.

the scalability of our algorithm.

3.3.2 Properties

Proposition 1. *The protocol of Sec. 3.3.1 implements suffix causal consistency.*

Proof. (Sketch) Let R_1 be the first rule R matched to a packet pkt with $R.\mathsf{epochCtr} = pkt.\mathsf{epochCtr}$. If R_1 is a send-back rule, then pkt will follow a chain of send-back rules R, each requiring pkt to be buffered awaiting the next by setting $pkt.\mathsf{tstamp} \leftarrow R.\mathsf{epochCtr}$. This chain delivers pkt back to a switch on the new path for pkt in the epoch with index $pkt.\mathsf{epochCtr}$. If R_1 is not a send-back rule, then this rule is already on a switch that is on this new path.

Let R_2 be the last rule R matched to a packet pkt with $R.\mathsf{epochCtr} = pkt.\mathsf{epochCtr}$. Then, each rule R_3 to which pkt is matched at downstream switches has $R_3.\mathsf{epochCtr} < pkt.\mathsf{epochCtr}$ (as otherwise, R_2 would not be the last such rule). Note that R_2 can therefore not be a send-back rule — each send-back rule has a tstamp field equal to $pkt.\mathsf{epochCtr}$, which would preclude the next rule R to match pkt having $R.\mathsf{epochCtr} < pkt.\mathsf{epochCtr}$. Now suppose for a contradiction that R_3 is the first downstream rule (possibly equal to R_2) matched to pkt that directs pkt differently than the rule $R_4 \in \mathcal{R}^{\mathsf{new}}$ would have (i.e., for which $match(\mathcal{R}^{\mathsf{new}}, R_4, pkt)$ is true), where $\mathcal{R}^{\mathsf{new}}$ refers to that set of rules as generated in epoch $pkt.\mathsf{epochCtr}$. Then, $R_3 \in \mathcal{R}^{\mathsf{del}}$ and $R_4 \in \mathcal{R}^{\mathsf{add}}$ in that invocation. By inductive application of the "backward closure" rule, $R_2.\mathsf{tstamp} = pkt.\mathsf{epochCtr}$, contradicting the assumption that R_2 is the last rule R matched to pkt with $R.\mathsf{epochCtr} = pkt.\mathsf{epochCtr}$. □

Higher-level properties One strength of SCC is that it implies a number of other desirable properties for routing. Among them is *black-hole freedom* [62], i.e., the property that packets are not dropped during the transition from an old routing configuration to a new configuration. This property is, of course, contingent on no packet being black-holed intentionally, i.e., that the routing policy in each epoch provides a viable path for every packet. Assuming this, then, the guarantee that each packet will traverse a suffix of the path

prescribed for it in the most recent epoch for which it encounters a rule will guarantee that the packet is not dropped.

A second property implied by SCC is *bounded looping* (cf., [101]). More specifically, since SCC ensures that each packet exits the network on a suffix of the most recently specified covering path for which it encounters a rule, the packet will not loop unless the packet forever encounters a rule created due to another, more recently specified path. In other words, once the network stabilizes and no more paths are specified for long enough, the packet will exit the network and cannot loop indefinitely.

A final property that we discuss is *relaxed waypoint correctness* (we adjust the definition of waypoint property in [84, 5, 24]), of which we consider two varieties. The first requires that certain flows be routed through a series of middleboxes (the "waypoints") that remain fixed, even though the paths between the waypoints are adjusted over time. In this case, each path segment between consecutive waypoints can be treated as an individual path in the routing policy of an epoch, i.e., treating each waypoint as the egress node for one "path" and the subsequent ingress node on another "path" to re-enter the network on its way to the next waypoint. SCC then guarantees that every packet reaches waypoints in order.

The second variant of relaxed waypoint correctness allows waypoints to change, in addition to the paths between them. In this case, we cannot treat each path segment between consecutive waypoints individually for the sake of routing. However, we can accommodate this version of the problem by modifying the order in which the controller deploys new rules to the network, to ensure that the ingress switch of the waypoint-bound packets in each epoch will be updated before any other switch is. In this way, SCC's promise that packets will be routed along a suffix of the most recently specified path for which they encounter rules equates to these packets being routed along the *entire* path. Implicit in this statement is the requirement that a subsequent epoch cannot "catch up to" a packet routed at its ingress switch using the previous epoch's rules. To ensure this, after the controller installs new rules at the ingress switch, it must wait to install new rules at subsequent switches until

all packets routed at the ingress using old rules would have had time to exit the network. We use this variant of SCC algorithm for implementation in Chapter 4. We also discuss the difference between relaxed waypoint correctness and waypoint correctness in Chapter 4.

3.3.3 Timestamp Reset

Since the number of bits in each packet header to maintain the timestamp $pkt.\mathsf{tstamp}$ is limited, the packet timestamps will eventually approach their maximum value. It is thus necessary for the controller to periodically reset the timestamps in rules in the network which, in turn, will reduce the values of timestamps carried in packets. Specifically, the controller executes the following steps periodically, during which time new epochs are not initiated. First, the controller issues commands to all the switches concurrently to reset the tagging timestamp $R.\mathsf{tstamp}$ of each deployed rule R to $R.\mathsf{tstamp} \leftarrow 0$ and awaits an acknowledgment from each switch. Second, after a delay of sufficiently long to ensure that packets pkt remaining in the network have $pkt.\mathsf{tstamp} = 0$, the controller issues commands to update the $R.\mathsf{epochCtr}$ fields of all deployed rules R to $R.\mathsf{epochCtr} \leftarrow 0$. Since each packet pkt traveling the network has $pkt.\mathsf{tstamp} = 0$, resetting $R.\mathsf{epochCtr} \leftarrow 0$ for all rules does not result in packet drops or delays.

3.4 Implementation

We implemented our algorithm in both the P4 switch [9] and Open vSwitch [78]. To issue updates to the switches, we utilized P4 Runtime [79] and the Ryu controller [87], respectively.

3.4.1 P4

We used the BMv2 switch target (i.e., behavioral-model [8]) as our switch model and the P4 language, which is a declarative language to express how packets are processed by the pipeline of switches. Specifically, we defined a packet header field to store a timestamp, i.e., $pkt.\mathsf{tstamp}$. Upon packet arrival, the parser of the switch extracts $pkt.\mathsf{tstamp}$ along with other header fields, and passes the packet to the ingress pipeline. Once the ingress pipeline determines the rule R matching pkt using the standard packet-matching logic, it records the values $R.\mathsf{epochCtr}$ and $R.\mathsf{tstamp}$ as metadata for this packet. If $R.\mathsf{epochCtr} \geq pkt.\mathsf{tstamp}$,

then the ingress pipeline forwards the packet to the egress pipeline with instructions to tag the packet with $pkt.\mathsf{tstamp} \leftarrow R.\mathsf{tstamp}$ and forward the packet to the outbound port defined by $R.\mathsf{sendTo}$. Otherwise (i.e. $pkt.\mathsf{tstamp} > R.\mathsf{epochCtr}$), the ingress pipeline resubmits the packet to the parser (even though pkt has not been changed) to go through the table again to see if the rules have been updated. $R.\mathsf{epochCtr}$ is provided as a parameter to the action field of R rather than as a match field, so that it can be incorporated into the logic of the rule's action.

We made several modifications to the BMv2 switch model to reduce performance overhead and achieve atomic rule updates. Specifically, to reduce the cost caused by resubmitting a packet too frequently, we used two input queues. The packets received by the switch are pushed to the first queue, while the resubmitted packets are buffered in the second one. The ingress pipeline then pops packets from the second queue much less frequently than it does from the first, to reduce the cost of resubmitting packets while waiting for new rules. Moreover, to achieve atomic rule updates (i.e., operation bundling, see Sec. 3.2), we used the approach proposed by Han et al. [29]. Specifically, when receiving multiple rule updates from the controller, the switch (i) makes a copy of the currently active rule table, (ii) applies the updates to the copy, (iii) atomically updates an active-table pointer to point to the (now updated) copy, and (iv) frees the original table. Although this approach requires double the memory space, it does not disrupt traffic during the update.

3.4.2 Open vSwitch

In contrast to the P4 implementation above, which uses a new header field to store $pkt.\mathsf{tstamp}$, our Open vSwitch (OVS) implementation leverages unused header bits to store $pkt.\mathsf{tstamp}$ in each packet. These header bits, which are the same as those used by CONCONUT [25], include 12 bits of the VLAN tag (also used by CU [86]) and 19 bits of the MPLS label. We modified OVS to extract these bits from the packet header and to set these bits during forwarding, according to the rule to which it was matched. The $R.\mathsf{epochCtr}$ value for each rule R is embedded into the matching field of the rule to avoid modifying to the

OpenFlow specification, but this value is not used for matching; instead, it is extracted from the rule during rule installation (and masked during matching). If for the rule R to which pkt is matched, $pkt.\mathsf{tstamp} > R.\mathsf{epochCtr}$, then the thread handling this packet pauses for 1ms and resubmits the packet for matching to the rule table again.

Since OVS allows a rule to direct packets back to the port on which it arrived *only* by specifying the directive "`in_port`" as the outbound port, it was necessary to implement each send-back rule (as described in Sec. 3.3) using two separate rules. One rule handles a packet arriving from the upstream switch on an old path and so that must be forwarded to the same port on which it arrived, using the `in_port` directive. The second rule handles a packet arriving from the downstream switch on the old path and so that must be forwarded to the upstream switch on that path.

For atomic rule updates, we leveraged OVS' bundle operation, which buffers packets while applying changes. Compared with our P4 implementation, this method increases packet latencies, but does not require extra memory resources.

3.4.3 Controller

The controller provides an interface to the applications and transforms a new routing policy into multiple rule modification commands. The controller does this by first computing forwarding rules based on the new epoch's routing policy, using the algorithm of Kang et al. [42]. The output rules ($\mathcal{R}^{\mathsf{new}}$) and the rules already deployed to the network ($\mathcal{R}^{\mathsf{old}}$) are the inputs to our algorithm described in Sec. 3.3.

Our P4 implementation uses the P4 runtime, which is a protocol-independent API using Protobuf [83] and gRPC [27] to issue rule updates to the P4 switches. For our OVS implementation, we leveraged the Ryu controller with the OpenFlow protocol [77]. The main difference between these options is that, while OpenFlow only gives us a way to populate switch tables, the P4 runtime can also push a new P4 program to reconfigure the forwarding behavior of the switches.

For our P4 implementation, we utilized gRPC in Python to issue rule modification com-

mands, while the Ryu controller uses a REST API to update rules in OVS. gRPC allows multiple table-entry updates to be included in one message. In contrast, the Ryu controller uses a bundle operation to issue a sequence of rule modifications. Specifically, the Ryu controller issues a bundle-control message to open a bundle for each switch and then one or more bundle-add messages to indicate which rules need to be modified, followed by a bundle-control message to commit and close the bundle. The flag OFPBF_ORDERED and OFPBF_ATOMIC are specified to ensure that updates are applied in the order sent and atomically. After committing the rule updates, the switch sends a confirmation message to the controller to allow the controller to update its records of switch states.

3.5 Evaluation

In this section we evaluate our design, specifically to show the following benefits of SCC:

- SCC compares favorably to COCONUT [25] and to the original, uncoordinated approach to rule updates in terms of the packets dropped during an update (Sec. 3.5.2), and favorably to COCONUT, CU [86], and TSU [59] in terms of the packets dropped due to link failures (Sec. 3.5.3).

- SCC deploys rules more quickly than CU, COCONUT, or TSU (Sec. 3.5.4) and imposes less rule storage overhead in amount and/or duration than these alternatives (Sec. 3.5.5).

- The rule generation time of our algorithm scales across a range of both fat-tree and ISP topologies (Sec. 3.5.6).

- The buffering overhead imposed by our algorithm is manageable for today's switches (Sec. 3.5.7).

3.5.1 Setup

For the tests in Secs. 3.5.2–3.5.5 and 3.5.7, our experiments were conducted on topologies emulated in Mininet [68] on a 2.1GHz quad-core CPU with 8GB of memory. We used a fat-tree topology with $K = 8$ ports per switch, and one ISP topology (DFN from Topology

Zoo [95]) for these tests. The $K = 8$ fat-tree contained 80 switches, and IP addresses were assigned as prescribed by Al-Fares et al. [2]. The DFN topology contained 58 switches and 87 links. To simulate the delay between the controller and switches, we randomly sampled values from a normal distribution measured by Huang et al. [35], specifically with mean 150ms and standard deviation of 7.1ms. To create realistic path changes on the fat-tree networks, we replayed a log of route changes collected from Facebook's network [12]. For the ISP topologies, we used shortest-path routing and induced route changes by breaking links. We used Hping [33] to craft packets in our Open vSwitch tests. We used Scapy [88] in our P4 tests, as this tool enabled us to craft custom packet headers. In our tests, there was one flow per source-switch/destination-switch pair, and we parameterized many of our tests by the packet-sending rate per flow. So, for example, a rate of 1000 packets per second indicates that 1000 packets flowed from each source switch to each other destination switch per second.

The tests in Sec. 3.5.6 focused on rule generation times and so did not require a network emulator. These tests used fat-tree topologies ($K = 8$ and $K = 6$) with routing changes again taken from the Facebook dataset, as well as multiple ISP topologies taken from Topology Zoo. These tests were executed on a 32-core, 2.1GHz computer with 256GB of memory, though did not require such a heavily resourced machine.

In our evaluations in Secs. 3.5.2–3.5.5, we primarily compare our algorithm (SCC) to COCONUT [25], CU [86], TSU [59] and "original" deployment, based on our own implementation of each. To interpret the results we report, it is therefore useful to briefly recall how these designs update routing policies. We use an example in Fig. 3.6 to show how each algorithm works. In the example, flow f changes its path from $S_1 \to S_2 \to S_3 \to S_4 \to S_5 \to S_6$ (i.e., the dashed line) to $S_1 \to S_2 \to S_7 \to S_8 \to S_5 \to S_6$ (the solid line).

CU Consistent Updates (CU) uses a two-phase commit to apply rule updates atomically throughout a network. Each ingress switch timestamps each inbound packet with a times-

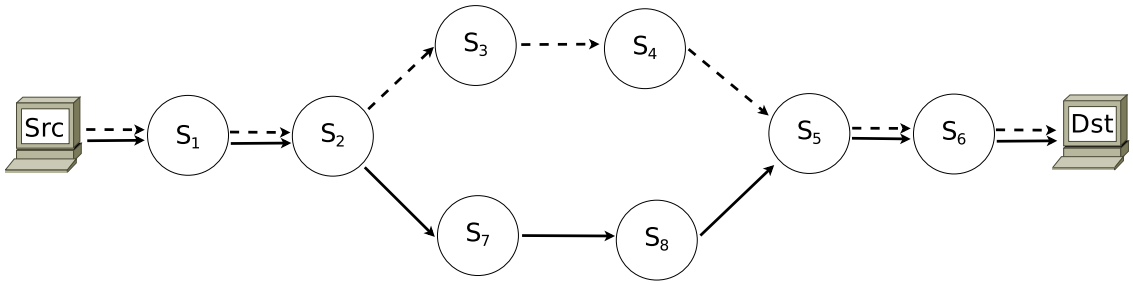


Figure 3.6: Example of CU, TSU, and COCONUT. The dashed line is the old path, and the solid line is the new path.

tamp indicating the current epoch, and each downstream switch uses rules with the same epoch timestamp to match the packet. In Fig. 3.6, S_1 tags packets of f with a timestamp 5 initially and the rule at each downstream switch (e.g., S_2) uses the timestamp 5 to match the packets. The timestamp carried by the packet will not be changed as it traverses the network. Therefore, packets carry the timestamp 5 all the way to S_6. S_6 removes the timestamp from the packets and forwards them to the destination. The update phase deploys—but does not yet enable—rules for the new epoch (with a new timestamp) at all switches. The controller installs a new rule with the timestamp 6 on S_2, S_7, S_8, S_5, and S_6 separately. After all the new rules have been installed, the controller updates the ingress switch (i.e., S_1) to start tagging packets with the new epoch timestamp (6), resulting in the new epoch's rules being applied to any packet carrying the new epoch timestamp. Packets are thus forwarded through the new path. This atomic update will make each packet traverse either its old or new path in its entirety. After the controller learns that all ingress switches are now timestamping with the new epoch timestamp, and after waiting sufficient time for any packets timestamped for the old epoch to have departed the network, the controller instructs all switches to delete the old epoch rules. In Fig. 3.6, S_2, S_3, S_4, S_5, and S_6 remove old rules with the timestamp 5.

