Inter-Flow Consistency: A Novel SDN Update Abstraction for Supporting Inter-Flow Constraints

Weijie Liu*, Rakesh B. Bobba†, Sibin Mohan* and Roy H. Campbell*
* University of Illinois at Urbana-Champaign, IL USA, †Oregon State University, OR USA
{wliu43, sibin, rhc}@illinois.edu, rakesh.bobba@oregonstate.edu

***Abstract*—Software Defined Networks (SDNs) have opened up a new era for networking by decoupling the control and data planes. With a centralized controller, the process of updating networks becomes much more convenient when compared to traditional networks. However, even with SDNs, transitional network states during network updates may still cause problems. Such states may result in a breakdown of isolation guarantees or other critical constraints and this could lead to incorrect behavior or even security vulnerabilities.**

In this paper, we propose a novel abstraction for network updates, *inter-flow consistency*, that can account for relationships and constraints among different flows during updates. We present a generic inter-flow consistency constraint, *version isolation*, and a special case, *spatial isolation*. We propose update scheduling algorithms based on dependency graphs and a data structure that captures dependencies among different update operations & network elements. We also implemented a prototype system on a Mininet OpenFlow network and Ryu SDN controller to evaluate our approach. Experimental results show that our approach is able to enforce inter-flow consistency constraints with reasonable overheads and that overheads for version isolation are higher than for spatial isolation. Furthermore, when only spatial isolation constraints are in use, overheads on update times for flows that have no isolation constraints are very small (around 1%).

I. Introduction

Software-Defined Networking (SDN) [17] changed the way we view networking by decoupling the control plane from the data plane. SDNs provide flexible traffic management and fine-grained network monitoring to enable various network applications, *e.g.,* virtual machine migrations [3], traffic engineering [20], access control [19], server load balancing [12], *etc.* Also, network setup, maintenance and updates become much more convenient. However, even with SDNs in place, there is no guarantee of consistency during the network update process which may throw up transient states, where isolation and consistency guarantees may break down, *e.g.,* traffic flows may be processed by a mixture of old and new rules [15], [21].

This is because even though SDN provides a centralized mechanism (network controller) to manage and update the network, the overall system/network is distributed in nature and there is no guarantee that all elements will receive the updates at the same time. Hence, without additional mechanisms in place configuration changes are not synchronous across the network infrastructure. These inconsistencies can not only adversely impact the stability and availability of the network (by causing transient black holes and packet loops) but also its security. For example, incorrect routing of packets during network transitions may create a vulnerability where packets may go around a security middlebox such as a firewall or a network intrusion detection system [21]. This issue becomes even more problematic when considering isolation guarantees for some critical flows (say configuration data or control packets in safety-critical systems).

Prior work [21] proposed two correctness abstractions for network updates in SDNs: *per-packet* and *per-flow* consistency. Per-packet consistency means that each packet in the network will be processed either by the old configuration or the new one but never a mixture of the two. Per-flow consistency generalizes per-packet consistency and guarantees that each flow in the network will be processed by the old configuration or the new one, but not a mixture of the two. This work also proposed mechanisms to implement these update abstractions based on the OpenFlow [17] protocol. There is follow up work that builds upon these two abstractions so that network systems can guarantee update correctness and efficiency for different application scenarios (*e.g.,* [7], [13], [15], [16], [18]).

However, per-packet and per-flow consistency alone are not sufficient for meeting certain security and reliability requirements. Most networks have multiple flows and there are often cases where we would like to *preserve some relationships or constraints across different flows* during network updates to maintain security and reliability – these two abstractions do not consider the relationships across multiple flows. For instance, in power systems, it is often desirable to separate certain critical real-time control flows from engineering ones so that they never share a common link or a common device – this is to increase the reliability of real-time operations. Similarly, it is desirable to isolate two flows from each other if one is a back-up flow for the other – again to increase reliability. As another example, in multi-tenant data centers, if flows belonging to different tenants from competing organizations share the same forwarding paths, a tenant may be able to infer the information of the other with certain metrics, *e.g.,* round-trip time and packet loss rate. Also, a tenant may transmit elephant flows to cause network congestion to slow down the transmission of others. Therefore, it might be necessary to separate flows belonging to different tenants to provide security

isolation required by SLAs.

Furthermore, the correctness of some distributed applications is based on the right ordering of certain network events or relationships among traffic flows. For instance, the correct operation of stateful firewalls depends on relationships across multiple, different flows. In such cases, processing flow updates using only per-packet or per-flow consistency may result in a network state that combines old configurations for some flows and new configurations for others leading to unexpected results such as forwarding loops, packet loss and even incorrect application execution [5].

In this work, we propose a novel update abstraction, *inter-flow consistency* that accounts for relationships and constraints among different traffic flows during network updates and maintains the necessary consistency requirements. To the best of our knowledge, we are the first to study update consistency across different flows, especially those with *inter-flow constraints*, for security and reliability. We present a generic inter-flow consistency constraint, *Version Isolation* and a special case, *Spatial Isolation*[1] (Section II). To achieve these two types of inter-flow consistency constraints during SDN updates, we developed a dynamic update scheduling approach leveraging dependency graphs [7] (Section III). We also implemented a prototype system using the Mininet OpenFlow network [11] and Ryu SDN controller [1] to evaluate the performance of the proposed approach (Section IV).

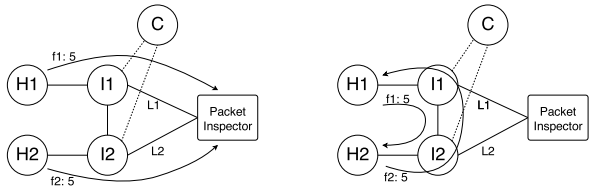
Our evaluation (Section IV) shows that, the overheads incurred are reasonable, less than 30% update time increase for the base case. Furthermore, we are able to provide these improved update abstractions with very low overheads on flows that don't have any constraints. For example, overheads on unconstrained flows in our approach when considering only spatial constraints is less than 1%. Finally, while we primarily motivate the need for such an abstraction through security and reliability requirements, the abstraction and the proposed mechanisms are generally applicable.

II. Inter-Flow Consistency

Inter-flow consistency for updates is a guarantee that specifies inter-flow constraints are preserved during network updates[2]. While there can be many situations that call for inter-flow consistency, we consider those motivated by security and reliability requirements. Specifically, in this work we discuss two forms of inter-flow consistency: *(a) version isolation* and *(b) spatial isolation*. Version isolation is a generic way to think about inter-flow consistency, while spatial isolation is a special case with a more efficient solution.

