Generating Consistent Updates for Software-Defined Network Configurations

Yifei Yuan†, Franjo Ivančić†, Cristian Lumezanu†, Shuyuan Zhang†, Aart 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 in-per-packet consistent update. It uses a special-purpose timestamp during the network update is started, as well as any new packet that arrives when any 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.

2. GENERATING CONSISTENT UPDATES
We define a rule \( r = (\mu, \alpha, \rho) \) to have 3 components: \( \mu \) is the set of packets that match rule \( r \) (the flow-space of a rule), \( \alpha \) is the action performed on matched packets (including \emph{drop} and \emph{forward} to a port), and \( \rho \) 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 {\((s_{i_1}, r_{i_1}), \ldots, (s_{i_k}, r_{i_k})\)} such that \([s_{i_1}, \ldots, s_{i_k}]\) 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 \( C^O = \{ T_i^O : i = 1, \ldots, n \} \) and a new configuration \( C^N = \{ T_i^N : i = 1, \ldots, n \} \), 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 \emph{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
of all equivalence flow-spaces under \( C \), and \( E_{C,O,T,N} \) denote the set of equivalence flow-spaces under both \( C^O \) and \( C^N \). 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 \( f \in E_{C,O,T,N} \) do
2: let \( C^O(f) = \{(s_{i_1}, r^O_{i_1}), \ldots, (s_{i_m}, r^O_{i_m})\} \)
3: let \( C^N(f) = \{(s_{j_1}, r^N_{j_1}), \ldots, (s_{j_m}, r^N_{j_m})\} \)
4: let \( j \) be the first index such that \( \alpha(v^O_j) \neq \alpha(v^O_j) \)
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 \( k = 1, \ldots, j \) do
2: for all \( r^k \) in \( T^N_{i_k} \) and \( r'' \in (r^N_{i_k}) \) do
3: if \( \mu(r') \cap f \neq \emptyset \) and \( \alpha(r^N_{i_k}) \neq \alpha(r'') \) and \( \alpha(r''_{i_k}) \neq \alpha(r^O_{i_k}) \) then
4: add edge \((r^N_{i_k}, r'')\)
5: end if
6: end for
7: end for

Given an equivalence flow-space \( f \) as input, algorithm 2 considers 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 \( r^N_{i_k} \) (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.

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 \( k = j + 1, \ldots, m \) do
2: for all rule \( r'' \in T^N_{i_k} \) do
3: if \( f \cap r'' \neq \emptyset \) and \( \alpha(r'') \neq \alpha(r^O_{i_k}) \) then
4: add edge \((r^N_{i_k}, r'')\)
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 \( r'' \) corresponding to a node in \( S \), it installs a rule \( r'' \) with the match and priority of \( r'' \), 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 \( r^N_v = \langle \mu(r_v), Ctrl, r(v) \rangle \) 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


