© 2015 Weijie Liu

INTER-FLOW CONSISTENCY: NOVEL SDN UPDATE ABSTRACTION FOR SUPPORTING INTER-FLOW CONSTRAINTS

BY

WEIJIE LIU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisers:

Professor Roy H. Campbell
Assistant Professor Rakesh Bobba

ABSTRACT

Software Defined Networks (SDN) are opening a new era in the world of networking by decoupling the data plane and control plane. With the centralized control plane, updating the networks becomes much more convenient to the network operators. However, due to the distributed nature of the data plane, people fail to avoid transitional states of SDN during network updates. The transitional states may be a combination of the old and the new network configurations, which may lead to incorrectness in forwarding behaviors and security vulnerabilities.

This thesis complements the large body of consistent update mechanisms of SDN by proposing a novel network update abstraction, *inter-flow consistency*, which can guarantee certain relationships and constraints among different flows during network updates. To the best of our knowledge, we are the first to study the update consistency abstraction across different flows. We propose an update scheduling algorithm based on dependency graphs, a data structure revealing dependency among different update operations and network elements, in order to guarantee two basic inter-flow consistency, *spatial isolation* and *version isolation*. Also, we implement a prototype system with a Mininet OpenFlow network and Ryu SDN controller to evaluate the performance of our approach.

To my parents and sister, for their love and support.

ACKNOWLEDGMENTS

This thesis would not be possible without the support and help of many people.

First of all, I would like to thank my advisor, Roy H. Campbell, for his support, patience and trust. He offered me with research assistantship during the two years of my master program. He is the most friendly professor I have met in UIUC and always ready to help with my problems. He gave my lots of insight suggestions about how to find my research topic, improve my algorithms and analyze the experiment results.

Second, I would like to thank co-advisor Rakesh Bobba and Sibin Mohan. We work in the same research team for my thesis project. I could always get valuable advice from them when I got stuck in the project. We met nearly every week and they contributed a lot in finding the research topic, designing the approach and conducting the experiments.

Last but not least, I would like to thank my lab mates, Smruti Padhy, Devin Akman, Shane Rogers and Faraz Faghri. They helped me set up the environment for my system development and experiments. Also, I want to Wenxuan Zhou, who is a PhD candidate in UIUC. She gave me a lot of suggestions on how to design and conduct the experiments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Network Update

Networking is the most significant technological foundation for the Internet, mobile computing, and cloud computing. Networks are dynamic. Network infrastructure and data flows are evolving constantly. Network operators need to reconfigure their networks frequently to support dynamic access control, traffic engineering, security updates, and infrastructure maintenance and updates among other things. All of these updates need to be carefully planned and performed in order to minimize transitional network states which might cause disruptions, e.g., broken links, incorrect access control, traffic congestion and packet loops.

The problems of reconfiguring networks is well documented. For example, according to a statistics from a major VPN service provider in [1], 80% of Provider Edge routers' (PE) "failures" are caused by planned network update for maintenance. In a commercial Internet network, 58% of router failures happen in the maintenance window from 10 pm to 6 am and most of these failures are due to network updates for maintenance. According to [2], network operators often need to perform large-scale Interior Gateway Protocols (IGPs) reconfiguration which may result in significant traffic losses and it is very difficult for them to understand the interactions between updated and non-updated devices. Another study in [3] shows that Border Gateway Protocol (BGP) updates account for up to 30% packet-loss for two minutes or more during a routing change. All of these examples indicate the significance of a careful, safe and efficient plan for network updates.

However, it is extremely difficult to guarantee that all these problems mentioned above will not happen during network updates in traditional networks, where data and control are transmitted over a common fabric and control is distributed. Lots of research efforts seek solutions in this area [1, 2, 3, 4]. But

each of these focus on one specific network protocol, e.g., OSPF and BGP, with a limited set of properties, e.g., packet loss, congestion and forwarding loops. Also, all of these solutions are heavyweight and can make network systems much more complicated. Thus, network innovations are required to support a general, safe and efficient framework for network updates.

1.2 Software Defined Networking

Software Defined Networking (SDN) [5], a novel networking architecture, decouples the control plane from the data plane. SDN changes the way we view the networks and brings network operators lots of benefits in building various network applications, *e.g.*, virtual machine migrations [6], traffic engineering [4], access control [7] and server load balancing [8].

The decoupling of the control plane and data plane provides great flexibility to network management and maintenance. For example, in the traditional networking, if a network administrator would like to deploy a new routing protocol in his network, he has to configure his switches or routers one by one. Then the data plane runs the distributed routing algorithm to set up the forwarding state of the network. Now with SDN, he just needs to write a piece of software running on the SDN controller; then the controller will send messages to the devices in the data plane to establish the forwarding routes. This also can avoid the convergence time of the traditional distributed algorithms and increases the service provisioning speed. Second, SDN simplifies the data plane because there is no need for the switches of data plane to support the large set of complicated network protocols. The only two basic requirements for SDN switches are: (1) communication with the controller; (2) forwarding, modifying or dropping packets according to the flow table, an entity in SDN switches that defines the forwarding behaviors of the switches. Moreover, with a standardized interface between control plane and data plane, SDN can enhance network security using promising solutions like network middlebox, network monitoring and network verification [7, 9]. For example, it is very convenient for the SDN controller to redirect traffic to a newly deployed middlebox and to query the flow forwarding rules along with flow statistics of a switch.

SDN has challenges in security and reliability. First of all, the centralized management afforded by the SDN controller is a double-edge sword. The con-

troller is a single point of failure and offers an attack surface [10]. Second, lack of security mechanisms for the communications between the control and data plane might give rise to various attacks, *e.g.*, denial-of-service (DoS) and man-in-the-middle. Using TLS/SSL between the control plane and data plane is an option; but it also has obvious weaknesses, *e.g.*, a self-signed certificate or a compromised certificate authority. Third, SDN control plane provides northbound interface for network applications. However, there is no guarantee that control logic of one application will not contradict the others. Even though each individual control application runs correctly with the enforcement of security policies, the combination of the control logic from multiple applications might cause abnormality, *e.g.*, forwarding loop, traffic black hole and forwarding inconsistency. [11] prioritizes network applications to resolve the contradiction and [12] presents an OpenFlow development framework to provide security modules on which control applications can be built.