A. Version Isolation

Version Isolation means that packets from different related flows cannot be processed by two different *versions of flow rules* during their passage through the network. Such a situation may naturally arise because of the asynchronous nature of network updates. That is, updates to some flows among those related flows may complete before updates to the rest. Imagine a scenario with 2 flows, A and B; let the states of Flow A before and after an update be R_{A1} and R_{A2}, respectively. Let the states of Flow B before and after an update as R_{B1} and R_{B2}, respectively. The network can have $R_{A1}R_{B1}$ or $R_{A2}R_{B2}$, but not $R_{A1}R_{B2}$ or $R_{A2}R_{B1}$ at any point in time.



(a) original state (b) target state

Figure 1. An example for a temporal relationship of flows

An example is shown in Figure 1, which is a revised version of a case in [5]. H_1 and H_2 represent two hosts each of which sends out a flow (f_1 and f_2, respectively). Each flow consumes 5 units of bandwidth while the bandwidth of each link is 10 units. There are two ingress switches, I_1 and I_2, with a controller, C. Both of the switches are connected to a server running a packet-inspection application. At first, both of the two hosts send some verification packets to the inspector (shown in Figure 1(a)). After inspection, the application can ask the controller to modify the rules in the 2 switches so that the two hosts can communicate with each other through $I1$ and $I2$ (shown in Figure 1(b)). However, the forwarding rules in the two switches may be not updated at the same time. Imagine that if the rules of f_2 have been updated and f_2 is forwarded to H_1; but the updates of the rules for f_1 have not been finished yet. Receiving packets from H_2, H_1 might think that the packet inspection is completed and then transmits normal application packets other than verification packets to H_2. However, since the rules of f_1 have not been updated yet, these packets will be forwarded to the inspector–an unexpected outcome.

Hence, we need to guarantee that new configurations and old ones should not exist at the same time for the two flows. Note that while this can be rare in practice, an extreme case of version isolation is one where all the flows in the network require version isolation. That is, all the packets in the network should be processed by either the initial flow rule configuration or the target flow rule configuration but not a mix. If it is assumed that the initial and final flow configurations are designed to satisfy necessary inter-flow constraints then in this extreme case version isolation implies inter-flow consistency.

B. Spatial Isolation

Spatial Isolation represents the requirement that certain flows are not allowed to share a link or a switch before, during and after an update for security and/or reliability reasons. For example, if a flow has critical latency requirements (it carries control messages in a power grid substation) then sharing links on its path with another flow that carries, say debugging or engineering traffic, may result in problems for the former flow. For instance, if there is a surge in information being sent over the engineering flow because of say firmware upgrades then we don't want critical control messages suffering delays

[1]We presented early versions of these concepts in a workshop paper [14]

[2]It is assumed that these constraints are satisfied by both the initial configuration and the target configuration

and/or dropped packets. Another scenario that calls for spatial isolation is that of shared data centers where it may be necessary to separate flows belonging to tenants from competing organizations to provide security isolation required by SLAs. Other examples include situations where hackers could try to use information about their own flows (*e.g.,* round-trip time or packet loss rate) to infer details about critical flows. In our work, we assume that the original flow configurations are consistent with spatial isolation requirements, and that updated flow configurations will also satisfy them. The problem arises during the *transitional states*. Hence, we need mechanisms to ensure that spatial isolation requirements are satisfied at all points during the update phase.

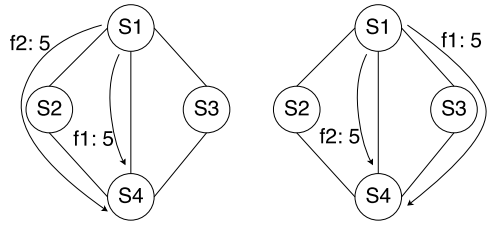


Figure 2. An example for a spatial relationship of flows

Consider the simple example in Figure 2. It shows a network with 4 switches and two flows, f_1 and f_2. Lets assume that flows f_1 and f_2 are not allowed to share a same link; in order words, f_1 and f_2 should demonstrate spatial isolation properties. In this example, each the flow consumes 5 units of bandwidth while the bandwidth of each link is 10 units. Originally, f_1 passes through the link, S_1S_4, and f_2 passes through S_1S_2 and S_2S_4. Now if we are to update the network to a new state (shown in Figure 2(b)) so that f_2 passes through S_1S_4 and f_1 passes through links S_1S_3 and S_3S_4. It is clear that the old as well as the new configurations of this network can guarantee spatial isolation. However, due to the asynchronous nature of flow updates we might have a transitional state where f_2 is updated before f_1. In that case, both the two flows pass through the same link, S_1S_4, for some finite time violating the inter-flow consistency requirements. Thus, update mechanism must guarantee that the spatial relationships (if any) between any two flows is preserved during the update process.

III. Approach

A. Dependency Graph

We use dependency graphs (DGs) [7] to model network update operations. A DG represents update operations, resources and paths as well as the dependencies among them. In the original version [7] (Figure 3(a)), there are 3 types of nodes: *resource*, *operation* and *path*. Resource nodes (rectangles) represent quantities such as link capacity and memory space of a switch. The number in the rectangle represents the amount of available resources of that node. Operation nodes (circles) represent addition, deletion, or modification of a forwarding rule. Path nodes (triangles) represent a group of operations and resources related to a certain path.

A DG has directed edges of different types. The edges operation nodes A and B means that the latter can not be scheduled before A completes. Edges between resource nodes and operation nodes represent a *resource dependency*. An edge from a resource node to an operation node represents the number of resources required for that operation. Edges from an operation node to a resource node represents the amount of resource that will be freed by that operation. Similarly, an edge from a path node to a resource node represents a certain amount of resource will be freed by the deletion of that path. An edge from a resource node to a path node shows how the addition of that path will consume a certain amount of resource. The edges between operation nodes and path nodes represent the proper update ordering. An edge from a path node to an operation node represents that operation cannot be scheduled before removing that path. An edge from an operation node to a path node represents a path cannot be used until that operation completes.

