HotSwap: Correct and Efficient Controller Upgrades for Software-Defined Networks

Laurent Vanbever
Princeton University
vanbever@cs.princeton.edu

Joshua Reich
Princeton University
jreich@cs.princeton.edu

Theophilus Benson
Duke University
tbenson@cs.duke.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

ABSTRACT

Like any complex software, SDN programs must be updated periodically, whether to migrate to a new controller platform, repair bugs, or address performance issues. Nowadays, SDN operators typically perform such upgrades by stopping the old controller and starting the new one—an approach that wipes out all installed flow table entries and causes substantial disruption including losing packets, increasing latency, and even compromising correctness.

This paper presents HOTSWAP, a system for upgrading SDN controllers in a disruption-free and correct manner. HOTSWAP is a hypervisor (sitting between the switches and the controller) that maintains a history of network events. To upgrade from an old controller to a new one, HOTSWAP bootstraps the new controller (by replaying the history) and monitors its output (to determine which parts of the network state may be reused with the new controller). To ensure good performance, HOTSWAP filters the history using queries specified by programmers. We describe our design and preliminary implementation of HOTSWAP, and present experimental results demonstrating its effectiveness for managing upgrades to third-party controller programs.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; C.2.3 [**Network Operations**]: Network management

Keywords

Software-Defined Network; Controller Upgrade; Dynamic Software Updating;

1. INTRODUCTION

In a software-defined network (SDN), a logically-centralized controller responds to changes in network conditions by updating the packet-processing rules installed on switches. However, any software system requires upgrades to fix bugs, add features, and improve performance. SDN controllers are no exception. In today's SDNs, upgrading a controller involves stopping and restarting the controller. Unfortunately, during the transition, the network cannot handle link failures, rule timeouts, or packets diverted to the controller. Even worse, the default way to start a new controller is to clear the existing rules in the switches, to avoid inconsistencies with previously installed rules. But deleting rules forces the new controller to handle a large number of events, leading to high overhead and performance disruptions. Moreover, for stateful applications like some firewalls, a newly launched controller may behave incorrectly because it does not have visibility into past events.

Upgrading control-plane software is not a new problem. High-end commercial routers minimize control-plane downtime using so-called In-Service Software Upgrades [1, 4] where the new control software is installed on a separate blade, while the old software continues to run. Many standard routing protocols provide "graceful restart" mechanisms for (re)establishing peering relationships and routing state when new control software starts (e.g., [10, 14]). Because routing protocols typically depend only on the current "snapshot" of the topology or routes, rather than the timing and ordering of previous messages, it is relatively easy to bootstrap the control-plane state by having each peer reannounce its current routing state. However, each of these mechanisms only applies to a single routing protocol, leading to a "cottage industry" of point solutions for control-plane upgrades.

In contrast to routers, SDN controllers are harder to upgrade. They have a wider range of functionality, as well as internal state and dependencies on past events. Hence, simply replicating the controller does not address the upgrade problem—cloning a controller would be relatively easy, but the new controller would need to run the same software as the old one. Instead, our goal is to develop techniques for upgrading controllers with *new software*, to address performance problems, fix bugs (e.g., changing the rules installed on switches), move to a new controller platform (e.g., from Floodlight to POX), or change the application control logic and data structures (e.g., to a more scalable implementation). As such, we must bootstrap the internal state of the new controller, and update whatever switch rules have changed.

Despite these challenges, SDN provides a unique opportunity to solve the controller upgrade problem once and for all, without customizing the solution to specific protocols or control logic. The open API between the controller and the switches provides a general way to learn the network state, record network events over time, and (re)play sequences of events to bootstrap a new controller. Of course, simply recording and replaying all events would introduce substantial storage overhead and long delays in starting the new controller, so a naive solution like this is not likely to scale.

HotSDN'13, August 16, 2013, Hong Kong, China.

