Efficient and Safe Network Updates with Suffix Causal Consistency

Sheng Liu
University of North Carolina
Chapel Hill, NC
shengliu@cs.unc.edu

Theophilus A. Benson
Brown University
Providence, RI
tab@cs.brown.edu

Michael K. Reiter
University of North Carolina
Chapel Hill, NC
reiter@cs.unc.edu

Abstract

Though centrally managed by a controller, a software-defined network (SDN) can still encounter routing inconsistencies among its switches due to the non-atomic updates to their forwarding tables. In this paper, we propose a new method to rectify these inconsistencies that is inspired by *causal consistency*, a consistency model for shared-memory systems. 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 but functionally equivalent version of this property called *suffix causal consistency* (SCC) and evaluate an implementation of SCC in Open vSwitch and P4 switches, in conjunction with the Ryu and P4Runtime controllers. Our 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 (black-hole freedom, bounded looping, etc.).

CCS Concepts • **Networks → Network protocol design**; *Control path algorithms*; *Network manageability*.

Keywords software-defined networking, consistent update, causal consistency, model checking

ACM Reference Format:
Sheng Liu, Theophilus A. Benson, and Michael K. Reiter. 2019. Efficient and Safe Network Updates with Suffix Causal Consistency. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303965

1 Introduction

Networks require fast and efficient consistent update for a variety of reasons, ranging from reacting to simple failures to enforcing complex security policies. In many of these situations, a slow or inconsistent network update will result in packet loss or, worse, violations of network policies.

The traditional approach to update consistency [35], on which most other update mechanisms [12, 14, 16, 20, 29, 33] 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 [16, 31, 36] or congestion [12, 20]; others focus on finding better update orderings [8, 14, 17, 19, 23, 24, 33]. 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.

In this paper, we investigate an alternative consistent update abstraction in which packets are allowed to traverse a combination of both paths, thus relaxing the consistency model and speeding up update times. Our approach 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 [18]; 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 not change). We thus propose a novel method of managing these timestamps to limit the number

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

EuroSys '19, March 25–28, 2019, Dresden, Germany
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6281-8/19/03.
https://doi.org/10.1145/3302424.3303965

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 [5] that unlock flexible functionality, support for causal consistency poses several challenges. Specifically, supporting causal consistency requires switches to detour packets and temporarily buffer packets. Efficiently supporting these features requires exploiting language constructs in non-intuitive ways.

To demonstrate the effectiveness and efficiency of suffix causal consistency, we developed prototypes for both Open vSwitch (OVS) and P4 [5]. We evaluate our prototypes against realistic workloads and topologies. Our analyses show that SCC deploys updates faster than state-of-the-art alternatives (COCONUT [10], TSU [23] and CU [35]) 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.

The rest of this paper is structured as follows. We discuss related work in Sec. 2, and detail our network model and goals in Sec. 3. The components of our framework are outlined in Sec. 4, and Sec. 5 presents our rule generation and deployment algorithm in detail. We describe our implementation in Sec. 6, evaluate our system in Sec. 7, and conclude in Sec. 8.

2 Related Work

Consistent network update in SDN networks has received considerable attention (e.g., [14, 16, 23, 25, 27–29, 31, 35, 38]). Most approaches provide either strong consistency in the sense that packets traverse either the old path or the new path (but not a mix) [16, 29, 35] or ensure specific properties (e.g., loop freedom, congestion freedom) via weaker, transient consistency [8, 12, 14, 17, 20, 23, 24, 36]. An example of the former class is Consistent Update (CU) [35], which uses 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. An example of the latter class of solutions is Transiently Secure Network Updates (TSU) [23], which provides specific properties (e.g., loop freedom, waypoint enforcement) via weaker consistency, though still 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.

We compare empirically to CU and TSU in Sec. 7, 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 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.

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 [18]). Causal consistency is widely used to ensure consistency and high availability of data objects with minimal delay across a wide-area network [4, 21, 22]. Our work adapts this principle to network routing specifically, introducing improvements to reduce the extent and hence delays associated with network updates.

Due to its shared basis in causal consistency, in Sec. 7 we will also empirically compare to COCONUT [10], which seeks to enable seamless scaling of logical network elements to multiple physical replicas. The core technical problem that COCONUT tackles is 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 [7, 26] in packets, with one component per logical rule undergoing 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 work does not focus on causal relationships between flows, but instead ensures that each flow "reads" (is matched to) rules in causal order across 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. 7 and compare to it empirically, as another point in the design space.

3 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). We then motivate and define our main goal in

this paper, a property that we call *suffix causal consistency* (Sec. 3.2).

3.1 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., [35]).

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 *sid* 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 *sid* as *sid*.ruleSet; e.g., $sid.\text{ruleSet} = \{R_1, R_6, R_{10}\}$ means that switch *sid* 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:

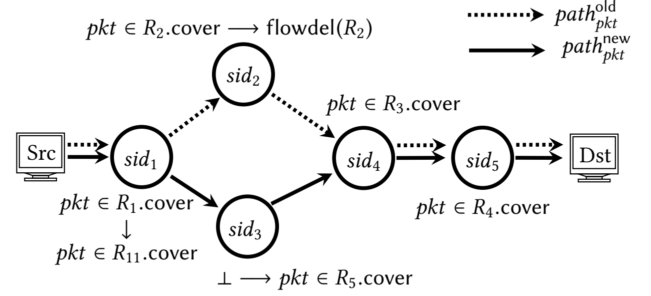


Figure 1. Example of route change.