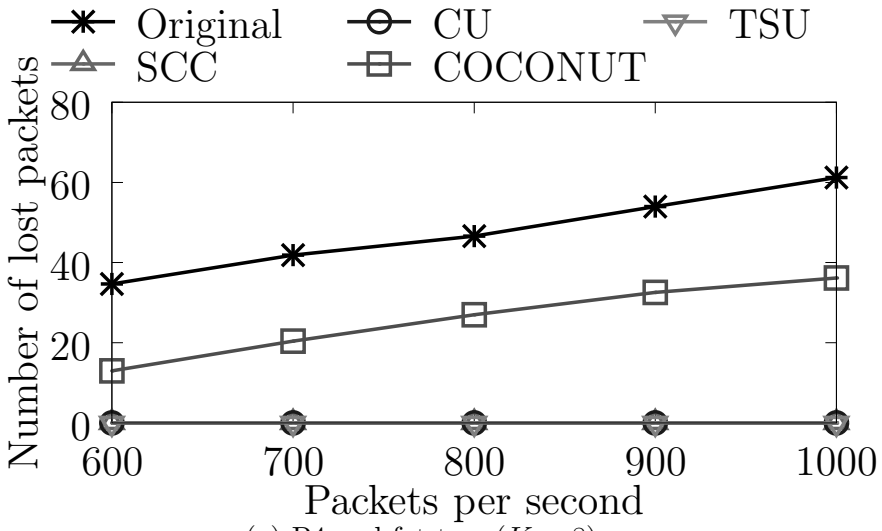
COCONUT As discussed in Sec. 2.3, COCONUT has somewhat different goals than SCC. However, by treating the entire network as a "logical device" and each epoch as a "logical rule" (in their terminology), we can adapt the COCONUT design to implement properties similar (though not identical) to ours. If interpreted this way, COCONUT behaves similarly

to CU by operating in phases: the controller deploys (but not yet enables) a new epoch's rules to switches in a first phase, then enables the new rules in a second, and then deletes the old rules in a third. The controller begins each phase after the previous completes. In Fig. 3.6, the controller first installs rules for the new routing policy in S_1, S_2, S_7, S_8, S_5, and S_6, but not yet enables the switches to use these rules. Second, the controller sends commands simultaneously to these switches to enable the new routing policy. Third, the controller removes rules for the old routing policy in S_1, S_2, S_3, S_4, S_5, and S_6. One difference from CU is that, during the second step, a packet previously routed by old rules can transition to being routed by new rules, at which point it will continue to be routed by new rules (since it carries a vector timestamp with a single component or, in other words, a traditional Lamport timestamp for the new epoch). For this reason, the controller need not delay between the second and third phases to give packets routed by old rules time to depart the network. There is a fourth stage in COCONUT that adjusts new rules' priorities, due to its implementation strategy to leverage priorities to ensure that a packet matches a new rule even when both old and new rules covering it are still installed in the switch.

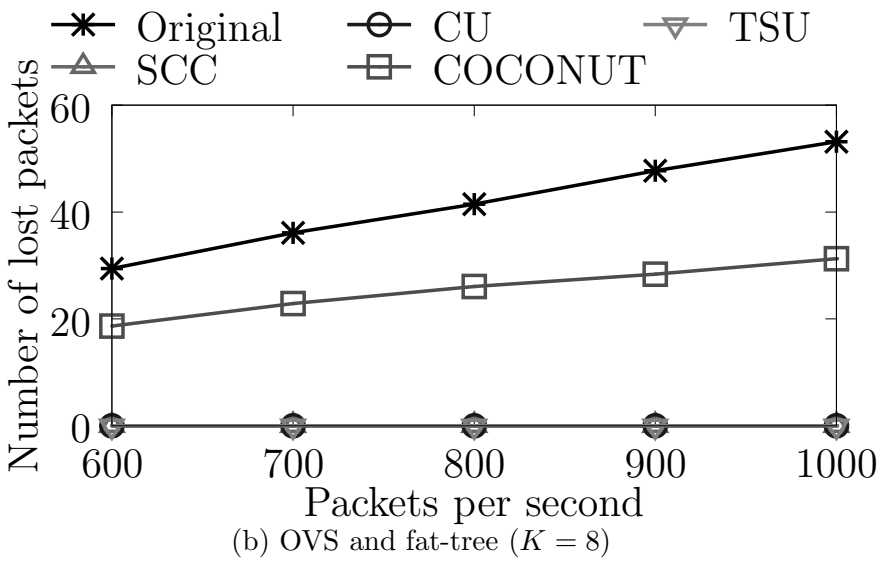
TSU Transiently secure network updates (TSU) commits network updates incrementally and so in multiple steps. In each step, the controller sends updates to a subset of switches, and waits for their installation before the next step. Therefore, during the update, old rules may be used by some switches, while other switches may forward packets according to the new policy. However, this scheme uses mixed-integer programs to compute an update schedule that minimizes the update steps while still guaranteeing some desirable properties (e.g., loop freedom and waypoint enforcement). Consider a loop-freedom update for the example in Fig. 3.6. S_7 and S_8 are first updated to install rules for the new routing policy. Then S_2 is changed to forward packets to the new path. Finally, rules for the old routing policy in S_3 and S_4 are deleted. Although there may not always be a feasible update schedule that achieves these properties, TSU does not require packet tagging and does not need extra

rules on the switches.

Original "Original" deployment commits network updates in only one step. Specifically, the controller sends updates to switches (Fig. 3.6, S_2, S_3, S_4, S_7, and S_8) without using any synchronization algorithm. Since messages from the controller to switches may suffer different delays, this deployment does not enforce any consistency.



(a) P4 and fat-tree ($K = 8$)



(b) OVS and fat-tree ($K = 8$)

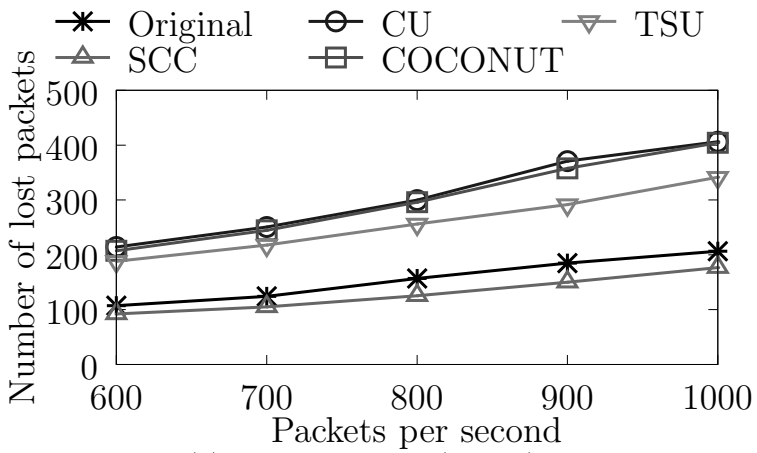
Figure 3.7: Packet loss during normal update. Each data point is an average of 100 runs.

3.5.2 Packet Loss During Regular Update

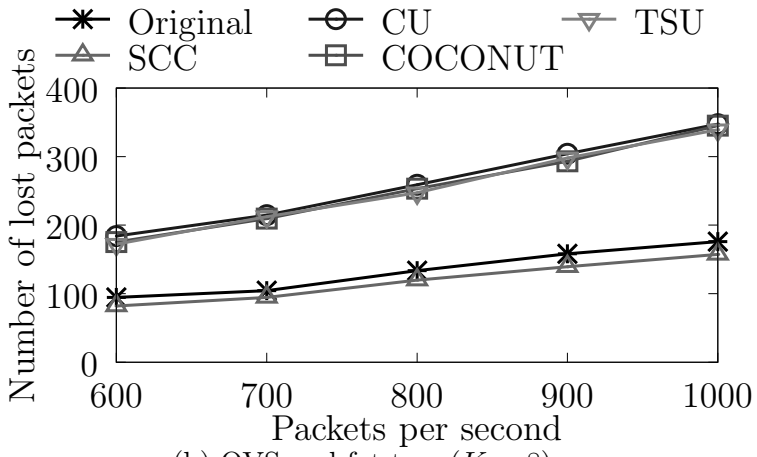
We measured the number of lost packets based on different packet sending rates when pushing updates to the switches concurrently to change the paths of packets in the fat-tree topology ($K = 8$). As shown in Fig. 3.7, our protocol (SCC), TSU, and CU incur no packet loss because these two mechanisms maintain black-hole freedom. In addition, we set the TTL of each packet to be twice its old path length plus its new path length. So no packet loss also indicates bounded looping, since the packet will be dropped if its TTL reaches zero. In contrast, the normal deployment without any consistent update mechanism and COCONUT dropped packets because they do not prevent the case where a switch forwards packets using an old rule to a switch that is not on the new path for this packet and that has already deleted (or deprecated) its old rule. The "original" deployment approach also drops packets in other cases that COCONUT addresses (and that CU, TSU, and SCC also address).

3.5.3 Packet Loss During Link Failure

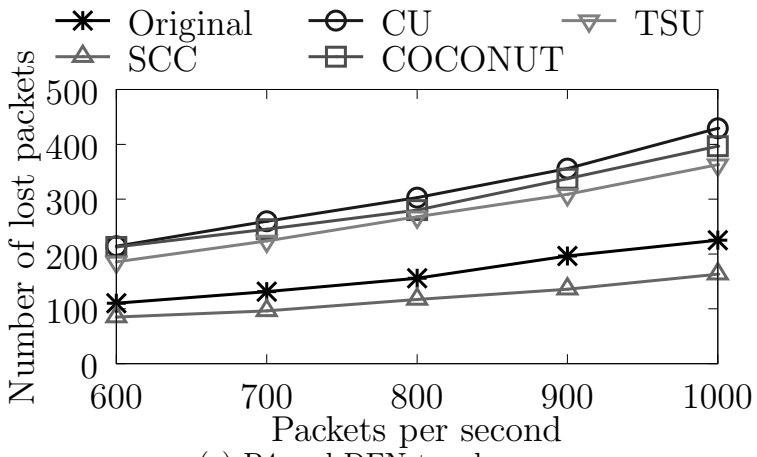
In the tests reported in this section, we broke one randomly chosen link of some existing path, forcing a new epoch with new paths for the flows traversing that link. During the delay to update the network with the new paths, packets on those flows were lost. For the $K = 8$ fat-tree topology (Figs. 3.8(a)–3.8(b)), the response time included rule generation (i.e., the delay for executing our algorithm), while we pre-computed the rules for the DFN topology (Fig. 3.8(c)). The DFN results for Open vSwitch are similar to those for P4 and so are omitted for brevity. As we can see in Fig. 3.8, SCC dropped fewer packets than COCONUT, TSU, and CU because our protocol has smaller delay to put the new configuration in place. Specifically, in SCC, new rules can be applied as soon as they are installed in the switch. However, CU and COCONUT require that updated rules reach all switches on the new paths before any of them can start to be used to route packets, and TSU deploys new rules over multiple steps. Also, SCC outperforms the "original" deployment due to the consistency that SCC offers; e.g., SCC prevents packets forwarded using a new rule from then being matched to an old rule or otherwise dropped.



(a) P4 and fat-tree ($K = 8$)



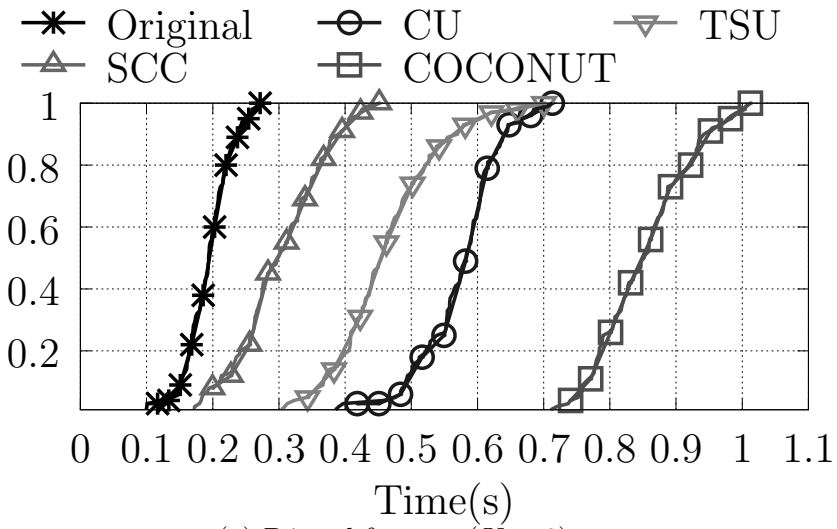
(b) OVS and fat-tree ($K = 8$)



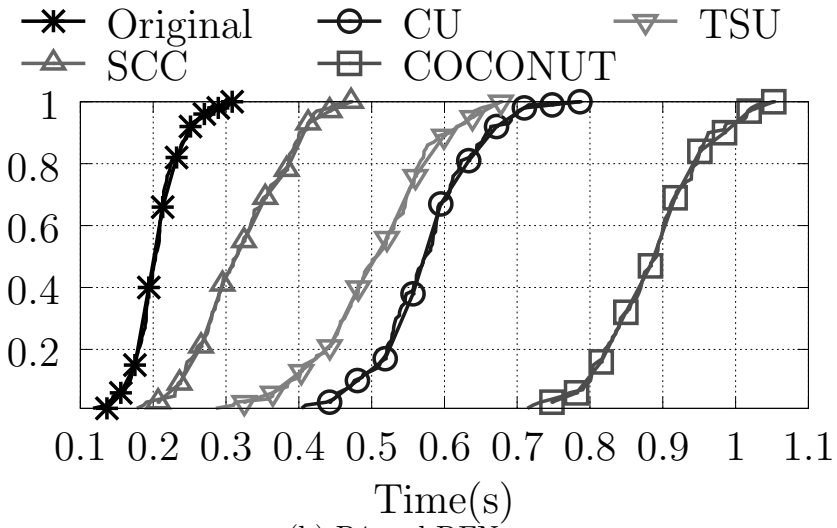
(c) P4 and DFN topology

Figure 3.8: Packet loss during link failure. Each data point is an average of 100 runs.

3.5.4 Rule Deployment Time



(a) P4 and fat-tree ($K = 8$)



(b) P4 and DFN

Figure 3.9: CDF of rule deployment times over 100 epochs.

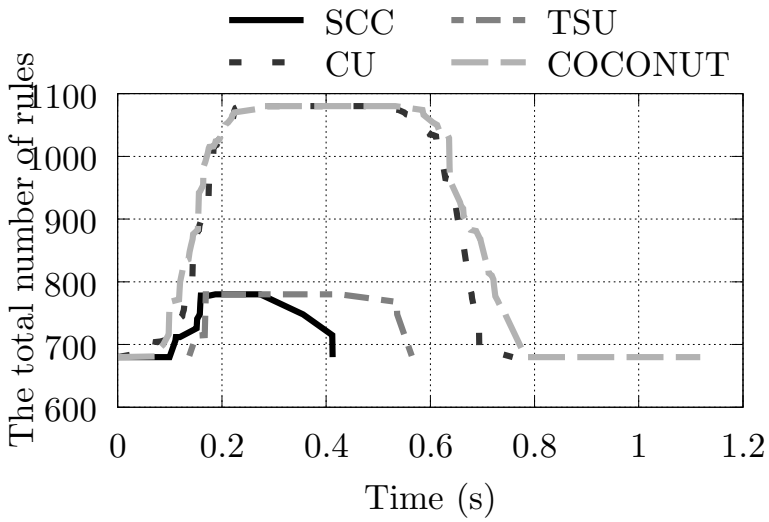
In the tests reported in this section, we measured the deployment time of rule updates, including both the new rule installation and old rule cleanup (and, in our case, send-back rule cleanup). Each epoch in these tests involved one path change, and Fig. 3.9 shows the distribution of rule deployment times for 100 such epochs for the fat-tree topology ($K = 8$). SCC rule deployment is considerably faster than TSU, CU and COCONUT, with the vast majority of the 100 SCC deployments completing before even a minority of the TSU and CU

deployments and well before any COCONUT deployments. Total completion time of SCC is only slightly larger than for the "original" protocol, owing to extra rule cleanup.