This paper presents HOTSWAP, a new system that provides seamless upgrades for SDN controllers. HOTSWAP is a hypervisor that sits between the controller and the switches. HOTSWAP handles upgrades in three phases: (i) it efficiently records network events; (ii) it relays these events to the new controller, while the old controller continues to run; (iii) it updates the rules to reflect any necessary changes in forwarding policy; and (iv) it replaces the old controller with the new one. To reduce overhead and delay, HOTSWAP incorporates mechanisms for filtering the sequence of events and (where possible) capitalizing on existing rules installed in switches, while still correctly bootstrapping the new controller. The right solution depends on whether and how the control logic depends on *past events*, and precisely how *correctness* is defined. We identify a few canonical classes of controllers, and design general solutions for updating the applications in each class correctly and efficiently. Application developers need only provide simple, high-level information to enable our system to apply the right replay technique. In summary, this paper makes four main contributions:

- *Upgrade mechanisms:* We describe disruption-free, yet correct controller upgrade procedures (§3).
- *Upgrade correctness:* We develop a formalism that captures the key correctness properties of controller upgrades (§3).
- *Implementation:* We describe a controller-independent implementation of HOTSWAP (§4).
- *Experiments:* We measure the disruptions caused by restarting an SDN controller and show that no disruption is observed when using HOTSWAP (§2, §5).

2. CONTROLLER UPGRADE PROBLEMS

This section presents examples that illustrate the main problems that arise with naive upgrades in controllers today: service disruptions and correctness bugs.

Service disruptions. In a naive upgrade, the operator simply halts the old controller and starts the new one. To avoid issues where the rules installed by the old controller are inconsistent with the rules specified by the new controller, most controllers issue commands to delete all rules on every switch after completing the OpenFlow handshake. Unfortunately, this can lead to massive disruptions since the new controller must process a large number of events while it reconstructs the network state.

To quantify the disruption that can happen during a naive upgrade, we injected traffic in an emulated network running in Mininet HiFi [5] and measured the amount of traffic lost and the median latency during a controller restart. We configured the emulated network as a FatTree topology with 20 switches and 32 hosts. As controller, we used Floodlight [2] running the default forwarding application. We generated traffic between 200 randomly selected pairs of hosts by sending 64 bytes `ping` probes every 10 ms, corresponding to roughly 10 Mbps of traffic overall. We used an Intel Xeon W5580 processor with 8 cores and 6GB of RAM for Mininet, an Intel i7 processor with 8GB of RAM for the controller, and repeated each experiment 15 times.

The results of the experiments are shown in Figure 1. Overall, the network experienced substantial packet loss and increased latency for about 15 seconds after the restart. To determine the cause of this disruption, we inspected the Open vSwitch logs and found that the switches were dropping a large number of packets due to full buffers. In the worst case, up to 90% of the traffic was lost, while the median latency increased by a factor of 10^5, going from 0.1ms to 10s! Clearly such a disruption would be completely unreasonable in settings where high availability is required.

```
if ip_src in H:
  insert(10,ip_src,
   ip_dst,fwd)
  insert(5,ip_dst,
   ip_src,fwd)
else:
  insert(15,ip_src,
   ip_dst,drop)
```

```
if ip_src in H:
  insert(10,ip_src,
   ip_dst,fwd)
  insert(5,ip_dst,
   ip_src,fwd)
elif scan[ip_src]>3:
  insert(15,ip_src,
   ip_dst,drop)
else:
  insert(15,ip_src,
   *, drop)
  scan[ip_src]+=1
```

Figure 2: Depending on the network traffic, these two simple SDN applications can behave incorrectly after a controller restart. Restarting the stateful firewall (left) causes previously allowed traffic to be blocked, while restarting the scan defense application (right) causes blocked traffic to be allowed.