- sid.flowadd(R_j) inserts rule R_j into *sid*.ruleSet. This command fails with no effect if $R_j.\text{switch} \neq sid$ or if *sid*.ruleSet already contains a rule $R_{j'}$ such that $R_{j'}.\text{priority} = R_j.\text{priority}$ and $R_j.\text{cover} \cap R_{j'}.\text{cover} \neq \emptyset$.
- sid.flowdel(R_j) removes rule R_j from *sid*.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 $sid_i \rightarrow \ldots \rightarrow sid_{i'}$, as directed by the rules on these switches, written $R_j \rightarrow \ldots \rightarrow R_{j'}$. For example, in Fig. 1, the rules applied to packet *pkt* on path $sid_1 \rightarrow sid_2 \rightarrow sid_4 \rightarrow sid_5$ are $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4$, where rule R_1 directs switch sid_1 to send the packet *pkt* to switch sid_2, and so on. If the application wants to change the path of packet *pkt* from $sid_1 \rightarrow sid_2 \rightarrow sid_4 \rightarrow sid_5$ (the dashed line in Fig. 1, denoted $path^{old}_{pkt}$) to $sid_1 \rightarrow sid_3 \rightarrow sid_4 \rightarrow sid_5$ (the solid line, denoted $path^{new}_{pkt}$), 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:

- sid_1.flowdel(R_1) and sid_1.flowadd(R_{11}) where $R_{11}.\text{sendTo} = sid_3$ and $pkt \in R_{11}$.cover;
- sid_2.flowdel(R_2) to delete rule R_2 from sid_2; and
- sid_3.flowadd(R_5) to instruct sid_3 to send *pkt* to sid_4 (i.e., $R_5.\text{sendTo} = sid_4$ and $pkt \in R_5$.cover).

Rule R_3 instructing sid_4 to send *pkt* to sid_5 need not be changed (assuming R_3.cover does not specify the inbound port on sid_4). Nor does rule R_4 instructing sid_5 to send *pkt* to the destination node.

3.2 Goals

In the example of Fig. 1, if sid_1 is updated prior to the addition of R_5 to sid_3, then *pkt* might be directed to sid_3 before sid_3 has a rule to handle it. Similarly, if switch sid_1 is not yet

updated but switch sid_2 has already deleted the rule R_2, then switch sid_2 would not have a rule to match *pkt* upon its arrival. Our central goal in this paper is to develop a rule-update framework that avoids such inconsistencies.

More specifically, the property we seek to implement in this paper, 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*.epochCtr denote the largest value of *R*.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*.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. 1, discussed above. In the first source of inconsistency considered there, sid_1 is updated with rule R_{11} before R_5 is added to sid_3, causing *pkt* to be directed to sid_3 before sid_3 has a rule to handle it. Since *pkt*.epochCtr = R_{11}.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 $path_{pkt}^{\text{new}}$. To do so, our framework ensures that sid_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 sid_2 already deleted R_2 but sid_1, having not yet been updated, still forwards *pkt* to sid_2. Since R_2 is gone from sid_2, there is no hope of forwarding *pkt* further along the old path, $path_{pkt}^{\text{old}}$. 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. 1), with R_6.epochCtr reflecting the existence of a new path $path_{pkt}^{\text{new}}$ 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 sid_2 alongside the deletion of R_2, expressly for the purpose of forwarding *pkt* back toward the switch at which $path_{pkt}^{\text{new}}$ departed from $path_{pkt}^{\text{old}}$ (which is sid_1 in this example). The *pkt* will then pick up the new path at that departure point, traveling the suffix of $path_{pkt}^{\text{new}}$ beginning there. R_6 will need to remain in sid_2 only temporarily.

As we will discuss in Sec. 5.2, 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.

4 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*.epochCtr $\geq$ *pkt*.tstamp, then the switch tags the packet with the rule's *tagging timestamp* *R*.tstamp (i.e., *pkt*.tstamp $\leftarrow$ *R*.tstamp; see Sec. 5) and forwards *pkt* to *R*.sendTo. Otherwise, the packet is buffered by the switch and until its highest-priority rule *R* covering the packet satisfies *R*.epochCtr $\geq$ *pkt*.tstamp.

The initial insertion and the final removal of the packet timestamp *pkt*.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*.tstamp for the rule *R* to which it matches the packet. For example, when packet pkt_1 first arrives at switch sid_1 in Fig. 1, the switch tags the packet according to the rule R_1 to which it is matched (i.e., pkt_1.tstamp $\leftarrow$ R_1.tstamp). Then switch sid_5 can remove the tag from the packets by setting the corresponding header field to a default value (e.g., pkt_1.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., *sid*.flowadd and *sid*.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.

5 Algorithm Description

In this section we provide an algorithm for preventing inconsistencies during path updates such as those described in Sec. 3, and specifically to implement SCC. In our framework, we add to each rule a *tagging timestamp* *R*.tstamp (an

integer) with which a switch tags packets matched to that rule before forwarding them. Each packet thus includes a new field *pkt*.tstamp to hold this timestamp. This timestamp plays a role similar to a Lamport timestamp [18], 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.tstamp that, when carried forward by the packet (in *pkt*.tstamp), will not induce downstream switches to await a new rule from the controller. In the remainder of this section we detail this algorithm.