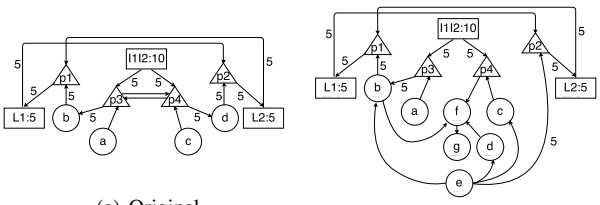
B. Enforcing Inter-flow Consistency

1) Version Isolation: We adopt a forward-to-controller update approach [16] to deal with version isolation constraints. Imagine a situation where we have a set of flows that require version isolation. We call this set a *Version Isolation Set*. The forward-to-controller approach is: *(a)* we pick one flow, say f_0, among those in the version isolation set, and forward the others in that set to the controller for storage; *(b)* update the flow rules of all the flows – all the flows in version isolation set except f_0 are forwarded to the controller; *(c)* once all updates are complete the controller transmits the cached packets into the network. The intuition is simple; by forwarding flows with version isolation constraints to the controller for caching while flow rules are updated, we avoid the situation where flows encounter different versions of rules. Picking and forwarding one flow among the constrained set of flows during the update process while the rest are cached is just an optimization as will become evident later. When the packets buffered in the controller are transmitted back into the network, all of the flows will be now processed with the new rules.

Consider the example in Figure 1 for generating a DG with version isolation. First, we calculate the update operations shown in Table I by comparing the old and new network states. Then we generate the DG in Figure 3(a). The two arrows between p_3 and p_4 means that the two flows must be updated with version isolation. But we fail to find a topological order of the operations in Figure 3(a) because of a loop. Based on the forward-to-controller approach, we first transmit packets of f_2 to the controller and then update the rules of f_1 and f_2. The controller then transmits the traffic of f_2 back to I_2. Even though f_1 gets updated before f_2, there is no f_2 packet processed by the old rules while f_1 is being updated[3]. e and g represent the action of sending f_2 to controller and transmitting it back to the network, respectively.

Figure 3(b) shows a revised DG. A new operation, e, is added with edges from it to the other operations related to the two

[3]**Note:** One can envision a stronger notion of version isolation that requires that all flows in the constrained set be processed together but we leave this for future work.



(a) Original

(b) Revised

Figure 3. Dependency Graph from Figure 1

paths. It means that the system should first forward f_2 to the controller before the rules are updated. This can prevent any packet loss due to updates. The other new operations, f and g, are added. The directions are from b, d and p_4 to f, which means that only after new rules are installed can we replay the packets of f_2 back to our network. One valid sequence of update operations is $e \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow f \rightarrow g$.

Table I. UPDATE OPERATIONS FOR FIGURE 1

ID	Entity	Update Operation
a	I_2	Add: forward f_1 to H_2
b	I_1	Modify: forward f_1 to I_2
c	I_1	Add: forward f_2 to H_1
d	I_2	Modify: forward f_2 to I_1
e	I_2	Modify: forward f_2 to C
f	I_2	Delete the rule of forwarding f_2 to C
g	C	forward all the cached traffic of f_2 to I_1

2) Flow Classification: Imagine a situation where we have multiple version isolation sets, each of which contains different number of flows. We should carefully schedule the ordering of flows to be updated. For example, an input with two version isolation sets, $V_1 = \{f_1,\ f_2\}$ and $V_2 = \{f_1,\ f_3\}$ means that, at any point of time, f_1 and f_2 are allowed to be both processed by the old forwarding rules, or both processed by the new ones but not mixed. The constraint between f_1 and f_3 is similar. However, since f_2 and f_3 are not in the same set, there is no version isolation constraint between them. Thus, we can find a valid update sequence that minimizes the number of flows that are forwarded to the controller: *(i)* forward f_1 to the controller; *(ii)* update all the flow rules and *(iii)* send buffered packets of f_1 back to the network.

We can formalize this a classification problem. Let $F = \{f_1,\ f_2,\ f_3, \dots f_n\}$ be the set of all flows in the network where f_i $(i = 1, \dots n)$ denotes an individual flow. Let V_j for $j = 1, \dots m$ denote m version isolation sets defined for F. Then $V_j \subseteq F$ and $V_j \neq \emptyset$, $\forall j = 1, \dots m$. The problem is to classify the n flows into two classes, *Class I* and *Class II*. Let c_1 and c_2 denote the flows in sets *Class I* and *Class II* respectively. c_1 represents the flows updated without being forwarded to the controller. c_2 represents the flows which should be forwarded to the controller. For version isolation, we should guarantee that if $|c_1| > 1$, any two flows in c_1 are not in the same V_j where $j = 1 \dots m$. To minimize the amount of traffic forwarded to the controller, the solution to this classification problem must minimize the sum of the rate of the flows in c_2.

We design a greedy algorithm (shown in Algorithm 1) to solve this classification problem. First, $c_1 = \emptyset$ and $c_2 = \emptyset$. Put all the flows which are not included in any V_j into c_1. Then $c_2 = F - c_1$. Second, we define the penalty factor, p_i, of f_i:

$$p_i = \sum r_k - r_i \tag{1}$$

Algorithm 1 ClassifyFlows(F, $V_j,\ j = 1, 2, ..., m$)

Require: F: a set of all the flows
Require: $V_j,\ j = 1, 2, ..., m$: m sets of flows with version isolation
1: $c_1 \leftarrow$ all the flows in F but not in $V_j,\ j = 1, 2, ..., m$
2: $c_2 \leftarrow F - c_1$
3: Sort flows in c_2 in penalty factor's ascending order
4: **for** each flow, f, in c_2 **do**
5: $\quad shouldMoveFlag \leftarrow TRUE$
6: $\quad$ **for** each flow, f', in c_1 **do**
7: $\quad\quad$ **if** f' and f are in the same V_j **then**
8: $\quad\quad\quad shouldMoveFlag \leftarrow FALSE$
9: $\quad\quad\quad$ break
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: $\quad$ **if** $shouldMoveFlag == TRUE$ **then**
13: $\quad\quad c_2 \leftarrow c_2 - f$
14: $\quad\quad c_1 \leftarrow c_1 + f$
15: $\quad$ **end if**
16: **end for**
17: **return** c_1, c_2

where r_i is the rate of f_i, and r_k is the rate of f_k such that $f_i, f_k \in V_j$ for any $j \in [1, m]$. The intuition is that if we classify f_i in *Class I*, we will increase the aggregated rate of flows forwarded to the controller by $\sum r_k - r_i$. Then we sort all the flows in c_2 in the ascending order of their penalty factors. Third, for each flow, $f \in c_2$, move it from c_2 to c_1 if and only if there isn't a flow f' in c_1 such that f' and f are in the same V_j for some $j \in [i, m]$. The correctness of Algorithm 1 is shown in Proof of Algorithm 1.