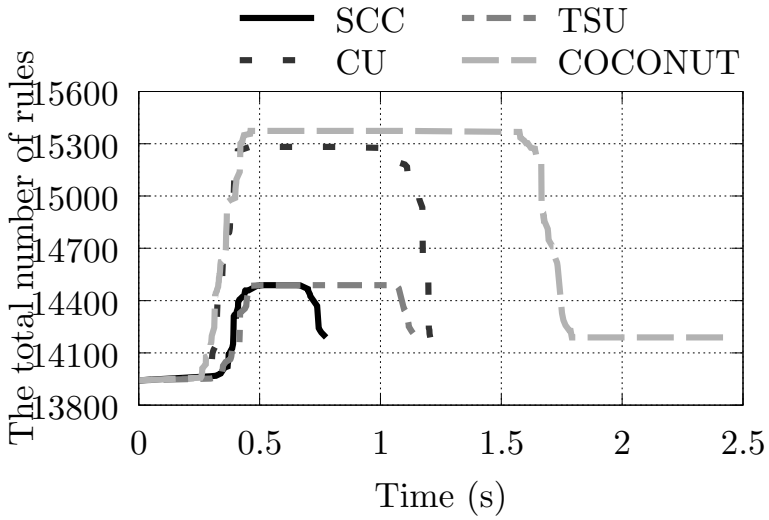
SCC outperforms CU and COCONUT during rule deployment for two reasons. First, SCC simply deploys rules to fewer switches, since it attempts to minimize the number of switches to which it must do so. Second, SCC involves fewer phases of communication between the controller and switches. Here, COCONUT is worse than CU because it requires more time to clean up old rules. TSU is better than CU because CU needs to update more switches.

3.5.5 Memory Overhead in Switches

To evaluate the number of rules imposed on the switches by each algorithm, we examined the per-switch logs of rule installations and deletions over 100 consecutive path changes in the fat-tree topology ($K = 8$). We computed a time series of the total number of rules installed across all switches in the network, if all 100 path changes were included in one epoch. This time series for each of SCC, CU, and COCONUT is shown in Fig. 3.10(a). We repeated this evaluation on the DFN topology, but by breaking the "busiest" link; see Fig. 3.10(b). Here, the "busiest" link is the link that causes the most path changes when the link fails, which was 306 path changes in this case. As these figures show, SCC induced a rule overhead of significantly fewer rules than CU and COCONUT, because SCC installs extra, send-back rules only on selected switches on old paths. In particular, SCC does not temporarily retain old rules along with new rules, like CU and COCONUT do. Though TSU does not add extra rules on switches, old rules are kept longer because TSU executes in multiple steps.



(a) P4, fat-tree, 100 path changes



(b) P4, DFN, 306 path changes

Figure 3.10: Rules in the network during epoch installation.

3.5.6 Rule Generation Time

Rule generation times for SCC are shown in Fig. 3.11. There are two groups of box-plots shown in Fig. 3.11: one for fat-tree topologies, and one for ISP topologies. Of the listed ISP topologies, the two numbers following each topology name (e.g., "40" and "61" in "Geant2012(40,61)") are the number of switches and links in the topology, respectively. The numbers in Fig. 3.11 for the fat-tree topologies are rule-generation times where the new epoch differs from the old epoch by a sequence of 100 route changes present in the Facebook

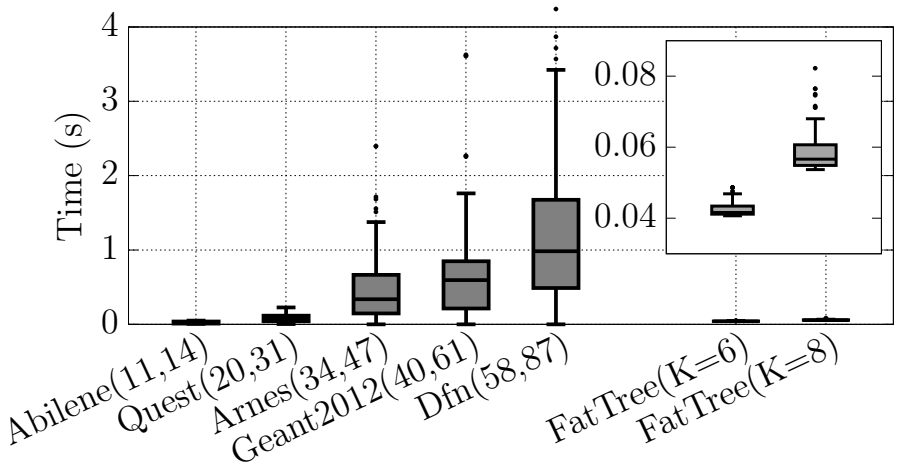


Figure 3.11: Distributions of SCC rule-generation times (100 path updates in fat-tree topologies; random link failure in ISP topologies).

data (as in Fig. 3.10). The experiments with ISP topologies reflect the cost of rule generation when a random link fails, causing all routes traversing it to change.

Fig. 3.11 shows distributions of rule-generation times as box-plots. Each box shows the first, second (median), and third quartiles, and whiskers extend to cover points that fall within 1.5× the interquartile range. Outliers are shown as dots.

Rule-generation times were minimal for the fat-tree topologies, even for epochs modifying 100 routing paths. Rule generation times for ISP topologies were more substantial, but typically completed in under 1s for all but one topology (DFN). Rule generation for DFN rarely exceeded 3s, but in these cases, the link failure induced changes in over 250 routes. While we are encouraged by these results, there are numerous opportunities for optimization in our current codebase (e.g., parallelization).

3.5.7 Buffer Overhead

In the tests reported in this section, we measured the peak number of buffered packets on any switch, for different packet sending rates. Each epoch in these tests involved ten path changes, and Fig. 3.12 shows the distribution of the maximum number of simultaneously buffered packets at any switch for 100 such epochs for the fat-tree topology ($K = 8$). As shown in the figure, the number of buffered packets per switch rarely exceeded 200 and

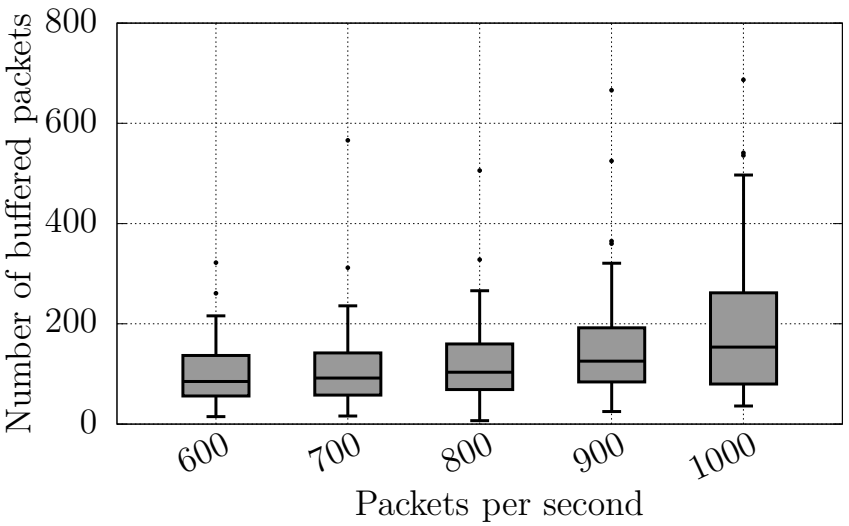


Figure 3.12: Distributions of SCC maximum switch buffering during 10 path updates in fat-tree ($K = 8$).

the buffering grew linearly as a function of packet sending rate. Buffer sizes of commodity switches are usually in the MB or even GB range [10], and so these buffering obligations should not be problematic.

CHAPTER 4: NIMBLE: FAST AND SAFE MIGRATION OF NETWORK FUNCTIONS

Stateful network functions (NFs) are a staple of modern network infrastructures. For example, network intrusion detection/prevention systems (NIDS/NIPS) is critical to ensure network security. However, the consequences of missing packets can be significant, and methods to sneak attacks past NIDS/NIPS to destination targets have a long history (e.g., [11, 13, 17]). The risk of traffic sidestepping intended NFs is particularly acute during routing-policy updates. Even if NFs remain in the same place during the update, packets that transition from a point upstream of the NF on the old routing path to a point downstream from the NF on the new routing path can result in an NF missing these packets. Routing-policy update algorithms that ensure *consistent update* (e.g., [86, 59]) can guarantee that all traffic gets processed (again, when the NF doesn't itself move), for example by ensuring that each packet traverses either its old path in its entirety or its new path in its entirety.

In this chapter, we provide a method and accompanying implementation, called Nimble, for interleaving routing-policy update and NF migration in a software-defined network (SDN), in a way that significantly reduces the latency to achieve both and without permitting packets to evade processing by NFs. Our technique works with any route-update protocol that implements a property we call *relaxed waypoint correctness*, which includes consistent-update protocols like CU [86] and our SCC algorithm. However, we provide a route-update protocol that is customized to achieve relaxed waypoint correctness without conforming to conventional "consistent update" semantics, as typically defined for such protocols.

As we will show, permitting both routing updates and NF migrations to proceed concurrently is a delicate endeavor that is fraught with corner cases. To holistically address these

cases while synchronizing these tasks as little as possible, Nimble leverages targeted buffering and packet marking in the network to coordinate packet processing with NF migration. The benefits to this approach are myriad, however, including lower latency for completion of both tasks and, depending on the routing-update protocol with which NF migration is being deployed and the circumstances requiring their update, reduced packet loss and/or reduced rule overhead in switches.

We have implemented Nimble on Open vSwitch [78] and the Ryu controller [87]. We evaluate implementations of Nimble building on both CU [86] and SCC, as well as a route-update protocol of our own design to satisfy specifically the relaxed waypoint correctness property that we define. We empirically compare our implementations of Nimble to OpenNF [24] and SwingState [61], and demonstrate the benefits of our design in both FatTree and ISP topologies.

4.1 Framework and Goals

4.1.1 SDN Model

We adopt an SDN model, in which a *controller* deploys *rules* to distributed *switches* to implement routing policy.

Flows As is standard, we define a *flow* to consist of packets with the same addressing information, i.e., IP 5-tuple. The space of all possible such 5-tuples, and so the space of all flows, is denoted $\mathbb{F}$. We denote by $\mathbb{F}^* \subset \mathbb{F}$ the space of possible flows between switches or between a switch and the controller. When convenient, we treat a flow f as a set of all possible packets with addressing information defined by f and use $pkt \in f$ to denote a packet pkt with the addressing information of f.

Controller The network has a logically centralized controller that is responsible for configuring the switches to update the route of each flow. The controller executes an SDN application consisting of a *route generator* and an *update scheduler*. The route generator

decides whether to change the routes of flows by monitoring the network conditions and topology changes. The output of the route generator is the routing path of each flow f.

The SDN update scheduler produces *rules* (defined below) to be deployed on each switch, and to schedule these switch updates to preserve certain network routing properties when transitioning from the old routing policies to the new. Specifically, the update scheduler outputs a schedule of rule deployment in ℓ steps, denoted as $U_1, U_2, ..., U_\ell$. Each U_r includes a set of flowadd and flowdel commands to add or remove rules on switches; we discuss these commands and the structure of rules below. The commands in U_r are issued simultaneously to switches in the network, and updates of U_r are finished before U_{r+1} begins.

We refer to an invocation of the SDN application to reconfigure routing policy as a new *epoch*. We assume that each routing policy change completes—i.e., its rules are deployed throughout the network—before the next epoch begins.

Switches Similiar to the switch model for SCC algorithm in Chapter 3, each switch maintains a flow entry table which stores a set of rules (see below) for flow management. We denote the set of rules in the flow table of switch S as S.ruleSet; e.g., $S.\mathsf{ruleSet} = \{R_1, R_6, R_{10}\}$ means that switch S includes rules R_1, R_6 and R_{10}. The controller modifies this set by invoking the following interface, which is similar to that provided by OpenFlow:

- $S.\mathsf{flowadd}(R_j)$ inserts rule R_j into $S.\mathsf{ruleSet}$. This command fails with no effect if $R_j.\mathsf{switch} \neq S$ (i.e., R_j should not be installed at S) or if $S.\mathsf{ruleSet}$ already contains a rule $R_{j'}$ such that $R_{j'}.\mathsf{priority} = R_j.\mathsf{priority}$, $R_{j'}.\mathsf{cover} \cap R_j.\mathsf{cover} \neq \emptyset$ (i.e., some packets can be matched by both R_j and $R_{j'}$). The meanings of these rule fields are described below.
- $S.\mathsf{flowdel}(R_j)$ removes rule R_j from $S.\mathsf{ruleSet}$.

We assume that switches support *bundling*, i.e., that a set of invocations from the controller will collectively be performed as a single atomic transaction with respect to packet processing by the switch.

Rules The instructions for how a switch should treat packets are specified by *rules*. When a packet arrives at a switch, it can be *matched* to at most one rule installed on the switch, which determines what happens to the packet. Similiar to the definition of rules in Chapter 3, each rule R includes (at least) the following fields, all of which are immutable:

- $R.\mathsf{switch}$ specifies the unique switch S into which R can be installed.
- $R.\mathsf{priority}$ specifies the priority of this rule, with higher priorities indicated by larger numbers, and with a special priority ∞ to represent the maximum priority, which can be used only by our algorithm;
- $R.\mathsf{cover} \subseteq \mathbb{F}$ specifies the flows to which this rule can be matched, i.e., a packet pkt can be matched to R only if $f \in R.\mathsf{cover}$ for the flow f containing pkt.
- $R.\mathsf{sendTo}$ specifies the switch identifier (in practice, an outbound port) to which packets matched to this rule should be forwarded. If $R.\mathsf{sendTo}$ is switch S_j, then there must be a link between $R.\mathsf{switch}$ and S_j.

4.1.2 Network Functions

Our goal is to extend the SDN model described above to support *network functions*. Below we describe the form of these network functions and the basic correctness requirement we have for their traversal.

Network functions A *network function* NF_i is an object with an immutable field $NF_i.\mathsf{flowSpec} \subseteq \mathbb{F}$ and a method $NF_i.\mathsf{processPkt}$ that takes as input a packet pkt in some $f \in NF_i.\mathsf{flowSpec}$ and outputs a (possibly empty) set of packets P, also in f. If pkt is part of a flow $f \notin NF_i.\mathsf{flowSpec}$, then $NF_i.\mathsf{processPkt}(pkt)$ has no effect. Correctness of NF_i is defined by a *sequential specification* that specifies its correct behavior when $NF_i.\mathsf{processPkt}$ is invoked sequentially, i.e., so that each method invocation returns before the next invocation begins, and that execution of $NF_i.\mathsf{processPkt}$ is linearizable [31]. Let n be the number of network functions; i.e., the network functions are $NF_1, \ldots, NF_n$.

Waypoint correctness Let $[w] = \{1, \ldots, w\}$. For each flow f, there is a specified injective function $\mathsf{wp}_f : [n_f] \to [n]$ where n_f is the number of network functions that should process packets of f sequentially on the entire path ($n_f \geq 0$) and $\mathsf{wp}_f(k)$ is the k-th network function that packets in f must traverse. We require that if $\mathsf{wp}_f(k) = i$, then $f \in NF_i.\mathsf{flowSpec}$. Our correctness condition is that the network enforce the waypoint property, i.e., for any f and any packet pkt in f that enters the network,

- if $n_f > 0$, then $NF_i.\mathsf{processPkt}(pkt)$ is invoked for $i = \mathsf{wp}_f(1)$;
- for each k, $1 \leq k < n_f$, if $NF_i.\mathsf{processPkt}(pkt')$ is invoked for $i = \mathsf{wp}_f(k)$, outputting P, then $NF_{i'}.\mathsf{processPkt}(pkt'')$ is invoked for $i' = \mathsf{wp}_f(k+1)$ and for every $pkt'' \in P$;
- no other invocations of any network function occur except by the above two rules; and
- if $NF_i.\mathsf{processPkt}(pkt')$ is invoked for $i = \mathsf{wp}_f(n_f)$, producing output P, then every pkt'' $\in P$ is forwarded to its destination.

The first condition guarantees that if packets of f need to be processed by at least one network function, they must be processed by the first NF $\mathsf{wp}_f(1)$. Together with the first condition, the second condition ensures that packets of f are processed by network functions sequentially. The third condition prevents packets from being processed by network functions that are not specified. We use the last condition to guarantee the delivery of packets to the destination. Let $n_{\mathsf{max}} = \max_f n_f$, i.e., n_{max} is the maximum number of waypoints that any flow can be required to traverse.

4.2 Migratable Network Functions

Our framework uses an existing SDN route-update algorithm to generate rules and schedules to update routing policy. Specifically, with new routing policies as input, the route-update algorithm generates a schedule of rule deployment to update switches. Most prior works on SDN routing-policy updates that achieve waypoint enforcement (e.g., [86, 59, 60]) do so assuming that NFs remain at fixed locations of the network during the routing-policy update. On the contrary, here we allow NF locations to change from one epoch to the next, and our contribution lies in ensuring that waypoint enforcement continues to hold.