5.1 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. [15]). This algorithm outputs a rule set $\mathcal{R}^{\text{new}}$, and let $\mathcal{R}^{\text{old}}$ denote the rules already deployed to the network.

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

Then, the controller computes $\mathcal{R}^{\text{add}}$ and $\mathcal{R}^{\text{del}}$ 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 $\mathcal{R}^{\text{add}}$, $\mathcal{R}^{\text{del}}$, and $\mathcal{R}^{\text{keep}}$ to initial values. The "backward closure" step updates $\mathcal{R}^{\text{add}}$ to include rules from $\mathcal{R}^{\text{new}}$ that precede (on routing paths) those already in $\mathcal{R}^{\text{add}}$ in certain circumstances, thereby propagating the installation of new rules "backward" along routing paths. The "forward closure" step then updates $\mathcal{R}^{\text{add}}$ to include rules from $\mathcal{R}^{\text{new}}$ that follow (on routing paths) those already in $\mathcal{R}^{\text{add}}$ in other circumstances, thus propagating the installation of new rules "forward" along routing paths. The "set tagging timestamps" step sets the R.tstamp field of rules $R \in \mathcal{R}^{\text{add}}$ that have not been set in the preceding steps. Finally, the "send-back rules" step adds new rules to $\mathcal{R}^{\text{add}}$ 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 $\mathcal{R}^{\text{keep}}$, $\mathcal{R}^{\text{add}}$, and $\mathcal{R}^{\text{del}}$ that it will then update throughout the remainder of the algorithm. By the end of the algorithm, $\mathcal{R}^{\text{add}}$ will contain those rules that the controller will install via *sid*.flowadd invocations, and $\mathcal{R}^{\text{del}}$ rules will contain those rules that controller will remove via *sid*.flowdel invocations. (As we will discuss below, some of the added rules will also be deleted afterwards.) The rules in $\mathcal{R}^{\text{keep}}$ at the end of the algorithm will be those that the controller leaves in place.

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

Taking the backward closure of $\mathcal{R}^{\text{add}}$ The next stage of the algorithm is motivated by situations like that shown in Fig. 2, where $path^{\text{old}}_{pkt} = (sid_1 \rightarrow sid_2 \rightarrow sid_5)$ is the path taken by *pkt* under the previous routing policy and the new path $path^{\text{new}}_{pkt} = (sid_1 \rightarrow sid_3 \rightarrow sid_4 \rightarrow sid_5)$ is the path it will take under the new routing policy. At this stage of the algorithm, $R_1 \in \mathcal{R}^{\text{keep}}$ and $R_2 \in \mathcal{R}^{\text{add}}$. If we keep R_1 unchanged and if the packet *pkt* arrives at sid_3 on the new path, it will be tagged using the old timestamp (e.g., pkt.tstamp $\leftarrow R_1$.tstamp = 8). Therefore, upon the arrival of the packet at sid_4, if sid_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., sid_6) or to a switch (e.g., sid_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 $\mathcal{R}^{\text{add}}$, so that upon passing sid_3, the packet will carry a new timestamp pkt.tstamp $\leftarrow R_3$.tstamp = 10, ensuring that new rule R_2 will be applied to it at switch sid_4.

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

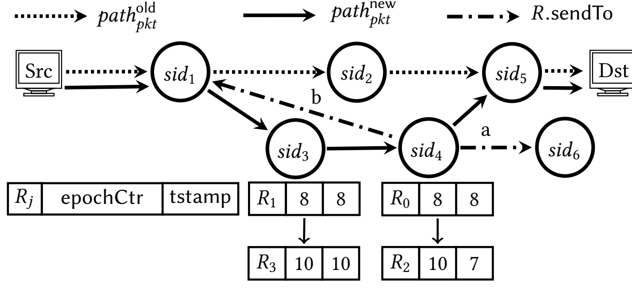


Figure 2. Example motivating backward closure.

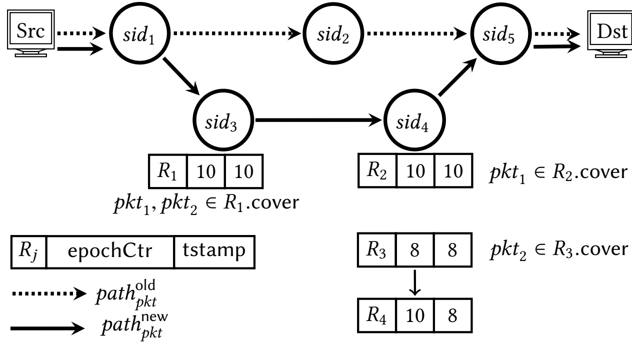


Figure 3. Example motivating forward closure.

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.

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

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

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

Taking the forward closure of $\mathcal{R}^{\text{add}}$ To understand the need for the next stage of our algorithm, consider Fig. 3, where there is a rule $R_1 \in \mathcal{R}^{\text{add}}$ at sid_3 matching two packets pkt_1 and pkt_2, another rule $R_2 \in \mathcal{R}^{\text{add}}$ at sid_4 matching pkt_1, and another rule $R_3 \in \mathcal{R}^{\text{keep}}$ at sid_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 sid_4.