Correctness problems. There is another subtle issue that also arises with naive upgrades: simply stopping the old controller and starting the new one can introduce new bugs. In particular, invariants that depend on the controller correctly tracking certain pieces of state can be broken after the upgrade. As an example, consider a network that consists of a single switch connected to several hosts. The controller program is a stateful firewall that allows internal hosts to initiate communication with external hosts, but blacklists external hosts that attempt to send unsolicited traffic to internal hosts. A pseudocode implementation of this program is given in Figure 2 (left). Now consider the following sequence of events: internal host i_1 sends traffic to external host e_1, the operator restarts controller, and then e_1 sends a reply back to i_1. Because the restarted controller receives the return traffic first, it will erroneously blacklist e_1. That is, restarting the controller causes *allowed* traffic to be incorrectly *blocked* by the network.

For another example, consider the same network, but now assume that the controller program is a stateful firewall that additionally blacklists any external hosts that attempt to initiate three unsolicited connections toward any internal host for all time. A pseudocode implementation of this program is given in Figure 2 (right). Intuitively, this program implements an extremely simple defense against network scans. Now consider the following sequence of events: external host e_1 attempts to connect to internal hosts i_1, i_2, and i_3, and is blacklisted; the operator restarts the controller; and then i_1 sends traffic to e_1. Because the restarted controller sees the outgoing traffic first, it will incorrectly allow communication between i_1 and e_1, even though the latter was previously blacklisted. That is, restarting the controller causes *forbidden* traffic to be incorrectly *allowed* by the network.

3. THE HOTSWAP SYSTEM

This section presents the design of HOTSWAP, a software hypervisor that enables correct and efficient SDN controller upgrades. The key idea in HOTSWAP is to "warm up" the new controller by replaying a history of past events rather than giving it control of the network immediately. This makes it possible to reuse existing forwarding rules installed by the old controller, and also avoids correctness problems, since the history can be engineered to ensure that critical pieces of internal state are reconstructed on the new controller before it assumes control of the network. We first de-

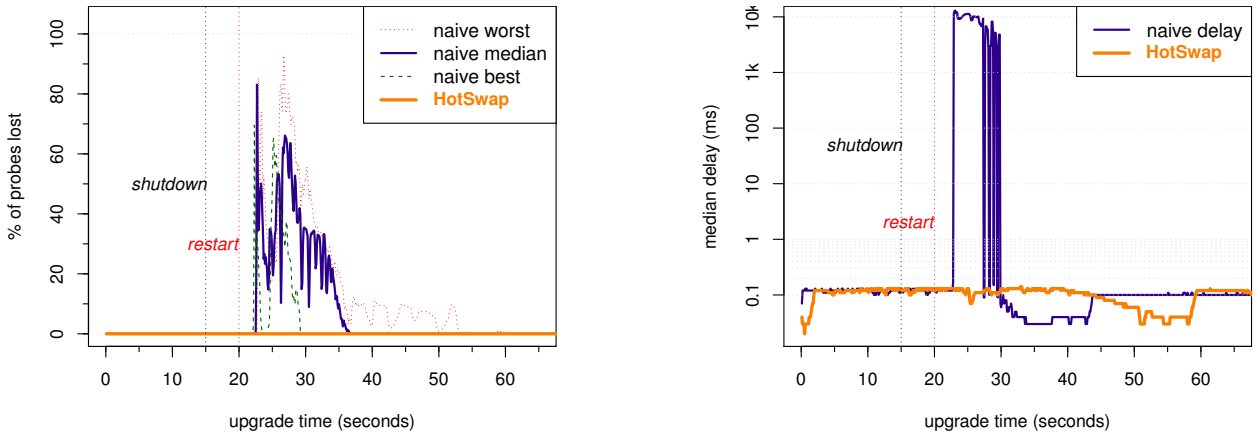

Figure 1: Restarting a controller can create massive network disruption involving packet losses (left) and increased network delays (right) during tens of seconds. In the worst case, *up to 90% of the entire network traffic* is dropped.

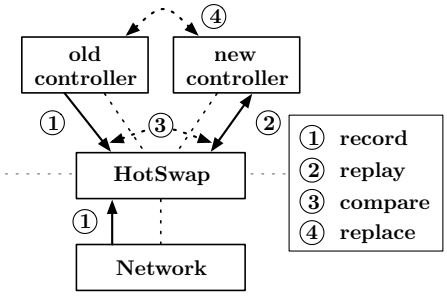

Figure 3: HOTSWAP architecture and upgrade phases.

scribe the overall design of HOTSWAP, and then explain its mechanisms for efficiently recording and replaying state during upgrades.