Proof of Algorithm 1: Without loss of generality, say that two flows, f_1 and f_2, have version isolation constraint.

Case I: Both of f_1 and f_2 are classified in c_2, which means that both of the two flows are buffered in the controller and then they will be sent back into the network. According to our schedule algorithm, only after all the update operations complete can the buffered packets be sent back into the network. Thus, after leaving the controller, both of f_1 and f_2 will be processed with new forwarding rules.

Case II: f_1 and f_2 are classified differently. Without loss of generality, assume f_1 classified in c_1 while f_2 in c_2. Before f_2 is sent back to the network, it is uncertain that f_1 is processed by the original configuration or the new. But there are no f_2's packets in the network. Also, it is clear that when buffered packets of f_2 are sent back into the network from the controller, all the updates of f_1 and f_2 are finished. Thus, they will be processed with new forwarding rules.

Last but not least, Algorithm 1 eliminates the possibility that both of f_1 and f_2 are classified in c_1. Thus, our algorithm can guarantee version isolation. ■

3) Spatial Isolation: For spatial isolation relationships, we define a new type of node, a *mutex node* (represented using diamonds) capturing isolation requirements between different flows. This node has a similar function to that of a resource mutex in operating systems, *i.e.,* mutex is *available* only when no one occupies it. A mutex node can be considered as a special resource node. We only have edges between path nodes and mutex nodes. An edge from a mutex node to a path node means that that path cannot be used until the mutex is freed. An edge from a path node to a mutex node means that the mutex will be released after the update of that path.

We can generate the DG shown in Figure 4 for the schedule problem in Figure 2. First, we calculate the update operations shown in Table II by comparing the old network state and the new one. Path Node p_1 represents the path of f_1 before updates while p_2 represent that of f_2 before updates; p_3 and p_4 represent the path of f_1 and f_2 after updates, respectively. There is a common link between p_1 and p_4; thus, there is a mutex node representing the common link, S_1S_4, between p_1 and p_4. The edge from Node p_1 to S_1S_4 means after updates f_1 will not pass through the link, S_1S_4. The edge from Node S_1S_4 to p_4 means that after B's updates f_2 will pass through S_1S_4. In Figure 4, with *topological sorting*, it is clear that a valid order is first to update p_1 and then to update p_4. A valid order of the update operations is $a \rightarrow b \rightarrow c \rightarrow d$. The proof of the correctness of this approach is discussed next.

Table II. UPDATE OPERATIONS FOR FIGURE 2

ID	Entity	Update Operation
a	S_3	Add: forward f_1 to S_4
b	S_1	Modify: forward f_1 to S_3
c	S_1	Modify: forward f_2 to S_4
d	S_2	Delete rules of f_2

Proof of Spatial Isolation: Imagine that f_1 and f_2 are required to be spatially isolated during update process under the assumption that they are spatially isolated both in the original network configuration and the target configuration. But the new path of f_2 and the old path of f_1 have at least one common link or switch. Thus, the valid update ordering should be: first, update f_1 and then update f_2.

Assume that our algorithm outputs an update ordering: first update f_2 and then update f_1. Because a mutex node is a special resource node with the maximal capacity as 1, the remaining capacity of that mutex node before updating f_1 is 0. That means that at least one update operation of f_2 requiring 1 unit of capacity of the mutex is scheduled with zero remaining capacity of the mutex node. This is a contradiction of our algorithm as it only schedules operations with sufficient resources. Thus, the algorithm always updates update f_1 and then update f_2. In other words, f_1 and f_2 are spatially isolated. ■

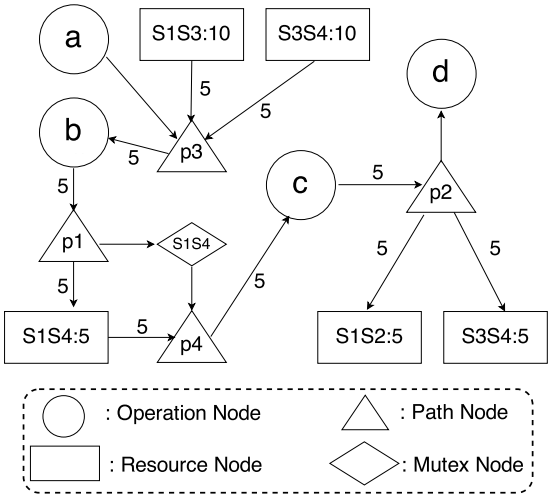


Figure 4. Dependency Graph from Figure 2

C. Dependency Graph Construction Algorithm

Algorithm 2 presents the process of constructing the DG for inter-flow consistency. First, we use Algorithm 1 to classify the flows into two classes, *Class I* and *Class II*. Then different algorithms are utilized to construct different DGs for the two classes. Last but not least, we need to add mutex nodes and some directed edges to represent the isolation constraints.

Algorithm 2 ConstructGraph(F, N_0, N_1, C_s, C_v)

Require: F: the flows whose forwarding rules to be updated
Require: N_0: the original network states of F
Require: N_1: the new network states of F
Require: C_s: constraint set of spatial isolation
Require: C_v: constraint set of version isolation
1: $F_1, F_2 \leftarrow ClassifyFlows(F,\ C_v)$
2: $G_1 \leftarrow ConstructGraphForClass1(F_1,\ N_0,\ N_1)$
3: $G_2 \leftarrow ConstructGraphForClass2(F_2,\ N_0,\ N_1)$
4: $AddVersionIsolationEdges(G_1,\ G_2,\ C_v)$
5: $G \leftarrow G_1 \bigcup G_2$
6: $AddMutexNodes(G,\ C_s)$
7: return G

1) Basic Dependency Graph: Algorithm 3 shows the basic algorithm for DG construction for flows that will not be forwarded to the controller. First, we create resource nodes in the graph. Second, by comparing the old and new paths of each flow, we can calculate the necessary update operations. Then edges are added between two nodes to represent the path-resource relationship or operation-resource relationship. Function $CreateEdges((s_1, d_1), (s_2, d_2), ..., (s_n, d_n))$ creates an edge from each element in s_i to each element in d_i, respectively. A *floodgate* operation is the update operation which will change the forwarding direction of a flow from the old path to the new one. It acts as the water floodgate changing water flows' directions in the real world. Generally speaking, we should first perform the adding operations to establish the new path and then perform the *floodgate* operation. Only after the direction of the flow is changed by the *floodgate* operation, can we delete all the forwarding rules of the old path.