OpenFlow [5] is a widely used standard of SDN that defines the functions of the control plane and data plane as well as the interactions between the two planes. In OpenFlow, the switch has one or more flow tables containing flow forwarding rules that describe how to handle certain incoming data flows. Each entry in a flow table or so-called a forwarding rule at least contains three fields: (1) Packet Header, which defines the flow processed with this rule, (2) Action, which defines how to process the packets matching the packet header, and (3) Statistics, which records the statistics of the matching packets, e.g., the number of packets, the bytes of the flow and the time since the last packet matched the packet header. Also, Openflow defines a security communication channel between the controller and the switches. An OpenFlow controller is able to add, remove or modify the entries in flow tables as well as to query the statistics of the switches. A large range of network device manufacturers, e.g., HP, Cisco, Arista and Brocade, have produced switches supporting OpenFlow.

1.3 SDN Update Consistency

Even with SDN in place, there is no guarantee regarding the *correctness* of traffic flows and network state during the update process [13, 14] even when the initial configuration and final configuration are verifiably correct. This is because even though SDN provides a centralized place, namely, the net-

work controller to manage and update the network, the controller itself has to distribute the configuration to switches and routers that are distributed. As such without additional mechanisms configuration changes are not synchronous across the network infrastructure. Lack of consistency during the network update process can not only adversely impact the stability and availability of the network (by causing transient black holes and loops) but also its security. For example, incorrect routing of packets during network transition may cause packets to go around a security middlebox such as a firewall or a network intrusion detection system and create a security hole.

Early work addressing this problem [14] 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 the mixture of the two. This work also proposed mechanisms to implement these update abstractions based on the OpenFlow [5] protocol. Since then a lot of research effort has gone into network systems that can guarantee update correctness for different application scenarios building on these two update abstractions.

1.4 Inter-flow Consistency

In this thesis we argue that per-packet and per-flow consistency alone are not sufficient for meeting requirements imposed by security and reliability. Most networks have multiple flows and there are often cases where we would like to preserve some relationships among the flows during network updates to maintain security and reliability. Per-packet and per-flow consistency abstractions do not account for the relationships between flows and are thus not sufficient to meet such security and reliability requirements. For instance, in control networks of power systems, it is desirable to separate certain critical real-time control flows from engineering flows, *i.e.*, ensure that they never share a link, for 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. As another example, in data centers it may be necessary to separate flows belonging to tenants

from competing organizations to provide security isolation required by SLAs.

Another common scenario is where a (distributed) application imposes a relationship among network flows that needs to be preserved during the update process. Stateful firewalls are one example of application where their correct operation can depend on relationship between flows. In such cases, processing flow updates using per-packet or per-flow consistency may lead to a state of the network that is a combination of old configuration for some flows and new configuration for others leading to unexpected results, *i.e.*, forwarding loops, packet loss and incorrect application execution [15].

In this thesis, we argue that a novel update abstraction, namely, *inter-flow consistency* that accounts for relationships and constraints among different flows is needed to address this problem. We present two special cases of inter-flow consistency, namely *spatial consistency* and *version consistency*, and design algorithms based on dependency graphs of [16] to achieve inter-flow consistency during SDN updates. While we primarily motivate the need for such an abstraction through security and reliability requirements the abstraction and the proposed mechanisms are generally applicable.

CHAPTER 2

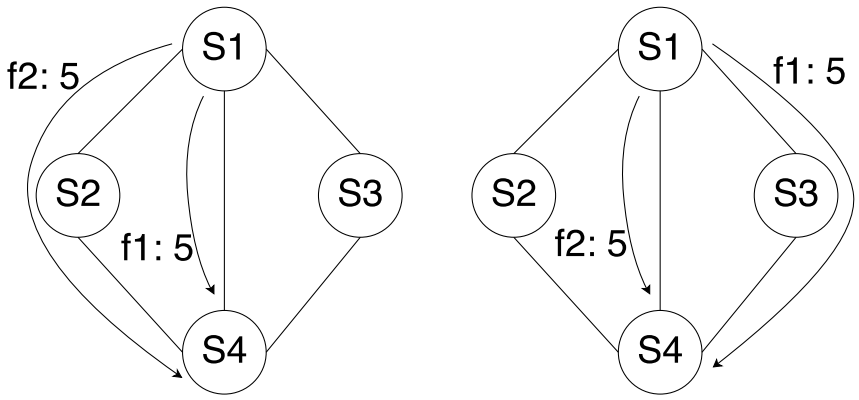
INTER-FLOW CONSISTENCY PROBLEM

Inter-flow consistency for updates is a guarantee that specified inter-flow constraints are preserved during network updates. While there can be many kinds of inter-flow constraints that are needed, we discuss two specific types in this work that are motivated by security and reliability requirements: 1) Spatial Isolation and 2) Version Isolation.

2.1 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 instance, if a flow has certain reliability 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 some firmware upgrades then we don't want critical control messages suffering delays and/or dropped packets. Other examples could include situations where hackers could try to use information about their own flows (*e.g.*, round-trip times or packet loss rates) to infer details about the critical flows. In our work, we assume that the original flow configurations were consistent with these requirements and the new, updates flows will also satisfy them. The problem arises during the *transitional states*. Hence, we need mechanisms to ensure that the spatial isolation requirements are satisfied at all points during the update phase.