Architecture. HOTSWAP is a software hypervisor that sits between the network and the controllers. Figure 3 depicts the architecture of the system graphically. To upgrade from an old controller C to a new controller C', HOTSWAP proceeds in four steps:

1. In the *record* phase, the hypervisor executes the old controller C, collects a trace T of network events, and maintains the network state N generated by C. Network events include switch-to-controller messages including topology changes, traffic statistics, and diverted packets. The network state consists of the forwarding rules installed on switches.
2. After the upgrade is initiated, the system transitions to the *replay* phase. It replays the collected trace T on C', but intercepts controller-to-switch messages such as flow modifications rather than actually sending them to the switches. Using these messages, it builds up a representation of the network state N' generated by C'.
3. After the full trace T has been replayed to the new controller C', the system transitions to the *compare* phase. It computes a sequence of flow insertions and deletions that transition the network from N to N'.
4. Finally, in the *replace* phase, the system transitions the network to the new state N' using a consistent update [12]. It then detaches the old controller C and re-enters the record phase, but this time with C' attached as the current controller.

Taken together, these steps enable to perform upgrades while avoiding the severe disruptions that occur with naive upgrades. In particular, when the old and new controllers are running similar programs, the two versions of the network state N and N' are often nearly identical, so much of the state can be safely reused. Even when the two versions of network state are dissimilar—e.g., because the new controller fixes a bug in the old program, precomputing the network state enables a more graceful upgrade process.

Note that during the upgrade, HOTSWAP must main relative timeouts and counters values for the rules that are equivalent (and thus preserved) between the C and C' controllers. For example, if C' pushes a rule with a soft timeout of 10 seconds and the corresponding rule installed by C expires earlier, HOTSWAP must transparently install a rule with a soft timeout equivalent to the relative timeout. Similar techniques can be used to handle traffic statistics counters: upon receiving a flow modification from C', HOTSWAP registers the current counter values (if any) and rewrites subsequent requests to reflect the difference.

Histories. Naively recording a full trace containing every network event for all time would clearly not scale—even just storing the full trace would quickly become unwieldy. To address these issues, HOTSWAP also includes mechanisms that allow an operator to collect a restricted *history* H. For example, in a learning switch, it suffices to only record the last packet diverted to the controller from each host since the behavior of the application does not depend on other events; in a stateful firewall with scan defense, it suffices to record the diverted packets for current outgoing flows and all events used to classify scanning hosts as malicious; and in a static routing application it suffices to only record the last event for each element of the network topology since other events such as traffic statistics and diverted packets do not affect the overall behavior in any essential way. HOTSWAP provides basic operators for extracting histories H from traces T, including recording the last element of a particular kind of event, and recording events permanently. For example, in the stateful firewall with scan defense we might permanently record the number of times the external host has attempted to connect to an internal host.

Upgrade correctness. Correctness is a critical issue in controller upgrades. As the stateful firewall examples in the preceding section showed, restarting a controller abruptly loses internal state, which can break important application invariants. Intuitively, when

we upgrade from the old controller C to a new controller C', we would like the network to behave as if the new controller had been running all along. Let $T@T'$ denote the trace obtained by concatenating T and T', and let $\sqsubseteq$ be a relation on controller outputs that captures a notion of "acceptable" differences. The intuitive condition described above can be captured formally using the following formula:

$$\forall T'.\ C'(H@T') \sqsubseteq C'(T@T')$$

It captures the notion that the network state obtained by replaying the history H on C' is "acceptable" on all future events. In particular, the critical pieces of the internal controller state are correctly reconstructed by replaying with H. Note that this condition does not specify that the internal state of C' must be identical to the state that would have been obtained if the new controller was running all along. But because the condition is required to hold for all future traces T', any discrepancies that would eventually result in observable differences on controller output are ruled out.