2) Dependency Graph for Inter-flow Constraints: For a flow's spatial isolation, it is clear that we should create mutex nodes between two isolated paths which might occupy the same resource. On the other hand, it is much more tricky to

Algorithm 3 ConstructGraphForClass1(F, N_0, N_1)

Require: F: the flows whose forwarding rules to be updated
Require: N_0: the original network states of F
Require: N_1: the new network states of F
$G \leftarrow \emptyset$
for each link, l, in N_0 and N_1 **do**
  Create a resource node, v, representing l and its remaining capacity
end for
for each f in F **do**
  $p_0 \leftarrow$ the old path of f in N_0
  Create edges from p_0 to each related resource node
  $p_1 \leftarrow$ the new path of f in N_1
  Create edges from each related resource node to p_1
  $o_f \leftarrow$ the *floodgate operation* of p_0 and p_1
  $O_0 \leftarrow$ the operations for removing p_0
  $O_1 \leftarrow$ the operations for creating p_1
  $CreateEdges((o_f, p_0), (p_1, o_f), (p_0, O_0), (O_1, p_1))$
end for
return G

guarantee a flow's version isolation. We first use traditional methods in Algorithm 3 to construct the subgraph for flows in *Class I* that are not forwarded to the controller; then we construct the subgraph for flows in *Class II* with Algorithm 4. Finally, we need to use Algorithm 5 to create edges between flows in *Class I* and *Class II* enforcing version isolation. In Algorithm 4, we add an operation node, o_m, which forwards all the relevant flows to the controller. Edges are created from o_m to other operations of the relevant flows. After all the new forwarding rules are installed, we delete the rules of o_m and then retransmit cached traffic into the network.

Algorithm 4 ConstructGraphForClass2(F, N_0, N_1)

Require: F: the flows in *Class II*
Require: N_0: the original network states of F
Require: N_1: the new network states of F
$G \leftarrow \emptyset$
for each link, l, in N_0 and N_1 **do**
  Create a resource node, v, representing l and its remaining capacity
end for
for each f in F **do**
  $p_0 \leftarrow$ the old path of f in N_0
  Create edges from p_0 to each related resource node
  $p_1 \leftarrow$ the new path of f in N_1
  Create edges from each related resource node to p_1
  $O_f \leftarrow$ the operations including *floodgate operation* of p_0 and p_1, removing p_0, creating p_1
  $o_m \leftarrow$ the operation for forwarding f to the controller
  $o_d \leftarrow$ the operation to delete the rule of forwarding f to the controller
  $o_r \leftarrow$ the operation for the controller to replay f onto the network
  $CreateEdges((o_m, p_0),\ (o_m, O_f),\ (O_f, o_d),\ (p_1, o_d),\ (o_d, o_r))$
end for
return G

Algorithm 5 AddVersionIsolationEdges(G_1, G_2, C)

Require: G_1: the dependency graph for flows in Class 1
Require: G_2: the dependency graph for flows in Class 2
Require: C: version isolated constraints
for each isolation set, s, in C **do**
  for each flow f of Class 2 in s **do**
    for each other flow f_{other} in s **do**
      if f_{other} in Class 1 **then**
        Add an Edge from o_m of f to o_f of f_{other}
        Add an Edge from o_f of f_{other} to o_d of f
      else if f_{other} in Class 2 **then**
        Add an Edge from o_m of f to o_d of f_{other}
      end if
    end for
  end for
end for

D. Scheduling Algorithm

We revise the scheduling algorithm [7] with considerations for inter-flow relationships. Because the DGs represent the dependency relationships between update operations, operations cannot be scheduled until they satisfy two conditions: *(a)* they have no ancestor nodes that are operation nodes and *(b)* the necessary resource are available. With the scheduling algorithm, we can divide the updates schedule into different rounds. In each round, we can schedule several update operations. Another more question is about the update order of the operations within one round. We adopt *critical-path* method [7]. In the DG, we assign weights to the nodes: the weight w of an operation node is 1 while weights of resource, mutex and path nodes are 0. Then we can calculate the CPL for each node i [7] recursively:

$$CPL_i = w_i + \max_{j \in children(i)} CPL_j \tag{2}$$

After sorting the nodes with their $CPLs$ in decreasing order, in each scheduling round, we greedily schedule the operation nodes based on this order. The scheduling algorithm are shown in Algorithm 6. In each round we schedule the operations that have no ancestor operation nodes and can gain enough resources for updates in CPL decreasing order. The correctness of Algorithm 6 is based on the assumption that there is at least one correct order of updates *i.e.,* there are no deadlocks or cycles in the DG. Thus, in each round, we can always schedule some operations and reduce the number of nodes in the graph. The algorithm will not result in infinite loops. After one round of scheduling, we need to update the resource capacities and delete scheduled operation nodes in the DG. Also, we need to wait for a certain time threshold between two rounds for the completion of last round. We use average Round-Trip Time (RTT) as the threshold. The space complexity of our approach is $\mathcal{O}N_o^2$, where N_o is the number of update operations. Because in the some cases multiple operation nodes might be classified in *Class II* and we need to create edges from these nodes to other nodes in *Class I*; thus, the number of edges should be in the order of N_o^2. The time complexity of our approach is also $\mathcal{O}N_o^2$. This is because the running time of Algorithm 1 is $\mathcal{O}N_f \cdot log(N_f) + N_f^2 = \mathcal{O}N_f^2$, where N_f is the amount of

updated flows. The running time of all the other algorithms in our approach is bounded by $\mathcal{O}N_o^2$ because the number of edges being in the order of N_o^2. Also, N_o is in the order of N_f. Thus, the total running time is $\mathcal{O}N_o^2 + N_f^2 = \mathcal{O}N_o^2$.