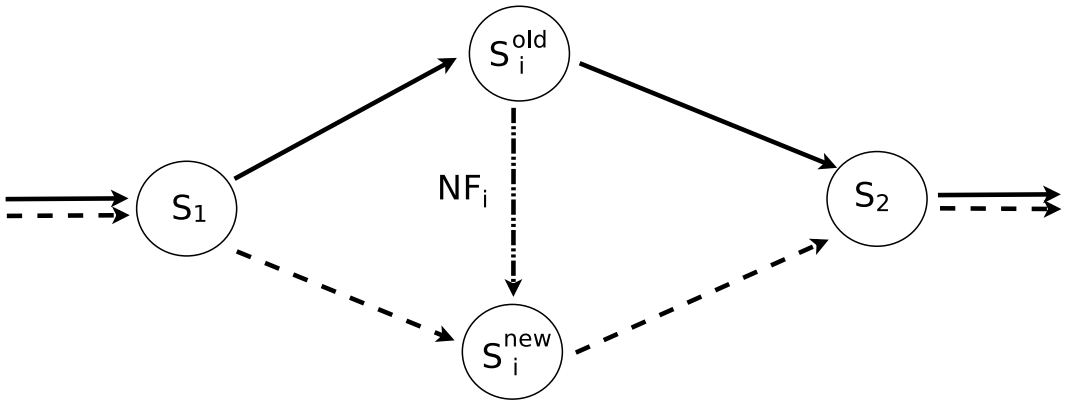


Figure 4.1: Example of NF migration

Several works (e.g., [85, 23, 61]) have explored the possibility of migrating network functions in concert with routing-policy updates. A strength of these approaches is that they make no assumptions regarding properties of the underlying route-update algorithm, except that it eventually deploys the rules to faithfully implement routing policies. However, because these NF-migration algorithms must tolerate any transitory behavior of the underlying route-update algorithm, they necessarily must be conservative in how they migrate NFs, to ensure that no packets bypass their NF waypoints while the routing policy is updated. In fact, for this reason, all of them permit routing-policy updates to proceed only after all NF migrations have completed.

To understand the challenges in permitting migration alongside route updates, consider a network function NF_i that migrates from the old position S_i^{old} to the new position S_i^{new}, shown in Fig. 4.1. The flow f, which should be processed by NF_i, also needs its path to be updated from $S_1 \rightarrow S_i^{\mathsf{old}} \rightarrow S_2$ to $S_1 \rightarrow S_i^{\mathsf{new}} \rightarrow S_2$. The path change can be accomplished by updating the rule matched to f at S_1. Migrating NF_i and updating the path of f without coordination can be harmful, however. For example, if the controller sends commands to migrate NF_i and updates S_1 simultaneously, S_1 might be updated before NF_i migrates to S_i^{new}. Then, packets of f might start to arrive at S_i^{new} before NF_i can process packets there, which may cause problems since packets can bypass NF_i. Also, if S_1 has not been updated by the time NF_i leaves S_i^{old}, packets may arrive at S_i^{old} with NF_i no longer there; depending on how S_i^{old} handles these packets, this could result in packet loss or packets bypassing NF_i.

Our goal is to migrate NFs and update routing-policies efficiently while ensuring packets are processed by NFs correctly. Specifically, our contribution here is an algorithm that leverages consistency properties of underlying routing-update algorithms to permit NFs to be migrated *alongside* rule deployment for new routing policies, more efficiently than simply serializing routing-policy update after NF migration. In particular, our approach demonstrates that by leveraging an SDN routing-policy update algorithm that provides a property that we call *relaxed waypoint correctness* (see Sec. 4.3.1), we can implement waypoint correctness when NFs are allowed to change locations much more efficiently than known approaches to achieving both.

4.2.1 Component Changes

To support NF migration, we require that the controller, switches, and rules be functionally enhanced in the following ways. Below we refer to each NF being hosted at a switch; this hosting could be implemented on the switch for a simple NF or at an attached middlebox for a more complex one.

Controller The output from the route generator is also provided to an NF application (see Fig. 4.2), to determine for each flow f the switch at which f will be processed by each of its waypoint NFs; each NF will need to be migrated to its corresponding switch as determined by the NF application. It is necessary to assume that there is at least one switch S such that for every $f \in NF_i.\mathsf{flowSpec}$, S is included in the path of f, as else there is no switch to where NF_i can be migrated to process every flow in $NF_i.\mathsf{flowSpec}$.

Switches We add three new switch interfaces to perform NF migration.

- $S.\mathsf{export}(i, j)$ marshals NF_i into a set P of packets with source address (the IP address) of S and destination address of S_j, and outputs P. $S.\mathsf{export}(i, j)$ executes only while no $NF_i.\mathsf{processPkt}$ invocations are underway at S, and S no longer permits invocations of $NF_i.\mathsf{processPkt}$ once $S.\mathsf{export}(i, j)$ completes.
- $S.\mathsf{import}(i, j)$ instructs switch S to await the arrival of packets P from S_j, from which to

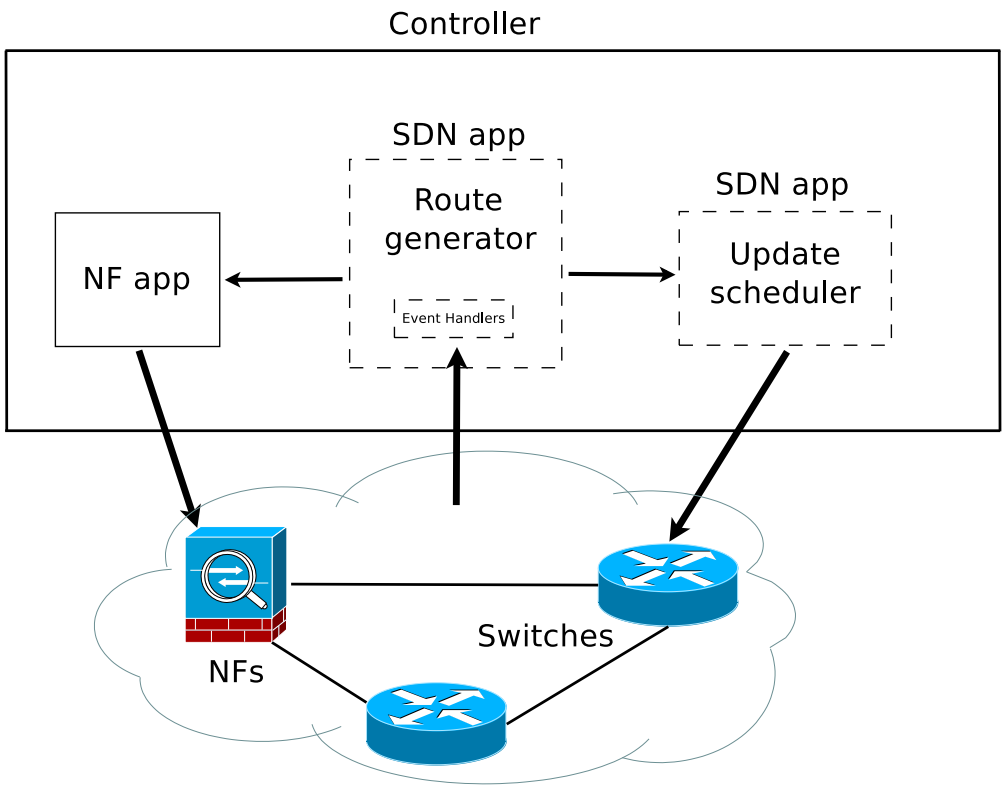


Figure 4.2: Typical components of a network controller

reconstitute function NF_i locally. This invocation causes S_j to allocate two buffers, an *inbound* buffer to hold packets to be processed in NF_i.processPkt invocations once NF_i is reconstituted locally, and an *outbound* buffer to hold packets output from NF_i.processPkt invocations. Starting from this invocation and until NF_i is reconstituted locally, packets matched to any $R \in S$.ruleSet for which R.sendTo $= NF_i$ (see below) are buffered in the inbound buffer for NF_i. Packets output from NF_i.processPkt invocations are buffered in the outbound buffer for NF_i, until a S.release(i) invocation.

- S.release(i) releases the packets buffered in the outbound buffer for NF_i to be matched against S.ruleSet, and disables buffering so that packets inbound to or outbound from NF_i are no longer buffered.

S.import, S.export, and S.release can be invoked by the controller, just like S.flowadd and S.flowdel.

Waypoint counters We add to each packet a field, called its *waypoint counter*, that can hold any value in $[n_{\mathsf{max}} + 1] = \{1, \ldots, n_{\mathsf{max}} + 1\}$. Upon arrival in the network, a packet's ingress switch initializes the packet's waypoint counter to 1. In brief, this counter is incremented in the packet as it is submitted to each of its waypoints for processing (see below). In this way, rules can treat a packet differently depending on how many of its waypoints it has already traversed.

Rules We extend rules to include a new field $R.\mathsf{wpCtr}$ that takes on a value in $[n_{\mathsf{max}}] \cup \{*\}$, and stipulate that a packet can be matched to this rule only if $R.\mathsf{wpCtr} = *$ or the packet's waypoint counter equals $R.\mathsf{wpCtr}$. As such, when packet pkt in flow f arrives at switch S, pkt is matched to the highest priority rule $R \in S.\mathsf{ruleSet}$ for which $f \in R.\mathsf{cover}$ and either $R.\mathsf{wpCtr} = *$ or the packet's waypoint counter equals $R.\mathsf{wpCtr}$; we denote this rule as $matchRule(S, pkt)$. If there is no $R \in S.\mathsf{ruleSet}$ to which pkt can be matched, then pkt is dropped.

We also extend rules to accommodate additional functionality related to the $R.\mathsf{sendTo}$ field.

- $R.\mathsf{sendTo}$ can be a network function NF_i, in which case for any packet pkt it matches to R, $S = R.\mathsf{switch}$ increments the pkt's waypoint counter and then submits pkt to NF_i in an $NF_i.\mathsf{processPkt}(pkt)$ invocation. If NF_i is not hosted locally at $R.\mathsf{switch}$, then the packet must be buffered (in the inbound buffer for NF_i, see above) until it is. Any packets returned from the $NF_i.\mathsf{processPkt}(pkt)$ invocation are matched again to $S.\mathsf{ruleSet}$. We do not require NF_i to process packets' waypoint counters, but we do require that any packets $NF_i.\mathsf{processPkt}(pkt)$ emits bear the same waypoint counter as pkt.
- $R.\mathsf{sendTo}$ can take on two more possible values, namely functions $\mathsf{encap}[f]$ and decap. If a switch S matches packet pkt to rule $R \in S.\mathsf{ruleSet}$ where $R.\mathsf{sendTo} = \mathsf{encap}[f]$, then the packet is encapsulated into a packet for flow f, which is then resubmitted for matching against $S.\mathsf{ruleSet}$ at this switch S. A packet matched to a rule $R \in S.\mathsf{ruleSet}$ where

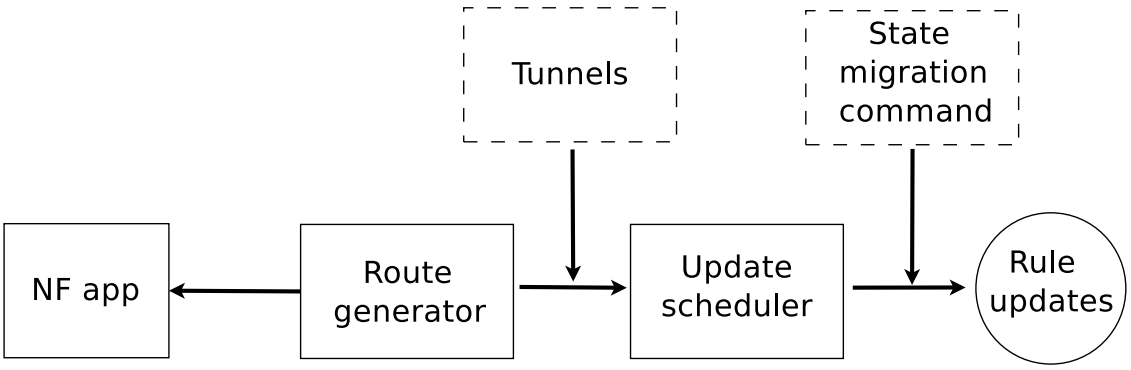


Figure 4.3: Conceptual additions by our algorithm

R.sendTo = decap is decapsulated (i.e., the existing packet header is removed) and the packet contained therein is then resubmitted for matching against S.ruleSet.

As before, all rule fields remain immutable.

4.2.2 Algorithm

Our algorithm augments the SDN framework outlined in Sec. 4.1 with two conceptual steps (see Fig. 4.3). The first instantiates routing policy for tunnels to migrate NFs from their old locations to their new locations and to relocate traffic that arrives at an NF's old location to its new location. Once these routes have been determined, the full routing policy (including these new routes) is then submitted to the routing-policy update scheduler, which produces the schedule for deploying rules to switches. The second phase of our algorithm then augments this update schedule with commands to bridge traffic on/off of tunnels as needed, to invoke each NF with packets destined for it at its new location, and to initiate migration of NFs. A later phase of our algorithm (not shown in Fig. 4.3) cleans up the bridging rules once they are no longer needed.

The first of these steps is implemented as follows, per NF_i that migrates from S_i^{old} to S_i^{new} in this epoch.

Migration routes: The controller constructs a route from S_i^{old} to S_i^{new} to carry flows f_i^{mig}, f_i^{tun} with source S_i^{old} and destination S_i^{new}. f_i^{mig}, f_i^{tun} and their associated route are added to the routing policy that is input to the update scheduler.

f_i^{mig} will be used to migrate NF_i from S_i^{old} to S_i^{new}, and f_i^{tun} will be used to tunnel

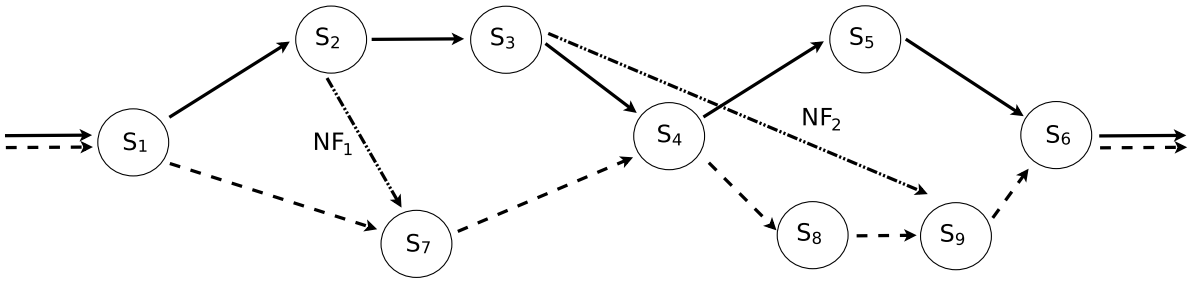


Figure 4.4: Example for algorithm description

packets from S_i^{old} to S_i^{new} that should be processed by NF_i. Because we assume that the IP addresses of S_i^{old} and S_i^{new} are distinct from the source and destination addresses of flows routed according to the policies output from the route generator, the routes chosen to carry f_i^{mig}, f_i^{tun} cannot contradict the routes output from the route generator.

Fig. 4.4 shows an example for this step. Suppose a flow f which is processed by NF_1 and NF_2 needs to be rerouted from the path $S_1 \to S_2 \to S_3 \to S_4 \to S_5 \to S_6$ (solid line) to the path $S_1 \to S_7 \to S_4 \to S_8 \to S_9 \to S_6$ (the dashed line). And consequently, the controller decides to migrate NF_1 from S_2 ($= S_1^{\mathsf{old}}$) to S_7 ($= S_1^{\mathsf{new}}$) and NF_2 from S_3 ($= S_2^{\mathsf{old}}$) to S_9 ($= S_2^{\mathsf{new}}$). $S_2 \to S_1 \to S_7$ can be selected for migration route between S_1^{old} and S_1^{new}. $S_3 \to S_4 \to S_8 \to S_9$ can be selected for migration route between S_2^{old} and S_2^{new}.

Recall that the update generator now outputs a schedule for rule deployment in ℓ steps $U_1, U_2, \ldots, U_\ell$, where each U_r includes a set of flowadd and flowdel commands. Continuing with the example of Fig. 4.4, to ensure that packets cannot bypass waypoints, the update scheduler might formulate the following three-step schedule. In the first step, S_7, S_8, and S_9 install new rules; i.e., U_1 should include the flow modification commands for these three switches. In the second step (U_2), S_4 is updated to send packets to S_8. In the third step (U_3), S_1 is modified and packets are sent through the new path. Recall that these update steps include commands output by the update scheduler to install rules to route the tunnels generated in the previous step of our algorithm.