Consider the simple example in Figure 2.1. Figure 2.1(a) 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



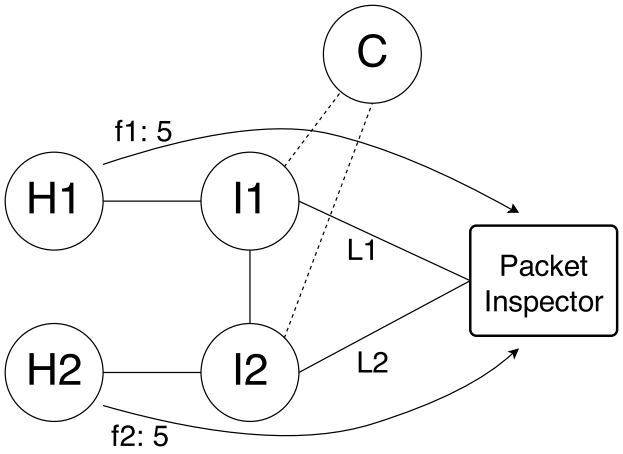
(a) original state (b) target state

Figure 2.1: An example for spatial relationship of flows

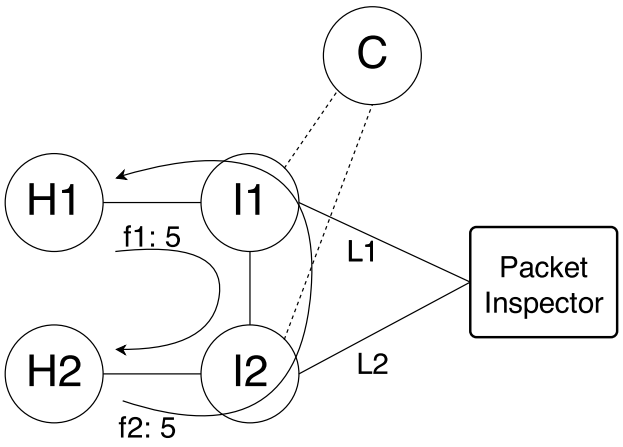
through the link, S_1S_4, and f_2 passes through S_1S_2 and S_2S_4. Now if we are to update the network to a new state (shown in Figure 2.1(b)) so that f_2 passes through S_1S_4 and f_1 passes through links S_1S_3 and S_3S_4. It is clear that the old as well as the new configurations of this network 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.

2.2 Version Isolation

Version isolation means that packets from different related flows cannot be processed by two different *versions of flow rules* during its passage through the network. This can happen because the network updates for certain flows have not been completed before they start routing packets. 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. We refer to this as *version isolation*.



(a) original state



(b) target state

Figure 2.2: An example for temporal relationship of flows

An example is shown in Figure 2.2, which is a revised version of a case in [15]. 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 2.2(a)). After inspection, the application can ask the controller to modify the rules in the two switches so that the two hosts can communicate with each other through $I1$ and $I2$

(shown in Figure 2.2(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, 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.

2.2.1 Assumptions

In this thesis, we assume that there always exists a correct order of update operations. This correct order can preserve bandwidth guarantees, spatial isolation and version isolation during network updates. Admittedly, this assumption may be violated in real world, e.g., by adding elephant flows to a network that already faces significant congestion, deadlocks among different flows – each of which holds a link and waits for others' links. We may integrate existing solution [16] to relax these assumptions but this deadlock-resolving technique is beyond the scope of the thesis.

CHAPTER 3

SYSTEM MODEL AND APPROACH

3.1 Dependency Graph

In order to represent the dependency among update operations and network elements, *e.g.*, link bandwidths and routing paths, we use dependency graphs [16] as the model of the updates. A dependency graph represents update operations, resources and paths as well as the dependencies among them. In the original version of dependency graph [16], there are 3 types of nodes: *resource nodes*, *operation nodes* and *path nodes*. Resource nodes (shown in rectangles) represent the resources in the network, *e.g.*, link capacity and memory space of a switch. The number in a resource node represents the amount of available resources of that node. Operation nodes (shown in circles) represent addition, deletion, or modification of a forwarding rule. Path nodes (shown in triangles) represent a group of operations and resources related to a certain path.

A dependency graph has directed edges of different types. The edges from an operation node A to another operation node B means that B can not be scheduled before A completes. The edges between resource nodes and operation nodes represent a *resource dependency*. An edge from a resource node to an operation node represents that the operation needs this amount of available resource. An edge from an operation node to a resource node represents that this amount of resource 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 represents the addition of that path will consume a certain amount of resource. The edges between operation nodes and path nodes represent the proper schedule order of the two. An edge from a path node to an operation node represents that operation cannot be scheduled until that path is removed. An edge from an operation node to a path node represents a path cannot be used until that operation completes.

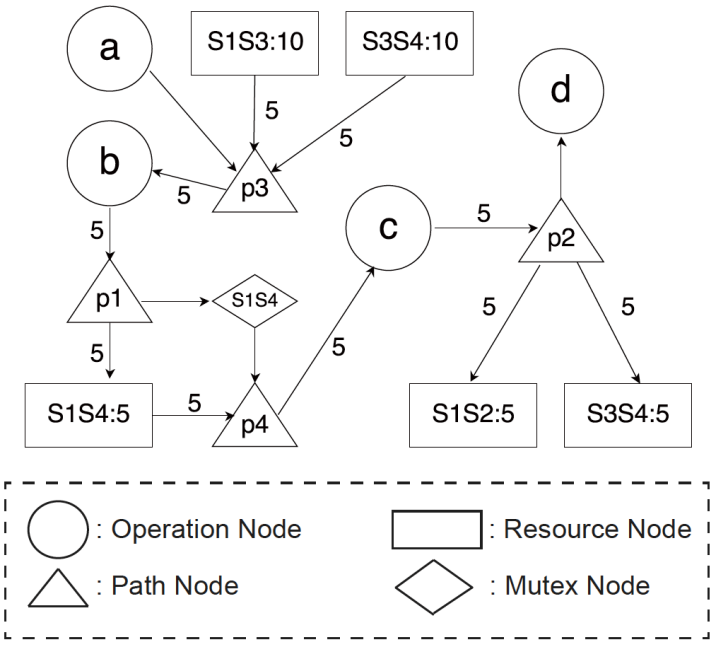


Figure 3.1: Dependency Graph from Figure 2.1

3.2 Approaches for Inter-flow Consistency

3.2.1 Spatial Isolation

In order to represent the flow spatial isolation relationships, we define a new type of node, *mutex nodes* (represented using diamonds) that capture the isolation requirements between different flows. The concept of a mutex is originated from the 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.