There are cases where deadlocks happen during the scheduling of a DG, *i.e.,* we fail to find a correct sequence of update operations because cycles exit in the graph. We have two kinds of resource nodes in our DG, one representing link capacity and the other representing mutex for spatial isolation. For a deadlock caused by link capacity we can resolve it by reducing the rate of certain flows [7]. For a deadlock containing mutex nodes and path nodes, we can leverage some classical graph algorithms to detect the directed cycles.

Algorithm 6 ScheduleUpdates(G)

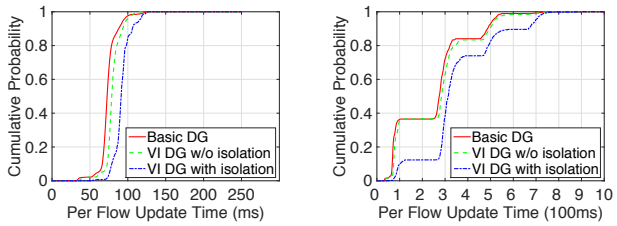
Require: G: the dependency graph
1: Calculate *CPL* for every node in G
2: Sort the operation nodes in the decreasing order of *CPL* and get a sorted order L
3: **while** $G \neq \emptyset$ **do**
4: **for** each operation node $O_i \in L$ **do**
5: **if** O_i has no ancestor operation nodes and can get the necessary resource for updates **then**
6: Schedule O_i
7: **end if**
8: **end for**
9: Delete scheduled operation nodes and corresponding path nodes as well as their edges
10: Delete resource nodes and mutex nodes without edges
11: Update the available amount in resource nodes
12: Wait for a time threshold for all scheduled operations to finish
13: **end while**

IV. Evaluation

A. Experiment Setup

Our prototype was implemented as a layer between SDN applications and the control plane. We used OpenFlow 1.3 and Ryu [1] version 3.20 as the controller. Our system accepts the configurations from the upper network applications and then uses Ryu's APIs to update the network. We also implemented the basic dependency graph (DG) approach in Dionysus [7] as a base line[4]. We used Mininet [11] as our evaluation framework and created a traditional 3-level tree topology to evaluate the performance of our system. There is 1 core switch with 5 aggregation switches linked to it. Each aggregation switch is linked to 5 ToR switches. There are 2 hosts connected to each ToR switch, and each host generates 20 flows with a bandwidth of 0.5 KB/s in each experiment. The number of flows per host is derived from literature [2].

Version isolation (VI) is evaluated for 3 characteristics, *viz.,*update time, number of rules installed, and number of flows forwarded[5] to the controller. These characteristics are studied by varying the following 3 parameters: *(i)* Update Percentage: the percentage of flows to be updated out of total 1000 flows; *(ii)* VI Percentage: the percentage of flows with version isolation out of all the flows to be updated and *(iii)* VI Set Size: the number of flows in a version isolation set. The base values for update percentage and VI percentage are 20% and 30%, respectively. The base case value for version isolation set size is a set, {2, 4, 6, 8}, meaning that the amount of flows in a version isolation set is 2, 4, 6 or 8. In each experiment, we emulate a VM migration scenario and run a shortest path routing application to generate the old and new network configurations. For instance, in the old network configuration, VM A in host 1 communicates with VM B in host 2. After the VM migration, VM B is in another host but with the same IP address and we need to update the forwarding rules in the network to preserve the communication between VM A and B. We randomly generated version isolation and spatial isolation constraints across different flows. In this section, VI is short for version isolation, SI for spatial isolation and DG for the basic DG approach. All of the error bars in this section show standard errors.



(a) VI w/o straggling switches (b) VI with straggling switches

Figure 5. Per Flow Update Time (Update Percentage=20%)

B. Experiment Results

1) Update Percentage in Version Isolation: We changed the update percentage from 10% to 30% to study its effects. In version isolation, there are two categories of flows: *(a)* those not in any version isolation set and *(b)* those with version isolation constraints. Figure 5(a) shows the cumulative distribution of the per flow update time with the update percentage as 20%. Update time of a flow means the elapsed time between the time updates begin and the time that the update operations of that flow are complete. We see that the distribution of update time for flows without version isolation in our approach is very close to that in the basic DG approach. We found that CDF curves for other update percentages exhibit a similar trend.

The average per-flow update time for different flow up-date percentages is shown in Figure 6. The increase in average update time of flows without version isolation constraints when using our approach ranged from 4.4% to 10.6% for the 5 update percentages evaluated. Both the approaches use the same graph structure to schedule the flows without isolation constraints. However, the difference in update time is caused by the flows forwarded to the controller, which slow down the processing of the controller. On the other hand, the increase in average update time of flows with version isolation constraints when using our approach ranged from 15.6% to 29.8%. The reasons for this difference are: (1) flows in *Class*

[4]**Note:** We only used the tunnel network algorithms from Dionysus so our results can't be considered as a full evaluation of the performance of Dionysus.

[5]Since all flows have the same rate in our experiment, number of flows captures bandwidth as well.

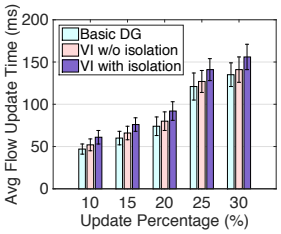


Figure 6. Avg Update Time on Update Percentages

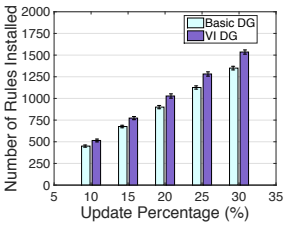


Figure 7. Avg Number of Rules Installed on Update Percentages

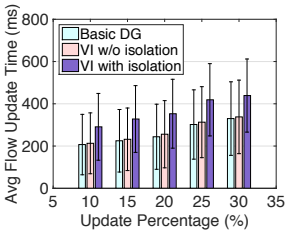


Figure 8. Avg Update Time on Update Percentages with Straggling Switches

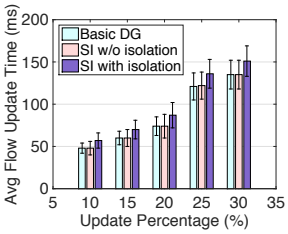
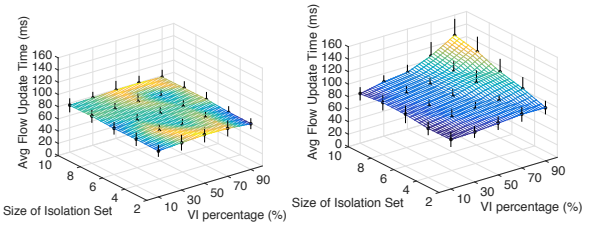