Continuing our algorithm, it first sets $R.\mathsf{wpCtr} \leftarrow *$ for any rule R in any flowadd or flowdel command in any step of the given update schedule $U_1, \ldots, U_\ell$. Then, for each NF_i to

be hosted at a switch S_i^{new} in this epoch different from the switch S_i^{old} where it was hosted in the last, the controller performs the following steps.

Rules for routing to NF_i: The controller constructs $n_{\mathsf{max}} + 2$ rules as follows. First, the controller constructs a rule R_i^{enc} with the following fields:

- $R_i^{\mathsf{enc}}.\mathsf{switch} \leftarrow S_i^{\mathsf{old}}$
- $R_i^{\mathsf{enc}}.\mathsf{cover} \leftarrow \left\{ f \;\middle|\; \begin{array}{rl} \exists pkt, R: & pkt \in f \\ \wedge & R \in S_i^{\mathsf{old}}.\mathsf{ruleSet} \\ \wedge & R = matchRule(S_i^{\mathsf{old}}, pkt) \\ \wedge & R.\mathsf{sendTo} = NF_i \end{array} \right\}$
- $R_i^{\mathsf{enc}}.\mathsf{wpCtr} \leftarrow *$
- $R_i^{\mathsf{enc}}.\mathsf{priority} \leftarrow \infty$
- $R_i^{\mathsf{enc}}.\mathsf{sendTo} \leftarrow \mathsf{encap}[f_i^{\mathsf{tun}}]$

In addition, the controller constructs the rule R^{dec} with the following fields:

- $R_i^{\mathsf{dec}}.\mathsf{switch} \leftarrow S_i^{\mathsf{new}}$
- $R_i^{\mathsf{dec}}.\mathsf{cover} \leftarrow \{f_i^{\mathsf{tun}}\}$
- $R_i^{\mathsf{dec}}.\mathsf{wpCtr} \leftarrow *$
- $R_i^{\mathsf{dec}}.\mathsf{priority} \leftarrow \infty$
- $R_i^{\mathsf{dec}}.\mathsf{sendTo} \leftarrow \mathsf{decap}$

Finally, for each $k \in [n_{\mathsf{max}}]$, the controller constructs a rule $R_{i,k}^{\mathsf{inv}}$ with the following fields:

- $R_{i,k}^{\mathsf{inv}}.\mathsf{switch} \leftarrow S_i^{\mathsf{new}}$
- $R_{i,k}^{\mathsf{inv}}.\mathsf{cover} \leftarrow \{f \mid \mathsf{wp}_f(k) = i\}$
- $R_{i,k}^{\mathsf{inv}}.\mathsf{wpCtr} \leftarrow k$
- $R_{i,k}^{\mathsf{inv}}.\mathsf{priority} \leftarrow \infty$
- $R_{i,k}^{\mathsf{inv}}.\mathsf{sendTo} \leftarrow NF_i$

In the example of Fig. 4.4, S_2 and S_3 should install rule R_1^{enc} and R_2^{enc}, respectively, to encapsulate packets of f onto f_i^{tun}. In this way, packets arriving at S_2 or S_3 can be relocated to new positions S_7 and S_9, respectively, during NF migration. Packets relocated through these tunnels to the new NF locations should be decapsulated back to the original flow f

such that they can traverse the remainder of the new path after being processed by the appropriate NF. Therefore, S_7 and S_9 need rules R_1^{dec} and R_2^{dec}, respectively.

$R_{i,k}^{\mathsf{inv}}$ has two functions. First, it sends packets that need to be processed (i.e., with waypoint counter k where $\mathsf{wp}_f(k) = i$) to NF_i. Note that packets on f output from NF_i will have a waypoint counter of $k+1$ and thus will not be matched to $R_{i,k}^{\mathsf{inv}}$ again. Second, $R_{i,k}^{\mathsf{inv}}$ prevents packets from being processed twice by NF_i. Continuing with the example of Fig. 4.4, since at the beginning of U_3, S_4 has been updated (in U_2) but S_1 has not yet been changed, packets on f traversing a part of old path ($S_1 \to S_2 \to S_3 \to S_4$) and a part of new path ($S_4 \to S_8 \to S_9 \to S_6$) encounter both the old (S_3) and new position (S_9) of NF_2. $R_{2,2}^{\mathsf{inv}}$ at S_9 ensures packets carrying waypoint counter 2 can be processed by NF_2, but packets with waypoint counter 3 (NF_2 already processed these packets at S_3) are forwarded to next switch S_6 immediately.

Now that these rules have been generated, we need to integrate them into the update schedule. To do so, the algorithm initializes $U_{\ell+1}$ to be empty, i.e., $U_{\ell+1} \leftarrow \{\}$. Then, for each migrating network function NF_i, the controller performs the following.

Update schedule: To deploy R_i^{enc}, R_i^{dec}, $\{R_{i,k}^{\mathsf{inv}}\}_{k\in[n_{\mathsf{max}}]}$ in the update schedule U_1, U_2, ..., $U_{\ell+1}$, the controller performs the following steps.

- The controller adds $S_i^{\mathsf{new}}.\mathsf{flowadd}(R_i^{\mathsf{dec}})$, $S_i^{\mathsf{new}}.\mathsf{import}(i,j)$, and $S_i^{\mathsf{new}}.\mathsf{flowadd}(R_{i,k}^{\mathsf{inv}})$ for each $k \in [n_{\mathsf{max}}]$ to U_1, where $S_i^{\mathsf{old}} = S_j$.
- The controller searches for the last step U_r in which rules to route f_i^{mig}, f_i^{tun} are deployed. It adds $S_i^{\mathsf{old}}.\mathsf{flowadd}(R_i^{\mathsf{enc}})$ and $S_i^{\mathsf{old}}.\mathsf{export}(i,j)$ to U_{r+1} where $S_i^{\mathsf{new}} = S_j$.
- The controller adds $S_i^{\mathsf{new}}.\mathsf{release}(i)$ to $U_{\ell+1}$.

The first bullet incorporates commands to prepare switches at new positions for NF migration. In the example of Fig. 4.4, the controller issues commands $S_7.\mathsf{import}(1,2)$, $S_7.\mathsf{flowadd}(R_1^{\mathsf{dec}})$ and $S_7.\mathsf{flowadd}(R_{1,1}^{\mathsf{inv}})$ to S_7. So S_7 waits for messages from S_2 and prepares to reconstruct NF_1 locally. The controller also performs similar operations on S_9.

The second bullet incorporates commands to migrate NF_i from its old to its new position.

This should be done after the rules implementing the migration route have been deployed (i.e., after step U_r). In the example of Fig. 4.4, assume the controller deploys rules to create a tunnel $S_2 \rightarrow S_1 \rightarrow S_7$ to migrate NF_1 from S_2 to S_7 in step U_1. Then, in step U_2, the controller can use the interface $S_2.\mathsf{export}(1, 7)$ to instruct S_2 to create a set of packets to marshal NF_1 and send these packets to S_7. Upon receiving packets from S_2, S_7 reconstructs NF_1 and starts to perform $NF_1.\mathsf{processPkt}$ invocations. Meanwhile, S_2 uses the rule R_1^{enc} to encapsulate the packets of f to packets of f_1^{tun}. Packets of f_1^{tun} are then forwarded to S_7 and S_7 uses the rule R_1^{dec} to decapsulate these packets back to packets of f. The packets have not been processed by S_2 and therefore should carry the waypoint counter 1. Thus, S_7 uses the rule $R_{1,1}^{\mathsf{inv}}$ to forward packets to NF_1. After processed by NF_1, these packets are buffered at S_7 with the waypoint counter 2.

The last bullet ensures that packets released from switches at new positions can be matched to rules implementing new routing policy at all downstream switches. In the example of Fig. 4.4, the controller sends $S_7.\mathsf{release}(1)$ in step U_4. S_7 then releases the buffered packets to the network since S_4, S_8 and S_9 have installed rules to forward packets to the destination.

The last step of our algorithm cleans up migration-related rules once they will no longer be used. Specifically, for each migrated NF_i, the following is performed:

Bridging rules cleanup: After sufficient time passes to ensure that f_i^{mig} and f_i^{tun} will contain no more packets, the controller issues $S_i^{\mathsf{old}}.\mathsf{flowdel}(R_i^{\mathsf{enc}})$ and $S_i^{\mathsf{new}}.\mathsf{flowdel}(R_i^{\mathsf{dec}})$ commands.

In Fig. 4.4, this step causes the deletion of R_1^{enc} and R_1^{dec} from S_2 and S_7, respectively, and the deletion of R_2^{enc} and R_2^{dec} from S_3 and S_9, respectively. The rules to implement the migration routes $S_2 \rightarrow S_1 \rightarrow S_7$ and $S_3 \rightarrow S_4 \rightarrow S_8 \rightarrow S_9$ can also be removed, if doing so does not disrupt other routing policy.

4.3 Update Scheduling

The algorithm described in the previous section adapts a given update schedule with additional flowdel, flowadd, export, import, and release commands to migrate NFs during

path updates. In this section, we explore the requirements for the given update schedule that, when combined with the algorithm of the previous section, ensures waypoint correctness as defined in Sec. 4.1.2. We define a sufficient condition in Sec. 4.3.1. Finally, in Sec. 4.3.2 we provide an update scheduling algorithm that is tailored to implement specifically this condition.

4.3.1 A Sufficient Condition for Waypoint Correctness

In this section we give a sufficient condition for the NF-migration algorithm of Sec. 4.2 to ensure the waypoint correctness property defined in Sec. 4.1.2. Recall that during a route change, each NF_i is migrated from its old location S_i^{old} to its new location S_i^{new}, while traffic to be processed by NF_i that arrives at S_i^{old} and matched to a rule R with $R.\mathsf{sendTo} = NF_i$ is transported from S_i^{old} to S_i^{new} to be processed once NF_i is reconstituted there. Whether traffic reaches S_i^{new} via this mechanism or by the new routing policy does not matter. Rather, all that really matters is that a packet on flow f with waypoint counter k reaches either $S_{\mathsf{wp}_f(k)}^{\mathsf{old}}$ or $S_{\mathsf{wp}_f(k)}^{\mathsf{new}}$. We call this property *relaxed waypoint correctness*:

Relaxed waypoint correctness An update scheduling algorithm satisfies *relaxed waypoint correctness* if during any route update, it ensures that for each flow f and each $k \in [n_f]$, each packet on flow f with waypoint counter k reaches $S_{\mathsf{wp}_f(k)}^{\mathsf{old}}$ or $S_{\mathsf{wp}_f(k)}^{\mathsf{new}}$.[1]

Our NF-migration algorithm in Sec. 4.2 guarantees the waypoint correctness property defined in Sec. 4.1.2, if the underlying update scheduling algorithm (used by the update scheduler in the controller) satisfies the relaxed waypoint correctness. An example of an update scheduling algorithm that implements this property is CU [86], which on its own ensures that each packet traverses either the old path in its entirety or the new path in its

[1]Strictly speaking, since a packet on flow f has its waypoint counter incremented to $k+1$ right *before* processing by NF_i for $i = \mathsf{wp}_f(k)$, it is possible that this property will fail for a packet that is not then output from NF_i (e.g., because NF_i drops it). We require this property for every packet on f output from NF_i, however.

entirety. When conjoined with our NF migration algorithm, a packet that is being routed along its old path might be tunneled from S_i^{old} to S_i^{new} for processing by NF_i, after which it will be buffered until the route update is complete. From that point forward, it will be routed along its new path. A natural question is whether there are route update algorithms that satisfy relaxed waypoint correctness without enforcing a packet to traverse only the path of its old flow or of its new one. In the next section we answer this question in the affirmative.

4.3.2 Update Scheduling for Relaxed Waypoint Correctness

In this section we provide an update scheduling algorithm, which we call RWC (for "relaxed waypoint correctness"), that is specifically designed to satisfy relaxed waypoint correctness, no more, no less, whenever it is possible to achieve this property while updating each switch only once during an epoch change. The algorithm is inspired by the TSU [59] routing update algorithm, though we have adapted it to accommodate NF migration and waypoint ordering.

The algorithm computes the update schedule $U_1,\ U_2,\ \ldots,\ U_\ell$ using an optimization expressed as a 0-1 integer linear program, which can be solved (if it has a solution) using solvers like CPLEX [18] or Gurobi [28]. This algorithm assumes that the old and new routing policies differ only in a single path; i.e., a flow f (or set of flows) transitions from the same old path to the same new path. (Multiple path changes can be implemented one-by-one in multiple updates.) Moreover, this algorithm assumes that both the old and new path are loop-free.

4.3.2(a) Integer program Let $\mathbb{S}^{\mathsf{old}}$ be the set of switches that appear on the old path; $\mathbb{S}^{\mathsf{new}}$ the set of switches that appear on the new path; $\mathbb{S} = \mathbb{S}^{\mathsf{old}} \cup \mathbb{S}^{\mathsf{new}}$; $\mathbb{P}^{\mathsf{old}} \subseteq \mathbb{S}^{\mathsf{old}} \times \mathbb{S}^{\mathsf{old}}$ the links comprising the old path; and $\mathbb{P}^{\mathsf{new}} \subseteq \mathbb{S}^{\mathsf{new}} \times \mathbb{S}^{\mathsf{new}}$ the links comprising the new path. Therefore, $\mathbb{P}^{\mathsf{old}} \setminus \mathbb{P}^{\mathsf{new}}$ is the set of links that will be *disabled* by the path change (i.e., that will no longer be traversed by the rerouted flow f), and $\mathbb{P}^{\mathsf{new}} \setminus \mathbb{P}^{\mathsf{old}}$ is the set of links that will

Minimize ℓ' subject to: (4.1)

$$\ell' \geq r \cdot x_j^r \qquad \forall r \in [m], S_j \in \mathbb{S}^{\pm} \quad (4.2)$$

$$1 = \sum_{r \in [m]} x_j^r \qquad \forall S_j \in \mathbb{S}^{\pm} \quad (4.3)$$

$$y_{j,j'}^r = 1 - \sum_{r' \leq r} x_j^{r'} \qquad \forall r \in [m], (S_j, S_{j'}) \in \mathbb{L}^- \quad (4.4)$$

$$y_{j,j'}^r = \sum_{r' \leq r} x_j^{r'} \qquad \forall r \in [m], (S_j, S_{j'}) \in \mathbb{L}^+ \quad (4.5)$$

$$y_{j,j'}^r = 1 \qquad \forall r \in [m], (S_j, S_{j'}) \in (\mathbb{P}^{\mathsf{old}} \cup \mathbb{P}^{\mathsf{new}}) \setminus (\mathbb{L}^+ \cup \mathbb{L}^-) \quad (4.6)$$

$$z_{j'}^{r,k} \geq z_j^{r,k} + y_{j,j'}^{r-1} - 1 \quad \forall r \in [m], k \in [n_f], S_{j'} \in \mathbb{S}, S_j \in \mathbb{S} \setminus \{S^{\mathsf{old}}_{\mathsf{wp}_f(k)}, S^{\mathsf{new}}_{\mathsf{wp}_f(k)}\} \quad (4.7)$$

$$z_{j'}^{r,k} \geq z_j^{r,k} + y_{j,j'}^{r} - 1 \quad \forall r \in [m], k \in [n_f], S_{j'} \in \mathbb{S}, S_j \in \mathbb{S} \setminus \{S^{\mathsf{old}}_{\mathsf{wp}_f(k)}, S^{\mathsf{new}}_{\mathsf{wp}_f(k)}\} \quad (4.8)$$

$$z_j^{r,k} = 1 \qquad \forall r \in [m], k \in [n_f], S_j \in \{in(f)\} \quad (4.9)$$

$$z_j^{r,k} = 0 \qquad \forall r \in [m], k \in [n_f], S_j \in \{out(f)\} \quad (4.10)$$

$$z_j^{r,k+1} \geq z_j^{r,k} \qquad \forall r \in [m], k \in [n_f - 1], S_j \in \mathbb{S} \quad (4.11)$$

Figure 4.5: RWC integer program for generating update schedule

be *enabled* by the path change. Let $\mathbb{S}^{\pm} \subseteq \mathbb{S}^{\mathsf{old}} \cap \mathbb{S}^{\mathsf{new}}$ contain the switches at which links to carry f must be both enabled and disabled, i.e., $S_j \in \mathbb{S}^{\pm}$ iff $S_j \in \mathbb{S}^{\mathsf{old}} \cap \mathbb{S}^{\mathsf{new}}$ and for some $S_{j'}$, $(S_j, S_{j'}) \in (\mathbb{P}^{\mathsf{old}} \setminus \mathbb{P}^{\mathsf{new}}) \cup (\mathbb{P}^{\mathsf{new}} \setminus \mathbb{P}^{\mathsf{old}})$. Let $\mathbb{L}^+$ be the new links enabled at the switches in $\mathbb{S}^{\pm}$, and let $\mathbb{L}^-$ bet the old links disabled at the switches in $\mathbb{S}^{\pm}$; i.e., $(S_j, S_{j'}) \in \mathbb{L}^+$ iff $S_j \in \mathbb{S}^{\pm}$ and $(S_j, S_{j'}) \in \mathbb{P}^{\mathsf{new}} \setminus \mathbb{P}^{\mathsf{old}}$, and $(S_j, S_{j'}) \in \mathbb{L}^-$ iff $S_j \in \mathbb{S}^{\pm}$ and $(S_j, S_{j'}) \in \mathbb{P}^{\mathsf{old}} \setminus \mathbb{P}^{\mathsf{new}}$. For a natural number w, let $[w] = \{1, \ldots, w\}$.