For example, We can generate the dependency graph shown in Figure 3.1 for the schedule problem in Figure 2.1. First, we need to calculate the update operations shown in Table 3.1 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 3.1, with *topological sorting*, it is clear that a proper update schedule is first to update p_1 and then to update p_4. A valid order of the update operations is $a \rightarrow b \rightarrow c \rightarrow d$.

Table 3.1: Update Operations for Figure 2.1

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

3.2.2 Version Isolation

We adopt a forward-to-controller update approach [17] to deal with version isolation. Imagine a situation where we have a set of several flows with version isolation requirement. We called this set as Version Isolation Set. The forward-to-controller method goes as follows: first, we pick one flow as f_0 among those with version isolation requirement, and forward the others in that set to the controller for storage. Then we update the flow rules of all the flow; during this time, all the flows except f_0 are forwarded to the controller. After all updates are completed, the controller transmits the cached packets into the network. The intuition behind this approach is that we choose one flow at first and forward all the other flows in the version isolation set to the controller so that the flows with new rules will not be mixed with those with old rules in the network. When the packets buffered in the controller are transmitted back into the network, all of the flows are already processed with the new rules. As an example, we can use the updates in Figure 2.2 to generate a dependency graph shown in Figure 3.2 with version isolation. First, we need to calculate the update operations shown in Table 3.2 by comparing the old network state and the new one. Then we can generate the dependency graph in Figure 3.2(a). The two arrows between p_3 and p_4 means that the corresponding two flows are required to be updated with version isolation. It is clear that we fail to find the topological order of the operations in 3.2(a) because of the existence of a

loop. Based on the forward-to-controller update approach [17], we can first transmit the packets of f_2 to the controller and then update the rules of f_1 and f_2. Then the controller transmits the traffic of f_2 back to I_2. During this process, even though f_1 gets updated before f_2, there is no f_2 packet processed by the old rules while f_1 has been updated. The operations e and g represent the action of sending f_2 to controller and transmitting it back to the network, respectively.

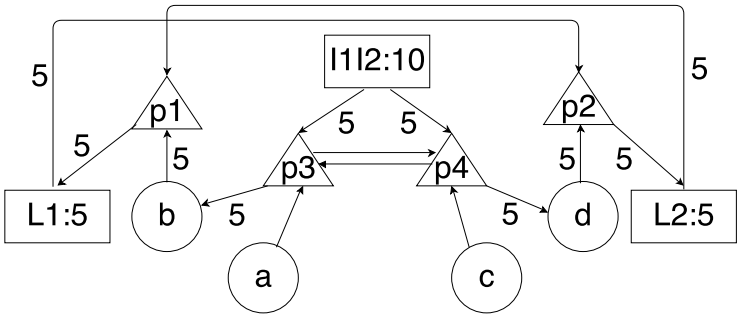
A revised dependency graph is shown in Figure 3.2(b). A new operation, e, is added with many edges from e to the other operations related to the two paths. It means that the system should first forward f_2 to the controller before the rules are updated. This can prevent any packet loss due to updates. 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 of the valid sequence of update operations is $e \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow f \rightarrow g$.

Table 3.2: Update Operations for Figure 2.2

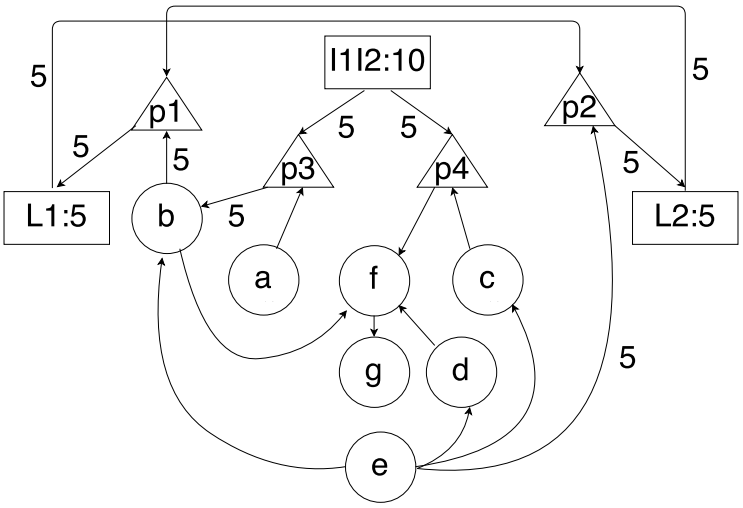
ID	Entity	Update Operation
a	I_2	Add: forward f_1 to H_2
b	I_1	Modify: forward f_1 to I_2
c	I_1	Add: forward f_2 to H_1
d	I_2	Modify: forward f_2 to I_1
e	I_2	Modify: forward f_2 to C
f	I_2	Delete the rule of forwarding f_2 to C
g	C	forward all the cached traffic of f_2 to I_1

3.2.3 Flow Classification

Imagine a situation where we have multiple flow sets, each of which contains multiple flows of version isolation. We should carefully schedule the ordering of flows to be updated. For example, an input with two sets, $F_1 = \{f_1,\ f_2\}$ and $F_2 = \{f_1,\ f_3\}$ 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 the situation where one is processed by the old rules while the other is processed by the new rules is forbidden. The constraint between f_1



(a) original dependency graph



(b) revised dependency graph

Figure 3.2: Dependency Graph from Figure 2.2

and f_3 is similar. However, since f_2 and f_3 are not in the same set, there is no constraint of version isolation between them. Thus, we can find out a valid update sequence: firstly, forward f_1 to the controller; secondly, update all the rules; last, send buffered packets of f_1 back to the network.