Figure 9. Avg Update Time on Different Update Percentages (Spatial Isolation)

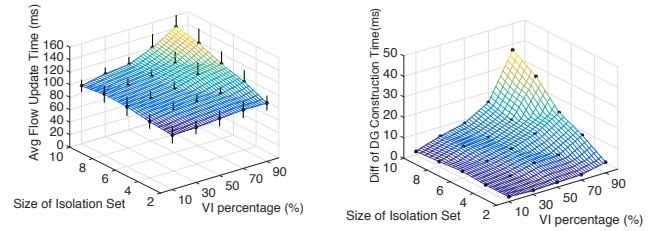
II are forwarded to the controller; and (2) flows in *Class II* have extra operations, namely rules for forwarding to the controller, removing those forwarding-to-controller rules, and re-transmission from the controller. Figure 7 shows that the average numbers of rules installed in our approach are about 14% more than those in basic DG approach.

2) Version Isolation with Straggling Switches: According to [7], the update time of some switches may be 10 to 100 times more than average. We call these switches as "straggling switches". Also, in the study of Google's wide area SDN [6], the 99th percentile of switch update time will be 5 times more than the median. Thus, we conducted experiments with 6 straggling switches (out of total 31 switches) and each straggling switch has 200 ms delay, which means that the elapsed time between the time when a straggling switch receives an OpenFlow update message and the time when the new forwarding rule is installed is 200 ms. From Figure 5(b), we can see that the cumulative distributions of the non-isolated flows in our approach and basic dependency approach are almost the same. That is because the overheads caused by our approach are trivial when compared to the delay time of the switches. The CDF curves are stair-like because the update time is mainly decided by the number of scheduling rounds containing straggling switches. The number of "stairs" of the curves of non-isolated and isolated flows are 3 and 4, respectively, and correspond to the maximum CPL of the two DGs which are exactly 3 and 4.

From Figure 8, we can see that the average update time of non-isolated flows of our approach is very close to that of basic DG. The differences in percentage are 2.9%, 3.1%, 4.9%, 3.6% and 2.4%, respectively. On the other hand, the average update time of the isolated flows is at least 33% more than that of non-isolated flows. This represents the trade-off between version isolation guarantee and update time.



(a) Avg Update, Basic DG (b) Avg Non-isolated Flow Update, VI DG



(c) Avg Isolated Flow Update, VI DG (d) DG Construction Time

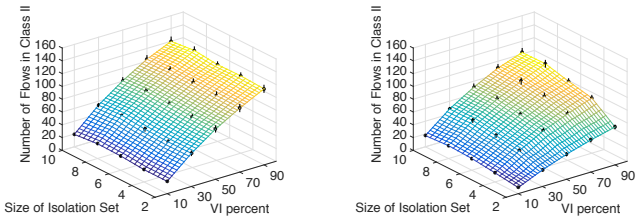
Figure 10. Update Times on Different VI Set Sizes & VI Percentages

3) VI Set Size and VI Percentage: To study the impact of VI Set Size and VI Percentage, with update percentage set at 20%, we change the Set Size from 2 to 10 and Percentage from 10% to 90%. Figure 10(a) shows that the average per-flow update time using basic DG approach doesn't change more than $5.7ms$ as a result, which is only 0.4% of the average value.That is because basic DG doesn't consider version isolation constraints.

Figure 10(b) shows that the average non-isolation flow update time increases as VI Set Size and VI Percentage increase. For example, with VI Set Size as 2 and VI Percentage as 10%, the average non-isolation flow update time of our approach is only 4.9% more than that of the basic DG approach; with VI Set Size as 10 and VI Percentage as 90%, this difference is 63.8% – because larger VI Set Size and larger VI Percentage mean more workload on the controller, which in turn slows down the updates of non-isolation flows. Figure 10(b) and Figure 10(c) share the same increasing trend. We also show the DG construction time difference between our approach and the basic DG approach (Figure 10(d)). The difference grows polynomialy as both flow classification and DG construction are polynomial algorithms. More than 50% of the difference in per-flow update time of non-isolated flows between our approach and the basic DG approach results from the difference in graph construction time.

4) Classification Algorithm: Figure 11 shows the average number of flows in *Class II* (and standard error) when using our greedy algorithm (Figure 11(b)), Algorithm 1, and when using a naive method of forwarding all version isolated flows to the controller (Figure 11(a), as Set Size is varied from 2 to 10 and Percentage from 10% to 90%. In order to prevent the controller becoming the performance bottle neck, we should minimize the volumes of the flows in *Class II*, as those flows will be forwarded to the controller during the updates. Figure 11(a) shows that the number of flows[6] of *Class II* increases as

[6]In our case number of flows represents the volume as all flows have the same rate.



(a) All VI flows to controller (b) Our greedy algorithm

Figure 11. Number of Class II Flows on Different VI Set Sizes and Percentages

the VI Percentage increases and it is clear that the VI set size has no impact.

Comparing Figure 11(a) and Figure 11(b), we can see that Algorithm 1 can effectively reduce the number of flows in *Class II*. Given a VI percentage, with a smaller VI set size, the difference of the two methods is larger. For example, when VI percentage is 50%, compared to the all-to-controller method, Algorithm 1 can reduce 12.4% of the average number of flows of *Class II* with VI set size as 10 and reduce 20.9% with VI set size as 6. That is because with a smaller number of flows in version isolation sets, the probability of having common flows among different sets is smaller; then it is more likely for our classification algorithm to classify some flow in *Class I*.

5) Spatial Isolation: To evaluate performance our approach with spatial isolation constraints, we changed the update percentage in our experiments. As seen in Figure 12, the cumulative distribution of per-flow update times for non-isolated flows using our approach is very close to that of the basic DG approach as there are no flows being forwarded to the controller as in the case of version isolation, which means less overhead on the controller side. The differences in average per-flow update times for non-isolated flows as compared to basic DG approach for various update percentages are all less than 1% as shown in Figure 9.

Furthermore, the overheads on per-flow update times with spatial isolation are lower than with version isolation. Thus while version-isolation may be able to achieve inter-flow consistency during updates in a more generic way (see Section II-A), it is more expensive when compared to enforcing very specific inter-flow consistency constraints such as spatial isolation. Figure 13 shows the number of SI constraints versus number of rules installed in the experiments.