The optimization, shown in Fig. 4.5, minimizes the number ℓ' of update steps subject to constraints (4.2)–(4.11). x_j^r is a binary indicator variable signaling whether switch $S_j \in \mathbb{S}^{\pm}$ is updated in step r; i.e., if the solution to the integer program has $x_j^r = 1$, then the controller will include its updates to S_j in U_r. Constraint (4.3) ensures that each switch in $\mathbb{S}^{\pm}$ is updated exactly once.

The binary variable $y_{j,j'}^r$ for each $(S_j, S_{j'}) \in \mathbb{P}^{\mathsf{old}} \cup \mathbb{P}^{\mathsf{new}}$ indicates whether the rerouted flows will be forwarded directly from S_j to $S_{j'}$ as of the end of update U_r. Constraint (4.4) ensures that $y_{j,j'}^r = 0$ once link $(S_j, S_{j'}) \in \mathbb{L}^-$ has been disabled, and constraint (4.5) ensures

that $y^r_{j,j'} = 1$ once link $(S_j, S_{j'}) \in \mathbb{L}^+$ has been enabled. Constraint (4.6) ensures that $y^r_{j,j'} = 1$ for any other link in $\mathbb{P}^{\mathsf{old}} \cup \mathbb{P}^{\mathsf{new}}$.

The binary variable $z^{r,k}_j$ indicates whether a packet on the rerouted flow f, upon reaching switch j after the end of update $r-1$ and before the end of update r, has yet to be processed by NF_i where $i = \mathsf{wp}_f(k)$. Constraints (4.7) and (4.8) ensure that if $y^{r-1}_{j,j'} = y^r_{j,j'} = 1$ and so the packet is forwarded directly from S_j to $S_{j'}$, and if the packet was not yet processed by NF_i upon reaching S_j (i.e., $z^{r,k}_j = 1$), then it still remains to be processed upon reaching $S_{j'}$ (i.e., $z^{r,k}_{j'} = 1$). Of course, this reasoning is valid only if $S_j \notin \{S^{\mathsf{old}}_i, S^{\mathsf{new}}_i\}$; if $S_j = S^{\mathsf{old}}_i$ then the packet will be processed by NF_i there, and if $S_j = S^{\mathsf{new}}_i$ then the packet will be buffered at S_j awaiting NF_i. Therefore, constraints (4.7) and (4.8) are included only for $S_j \notin \{S^{\mathsf{old}}_i, S^{\mathsf{new}}_i\}$. Constraints (4.9) and (4.10) indicate that the packets on flow f have yet to be processed by NF_i upon their arrival at their ingress $in(f)$ and must be processed by NF_i upon departing the network at their egress $out(f)$. Finally, constraint (4.11) ensures that if a packet has yet to be processed by NF_i for $i = \mathsf{wp}_f(k)$, then it also has yet to be processed by $NF_{i'}$ for $i' = \mathsf{wp}_f(k+1)$.

4.3.2(b) Generating the update schedule Given a solution to the integer program of Fig. 4.5, the update scheduler generates the update schedule as follows. We assume that $\mathbb{S} \setminus ((\mathbb{S}^{\mathsf{old}} \cap \mathbb{S}^{\mathsf{new}}) \setminus \mathbb{S}^{\pm})$ is the set of switches at which the new rules $\mathcal{R}^{\mathsf{new}}$ to implement the new routing policy differ from the rules $\mathcal{R}^{\mathsf{old}}$ already deployed to the network to implement the old routing policy.

- For each $S_j \in \mathbb{S}^{\mathsf{new}} \setminus \mathbb{S}^{\mathsf{old}}$, the update scheduler adds $S_j.\mathsf{flowadd}(R)$ to U_1 for each $R \in \mathcal{R}^{\mathsf{new}} \setminus \mathcal{R}^{\mathsf{old}}$ for which $R.\mathsf{switch} = S_j$.
- For $S_j \in \mathbb{S}^{\pm}$ for which $x^r_j = 1$, the update scheduler adds $S_j.\mathsf{flowadd}(R)$ to U_{r+1} for each $R \in \mathcal{R}^{\mathsf{new}} \setminus \mathcal{R}^{\mathsf{old}}$ for which $R.\mathsf{switch} = S_j$, and $S.\mathsf{flowdel}(R)$ to U_{r+1} for each $R \in \mathcal{R}^{\mathsf{old}} \setminus \mathcal{R}^{\mathsf{new}}$ for which $R.\mathsf{switch} = S_j$.
- For $S_j \in \mathbb{S}^{\mathsf{old}} \setminus \mathbb{S}^{\mathsf{new}}$, the scheduler adds $S_j.\mathsf{flowdel}(R)$ to $U_{\ell'+2}$ and for each

$R \in \mathcal{R}^{\mathsf{old}} \setminus \mathcal{R}^{\mathsf{new}}$ for which $R.\mathsf{switch} = S_j$.

After we obtain the update schedule U_1, U_2, ..., U_ℓ ($\ell = \ell' + 2$), it can then be turned over to the algorithm of Sec. 4.2.2 for adaptation as prescribed there.

4.4 Implementation

We implemented our NF migration algorithm (Sec. 4.2) using Open vSwitch [78] and the Ryu controller [87]. Packets' waypoint counters were stored in six bits of the VLAN tag, permitting up to $n_{\mathsf{max}} = 2^6 - 2$ waypoints per flow. PRADS [82] was used to instantiate network functions and was modified to provide APIs for migration. We used and implemented three underlying route-update algorithms, namely SCC, CU [86], and RWC to generate rule-deployment schedules to transition to a new routing policy. We incorporated our state migration algorithm into the rule-update schedule as described in Sec. 4.2 to achieve waypoint correctness. The RWC integer program in Fig. 4.5 was solved using Gurobi [28].

4.4.1 Route-Update Algorithms

SCC We optimized the implementation of our SCC algorithm described in Chapter 3. Our implementation leverages unused header bits, namely six bits of the VLAN tag, to store the timestamp in each packet. (The remaining six bits of the VLAN tag was used to record the packet's waypoint counter, as already mentioned.) We modified Open vSwitch(OVS) to extract these bits from each packet and to set these bits based on the action of the rule to which it is matched. The SCC rule timestamp value for each rule R is embedded into the corresponding field of the rule to avoid changing the OpenFlow protocol used between OVS and the controller. However, this value is not used for rule matching. Rather, it is extracted from the rule upon rule installation and masked during matching. If the timestamp of the rule to which the packet is matched is smaller than the packet timestamp, then this packet is buffered awaiting more up-to-date rules.

Since OVS does not provide packet buffering anymore, we connected each OVS with a local Ryu controller. Instead of buffering packets itself, OVS forwards packets to the local controller. The local controller is in charge of buffering packets and updating rules for this

switch. A globally centralized controller running our algorithm uses a RESTful API to issue rule modification commands to the local controller. Then the local controller updates OVS using OpenFlow and also sends buffered packets back to the switch when appropriate.

CU Like in SCC, our implementation leverages six bits of the VLAN tag to store the CU timestamp in each packet. However, unlike SCC, the value of the rule timestamp is used to match packets; i.e., the value of the rule timestamp must be equal to the value of timestamp carried by the packet in order to match to this packet. Also, since CU does not need to buffer packets, local controllers are not required. A centralized controller running the CU algorithm used OpenFlow to directly issue rule modification commands to OVS.

RWC We used Gurobi to solve the optimization formulated in Fig. 4.5. The centralized controller runs RWC and deploys updates in multiple steps. *RWC* does not use any timestamp to ensure consistency.

4.4.2 NF Migration

The centralized controller runs SCC, CU, or RWC to generate a rule-update schedule and incorporates our state migration rules into the update deployment as described in Sec. 4.2. We used the IP addresses of S_i^{new} and S_i^{old} to create rules to forward flows f_i^{mig}, f_i^{tun}. In our experiments, we used the Passive Real-time Assets Detection Systems (PRADS) to instantiate network functions. PRADS passively listens to network packets and collects information about hosts and services sending packets. We modified PRADS to permit import/export of portions of its state, such as per flow statistics. After receiving the $S_i^{\mathsf{old}}.\mathsf{export}$ command, S_i^{old} exported the relevant PRADS state and crafted packets on f_i^{mig}. Each PRADS instance executed on a host directly connected to S_i^{old} or S_i^{new}. S_i^{old} and S_i^{new} were in charge of forwarding packets to the PRADS instance. To implement encapsulation and decapsulation, we used the IP tunnel command to configure Generic Routing Encapsulation (GRE) tunnel on each host. Moreover, to guarantee that packets carrying NF state are delivered to destination NF instances, a TCP connection was used. S_i^{new} used the local controller to buffer packets until

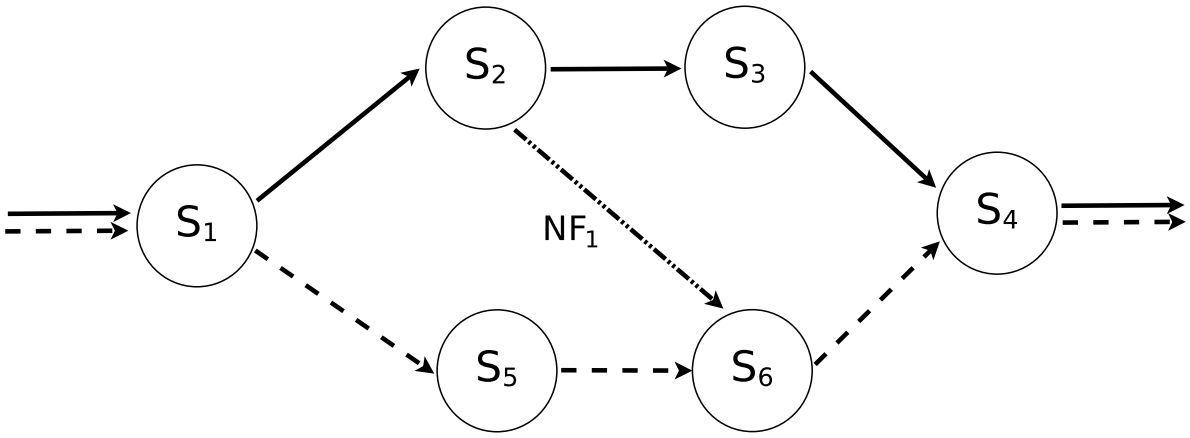


Figure 4.6: Example of SwingState and OpenNF. The solid line is the old path, and the dashed line is the new path.

receiving a $S.\mathsf{release}(i)$ invocation.

For comparison purposes in our empirical evaluations, we also leveraged implementations of SwingState [61] and OpenNF [24]. We use an example in Fig. 4.6 to describe them. Assume a flow f that is processed by NF_1 needs to change its path from $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$ (i.e., the solid line) to $S_1 \rightarrow S_5 \rightarrow S_6 \rightarrow S_4$ (the dashed line). So, NF_1 migrates from S_2 to S_6.

SwingState SwingState migrates an NF over a tunnel between its old and new locations. First, a GRE tunnel is built between S_i^{old} and S_i^{new}. Any route-update algorithm can be leveraged to deploy forwarding rules on switches. In Fig. 4.6, a tunnel $S_2 \rightarrow S_1 \rightarrow S_5 \rightarrow S_6$ is created to connect S_1^{old} and S_1^{new}. Then, S_i^{old} (i.e., S_2) starts to migrate the NF by prepending its state to the clone of incoming packets and forwarding those packets to S_i^{new} (S_6). Meanwhile, S_i^{old} (S_2) still forwards incoming packets to the destination through the old path (the solid line). When S_i^{new} (S_6) receives packets with NF state piggybacked on them, it instantiates the NF before processing these packets. After states are synchronized between S_i^{old} and S_i^{new}, S_i^{old} (S_2) forwards packets normally and meanwhile tunnels a copy of each incoming packet to S_i^{new} (S_6). Finally, the path change is performed. In Fig. 4.6, switches are updated to change the path of f from the solid line to the dashed line. Different from our algorithm, SwingState has to transfer NF states before path changes can begin.

OpenNF OpenNF utilizes the centralized controller as a relay node to transfer NF states and redistribute incoming packets. Specifically, before the Ryu controller issues a command to S_i^{old} (i.e., S_2) to export state for NF_i (NF_1), it deploys a rule for an OpenFlow packet-in event to S_i^{old} to redistribute affected packets to the controller. Then, the Ryu controller transfers the NF states to S_i^{new} (S_6). Next, incoming packets buffered on the controller are delivered to S_i^{new} (S_6) using OpenFlow packet-out messages. Finally, the path change is performed using some route-update algorithm. Similar to SwingState, OpenNF also separates state migration from path change.

4.5 Evaluation

In this section we evaluate our design and demonstrate our algorithm outperforms SwingState and OpenNF.

4.5.1 Setup

Our experiments were conducted on topologies emulated in Mininet [68] on a 32-core, 2.1GHz computer with 256GB of memory. We used a fat-tree topology with $K = 8$ ports per switch, and one ISP topology (Forthnet from Topology Zoo [95]) for these tests. The $K = 8$ fat-tree contained 80 switches, and IP addresses were assigned as prescribed by Al-Fares et al. [2]. The Forthnet topology contained 62 switches and 62 links. To simulate the delay between the controller and switches, we randomly chose the position of the controller and computed the path from the controller to each switch using a Spanning Tree Protocol. The delay for each hop was measured using a simple topology with one OVS switch and two hosts sending ping packets. Specifically, the delay between each switch and the controller was computed as $db + dh \times h$ where db is the control path delay measured by Huang et al. [35] for a setting similar to ours and $dh \times h$ is the delay for one extra hop multiplied by the number of hops. db was sampled from a normal distribution with mean 32ms and standard deviation 5.1ms and dh from normal distribution with mean 3ms and standard deviation 0.3ms.

To create realistic path changes on the fat-tree networks, we replayed a log of route

changes collected from Facebook's network [12]. For the ISP topologies, we used shortest-path routing and induced route changes by breaking links. In each case, NFs were reassigned from the old path to the new path randomly but constrained to appear on the new path in waypoint order.

In our evaluations, we primarily compare our state migration algorithm with OpenNF [24] and SwingState [61] when applied with three route-update algorithms, namely SCC, CU [86] and RWC [59], based on our own implementation of each.