We can formalize this problem as a classification problem. Let $F = \{f_1,\ f_2,\ f_3, ...,\ f_n\}$ is the set of all the n flows in the network and f_i denotes one flow. F_j denotes a set of version isolation, where $F_j \subseteq F$ and $F_j \neq \emptyset$. The problem is to classify all the n flows into two classes, *Class I* and *Class II*. Let c_1 and c_2 denote the flow set of *Class I* and *Class II*, respectively. c_1 represents the flows updated without being sent to the controller. c_2 represents the flows which

should be forwarded to the controller and then sent back to the network. For version isolation, we should guarantee that if $|c_1| > 1$, any two flows in c_1 are not in the same F_j. In order to minimize the amount of traffic forwarded to the controller, the goal of our solution to this classification problem is to 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 F_j into c_1. Then $c_2 = F - c_1$. Second, we define the penalty factor, p_i, of f_i:

$$p_i = \sum r_j - r_i \tag{3.1}$$

where r_j is the rate of f_j, a flow which is in the same version isolation set with f_i and we sum the rates of all f_j in any F_j. r_i is the rate of f_i. 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 $\sum r_j - r_i$. 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 F_j.

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

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

The correctness of Algorithm 1 is shown in Proof 3.2.3

Proof. Without loss of generality, imagine that two flows, f_1 and f_2, have the constraint of version isolation.

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. □

3.3 Dependency Graph Construction Algorithm

Algorithm 2 presents the process of constructing the dependency graph 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 dependency graphs for the two classes. Last but not least, we need to add mutex nodes and some directed edges to represent the isolation constraints.

3.3.1 Basic Dependency Graph

Algorithm 3 shows the basic algorithm of dependency graph 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 $CreateEdges((s_1, d_1), (s_2, d_2), ..., (s_n, d_n))$ cre-

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

ates 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.

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
1: $G \leftarrow \emptyset$
2: **for** each link, l, in N_0 and N_1 **do**
3: Create a resource node, v, representing l and its remaining capacity
4: **end for**
5: **for** each f in F **do**
6: $p_0 \leftarrow$ the old path of f in N_0
7: Create edges from p_0 to each related resource node
8: $p_1 \leftarrow$ the new path of f in N_1
9: Create edges from each related resource node to p_1
10: $o_f \leftarrow$ the *floodgate operation* of p_0 and p_1
11: $O_0 \leftarrow$ the operations for removing p_0
12: $O_1 \leftarrow$ the operations for creating p_1
13: $CreateEdges((o_f, p_0), (p_1, o_f), (p_0, O_0), (O_1, p_1))$
14: **end for**
15: **return** G

3.3.2 Dependency Graph for Inter-flow Constraints

For flows' 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 guarantee flows' version isolation. In a nutshell, we first use traditional method in Algorithm 3 to construct the subgraph for flows in *Class I*, which are not forwarded to the controller; then we construct the subgraph for flows in *Class II* with Algorithm 5. Last, we need to use Algorithm 6 to create edges between flows in *Class I* and *Class II* enforcing version isolation. In Algorithm 5, 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 the cached traffic into the network.

Algorithm 4 AddMutexNodes(G, C)

Require: G: the original dependency graph
Require: C: spatial constraints
1: **for** each constraint c in C **do**
2: **for** each common resource, r, shared by the flows specified by c **do**
3: Create one mutex node, r_m
4: Create an edge from the old path node occupying the common resource to r_m
5: Create an edges from r_m to the new path node which will occupy the common resource after updates
6: **end for**
7: **end for**
8: **return** G

3.4 Scheduling Algorithm

We revise the previous scheduling algorithm [16] with considerations for inter-flow relationships. Because the dependency graphs represent the dependency relationships between update operations, the simple idea behind the scheduling algorithm is that the operations cannot be scheduled until they satisfy two conditions: 1) they have no ancestor nodes that are operation nodes, and 2) the necessary resource are available. With the scheduling algorithm, we can

Algorithm 5 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 6 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

divide the updates schedule into different rounds. In each rounds, we can schedule several update operations.

Another more fine-grained question is about the update order of the operations within one round. We adopt the same *critical-path* method as in [16]. In a classical DAG using nodes to represent several jobs in a project, a node's critical-path length (CPL) is the maximal value among the distances from this node to other nodes. In order to minimize the total completion time, we tend to schedule the job with the largest CPL first. In our dependency graph, 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. With this, we can calculate the CPL for each node i [16] recursively:

$$CPL_i = w_i + \max_{j \in children(i)} CPL_j \tag{3.2}$$

After sorting the nodes with their $CPLs$ in decreasing order, in each round of scheduling, we greedily schedule the operation nodes based on this order. The scheduling algorithm are shown in Algorithm 7. We leverage the dependency graph to divide the updates into several rounds; in each round we schedule the available operations in CPL decreasing order. Available operations are operations that have no ancestor operation nodes and can gain enough resources for updates. The correctness of Algorithm 7 is based on the assumption in Section 2.2.1 that there is a correct scheduling order of updates. In other words, there is no deadlock in the dependency graph. 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 capacity and delete scheduled operation nodes in the dependency graph. Also, we need to wait a certain time threshold between two rounds for the completion of the operations in the last round. We use average Round-Trip Time (RTT) as the time threshold in our implementation.

Algorithm 7 ScheduleUpdates(G)

Require: G: the dependency graph

Calculate CPL for every node in G
Sort the operation nodes in the decreasing order of CPL and get a sorted order L
while $G \neq \emptyset$ **do**
  for each operation node $O_i \in L$ **do**
    if O_i has no ancestor operation nodes and can get the necessary resource for updates **then**
      Schedule O_i
    end if
  end for
  Delete scheduled operation nodes and corresponding path nodes as well as their edges
  Delete resource nodes and mutex nodes without edges
  Update the available amount in resource nodes
  Wait for a time threshold for all scheduled operations to finish
end while

CHAPTER 4

SYSTEM IMPLEMENTATION

We implemented a prototype system as the middle layer between SDN applications and the control plane. The system architecture is shown in Figure 4.1. We used OpenFlow 1.3 as the SDN protocol and Ryu [18] version 3.20 as the controller. Ryu is an open-source and component-based controller with lots of well-documented, developer-friendly APIs. Our system accepts the network states from the upper network application and then use Ryu's APIs to add, modify or remove forwarding rules in the data plane. The prototype is implemented with more than 3000 lines of Python code.