Note that the definition above is parameterized on a relation $\sqsubseteq$ that captures acceptable differences on controller output. This can be instantiated with different relations depending on the applications running on top of the controller. For example, the strictest relation is $=$, or equality, which mandates the network state N' obtained after replaying the history to be identical to N. A more relaxed relation is $\equiv$, or semantic equivalence, which mandates N' to behave the same as N after the replay without requiring a perfect identity. For instance, an operator might want to use the $\equiv$ relation when it updates her controller for one that aggregates contiguous forwarding entries without changing the actual forwarding paths taken by data packets. A third possible relation is $\prec$, or approximate semantic equivalence, which mandates that N' must behave the "same" as N in some relaxed sense—i.e. N' may contain less forwarding rules than N but where they are both defined they agree. For instance, when upgrading a learning switch, an operator may want to use the $\prec$ relation to allow the new controller to flood traffic for some destinations for a limited period of time. Of course, other relations are also possible.

Controller classes. In general, picking a correct record and replay mechanism depends on the type of controller application and the relation required for correctness. We describe here several common classes of controllers we have identified in our preliminary work on HOTSWAP.

At the highest level, a SDN application can be either *stateless* or *stateful* according to whether or not its behavior depends on network events.

Stateless applications. These applications include all purely proactive applications including static routing and static access control. These applications do not need to collect any history and can use the strictest relation, equality, on controller outputs.

Stateful applications can be further subdivided into three classes: (i) *topology-dependent*, (ii) *stationary* and, (iii) *general applications*. Each of these classes differs in the amount of state that HOTSWAP needs to maintain.

Topology-dependent applications. These applications depend only on the current topology graph, and not on the traffic that has been exchanged. This is the case of most routing applications (e.g., shortest-path). For these applications, HOTSWAP simply records the last version of each topology event and allows C' to discover the links between switches using existing OpenFlow (e.g., LLDP) mechanisms for topology discovery before setting it as primary controller.

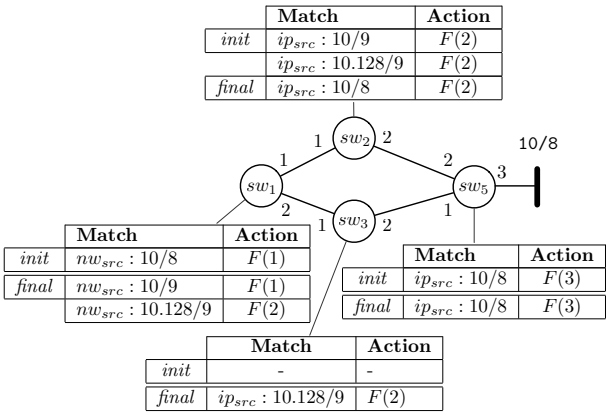

Figure 4: A simple traffic-engineering upgrade example

Stationary applications. These applications depend on the traffic exchanged, but only on the last network event of a given type. An example of such an application is a learning switch. Indeed, the state of a learning switch depends on the locations of hosts that have been learned by observing network traffic. HOTSWAP therefore only needs to keep track of the last piece of relevant state. Actually, in these applications, the network state N generated by the old controller C often contains all the information necessary to reconstruct the state in C'. For instance, in a network running a learning switch application, one can infer the last position of each host by looking at the forwarding rules. For these applications, HOTSWAP can use record and replay mechanisms where network events are automatically generated from the actual content of the old forwarding tables stored in N. This is attractive because it does not require recording any additional events beyond the network state. Sections 4 and 5 discuss an implementation and evaluation of this mechanism.

General applications. In general, the behavior of the old and new controller applications can depend on the order and timing of network events. For instance, in Figure 2, the state maintained by the scan defense application depends on the relative ordering between network events, but also on the history of past network events. To recreate the state of these applications, a strawman solution would be to record the full timing and ordering of all network events[1] but this "solution" has obvious scalability and efficiency problems as the storage requirements and replay time would quickly become enormous. Fortunately, using the mechanisms for constructing histories provided in HOTSWAP, many stateful applications can be handled, by only maintaining the relatively small subset of network events needed to reconstruct an equivalent network state.