4.5.2 NF Migration and Path Change Time

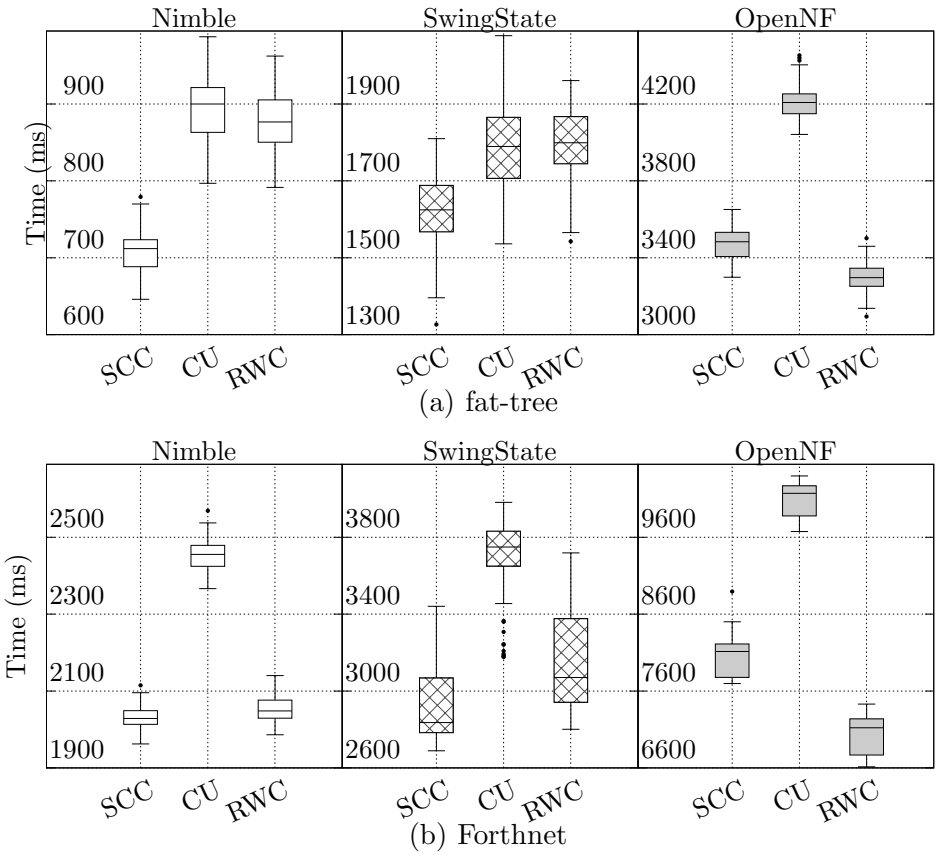


Figure 4.7: NF migration and path change times

We measured the performance of our algorithm for NF migration and path change, in comparison to other algorithms. Each evaluation involved 100 runs, in which hosts sent 100 packets per second for each flow. Fig. 4.7 demonstrates the times required to finish both

NF migration and path changes; Fig. 4.7(a) shows times for 100 path changes with two NF migrations per path change in a fat-tree topology ($K = 8$), and Fig. 4.7(b) shows times for 178 path changes with three NF migrations per path change in the Forthnet topology induced by breaking its "busiest" link carrying the most flows. Each boxplot in these figures represents 100 points, one per run; the box marks the first, second (median), and third quartiles, and whiskers extend to cover points within 1.5× the interquartile range. Outliers are shown as dots.

As can be seen in these figures, Nimble performed much faster than SwingState and OpenNF, since NF migration and path change were executed simultaneously. OpenNF required much more time to perform the updates because it uses a single controller to buffer and redistribute packets. Upon receiving a large number of incoming packets, the controller consumed a lot of resources to process these packets, which significantly slowed down the rule-update process.

4.5.3 Memory Overhead in Switches

To evaluate the number of rules (including rules to build tunnels) imposed on the switches by each algorithm, we examined the per-switch logs of rule installations and deletions over 100 consecutive path changes in the fat-tree topology ($K = 8$). We computed a time series of the total number of rules installed across all switches in the network, including the time cleaning up tunnels used to migrate NFs and to tunnel traffic. This time series for a representative run of each of SCC, CU, and RWC using Nimble, SwingState and OpenNF is shown in Fig. 4.8. We repeated this evaluation on the Forthnet topology, but by breaking the busiest link; see Fig. 4.9. Each curve is marked with the time when NF migration for all flows was done and the time when all path changes were completed. All three algorithms require installing rules on S_i^{old} and S_i^{new} to deal with incoming packets and also to clean up these rules. Different from OpenNF, Nimble and SwingState need to build tunnels to migrate NFs and to tunnel traffic. As these figures show, Nimble requires less time to finish NF migrations and path changes, because it performs both operations simultaneously. OpenNF needs more time to

finish path changes than SwingState since OpenNF buffers and redistributes all incoming packets in the controller, which induces a large delay before path change can be executed.

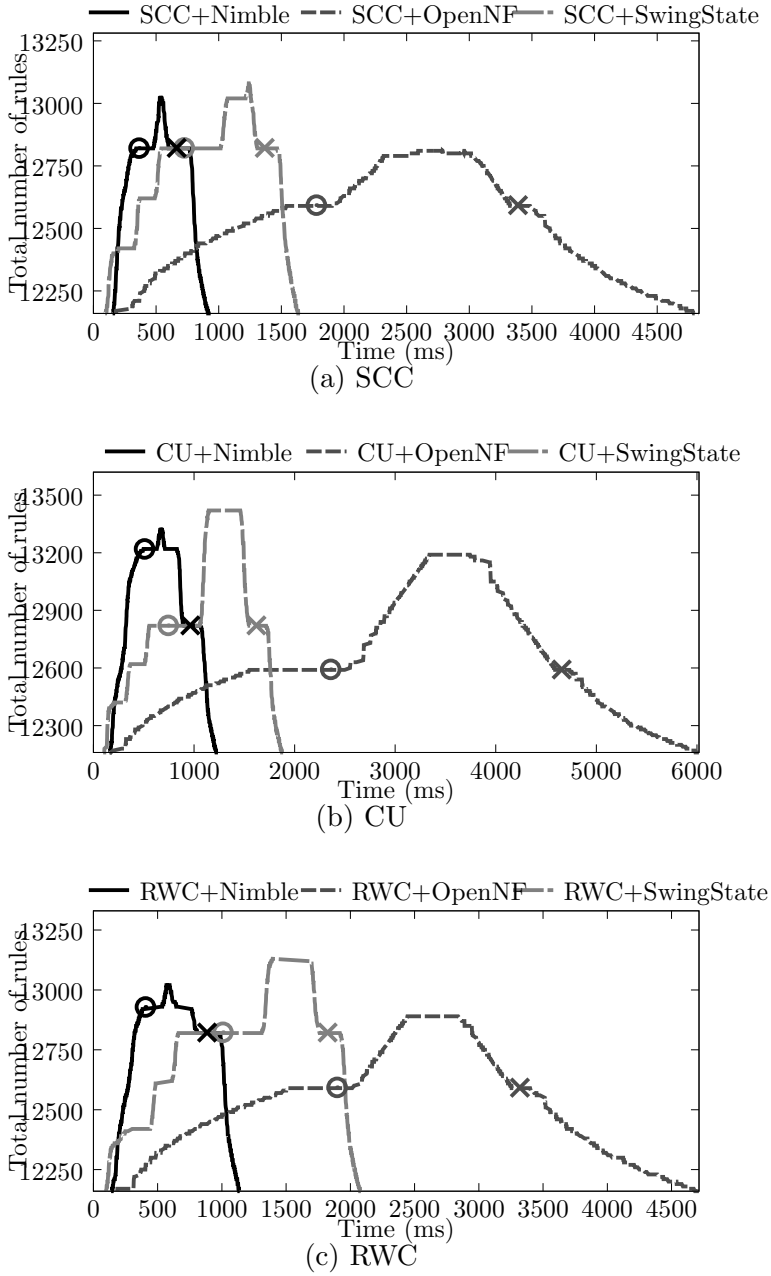


Figure 4.8: Rules in the network during 100 path changes and accompanying NF migrations for fat-tree topology; markers show completion of path changes (×) and NF migrations (○)

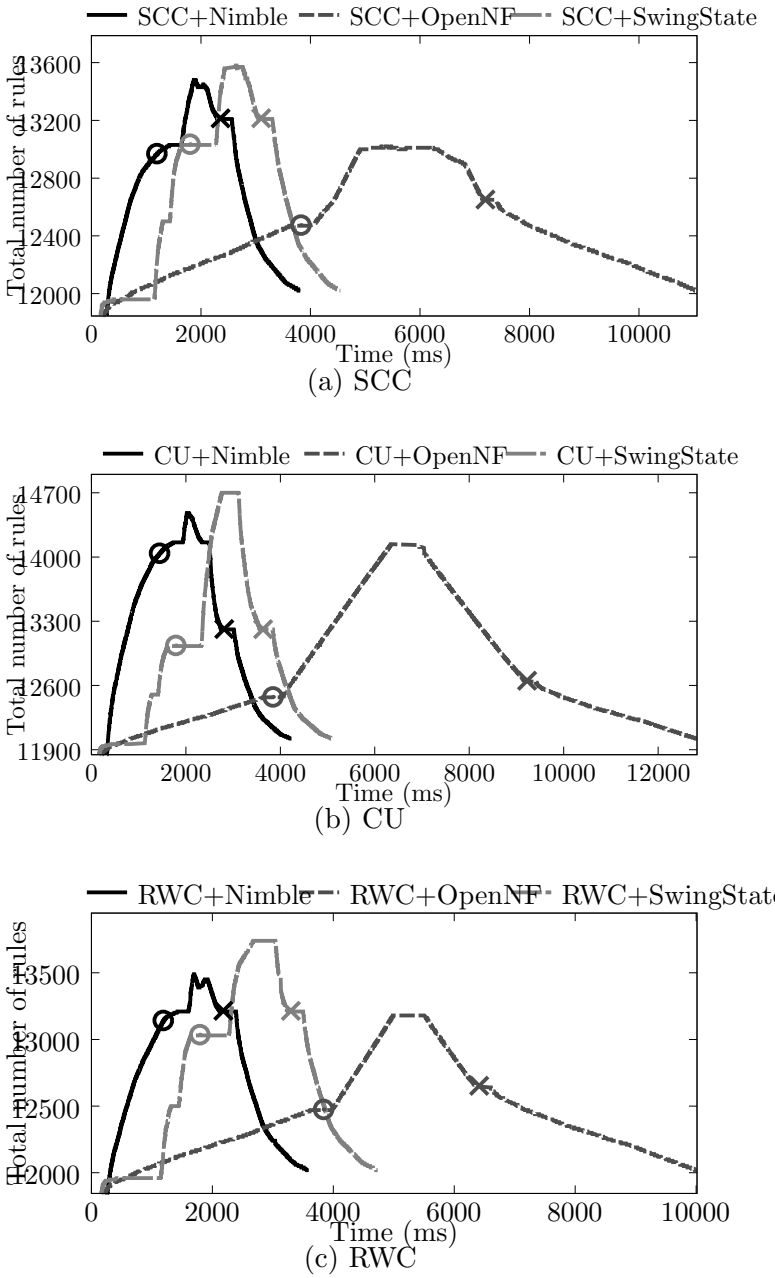


Figure 4.9: Rules in the network during 178 path changes and accompanying NF migrations for Forthnet topology; markers show completion of path changes (×) and NF migrations (○)

4.5.4 Packet Latency During Link Failure

To evaluate the latency imposed on each flow upon link failure by each algorithm, we measured the time required for a destination host to receive 10MB from a source host through a TCP connection. We broke one random link and selected 10 flows to update their routing

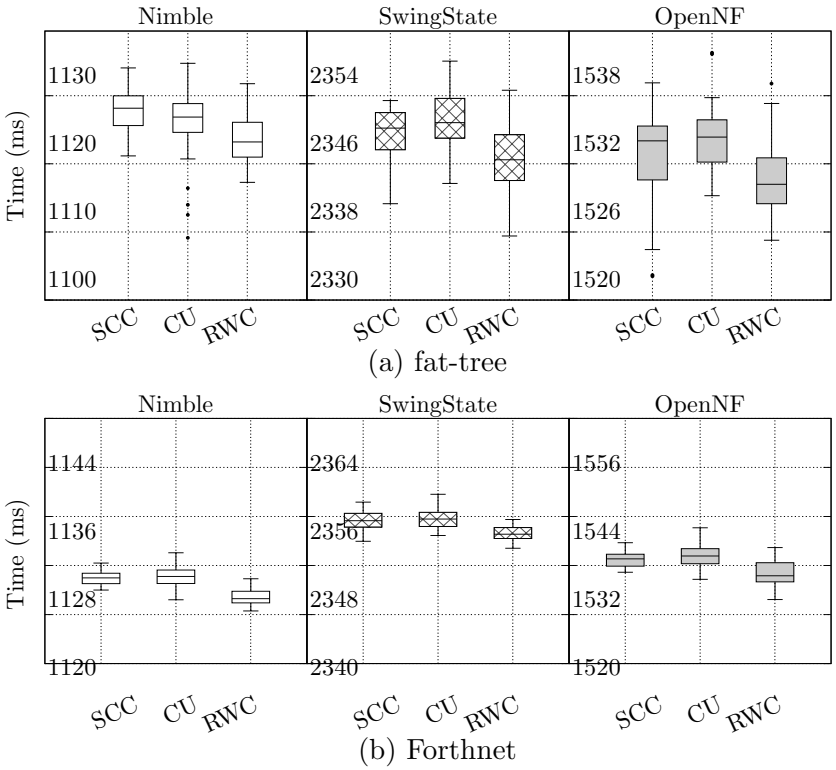


Figure 4.10: Times for receiving 10MB upon link failure

polices. The source host of each flow started to send a stream of packets at speed 100Mb/ sec to the destination host upon link failure. The time shown in Fig. 4.10 shows the times for the destination to receive 10MB using each algorithm. Nimble outperforms SwingState and OpenNF because Nimble recovers the new path more efficiently by performing NF migration and path update simultaneously. Though SwingState also uses tunnels to migrate NFs, it mirrors packets but still delivers packets through the old path during NF migration. Plus, OpenNF uses a single controller to redistribute incoming packets, which significantly slows down the speed.

CHAPTER 5: MODEL CHECKING[1]

It is important to prove the correctness of our algorithms. Specifically, we need to verify the suffix causal consistency property for SCC and the waypoint enforcement property for Nimble. In this chapter, we describe how to use model checking to demonstrate the correctness of SCC and Nimble, respectively. Model-checking tools have long been used to check for violations of network properties and automatically find bugs in network applications. Recent works [92, 3, 63] use model checking to verify network-wide properties for SDN. It is not easy to do so since these methods need to consider the ample space of switch states and the large space of input packets. An SDN switch maintains a flow table that may store a large number of rules and that processes a packet based on the matching rule of the highest priority. Also, the OpenFlow specification allows switches to match packets to rules based on multiple packet header fields, e.g., source IP address and destination IP address. Though prior works try to address these issues by using symbolic execution [49] or simplified switch and packet models, there are additional challenges that prevent existing tools [92, 3, 45, 46, 69, 63, 102, 48] from being directly applied to our algorithms.

First, we need to model unknown delays for each switch update to occur since updates are pushed to switches simultaneously (within a single update step for Nimble). This indicates that each switch can possibly have an old state or a new state, depending on the delay during the update. It is not efficient to naively leverage the model checking tool to explore all possibilities for switch states. So we use the property of our SCC algorithm to reduce the searching state space. Remember our SCC algorithm guarantees that, once a packet is

[1]This chapter is excerpted from previously published work [55].

matched to a forwarding rule in a switch (i.e., reads a switch state), it can be matched in downstream switches only to rules that are equally or more up-to-date (i.e., it cannot read a stale state for downstream switches). Therefore, we explicitly formulate these constraints to model our algorithms.

Second, our SCC algorithm operates differently from the OpenFlow since SCC needs to compare an extra header field $pkt.\mathsf{tstamp}$ of a packet with $R.\mathsf{epochCtr}$ when searching for a matching rule on a switch. This field $pkt.\mathsf{tstamp}$ may be updated by any switch depending on the value of $R.\mathsf{tstamp}$. Also, some rules remain unchanged during the previous configurations and thus $pkt.\mathsf{tstamp}$ can be any value from the range of $[1, \mathsf{epochCtr}]$. To tackle this, our model adds two fields $R.\mathsf{epochCtr}$, $R.\mathsf{tstamp}$ for each rule R and a field $pkt.\mathsf{tstamp}$ for each packet pkt. The model checking tool explores all possibilies for $pkt.\mathsf{tstamp}$ as long as the constraints defined for SCC protocol are satisfied.

Third, Nimble can work with any routing-update algorithm that guarantees relaxed waypoint correctness. The update of switches may be executed in multiple steps and in any order. Therefore, the model checking tool cannot focus on only a specific protocol but needs to model diverse protocol behaviors. Our model should also take into account the behavior of the tunnel protocol since Nimble uses tunnels to redistribute packets from old to new positions of network functions. We formulate the definition of relaxed waypoint correctness, simplify the behavior of tunnels and let the model checking tool explore all possible rule-update schedules. Next, we elaborate on how we formulate our models using Z3Py [100], a Python API for the Z3 solver [72], and how we use the Z3 solver to verify the desired property.

5.1 Model Checking for SCC

We subjected SCC to model checking to verify its enforcement of suffix causal consistency, as well as black-hole freedom and bounded looping. We constructed our model with ten switches in a mesh topology and with three flows. The maximum length of each routing path was six switches. Each switch was allowed ten rules ($|S_j.\mathsf{ruleSet}| \leq 10$), which was

adequate to accommodate the rules deployed by our algorithm for any well-formed routing policy for a system of this size. Deployed rules satisfied the constraints of Sec. 3.1.1; e.g., if $R_j, R_{j'} \in S_j.\mathsf{ruleSet}$ then $R_j.\mathsf{priority} \neq R_{j'}.\mathsf{priority}$ or $R_j.\mathsf{cover} \cap R_{j'}.\mathsf{cover} = \emptyset$. The fields of each rule were unspecified and so explored by the model checker; in particular, each rule could cover any number of flows. Each flow was routed from its ingress to its egress using normal switch behavior (e.g., a switch matches a packet to the highest priority rule that covers it). For each initial rule R, $R.\mathsf{epochCtr}$ and $R.\mathsf{tstamp}$ were allowed to range over $\{1, 2, 3\}$ (explored by the model checker). We modeled the effects of one new epoch[2] ($epochCtr = 4$) that implemented some different routing policy (i.e., at least one flow traveled a different path to its egress) using rules for which $R.\mathsf{epochCtr}$ and $R.\mathsf{tstamp}$ were set according to our algorithm.