There are 6 modules in the system. Update Operation Generator is the module that accepts the input network states from the applications. A network state contains the packet header information defining each flow in the network along with the routing paths of the flows. Update Operation Generator compares the new network state with the existing old network state and then generates the update operations we need to perform, *i.e.*, add, modify or remove a forwarding rule in a switch, in order to change the network from the old state to the new one. After that, it sends all these necessary operations to the Dependency Graph Constructor.

Dependency Graph Constructor (DGC) use Algorithm 2 to construct the dependency graph. It needs 3 kinds of information as input. First, DGC needs the information of all the update operations from Update Operation Generator. Second, in order to correctly generate resource nodes, DGC needs to fetch the topology information from a module called Network Information Base. Last but not least, the user needs to provide scripts showing the constraints across different flows so that DGC can generate a dependency graph to represent these constraints.

With the dependency graph from DGC, Update Scheduler (US) runs Algorithm 7 to perform the update operations round by round. With the signal from US, a Ryu application called Rule Installer calls Ryu's OpenFlow APIs

to send OpenFlow messages to the switches. On top of Ryu, there is a built-in application called "switches" which provides topology information to the Network Information Base.

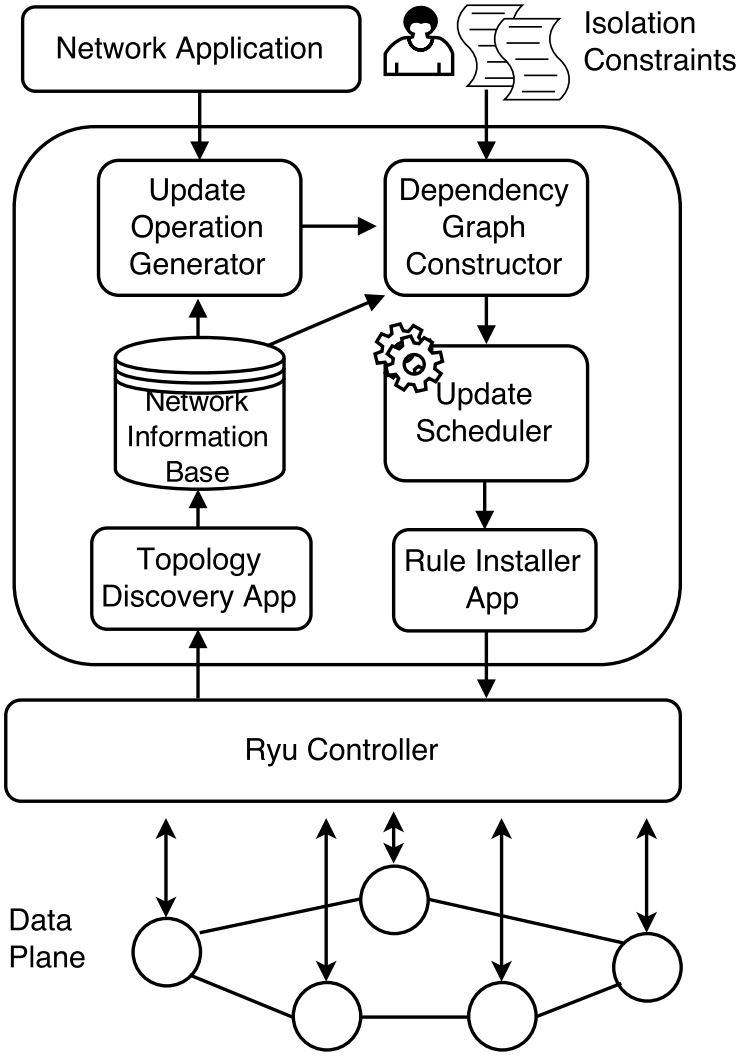


Figure 4.1: System Architecture

CHAPTER 5

EVALUATION

5.1 Experiment Setup

We used Mininet [19] as our evaluation framework and created a traditional 3-level tree topology to evaluate the performance of our system on top of Ryu. 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 connecting to each ToR switch and each host generate 20 flows each with a bandwidth of 0.5 KB/s in each experiment. The number of flows is derived from some studies of data centers [20, 21].

We have a default setting including basic 3 parameters for each experiment and change one parameter each time to study its influence of network updates. The 3 parameters are: (1) Update Percentage, the percentage of flows to be updated out of total 1000 flows; (2) Version Isolation Percentage, the percentage of flows with version isolation out of all the flows to be updated; (3) Version Isolation Set Size, the number of flows in a version isolation set. The default value of update percentage and version isolation percentage are 20% and 30%, respectively. The default value of version isolation set size is a set, $\{2,\ 4,\ 6,\ 8\}$, which means that a version isolation set will be randomly generated with the size of 2, 4, 6 or 8. In each experiment, we emulate the scenarios of the communication in VM migration and run a shortest path routing application to generate the old and new network configuration. 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.

In terms of version isolation, we randomly generated version isolation sets showing some flows should be updated with version isolation. In terms of spatial isolation, we also randomly generated the spatial isolation constraints

among different flows. But in order to generate a more practical scenario, any two flows, f_i and f_j, in spatial isolation sets must meet the following condition: (1) f_i and f_j are spatially isolated in both the old network configuration and the new network configuration; (2) there is at least one common link between the old route of f_i and the new route of f_j, or the old route of f_j and the new route of f_i.

We also implement the basic dependency graph approach in Dionysus [16] as the base line. Please note that we only use the algorithm for tunnel network from Dionysus; thus, our results can't be considered as the full evaluation of the performance of Dionysus. VI is short for version isolation. SI is short for spatial isolation and DG is short for the dependency graph approach.

5.2 Experiment Results

5.2.1 Update Percentage in Version Isolation

Update percentage is the percentage of flows to be updated during one experiments. It directly influences the workload of the network updates. In our experiments, we changed the update percentage from 10% to 30% to study its effect on our system.