So, the next step of the algorithm identifies cases in which some packets handled by a rule $R_1 \in \mathcal{R}^{\text{add}}$ will be handled at the downstream switch by another rule $R_2 \in \mathcal{R}^{\text{add}}$, while others handled by R_1 will be handled at the downstream switch by a rule $R_3 \in \mathcal{R}^{\text{keep}}$. 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 $\mathcal{R}^{\text{add}}$. If for any $R_1 \in \mathcal{R}^{\text{add}}$, there are

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

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

Setting tagging timestamps So far, only rules added to $\mathcal{R}^{\text{add}}$ 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 $\mathcal{R}^{\text{add}}$. In brief, the tstamp field for a rule R in $\mathcal{R}^{\text{add}}$ 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. 4. In Fig. 4a, assume we have $R_1 \in \mathcal{R}^{\text{add}}$ at sid_1 matching two packets, pkt_1 and pkt_2; $R_2 \in \mathcal{R}^{\text{keep}}$ at sid_2 matching pkt_1; and $R_3 \in \mathcal{R}^{\text{keep}}$ at sid_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.epochCtr and R_3.epochCtr, respectively. Therefore, we set R_1.tstamp $\leftarrow$ 8.

In Fig. 4b, assume $R_1 \in \mathcal{R}^{\text{add}}$ at sid_1 matches a packet pkt_1 and $R_2 \in \mathcal{R}^{\text{add}}$ at sid_2 matches pkt_1. According to forward closure, any R_3 at sid_2 matching packet pkt_2 which is also matched by R_1 should have R_3.epochCtr = *epochCtr* where *epochCtr* is the latest epoch counter. Therefore, we set the tagging timestamp of R_1.tstamp $\leftarrow$ *epochCtr*, i.e., R_1.tstamp $\leftarrow$ 12.

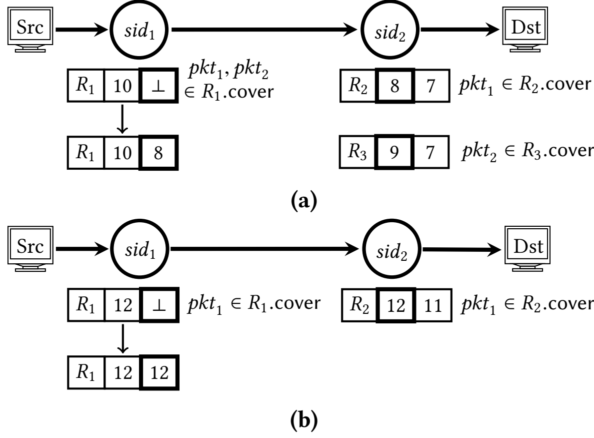


Figure 4. Example of tagging timestamps.

Tagging timestamps: For each $R_1 \in \mathcal{R}^{\text{add}}$ with R_1.tstamp $= \perp$ and each packet pkt such that $match(\mathcal{R}^{\text{add}} \cup \mathcal{R}^{\text{keep}}, R_1, pkt)$, let R_{pkt} be the rule with R_{pkt}.switch $= R_1$.sendTo such that $match(\mathcal{R}^{\text{add}} \cup \mathcal{R}^{\text{keep}}, R_{pkt}, pkt)$. Then, set $R_1.\text{tstamp} \leftarrow \min_{pkt}\{R_{pkt}.\text{epochCtr}\}$, 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 sid_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 sid_1 from which it came. This time, however, the send-back rule R at sid_2 will tag the packet with $pkt.\text{tstamp} = R.\text{tstamp} = epochCtr$, causing the packet to be buffered at sid_1 until a new rule arrives. This rule might, in fact, be another send-back rule.

An example is shown in Fig. 5, where a packet pkt travels along an old path until it reaches switch sid_3, which is not on pkt's new path. Rather than drop the packet, the send-back rule R_3 is added to sid_3 when the old rule R_2 matching pkt in the old configuration is deleted, to direct pkt back to sid_2. Because $R_3.\text{tstamp} = epochCtr = 9$, $pkt.\text{tstamp} = 9$ when it arrives back at sid_2, where it is buffered until the rule R_4 with $R_4.\text{epochCtr} = 9$ is installed. R_4 is also a send-back rule, causing pkt to be forwarded back to sid_1, where it will await the installation of a rule that will forward the packet on its new path.

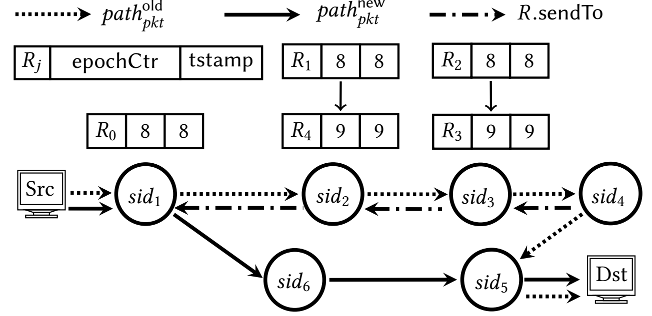


Figure 5. Example of send-back rules.

