Generating Consistent Updates for Software-Defined Network Configurations

Yifei Yuan*, Franjo Ivančić§, Cristian Lumezanu†, Shuyuan Zhang‡, Aarti Gupta†
†NEC Laboratories America *University of Pennsylvania ‡Princeton University §Google, Inc.

ABSTRACT

This paper addresses the problem of consistently updating network configurations in a software-defined network. We are interested in generating an *update sequence ordering* that guarantees per-packet consistency. We present a procedure that computes a safe update sequence by generating an *add-before rule dependency graph*. Nodes in the graph correspond to rules to be installed and edges capture dependency relations among them.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*

Keywords

Consistent network update, software-defined networking, add-before dependency graph

1. INTRODUCTION

Per-Packet Update Consistency. The notion of *per-packet update consistency* was introduced in [5]: A sequence of rule updates should be performed in such a way that any packet that traverses the network follows either the old configuration from network entry point (ingress) to network exit point (egress) or the new configuration. This requirement also applies to any in-flight packet when the network update is started, as well as any new packet that arrives during the network update process.

Two-phase commit protocol (2PC) [5, 4] was the first solution for per-packet consistent update. It uses a special-purpose timestamp tag to differentiate old and new rules. Katta et al. propose an incremental update version of 2PC [1]. McGeer has proposed a safe update protocol for SDNs [3], where all packets that are affected by the update are sent to the controller.

Proposed Approach. We propose a novel approach, where we identify dependency relations between rules on a single switch and between different switches, and build a graph of such dependencies. Each node in this graph corresponds to a rule to be installed. The meaning of an edge between a source node and a target node is that rule r_1 on switch s_1 corresponding to the source node should be installed before rule r_2 on switch s_2 corresponding to the target node. The topological ordering of the graph corresponds to a safe update sequence. If there is a cycle in the rule dependency graph, there may exist a potential inconsistency, i.e., we cannot find a safe ordering of rule updates. Our approach incorporates the possibility of using different methods to deal with the cyclic graph, e.g. using timestamp tags. We propose a simple solution where we break cycles by removing some nodes' outgoing edges from the graph. Removing a node's outgoing edges implies that we initially avoid applying its rule update. Instead, we install a temporary rule to forward the associated traffic to the controller. Next, we install rules according to a topological sort on the remaining cycle-free graph. After all rules associated with incoming edges of a removed node have been installed, we can install the original rule safely on the switch, and stop forwarding additional traffic to the controller. We leave other solutions to future research.

HotSDN'14, August 22, 2014, Chicago, IL, USA.
ACM 978-1-4503-2989-7/14/08.
http://dx.doi.org/10.1145/2620728.2620774.

2. GENERATING CONSISTENT UPDATES

We define a rule $r = (\mu, \alpha, \rho)$ to have 3 components: μ is the set of packets that match rule r (the *flow-space* of a rule), α is the action performed on matched packets (including *drop* and *forward* to a port), and ρ is the priority of the rule. We write $\mu(r)$ to denote the flow-space of r, $\alpha(r)$ to denote its action, and $\rho(r)$ to denote its priority. Consider a network N consisting of a set of switches $s_1, .., s_n$, a configuration $C = \{T_i : i = 1, .., n\}$ of N is the set of flow tables for every switch, where T_i denotes a flow table. For a packet p, we use $C(p)$ to denote the sequence of switch-rule pairs $[(s_{i_1}, r_{i_1}), .., (s_{i_k}, r_{i_k})]$ such that $[s_{i_1}, .., s_{i_k}]$ is the sequence of switches that p traverses under C, and r_{i_j} is the rule that processes p on switch s_{i_j}.

Given a network N, an old configuration $C^O = \{T_i^O : i = 1, .., n\}$ and a new configuration $C^N = \{T_i^N : i = 1, .., n\}$, we first compute an *add-before rule dependency graph* that captures the dependency relations between new rules. Second, we propose a generic framework to break cycles if the add-before dependency graph is cyclic. Without loss of generality, we assume that for every switch, each rule in its new flow table has a higher priority than every rule in the old flow table.

2.1 Add-Before Rule Dependency Graph

Algorithm 1 builds an add-before rule dependency graph. Instead of reasoning about each individual packet, we consider equivalence flow-spaces. A set of packets f is an *equivalence flow-space* under a configuration C, if under the configuration C, all packets in f traverse the network following the same path, and every switch on the path processes the packets using the same rule. We generalize the notation $C(p)$ to $C(f)$ in a natural way. Let E_C be the set

of all equivalence flow-spaces under C, and $E_{C^O \cap C^N}$ denote the set of equivalence flow-spaces under both C^O and C^N[1]. To build the dependency graph, we add three types of dependency edges between rules, by identifying situations where some packet would violate per-packet consistency in some transient configuration.

Algorithm 1 $build\_dependency(C^O, C^N)$

1: **for all** $f \in E_{C^O \cap C^N}$ **do**
2: let $C^O(f) = [(s_{i_1}, r^O_{i_1}), .., (s_{i_m}, r^O_{i_m})]$
3: let $C^N(f) = [(s_{i'_1}, r^N_{i'_1}), .., (s_{i'_l}, r^N_{i'_m})]$
4: let j be the first index such that $\alpha(r^O_{i_j}) \neq \alpha(r^N_{i'_j})$
5: $add\_intra\_dependency(f)$
6: $add\_fwd\_dependency(f)$
7: $add\_bwd\_dependency(f)$
8: **end for**

Intra-switch dependency. Algorithm 2 shows the addition of the first type of edges between two rules on the same switch.