4. IMPLEMENTATION

This section presents a prototype implementation of HOTSWAP as an extension of FlowVisor [16]. It starts by giving a general overview and then presents a detailed description of the *replay* and *compare* algorithms that form the core of HOTSWAP.

Overview. We implemented a first prototype of HOTSWAP by extending FlowVisor. Conceptually, FlowVisor is a hypervisor that sits between the controllers and switches. FlowVisor provides Open-

[1]Note that it would actually be necessary to record all *data traffic* as C forwarding rules might prevent HOTSWAP from receiving network events required to recreate C' state.

Flow networks with virtualization capabilities by slicing the network into different and isolated slices. Each slice can be configured with a different flow space using topology and packet header characteristics and is controlled by a different controller.

To implement HOTSWAP, we extended FlowVisor with two new types of slices: *primary* and *replayed* slices. HOTSWAP automatically records the network state as well as a history of network events for *primary* slices. Similarly, HOTSWAP automatically monitors the output for *replayed* slices, but prevents them from actually writing to the network. Before the upgrade, C is attached to a *primary* slice and C' to a *replayed* slice, both sharing the same flow space. After the upgrade, the C' (resp. C) slice is converted into a *primary* (resp. *replayed*) slice.

We implemented two new FlowVisor primitives to manage the process: (i) `replay <source> <target> <method>` and (ii) `replace <source> <target>`. The first triggers the replay of the history of events attached to the *primary* slice `source` to the *replayed* slice `target`. The `method` parameter designates a replay mechanism. The second transitions the network to the state computed by the slice `target` and swaps the roles of the two slices.

Replay algorithm. The current HOTSWAP implementation supports both generating events from an existing network state and replaying recorded events. Since replaying recorded events is relatively straightforward, we focus on former. We illustrate the execution of the algorithm by considering a simple upgrade scenario in which a traffic-engineering application reactively installs destination-based forwarding rules (Figure 4).

Using the current network state, the ***replay*** algorithm (Figure 5) generates packet-in events and sends them to C'. It iterates over the forwarding rules for each switch in decreasing order of priority, and generates one packet-in event per forwarding rule. To generate a packet-in event, the algorithm copies the pattern values to the corresponding fields in the IP packet. When a wildcard is encountered for a given field, a random but compatible value is used. For example, the replay procedure executing on sw_1 will generate an IP packet with a source address belonging to $10/8$ and random values for all the other fields. As rules can overlap, the algorithm keeps track of the generated packet to ensure that the same packet is not generated for more than one matching rule.

Observe that the current replay mechanism can generate packet-in events that would not have been received by the controller otherwise. This could be problematic for some SDN applications, such as the ones that generate an entire path when a packet arrives at an ingress switch. Indeed, these applications would not expect to see packet-in events for packets on interior ports. Yet, there would be rules in those interior switches. To handle this situation, HOTSWAP can generate packet-ins for rules matching ingress links first, see what internal rules get pushed in the network, and prevent packets-in events from being generated for these rules. Ingress links can be identified by building the forwarding graph corresponding to each forwarding equivalence class (using techniques similar to HSA [8]).

Compare algorithm. The ***handleFlowMod*** algorithm (see Figure 6) monitors the network state N' generated by C' during the replay. For any C' generated rule, the algorithm checks whether it matches exactly a C generated rule. In this case (lines 5–7), no action is required and the corresponding rule entry remains installed in the switch. When the forwarding rules generated by C' and C differ, the algorithm takes "reconciliation" actions (lines 9–23) to ensure that the resulting network state is compatible with C'

```
1: replay(switch)
2: generated_packets ← ∅
3: for fwd_rule ∈ sort(switch.fwd_table, key = priority) do
4:     if fwd_rule ∈ to_be_replayed then
5:         while packet_in ← generate_packet_in(fwd_rule) ∈
   generated_packets do
6:             packet_in ← generate_packet_in(fwd_rule)
7:         end while
8:         controller.send(packet_in)
9:         generated_packets ← generated_packets ∪ fwd_rule
10:    end if
11: end for
```