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

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

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

$$\begin{aligned} &R_3.\text{switch} \leftarrow R_2.\text{switch} \\ &R_3.\text{sendTo} \leftarrow R_1.\text{switch} \\ &R_3.\text{priority} \leftarrow \infty \\ &R_3.\text{cover} \leftarrow \bigcup \{pkt\} \qquad (1) \\ &R_3.\text{epochCtr} \leftarrow epochCtr \\ &R_3.\text{tstamp} \leftarrow epochCtr \end{aligned}$$

where the union in (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 $\mathcal{R}^{\text{add}}$ using flowadd commands and removes $\mathcal{R}^{\text{del}}$ 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[1], 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

[1]A delay of $linkLatency \times diameter$ should suffice, where $diameter$ is the length of the longest routing path in the network.

(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 $\mathcal{R}^{\text{old}}$, and similarly the "new" equivalence classes each consists of all flows that each match to the same sequence of rules in $\mathcal{R}^{\text{new}}$. 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 $path^{\text{old}}_{pkt}$ and $path^{\text{new}}_{pkt}$), 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 ($R_1 \in \mathcal{R}^{\text{old}}$ and $R_2 \in \mathcal{R}^{\text{new}}$ where $R_2.\text{switch} = R_1.\text{sendTo}$) and old and new paths of flows they cover (at most $c^{\text{old}} \times c^{\text{new}}$ pairs), and so very coarsely, "Backward closure" costs $O\left(c^{\text{old}}c^{\text{new}} \times \left|\mathcal{R}^{\text{old}}\right| \times |\mathcal{R}^{\text{new}}|\right)$ time. Similarly, "Send-back rules" considers pairs of adjacent, old rules ($R_1 \in \mathcal{R}^{\text{old}}$ and $R_2 \in \mathcal{R}^{\text{old}}$ such that $R_2.\text{switch} = R_1.\text{sendTo}$) and so incurs cost of $O\left(c^{\text{old}}c^{\text{new}} \times \left|\mathcal{R}^{\text{old}}\right|^2\right)$ time. So, the running time of our algorithm is theoretically dominated by steps of $O\left(c^{\text{old}}c^{\text{new}} \times \left(\left|\mathcal{R}^{\text{old}}\right| \times |\mathcal{R}^{\text{new}}| + \left|\mathcal{R}^{\text{old}}\right|^2\right)\right)$ 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. 7 we will empirically demonstrate the scalability of our algorithm.

5.2 Properties

Proposition 1. *The protocol of Sec. 5.1 implements suffix causal consistency.*

Proof. (Sketch) Let R_1 be the first rule R matched to a packet pkt with $R.\text{epochCtr} = pkt.\text{epochCtr}$. 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 $pkt.\text{tstamp} \leftarrow R.\text{epochCtr}$. This chain delivers pkt back to a switch on the new path for pkt in the epoch with index $pkt.\text{epochCtr}$. 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 $R.\text{epochCtr} = pkt.\text{epochCtr}$. Then, each rule R_3 to which pkt is matched at downstream switches has $R_3.\text{epochCtr} < pkt.\text{epochCtr}$ (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 $pkt.\text{epochCtr}$, which would preclude the next rule R to match pkt having $R.\text{epochCtr} < pkt.\text{epochCtr}$. 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 $R_4 \in \mathcal{R}^{\text{new}}$ would have (i.e., for which $match(\mathcal{R}^{\text{new}}, R_4, pkt)$ is true), where $\mathcal{R}^{\text{new}}$ refers to that set of rules as generated in epoch $pkt.\text{epochCtr}$. Then, $R_3 \in \mathcal{R}^{\text{del}}$ and $R_4 \in \mathcal{R}^{\text{add}}$ in that invocation. By inductive application of the "backward closure" rule, $R_2.\text{tstamp} = pkt.\text{epochCtr}$, contradicting the assumption that R_2 is the last rule R matched to pkt with $R.\text{epochCtr} = pkt.\text{epochCtr}$. □

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* [25], 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., [37]). 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 *waypoint enforcement* (e.g., [3, 9, 34]), 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 waypoint enforcement 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.

Model checking We subjected our algorithm (specified using a modeling language) to model checking to verify its enforcement of suffix causal consistency, as well as black-hole freedom and bounded looping. Specifically, we used Z3Py[2], a Python API for the Z3 solver [30], to model network updates, and let the Z3 solver verify black-hole freedom, bounded looping and suffix causal consistency.

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 ($|sid_i.\text{ruleSet}| \leq 10$), 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; e.g., if $R_j, R_{j'} \in sid_i.\text{ruleSet}$ then $R_j.\text{priority} \neq R_{j'}.\text{priority}$ or $R_j.\text{cover} \cap R_{j'}.\text{cover} = \emptyset$. 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, $R.\text{epochCtr}$ and $R.\text{tstamp}$ were allowed to range over $\{1, 2, 3\}$ (explored by the model checker). We modeled the effects of one new epoch[3] ($epochCtr = 4$) that implemented some different routing policy from the first (i.e., at least one flow traveled a different path to its egress) using rules for which $R.\text{epochCtr}$ and $R.\text{tstamp}$ 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 $pkt.\text{tstamp}$. Specifically, if $pkt.\text{tstamp} \leq R_1.\text{epochCtr}$ and $pkt.\text{tstamp} \leq R_2.\text{epochCtr}$, either rule could be applied to the packet, creating two branches. If $pkt.\text{tstamp} > R_1.\text{epochCtr}$ and $pkt.\text{tstamp} \leq R_2.\text{epochCtr}$, 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. 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. 5.1 to motivate the algorithm stages. For the algorithm presented in Sec. 5.1, however, after running about 6 days, the model checker successfully terminated and found no violations.