Z3 explored all possibilities for each new rule and, so, for the new path traversed by each flow, constrained only so that each flow's ingress was unchanged. To model unknown delays for switch updates to occur, each switch that had not yet applied a new rule to a packet could apply either an old rule R_1 or new rule R_2 to match the current packet pkt, according to $pkt.\mathsf{tstamp}$. Specifically, if $pkt.\mathsf{tstamp} \leq R_1.\mathsf{epochCtr}$ and $pkt.\mathsf{tstamp} \leq R_2.\mathsf{epochCtr}$, either rule could be applied to the packet, creating two branches. If $pkt.\mathsf{tstamp} > R_1.\mathsf{epochCtr}$ and $pkt.\mathsf{tstamp} \leq R_2.\mathsf{epochCtr}$, then only the new rule R_2 could be used to match the packet.

To test black-hole freedom, we set a condition that the trace of each packet should end with the egress node for the packet. The bounded looping property was defined to require that any unordered pair of switches cannot occur in the trace more than twice. The suffix causal consistency property was modeled to require that, once a packet arrives at a switch belonging to its new path but not its old path, it stays on the new path. We let the Z3 solver explore all possible switch configurations to check for violations of these properties.

[2]This is reasonable since we require that one epoch's rule changes are deployed to the network prior to starting the next.

In previous, incomplete versions of our algorithm, this model checking revealed corner cases that we had failed to consider and that resulted in property violations; several of these corner cases were used in the examples given in Sec. 3.3.1 to motivate the algorithm stages. For the algorithm presented in Sec. 3.3.1, however, after running about 6 days, the model checker successfully terminated and found no violations.

5.2 Model Checking for Nimble

We subjected our algorithm of Sec. 4.2 to model checking to verify its enforcement of waypoint correctness described in Sec. 4.1.2. We constructed our model with fifteen switches in a mesh topology and with three flows. Each routing path was eight switches and each path contained three NFs. We modeled the effects of one new epoch that implemented a routing policy with NF migration (i.e., each network function for each flow was moved to a different position) using rules in the old and new configuration.

The underlying route-update algorithm deployed rule updates on switches (i.e., deleting unused rules and installing new rules) in at most $\ell = 15$ steps, with each switch being updated at most once. Each switch utilized either old or new rules to process packets based on the step during which it received those packets. For example, if S_1 was scheduled to be updated in step U_3, S_1 used an old rule R_1 to match f_1 before U_3 began and a new rule R_2 after U_3 completed. During U_3, to model unknown delays for switch updates to occur, either R_1 or R_2 was used to match f_1 nondeterministically. The step at which packets were received by each switch through the network was non-decreasing. Moreover, the update schedule generated by the underlying route-update algorithm ensured relaxed waypoint correctness. Z3 explored all possible rule-update schedules constrained by the above conditions to enforce new routing policy of each flow. As such, the model checked was not dependent on any specific route-update protocol, but rather permitted any route-update strategy as long as it satisfied these properties.

Our algorithm incorporated NF migration into the rule-update schedule generated by the underlying route-update algorithm and used tunnels to redistribute packets from old NF

locations to new ones. For simplicity, the correctness of tunnels was assumed, and tunnels were not modeled explicitly. Specifically, when a packet on flow f arrived at S_i^{old} and was matched to rule R_i^{enc}, the packet was delivered to S_i^{new}, as if through the tunnel.

To model the delay caused by buffering packets at new NF locations, the specific step at which S_i^{new} forwarded packets using its new rule should not be earlier than the step at which $S_i^{\mathsf{new}}.\mathsf{release}(i)$ is invoked. For example, if a packet arrived at S_i^{old} for NF_i at step U_3 and then was forwarded through the tunnel to S_i^{new}, S_i^{new} could not release packets until $S_i^{\mathsf{new}}.\mathsf{release}(i)$ was deployed at step U_5. We let the Z3 solver explore all possible delays before the S_i^{new} released packets and check for violations of waypoint correctness as defined in Sec. 4.1.2. In previous, incorrect versions of our algorithm, this model checking revealed corner cases that we had failed to consider and that resulted in property violations. For the algorithm presented in the previous sections, however, after running about one day on a 32-core, 2.1GHz computer with 256GB of memory, the model checker successfully terminated and found no violations.

CHAPTER 6: CONCLUSION

Rapid NF migration and accompanying path changes can be critical for alleviating problems in a network, and doing so in a way that ensures that all traffic is processed by its required waypoints is important to avoid violations of network policy. Realizing this requires both efficient network forwarding-state update and safe NF migration. In this dissertation, we have proposed suffix causal consistency (SCC) as an interpretation of causal consistency for network forwarding-state updates in an SDN network. SCC ensures that a packet will be matched only to rules at least as recent as those to which it has been matched previously, thus ensuring that a packet will exit the network on a suffix of the most recent path's rules to which it was matched. Our algorithm implements this property without updating switches unnecessarily. We showed that SCC implements bounded looping and black-hole freedom during updates and formally verified that our algorithm achieves SCC as well as these additional properties. Through empirical tests with implementations in P4 and Open vSwitch, and using real traffic traces from Facebook, we showed that our algorithm supports faster rule deployment than CU, TSU and COCONUT, leading to fewer dropped packets during updates. SCC also requires the retention of fewer additional rules during the update, and its rule generation scales across a wide range of topologies.

To coordinate NF migration with the routing policy update, we have presented an algorithm that accelerates the deployment of these changes in SDN networks over current best solutions. Our design accomplishes this through a careful interleaving of NF migrations with path changes, and ensures the correctness of traffic processing provided that the route-update protocol on which we build ensures a property that we call *relaxed waypoint correctness*. We provided a route-update protocol designed to achieve this property, without enforcing other properties typically associated with consistent-update protocols. We showed the sufficiency

of this property through model checking, and then demonstrated the performance improvements achieved by our algorithm in empirical comparisons to state-of-the-art.

We believe our work paves the way for future research in SDN and network function migration. For example, is it feasible to balance the workload of each network function by choosing where NFs should be migrated to and which path traffic should be rerouted through? Is it possible to design a one big switch framework that can automatically and consistently manage state for programmable switches? We leave these open issues for future work.

REFERENCES

[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 1995.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, pages 63–74, Aug. 2008.

[3] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '10, pages 37–44, 2010.

[4] Amazon aws network function virtualization. `https://docs.aws.amazon.com/whitepapers/latest/ec2-networking-for-telecom/mapping-aws-services-to-the-nfv-framework.html`.

[5] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming slick network functions. In *1st ACM Symposium on Software Defined Networking Research*, pages 1–13, 2015.

[6] From next-gen to now: Sdn, white box and open source go mainstream. `https://about.att.com/innovationblog/2019/09/sdn_white_box_and_open_source_go_mainstream.html`.

[7] P. Bailis, A. Ghodsi, J. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *ACM SIGMOD International Conference on Management of Data*, pages 761–772, 2013.

[8] Behavioral model. `https://github.com/p4lang/behavioral-model/`.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, pages 87–95, July 2014.

[10] Switch buffer size. `https://people.ucsc.edu/~warner/buffer.html`.

[11] D. J. Chaboya, R. A. Raines, R. O. Baldwin, and B. E. Mullins. Network intrusion detection: Automated and manual methods prone to attack and evasion. *IEEE Security & Privacy*, 4(6), 2006.

[12] H. Chen and T. Benson. Hermes: Providing tight control over high-performance SDN switches. In *13th International Conference on Emerging Networking Experiments and Technologies*, pages 283–295, 2017.

[13] T.-H. Cheng, Y.-D. Lin, Y.-C. Lai, and P.-C. Lin. Evasion techniques: Sneaking through your intrusion detection/prevention systems. *IEEE Communications Surveys & Tutorials*, 14(4), 2012.

[14] Cisco network function virtualization infrastructure. https://www.cisco.com/c/en/us/solutions/service-provider/network-functions-virtualization-nfv-infrastructure/index.html.

[15] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation*, NSDI'05, pages 273–286, 2005.

[16] E. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. Model checking. 2018.

[17] I. Corona, G. Giacinto, and F. Roli. Adversarial attacks against intrusion detection systems: Taxonomy, solutions, and open issues. *Information Sciences*, 239, Aug. 2013.

[18] Ibm cplex optimizer. https://www.ibm.com/analytics/cplex-optimizer/.

[19] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, 2008.

[20] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 56–66, 1988.

[21] K. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid. Loop-free route updates for software-defined networks. *IEEE/ACM Transactions on Networking*, 26(1):328–341, Feb. 2018.

[22] K. Foerster, R. Mahajan, and R. Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *IFIP Networking Conference and Workshops*, 2016.

[23] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *ACM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 43–48, 2015.

[24] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM*, pages 163–174, 2014.

[25] S. Ghorbani and P. Godfrey. COCONUT: Seamless scale-out of network elements. In *12th ACM European Conference on Computer Systems*, Apr. 2017.

[26] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker. Transparent, live migration of a Software-defined network. In *ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[27] gRPC. https://grpc.io/.

[28] Gurobi solver. http://www.gurobi.com/.

[29] J. Han, H. J. Hun, P. Mundkur, R. Prashanth, C. Rotsos, G. Antichi, N. Dave, A. Moore, and P. Neumann. Blueswitch: Enabling provably consistent configuration of network switches. In *11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, May 2015.

[30] The reliable, high performance tcp/http load balancer. `http://www.haproxy.org/`.

[31] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[32] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, pages 15–26, Aug. 2013.

[33] hping. `http://www.hping.org/`.

[34] K. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Feb. 2020.

[35] D. Huang, K. Yocum, and A. Snoeren. High-Fidelity switch models for Software-defined network emulation. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, Aug. 2013.

[36] Microsoft azure virtual machines. `https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-hpc`.

[37] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, 2013.

[38] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[39] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, pages 539–550, Aug. 2014.

[40] K. Junaid, G. Aaron, M. Roney, A. Anubhavnidhi, and A. Aditya. Paving the way for NFV: Simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 239–253, Mar. 2016.

[41] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Conference on Networked Systems Design and Implementation*, Apr. 2017.

[42] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *9th ACM Conference on Emerging Networking Experiments and Technologies*, Dec. 2013.

[43] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, 2016.

[44] N. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *2nd ACM Workshop on Hot Topics in Software Defined Networking*, 2013.

[45] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 99–112. USENIX Association, 2013.

[46] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 9, 2012.

[47] J. Khalid and A. Akella. StreamNF: Performance and correctness for stateful chained NFs. In *16th USENIX Conference on Networked Systems Design and Implementation*, 2019.

[48] A. Khurshid, W. Zhou, M. Caesar, and B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. HotSDN'12, pages 49–54, 2012.

[49] J. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[50] D. Kreutz, F. Ramos, P. Verissimo, C. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. 103(1):14–76, 2014.

[51] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commununications of the ACM*, 21(7):558–565, July 1978.

[52] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization. In *10th ACM Conference on Emerging Networking Experiments and Technologies*, 2014.

[53] H. Li, H. Hu, G. Gu, G.-J. Ahn, and F. Zhang. vNIDS: Towards elastic security with safe and efficient virtualization of network intrusion detection systems. In *25th ACM Conference on Computer and Communications Security*, pages 17–34, 2018.

[54] H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. In *ACM SIGCOMM*, pages 411–422, Aug. 2013.

[55] S. Liu, T. A. Benson, and M. K. Reiter. Efficient and safe network updates with suffix causal consistency. In *14th ACM European Conference on Computer Systems*, Mar. 2019.

[56] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, 2016.

[57] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[58] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Conference on Networked Systems Design and Implementation*, pages 313–328, 2013.

[59] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. Transiently secure network updates. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 273–284, 2016.

[60] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *13th ACM Workshop on Hot Topics in Networks*, 2014.

[61] S. Luo, H. Yu, and L. Vanbever. Swing State: Consistent updates for stateful and programmable data planes. In *3rd Symposium on SDN Research*, pages 115–121, 2017.

[62] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. In *12th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[63] C. Marco, V. Daniele, P. Peter, K. Dejan, and R. Jennifer. A NICE way to test openflow applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, Apr. 2012.

[64] F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithsm*, pages 215–226, 1988.

[65] D. M. F. Mattos and O. C. M. B. Duarte. Xenflow: Seamless migration primitive and quality of service for virtual networks. In *2014 IEEE Global Communications Conference*, pages 2326–2331, 2014.

[66] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient synthesis of network updates. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, 2015.

[67] Microsoft azure network function virtualization. `https://docs.microsoft.com/en-us/windows-server/networking/sdn/technologies/network-function-virtualization/network-function-virtualization`.

[68] Mininet. http://mininet.org/.

[69] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev. SDNRacer: Detecting concurrency violations in software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR'15, pages 1–7, 2015.

[70] T. Mizrahi, O. Rottenstreich, and Y. Moses. TimeFlip: Using timestamp-based TCAM ranges to accurately schedule network updates. *IEEE/ACM Transactions on Networking*, 25(2):849–863, Apr. 2017.

[71] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates. In *1st ACM Symposium on Software Defined Networking Research*, 2015.

[72] L. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*, pages 337–340, Mar. 2008.

[73] Etsi network function virtualization. https://www.etsi.org/technologies/nfv/.

[74] Network function virtualization (nfv) market global forcast to 2024. https://www.marketsandmarkets.com/Market-Reports/network-function-virtualization-market-93929190.html.

[75] T. Nguyen, M. Chiesa, and M. Canini. Decentralized consistent updates in SDN. In *Symposium on SDN Research (SOSR)*, Nov. 2017.

[76] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, page 2941, 2011.

[77] Open Networking Foundation. Openflow switch specification, version 1.5.1. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, 26 March 2015.

[78] Open vswitch. http://openvswitch.org/.

[79] P4 runtime. https://github.com/p4lang/PI/.

[80] P. Pereíni, M. Kuzniar, M. Canini, and D. Kostić. Espres: Transparent SDN update scheduling. In *ACM HotSDN*, 2014.

[81] Pica8 switch. https://www.pica8.com/product.

[82] Passive real-time asset detection system. https://github.com/gamelinux/prads/.

[83] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[84] Z. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM SIGCOMM*, pages 27–38, 2013.

[85] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *10th USENIX Conference on Networked Systems Design and Implementation*, Apr. 2013.

[86] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, Aug. 2012.

[87] Ryu controller. `https://osrg.github.io/ryu/`.

[88] Scapy. `http://www.secdev.org/projects/scapy/`.

[89] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. PISCES: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 525–538, 2016.

[90] W. Shinae, S. Justine, H. Sangjin, M. Sue, R. Sylvia, and S. Scott. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Apr. 2018.

[91] Snort. `https://www.snort.org/`.

[92] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Model checking invariant security properties in openflow. In *2013 IEEE International Conference on Communications (ICC)*, pages 1974–1979, 2013.

[93] V. Stefano and C. Luca. Flip the (flow) table: Fast lightweight policy-preserving sdn updates. In *35th IEEE International Conference on Computer Communications*, Apr. 2016.

[94] Barefoot networks' tofino. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series/tofino-2.html`.

[95] Topology zoo. `http://www.topology-zoo.org/`.

[96] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central control over distributed routing. In *ACM SIGCOMM*, Aug. 2015.

[97] Vmware vcloud network function virtualization. `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/solutions/vmware-vcloud-nfv-datasheet.pdf`.

[98] K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with Espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Aug. 2017.

[99] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *10th USENIX Conference on Networked Systems Design and Implementation*, pages 29–42, 2013.

[100] Z3py. `https://github.com/ericpony/z3py-tutorial/`.

[101] K. Zarifis, R. Miao, M. Calder, E. Katz-Bassett, M. Yu, and J. Padhye. DIBS: Just-in-time congestion mitigation for data centers. In *9th ACM European Conference on Computer Systems*, 2014.

[102] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. *IEEE/ACM Trans. Netw.*, 22(2):554–566, Apr. 2014.

[103] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *12th USENIX Conference on Networked Systems Design and Implementation*, pages 73–85, 2015.