Figure 5.1 shows the cumulative distribution of the per flow update time with different update percentages. 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 completely performed. In our approach with version isolation, there are two categories of flows. One category are the flows that are not in any version isolation sets while the other category are those with version isolation constraints. From Figure 5.1, we can see that the distributions of flows without version isolation in our approach are very close to those in the basic dependency graph. From Figure 5.2, the average update time of flows without version isolation constraints in our approach is at most 10.6% and at least 4.4% more than that in basic dependency graph under the 5 update percentages. Two approaches use the same graph structure to schedule the flows without isolation. The difference in update time is caused by the flows forwarded to the controller, which slows down the processing of the controller. On the other hand, the average update time of flows with version isolation

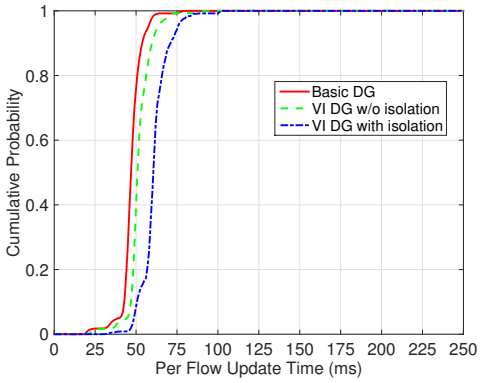
constraints in our approach is at most 29.8% and at least 15.6% more than that in basic dependency graph. The reasons for this difference is: (1) flows in *Class II* are forwarded to the controller; (2) flows in *Class II* have extra operations, namely forwarding to the controller, removing this forwarding-to-controller rule and retransmission from the controller. Figure 5.3 shows that the average numbers of rules installed in our approach are about 14% more than those in basic dependency graph.

5.2.2 Version Isolation with Straggling Switches

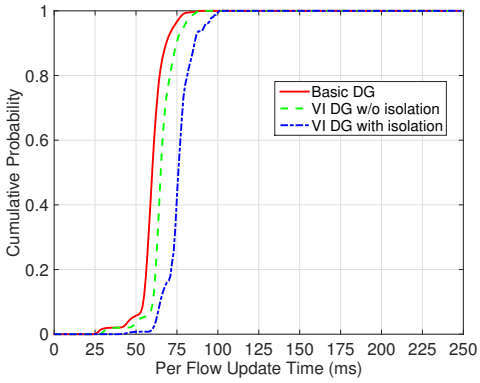
According to [16], the update time of some switches may be 10 to 100 times more than average. We call these switches as "straggling switches". Also, in the study of Google's wide area SDN [22], the 99th percentile of switch update time can be 5 times more than the median. Thus, we conducted experiments 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 5.4, we can see that the cumulative distributions of the non-isolation flows in our approach and basic dependency approach are almost the same. That is because the overheads caused by our approach is 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-isolation and isolation flows are 3 and 4, respectively. Because the maximum CPL in Section 3.4 of the two dependency graphs are exactly 3 and 4, respectively.

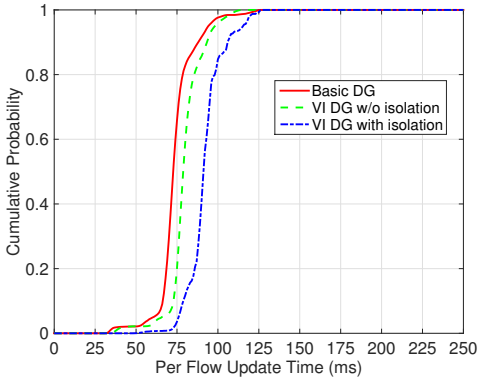
From Figure 5.5, we can see that the average update time of non-isolation flows of our approach is very close to that of basic dependency graph. The difference percentages are 2.9%, 3.1%, 4.9%, 3.6% and 2.4%, respectively. On the other hand, the average update time of the isolation flows is at least 33% more than that of non-isolation flows. This represents the trade-off between version isolation guarantee and update time.



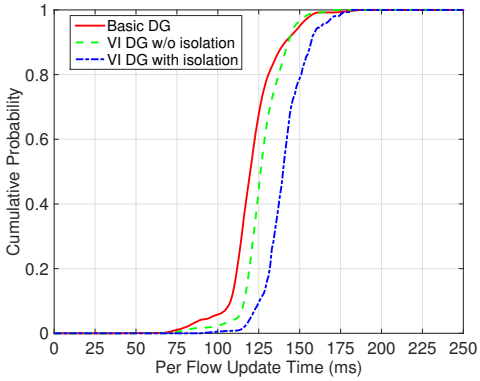
(a) Update Percentage = 10%



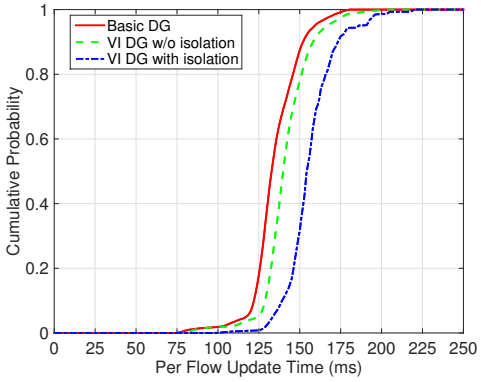
(b) Update Percentage = 15%



(c) Update Percentage = 20%



(d) Update Percentage = 25%



(e) Update Percentage = 30%

Figure 5.1: Cumulative Distribution of Per Flow Update Time with Different Update Percentages

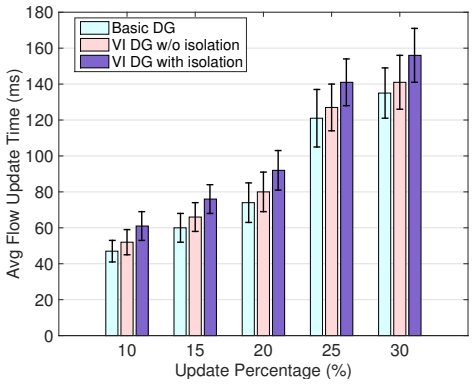


Figure 5.2: Average Update Time (standard error) with Different Update Percentages