5.3 Timestamp Reset

Since the number of bits in each packet header to maintain the timestamp $pkt.\text{tstamp}$ 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 $R.\text{tstamp}$ of each deployed rule R to $R.\text{tstamp} \leftarrow 0$ and awaits an acknowledgment from each switch. Second, after a delay of sufficiently long to ensure that packets pkt remaining in the network have $pkt.\text{tstamp} = 0$, the controller issues commands to update the $R.\text{epochCtr}$ fields of all deployed rules R to $R.\text{epochCtr} \leftarrow 0$. Since each packet pkt traveling the network has $pkt.\text{tstamp} = 0$, resetting $R.\text{epochCtr} \leftarrow 0$ for all rules does not result in packet drops or delays.

6 Implementation

We implemented our algorithm in both the P4 switch [5] and Open vSwitch[4]. To issue updates to the switches, we utilized P4 Runtime[5] and the Ryu controller[6], respectively.

6.1 P4

We used the BMv2 switch target (i.e., behavioral-model[7]) 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., $pkt.\text{tstamp}$. Upon packet arrival, the parser of the switch extracts $pkt.\text{tstamp}$ 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 $R.\text{epochCtr}$ and $R.\text{tstamp}$ as metadata for this packet. If $R.\text{epochCtr} \geq pkt.\text{tstamp}$, then the ingress

[2] http://ericpony.github.io/z3py-tutorial/guide-examples.htm

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

[4] http://openvswitch.org

[5] https://github.com/p4lang/PI/

[6] https://osrg.github.io/ryu/

[7] https://github.com/p4lang/behavioral-model/

pipeline forwards the packet to the egress pipeline with instructions to tag the packet with pkt.tstamp ← R.tstamp and forward the packet to the outbound port defined by R.sendTo. Otherwise (i.e. pkt.tstamp > R.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.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. 4), we used the approach proposed by Han et al. [11]. 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.

6.2 Open vSwitch

In contrast to the P4 implementation above, which uses a new header field to store pkt.tstamp, our Open vSwitch (OVS) implementation leverages unused header bits to store pkt.tstamp in each packet. These header bits, which are the same as those used by COCONUT [10], include 12 bits of the VLAN tag (also used by CU [35]) 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.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.tstamp > R.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. 5) 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.

6.3 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. [15]. The output rules ($\mathcal{R}^{\text{new}}$) and the rules already deployed to the network ($\mathcal{R}^{\text{old}}$) are the inputs to our algorithm described in Sec. 5.

Our P4 implementation uses the P4 runtime, which is a protocol-independent API using Protobuf[8] and gRPC[9] to issue rule updates to the P4 switches. For our OVS implementation, we leveraged the Ryu controller with the OpenFlow protocol [32]. 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 commands, 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.

7 Evaluation

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

- SCC compares favorably to COCONUT [10] and to the original, uncoordinated approach to rule updates in terms of the packets dropped during an update (Sec. 7.2), and favorably to COCONUT, CU [35], and TSU [23] in terms of the packets dropped due to link failures (Sec. 7.3).

[8] https://developers.google.com/protocol-buffers/
[9] https://grpc.io/

- SCC deploys rules more quickly than CU, COCONUT, or TSU (Sec. 7.4) and imposes less rule storage overhead in amount and/or duration than these alternatives (Sec. 7.5).
- The rule generation time of our algorithm scales across a range of both fat-tree and ISP topologies (Sec. 7.6).
- The buffering overhead imposed by our algorithm is manageable for today's switches (Sec. 7.7).

7.1 Setup

For the tests in Secs. 7.2–7.5 and 7.7, our experiments were conducted on topologies emulated in Mininet[10] 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[11]) 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. [13], 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 [6]. For the ISP topologies, we used shortest-path routing and induced route changes by breaking links. We used Hping[12] to craft packets in our Open vSwitch tests. We used Scapy[13] 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. 7.6 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. 7.2–7.5, we primarily compare our algorithm (SCC) to COCONUT [10], CU [35], TSU [23] 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.

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 timestamp indicating the current epoch, and each downstream switch uses rules with the same epoch timestamp to match the packet. The timestamp carried by the packet will not be changed as it traverses the network. The update phase deploys—but does not yet enable—rules for the new epoch (with a new timestamp) at all switches. Then, after all the new rules have been installed, the controller updates the ingress switch to start tagging packets with the new epoch timestamp, resulting in the new epoch's rules being applied to any packet carrying the new epoch timestamp. 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.

COCONUT As discussed in Sec. 2, 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. One difference from CU is that 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). 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.

[10] http://mininet.org/
[11] http://www.topology-zoo.org/
[12] http://www.hping.org/
[13] http://www.secdev.org/projects/scapy/

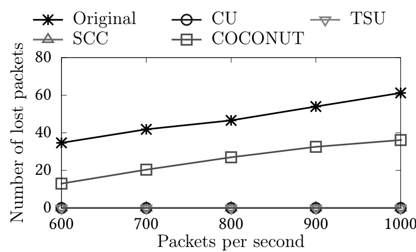
Original "Original" deployment commits network updates in only one step. Specifically, the controller sends updates to switches without using any synchronization algorithm. Since messages from the controller to switches may suffer different delays, this deployment does not enforce any consistency.