Algorithm 2 $add\_intra\_dependency(f)$

1: **for all** $k = 1, .., j$ **do**
2: **for all** r'' in $T^N_{i_k}$ and $\rho(r'') < \rho(r^N_{i_k})$ **do**
3: **if** $\mu(r'') \cap f \neq \phi$ and $\alpha(r^N_{i_k}) \neq \alpha(r'')$ and $\alpha(r'') \neq \alpha(r^O_{i_k})$ **then**
4: add edge $(r^N_{i_k}, r'')$, if no such edge yet
5: **end if**
6: **end for**
7: **end for**

Given an equivalence flow-space f as input, algorithm 2 considers the first j switches. Note that by the definition of j in algorithm 1, the $C^O(f)$ and $C^N(f)$ share the first j switches. For each new table $T^N_{i_k}$ in the first j switches, we add an edge from $r^N_{i_k}$ to any lower-priority rules r'', if r'' overlaps with f and it takes a different action from $r^N_{i_k}$ and $r^O_{i_k}$.

Intra-switch dependency guarantees that if the first j switches that the equivalence flow-space$f$ traverses in both the old and new configurations are the same, then any packet in f traverses those j switches in any transient configuration.

Forward dependency. Algorithm 3 adds the second type of edges, originating from one rule of an upstream switch on an old path (i.e. forwarding path under the old configuration) to another rule of a downstream switch on that path. Given the equivalence flow-space$f$ and the switch s_{i_j} which is the last switch that the old path and new path (i.e. forwarding path under the new configuration) for f share, the algorithm enumerates all new rules r' for each switch s_{i_k} downstream to s_{i_j} on the old path for f. It adds an edge from the rule $r^N_{i_j}$(which forwards f to the new path) to the rule r' if its action differs from the old rule on s_{i_k} for f and overlaps with f. The forward dependency ensures that if a packet in f starts traversing the network following the old path in some transient configuration, it should complete its traversal using the old path.

Backward dependency. Symmetric to forward dependency, backward dependency ensures that if a packet starts traversing the network following the new path in some transient configuration, it should complete its traversal using the new path. The algorithm for this case is very similar to that of forward dependency, and thus omitted due to space limitations.

[1]One can represent an equivalence flow-space and compute E_C efficiently using standard techniques such as [2]. $E_{C^O \cap C^N} = \{f : f = f^O \cap f^N, \forall f^O \in E_{C^O}, \forall f^N \in E_{C^N}\}$.

Next, we state a soundness result for this procedure when the add-before rule dependency graph is cycle-free.

THEOREM 1. *Updating the rules according to a topological ordering (if one exists) of nodes in the add-before rule dependency graph preserves per-packet consistency.*

Algorithm 3 $add\_fwd\_dependency(f)$

1: **for** $k = j + 1, .., m$ **do**
2: **for all** rule $r' \in T^N_{i_k}$ **do**
3: **if** $f \cap \mu(r') \neq \phi$ and $\alpha(r') \neq \alpha(r^O_{i_k})$ **then**
4: add edge $(r^N_{i_j}, r')$, if no such edge yet
5: **end if**
6: **end for**
7: **end for**

2.2 Handling Cyclic Dependencies

In this subsection, we propose a generic framework to consistently update the network by utilizing the controller when the dependency graph is cyclic. We leave other solutions towards cyclic dependencies, e.g. using timestamp tags, for future research.

Algorithm 4 describes the overall configuration update framework consisting of three phases. First, it picks a subset of nodes S in the dependency graph, such that by removing the set of edges E_S that originate in S, the add-before rule dependency graph becomes a directed acyclic graph G' (line 1). Second, it adds temporary rules corresponding to the nodes in S, such that the matching packets are sent to the controller. That is, for each rule r^N corresponding to a node in S, it installs a rule r' with the match and priority of r^N, and action `Ctrl`. Here, `Ctrl` is an action that sends traffic to the controller. Finally, it updates the network following a topological order of G'. After the new rules of G' are installed, the temporary rules are replaced by the associated new rules, thus yielding the final desired configuration C^N.

Algorithm 4 $update(G = (V, E))$

1: Pick $S \subseteq V$, such that $G' = (V, E \setminus E_S)$ is a directed acyclic graph; //$E_S = \{(u, v) \in E | u \in S\}$
2: **for all** $v \in S$ **do**
3: install $r'_v = (\mu(r_v), \texttt{Ctrl}, \rho(r_v))$ on s_v; //r_v, s_v are rule and switch for v, respectively
4: **end for**
5: install rules using topological order of G';

The following theorem states the correctness of this framework.

THEOREM 2. *Assuming the controller maintains per-packet consistency for packets that are sent to it, updating the rules using Algorithm 4 preserves per-packet consistency.*

3. REFERENCES

[1] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *HotSDN*, 2013.

[2] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static checking for networks. In *NSDI*. USENIX, 2012.

[3] Rick McGeer. A correct, zero-overhead protocol for network updates. In *HotSDN*, 2013.

[4] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[5] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *HotNets*. ACM, 2011.