Figure 5: The replay algorithm.

```
1: handleFlowMod(flow_mod, switch)
2: to_be_deleted ← ∅
3: to_be_installed ← ∅
4: if flow_mod ∈ switch.fwd_table then
5:     to_be_replayed ← to_be_replayed − {flow_mod}
6: else
7:     if flow_mod ⊂ switch.fwd_table then
8:         for rule ∈ flow_mod ⊂ switch.fwd_table do
9:             to_be_deleted ← to_be_deleted ∪ {rule}
10:            to_be_replayed ← to_be_replayed − {rule}
11:        end for
12:    else
13:        if switch.fwd_table ⊂ flow_mod then
14:            for rule ∈ switch.fwd_table ⊂ flow_mod do
15:                to_be_deleted ← to_be_deleted ∪ {rule}
16:                to_be_replayed ← to_be_replayed − {rule}
17:            end for
18:        end if
19:    end if
20:    to_be_installed ← to_be_installed ∪ {flow_mod}
21: end if
```

Figure 6: The HandleFlowMod algorithm.

intent. The algorithm distinguishes two cases: when a C' generated forwarding rule corresponds to (i) a subset of existing rule(s) (lines 9–13) and, (ii) a superset of existing rule(s) (lines 15–20). In the first case, the algorithm marks less specific rules for deletion. Indeed, since C' defines more specific behavior, the algorithm needs to make sure that the less specific forwarding rules installed by C are not hiding relevant packets to C'. For instance, consider again sw_1 for which a packet-in event corresponding to $10/8$ is generated during the replay. Without loss of generality, consider that the generated packet is `10.0.0.1`. Upon reception of this packet, C' pushes the $10/9$ rule. Since this rule is a strict subset of $10/8$, HOTSWAP marks the $10/8$ rule as to be deleted. Indeed, if HOTSWAP does not delete that rule, C' will never receive a packet-in event for $10.128/9$ as these packets would match the old rule.

In the second case, C' defines less specific behavior with respect to C. As in the previous case, HOTSWAP marks more specific rules generated by C as to be deleted to ensure that the traffic will be forwarded according to C'. As an example, consider sw_2 and consider that the replay algorithm generates first a packet-in matching $10.128/9$. Upon reception of this packet, C' pushes the $10/8$ rule. HOTSWAP therefore marks both the $10/9$ and $10.128/9$ entries as to be deleted. This prevents the packet-in matching $10/9$ to be generated, which reduces the number of replayed events. In both cases, it marks the rule generated by C' for installation (line 20).

5. EVALUATION

To evaluate our prototype, we used the restart experiment discussed in Section 2. We ran a Floodlight controller using the default forwarding application on top of HOTSWAP. After the network had reached a stable state, we launched a new controller instance and

used HOTSWAP to replay and swap the two controllers at times $t = 15s$ and $t = 20s$, respectively. We configured HOTSWAP to use the replay algorithm of Figure 5, and repeated the experiment 15 times. For each experiment, HOTSWAP managed to swap the controllers without losing a single packet or causing any increase in the network delay (see Figure 1). This clearly contrasts with the disruption that occurred with the naive approach. On average, HOTSWAP took only 375ms to generate and replay all network events and 7ms to perform the controller swap.

Since HOTSWAP's goal is to perform controller upgrade and not only restart, we used it to upgrade a Floodlight controller running v0.85 to v0.90. Again, we did not observe a single packet loss or any increase in the network delay. The network state remained perfectly intact despite the fact that the v0.90 release fixed 37 bugs [3].