7.2 Packet Loss During Regular Update

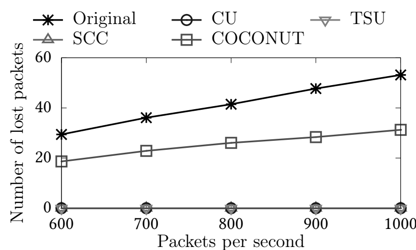
We measured the number of lost packets based on different packet sending rates when pushing updates to the switches concurrently to change the paths of packets in the fat-tree topology ($K = 8$). As shown in Fig. 6, our protocol (SCC), TSU, and CU incur no packet loss because these two mechanisms maintain 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).

7.3 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. 7a–7b), 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. 7c). The DFN results for Open vSwitch are similar to those for P4 and so are omitted for brevity. As we can see in Fig. 7, 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.

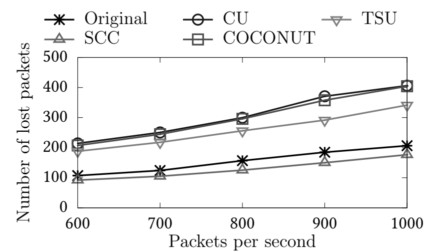


(a) P4 and fat-tree ($K = 8$)

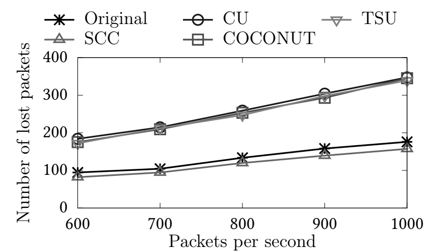


(b) OVS and fat-tree ($K = 8$)

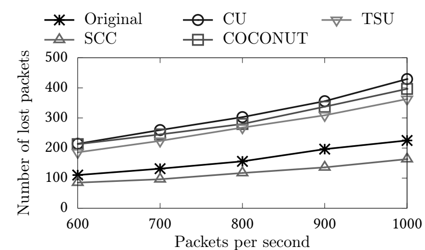
Figure 6. Packet loss during normal update. Each data point is an average of 100 runs.



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



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

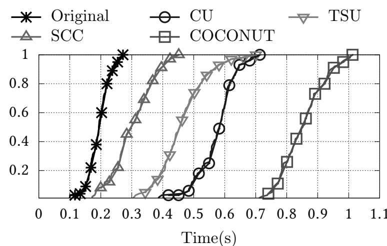


(c) P4 and DFN topology

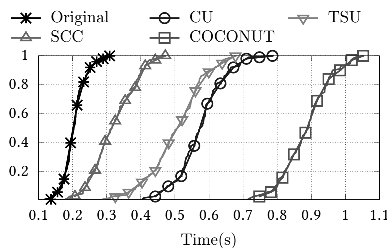
Figure 7. Packet loss during link failure. Each data point is an average of 100 runs.

7.4 Rule Deployment Time

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. 8 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.



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



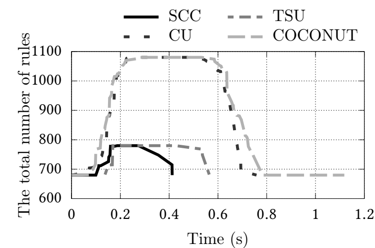
(b) P4 and DFN

Figure 8. CDF of rule deployment times over 100 epochs.

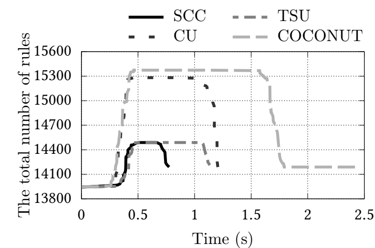
SCC outperforms CU and COCONUT during rule deployment for two reasons. First, SCC simply deploys rules to fewer switches, since it attempts to minimize the number of switches to which it must do so. Second, SCC involves fewer phases of communication between the controller and switches. Here, COCONUT 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.

7.5 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. 9a. We repeated this evaluation on the DFN topology, but by breaking the "busiest" link; see Fig. 9b. 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 9. Rules in the network during epoch installation.

7.6 Rule Generation Time

Rule generation times for SCC are shown in Fig. 10. There are two groups of box-plots shown in Fig. 10: 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. 10 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 data (as in Fig. 9). The experiments with ISP topologies reflect the cost of rule generation when a random link fails, causing all routes traversing it to change.

Fig. 10 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.

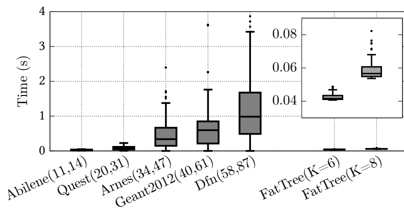


Figure 10. Distributions of SCC rule-generation times (100 path updates in fat-tree topologies; random link failure in ISP topologies).

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).

7.7 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. 11 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 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[14], and so these buffering obligations should not be problematic.

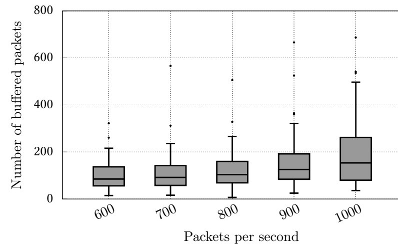


Figure 11. Distributions of SCC maximum switch buffering during 10 path updates in fat-tree ($K = 8$).