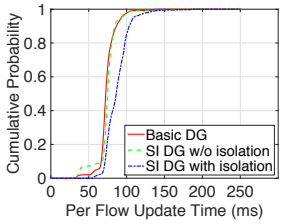


Figure 12. Per Flow Update Time (Update Percentage=20%, SI)

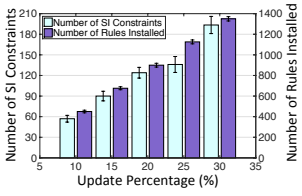


Figure 13. Number of SI Constraints & Rules Installed

V. Related Work

SDN [17] has opened up a new, diverse and exciting area for research. There has been considerable work on verification tools for SDNs (*e.g.,* [9], [10]) but they focus on verifying the final network states and do not consider transitional states.

While Reitblatt *et al.* [21] provided the first formal foundations of per-packet and per-flow consistency. We go beyond and focus on the issue of maintaining consistency across flows that have constraints between them. Mahajan and Wattenhofer [15] highlight the dependency among rules at different switches and develop an algorithm for loop-free guarantees. Jin *et al.* [7] present a solution for dynamic update scheduling and build a system with dependency graphs. Our work is complementary and provides additional guarantees.

Noyes *et al.* [18] proposed a tool for synthesizing network updates automatically while satisfying a specified collection of invariants during the transition. While this approach is more generic and can in theory support a range of inter-flow constraints, it is very expensive – updates synthesis is in the order of ten minutes versus hundred milliseconds in our case. We demonstrate efficient algorithms to handle the update process and also focus on inter-flow constraints. McGeer [16] proposed an update protocol that provides per-packet consistency and a weak form of per-flow consistency with the additional goal of conserving switch rulespace. Our approach for version isolation is very similar to their work albeit with different goals. An important distinction is that we re-inject the original packets back into the network at the source whereas they send the packets directly to the destinations. The latter approach may break application functionality, *e.g.,* security requirements if the flows needed to be routed through a firewall.

Katta *et al.* [8] point out the high space overhead in the two-phase method and propose to divide the update schedule into multiple rounds. Ghorbani and Caesar [4] and zUpdate [13] present update methods under bandwidth constraints. We use similar mechanisms in our work. While Ghorbani and Godfrey [5] point out the insufficiency for per-packet and per-flow update consistency abstractions and argue for newer update abstractions to account for end application level semantics, we have proposed a concrete approach and implemented a system to demonstrate these newer abstractions. Our framework accounts for end application semantics in a generic way. Our version isolation consistency in particular addresses the needs of some of the applications However, the coverage and limitations of the proposed update abstraction in terms of addressing the needs of various application classes needs to be further investigated.

VI. Limitations and Future Directions

One concern with the proposed approach to address version isolation is regarding forwarding the flows to the controller. This will create extra latency in the network that might be unacceptable for some time-sensitive applications like video and game streaming. Also, this results in additional processing and memory overheads at the controller, potentially impacting the performance of its other functions.

As future work, we will extend this forward-to-controller

method to leverage distributed controllers. We can leverage multiple controllers (even the hot standby controllers) to better guarantee inter-flow consistency without impacting the performance of other controller operations. Furthermore, in this work (see Algorithm 1), we only considered flow rates when choosing which flows are to be forwarded to the controller. However, in reality, different flows might have different priorities (that may not correspond to the flow rates). We plan to develop new flow classification algorithms that are able to take into account the priority or criticality of flows. This can provide better Quality of Service (QoS) for latency sensitive applications. We also plan to evaluate the proposed algorithms over a wider parameter range.

VII. Conclusion

By using the concepts presented here, SDNs can guarantee operational consistency even during transitional states – this has huge implications for isolation, criticality and security guarantees which would be broken otherwise. Using the results gleaned from our evaluation, designers of SDNs, especially those with additional constraints (*e.g.,* between groups of flows) can now provision the requisite resources ahead of time – thereby improving the overall design of such systems. We will continue to investigate issues like multiple isolation constraints, coverage, different application classes and other types of inter-flow consistency constraints.

Acknowledgments

The authors would like to thank Rhett Smith, Syed Faisal Hasan and Smruti Padhy for valuable discussions. Thanks are also due to the anonymous reviewers for their valuable feedback. The material in this paper is based upon work supported in part by the Department of Energy[7] under Award Number DE-OE0000679 and by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084.

References

[1] Ryu controller. http://osrg.github.io/ryu/. Accessed: 2014-11-01.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[3] David Erickson, Glen Gibb, Brandon Heller, David Underhill, Jad Naous, Guido Appenzeller, Guru Parulkar, Nick McKeown, Mendel Rosenblum, Monica Lam, et al. A demonstration of virtual machine mobility in an openflow network, 2008.

[4] Soudeh Ghorbani and Matthew Caesar. Walk the line: consistent network updates with bandwidth guarantees. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 67–72. ACM, 2012.

[5] Soudeh Ghorbani and Brighten Godfrey. Towards correct network virtualization. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2014.

[7] Disclaimer: The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

[6] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[7] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 539–550. ACM, 2014.

[8] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 49–54. ACM, 2013.

[9] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[10] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.

[11] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[12] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012.

[13] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 411–422. ACM, 2013.

[14] Weijie Liu, Rakesh B. Bobba, Sibin Mohan, and Roy H. Campbell. Inter-Flow Consistency: Novel SDN Update Abstraction for Supporting Inter-Flow Constraints. In *Proceeding of NDSS Workshop on Security of Emerging Networking Technologies (SENT)*. Internet Society, 2015.

[15] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 20. ACM, 2013.

[16] Rick McGeer. A safe, efficient update protocol for OpenFlow networks. In *Proceedings of HotSDN*, 2012.

[17] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[18] Andrew Noyes, Todd War, Pavol Černý, and Nate Foster. Toward synthesis of network updates. In *Proceedings of Workshop on Synthesis (SYNT)*, July 2013.

[19] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 27–38. ACM, 2013.

[20] Saqib Raza, Yuanbo Zhu, and Chen-Nee Chuah. Graceful network state migrations. *IEEE/ACM Transactions on Networking (TON)*, 19(4):1097–1110, 2011.

[21] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 323–334. ACM, 2012.