Finally, we evaluated HOTSWAP overhead with respect to an unmodified version of FlowVisor by using the `cbench` tool [13] (with the default configuration). When configured to record *every* packet-in, the unoptimized HOTSWAP implementation induced an overhead of approximately 25%. As discussed in Section 3, recording every single packet-in must be considered as a worst-case scenario as most applications only require a subset of the network events to be maintained. In contrast, HOTSWAP did not induce any significant overhead when using the stateless replay algorithm.

6. RELATED WORK

A number of different tools that record network traffic for the purpose of debugging SDN networks have been proposed in recent years. The `OFRewind` system [17] is probably the most closely related to HOTSWAP as it also records and replay (filtered) network events using a hypervisor. However, `OFRewind` is not concerned with recreating state in a correct manner and only provides coarse grained filtering ability. The `ndb` tool [6] provides a trace view of OpenFlow forwarding tables traversed by data packets in the network. Unlike HOTSWAP, `ndb` aims at troubleshooting the data plane, and not at reconstructing the state in the controller. Recent work by Scott et al. [15] aims at finding a minimal causal sequence of events responsible for triggering a bug. While HOTSWAP is concerned about filtering network events, our aim is to find the minimal set of events needed to create a correct controller state.

While distributed controller architectures such as `Onix` [9] and `ONOS` [11] can create new controller instances and synchronize state among them, they do not solve the controller upgrade when the entire architecture must be upgraded (e.g., where the new controller may use different data structures and algorithms than the old one). In contrast, nothing prevents HOTSWAP from bootstrapping a distributed controller.

Finally, there is a large body of work in the programming languages community on techniques for upgrading a software while it is running—see `Kitsune` [7] for a recent example. Applying these techniques to controller upgrade is possible, but would have to be considered on a per controller basis, while HOTSWAP is controller independent.

7. CONCLUSIONS

Dynamic software updates are notoriously difficult to implement correctly, and SDN controllers are no exception. And yet, SDN programmers need mechanisms for gracefully upgrading controller programs. This paper shows how HOTSWAP enables disruption-free software upgrade *in practice*, while ensuring key correctness properties. HOTSWAP is highly general as it does not make any assumption on the network applications or the controller and is easy to use as it only requires minimal input from the SDN programmer.

In ongoing work, we are exploring how HOTSWAP can be applied to contexts in which state synchronization is problematic, including distributed controllers. We also plan to develop a programming API that facilitates upgrades either by making state dependencies more explicit or by supporting run-time code patching.

Acknowledgements. The authors wish to thank the HotSDN referees for their helpful comments. Our work is supported in part by the NSF under grants CNS-1111698, CNS-1111520, CCF-1253165, and CCF-0964409; by the ONR under award N00014-12-1-0757; and by a Google Research Award.

8. REFERENCES

[1] Cisco IOS In Service Software Upgrade. `http://tinyurl.com/acjng7k`.

[2] Floodlight OpenFlow Controller. `http://floodlight.openflowhub.org/`.

[3] Floodlight v0.90 Release Notes. `http://tinyurl.com/aaya7yg`.

[4] Juniper Networks. Unified ISSU Concepts. `http://tinyurl.com/9wbjzhy`.

[5] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. CoNEXT. ACM, 2012.

[6] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *HotSDN*, 2012.

[7] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. OOPSLA, pages 249–264. ACM, 2012.

[8] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI'12*, Berkeley, CA, USA, 2012. USENIX Association.

[9] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*. USENIX Association, Oct. 2010.

[10] J. Moy, P. Pillay-Esnault, and A. Lindem. Graceful OSPF Restart. RFC 3623, 2003.

[11] On.Lab. ONOS: Open network operating system. `http://tinyurl.com/pjs9eyw`.

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

[13] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for Openflow Switch Evaluation. PAM, 2012.

[14] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter. Graceful Restart Mechanism for BGP. RFC 4724, Jan. 2007.

[15] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker. How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software. Technical Report UCB/EECS-2013-8, EECS Department, University of California, Berkeley, Feb 2013.

[16] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *OSDI*, Oct. 2010.

[17] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical*, June 2011.