8 Conclusion

We have proposed suffix causal consistency (SCC) as an interpretation of causal consistency for rule 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 retention of fewer extra rules during the update, and its rule generation scales across a wide range of topologies.

[14] https://people.ucsc.edu/~warner/buffer.html

Acknowledgements

We are grateful to Prof. Jon Crowcroft and the anonymous reviewers for their comments. This work was supported in part by NSF grant 1535917.

References

[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. 1995. Causal Memory: Definitions, Implementation and Programming. *Dist. Comp.* (1995).

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM.* 63–74.

[3] B. Anwer, T. Benson, N. Feamster, and D. Levin. 2015. Programming Slick Network Functions. In *1st ACM SOSR.* 1–13.

[4] P. Bailis, A. Ghodsi, J. Hellerstein, and I. Stoica. 2013. Bolt-on Causal Consistency. In *ACM SIGMOD/PODS.* 761–772.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. 2014. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Comp. Comm. Rev.* (July 2014), 87–95.

[6] H. Chen and T. Benson. 2017. Hermes: Providing Tight Control over High-Performance SDN Switches. In *13th ACM CoNEXT.* 283–295.

[7] C. J. Fidge. 1988. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference.* 56–66.

[8] K. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid. 2018. Loop-Free Route Updates for Software-Defined Networks. *IEEE/ACM Trans. Netw.* 26, 1 (Feb. 2018), 328–341.

[9] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *ACM SIGCOMM.* 163–174.

[10] S. Ghorbani and P. Godfrey. 2017. COCONUT: Seamless Scale-out of Network Elements. In *12th ACM EuroSys.*

[11] J. Han, H. Jong Hun, P. Mundkur, R. Prashanth, C. Rotsos, G. Antichi, N. Dave, A. Moore, and P. Neumann. 2015. Blueswitch: Enabling Provably Consistent Configuration of Network Switches. In *11th ACNS.*

[12] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. 2013. Achieving High Utilization with Software-driven WAN. In *ACM SIGCOMM.* 15–26.

[13] D. Huang, K. Yocum, and A. Snoeren. 2013. High-Fidelity Switch Models for Software-defined Network Emulation. In *ACM HotSDN.*

[14] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. 2014. Dynamic Scheduling of Network Updates. In *ACM SIGCOMM.* 539–550.

[15] N. Kang, Z. Liu, J. Rexford, and D. Walker. 2013. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *9th ACM CoNEXT*.

[16] N. Katta, J. Rexford, and D. Walker. 2013. Incremental Consistent Updates. In *2nd ACM HotSDN*.

[17] F. Klaus-Tycho, M. Ratul, and W. Roger. 2016. Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes. In *IFIP Netw.*

[18] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM* 21, 7 (July 1978), 558–565.

[19] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. 2014. Tango: Simplifying SDN Control with Automatic Switch Property Inference, Abstraction, and Optimization. In *ACM CoNEXT.*

[20] H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. 2013. zUpdate: Updating Data Center Networks with Zero Loss. In *ACM SIGCOMM*. 411–422.

[21] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *23rd ACM SOSP*. 401–416.

[22] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In *10th USENIX NSDI*. 313–328.

[23] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. 2016. Transiently Secure Network Updates. In *ACM SIGMETRICS*. 273–284.

[24] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. 2014. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies. In *13th ACM HotNets*.

[25] R. Mahajan and R. Wattenhofer. 2013. On Consistent Updates in Software Defined Networks. In *12th ACM HotNets*. 1–7.

[26] F. Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithsm*. 215–226.

[27] J. McClurg, H. Hojjat, P. Černý, and N. Foster. 2015. Efficient Synthesis of Network Updates. In *36th ACM PLDI*. 196–207.

[28] T. Mizrahi, O. Rottenstreich, and Y. Moses. 2017. TimeFlip: Using Timestamp-Based TCAM Ranges to Accurately Schedule Network Updates. *IEEE/ACM Trans. Netw.* 25, 2 (April 2017), 849–863.

[29] T. Mizrahi, E. Saat, and Y. Moses. 2015. Timed Consistent Network Updates. In *1st ACM SOSR*.

[30] L. Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th TACAS/ETAPS*. 337–340.

[31] T. Nguyen, M. Chiesa, and M. Canini. 2017. Decentralized Consistent Updates in SDN. In *ACM SOSR*.

[32] Open Networking Foundation. 2015. OpenFlow Switch Specification, Version 1.5.1. (26 March 2015).

[33] P. Pereíni, M. Kuzniar, M. Canini, and D. Kostić. 2014. ESPRES: Transparent SDN Update Scheduling. In *ACM HotSDN*.

[34] Z. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *ACM SIGCOMM*. 27–38.

[35] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. 2012. Abstractions for Network Update. In *ACM SIGCOMM*.

[36] V. Stefano and C. Luca. 2016. FLIP the (Flow) table: Fast lightweight policy-preserving SDN updates. In *35th IEEE INFOCOM*.

[37] K. Zarifis, R. Miao, M. Calder, E. Katz-Bassett, M. Yu, and J. Padhye. 2014. DIBS: Just-in-time Congestion Mitigation for Data Centers. In *9th ACM EuroSys*. 6:1–6:14.

[38] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. 2015. Enforcing Customizable Consistency Properties in Software-defined Networks. In *12th USENIX NSDI*. 73–85.