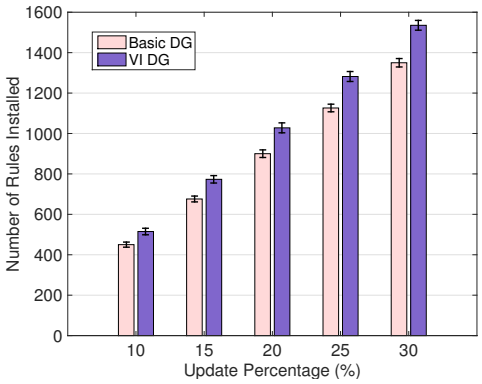
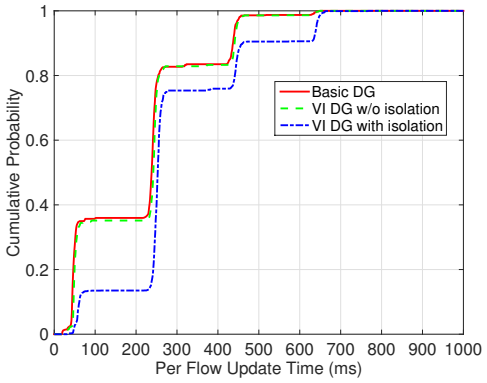


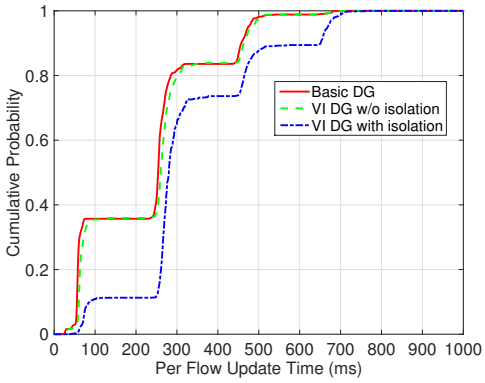
Figure 5.3: Average Number of Rules Installed (standard error) with Different Update Percentages

5.2.3 VI Set Size and VI Percentage

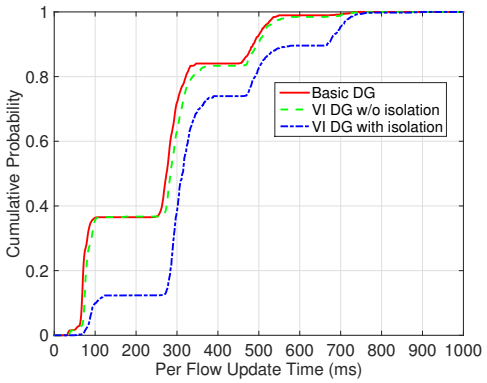
In order to study the impacts of Version Isolation Set Size and Version Isolation Percentage, with default update percentage as 20%, we change Version Isolation Set Size from 2 to 10 and Version Isolation Percentage from 10% to 90%. All the values in Figure 5.7(a), Figure 5.7(b) and Figure 5.7(c) are shown with standard error. Figure 5.7(a) shows that the average per-flow update time of the basic dependency graph approach with different values of the two parameters changes within 5.7 ms, which is only 0.4% of the average of per-flow update time. That is because the basic dependency graph doesn't consider version isolation constraints. Figure 5.7(b) and Figure 5.7(c) share the same trend: per flow update time increases as Version Isolation Set Size and Version Isolation Percentage increase. Figure 5.7(d) shows the difference between the graph construction time of our approach and basic dependency graph approach. 19 out of all the 25 difference values in graph construction time accounts for more than 50% of the differences between the update time per non-isolation flow of our approach and basic dependency graph approach. Figure 5.7(e) represents the difference in numbers of update rules between the two approaches. With a smaller version isolation set size, the increasing speed of the number of extra rules is slower. That is because if the number of flows in the version isolation sets is larger, then the probability of having common flows among different sets is larger; and it is less likely for our classification algorithm to classify some flow as the one not forwarded to the controller.



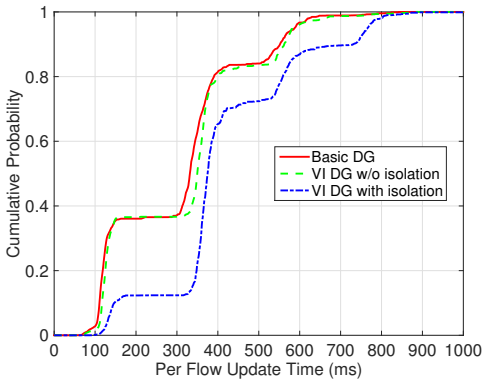
(a) Update Percentage = 10%



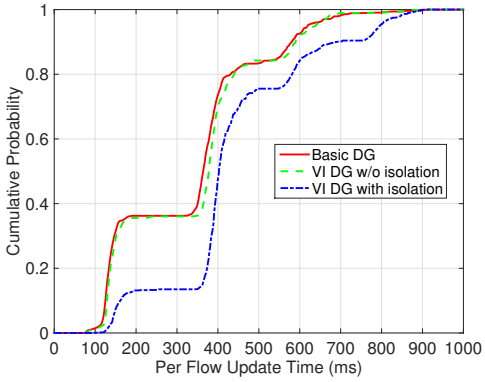
(b) Update Percentage = 15%



(c) Update Percentage = 20%



(d) Update Percentage = 25%



(e) Update Percentage = 30%

Figure 5.4: Cumulative Distribution of Per Flow Update Time with Different Update Percentages and Straggling Switches

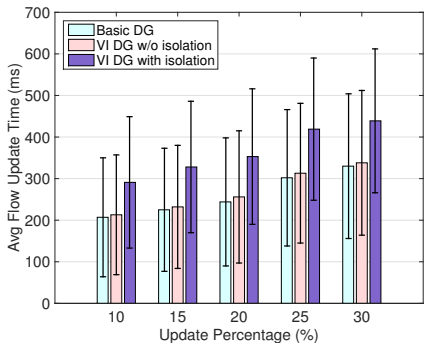


Figure 5.5: Average Update Time (standard error) with Different Update Percentages and Straggling Switches

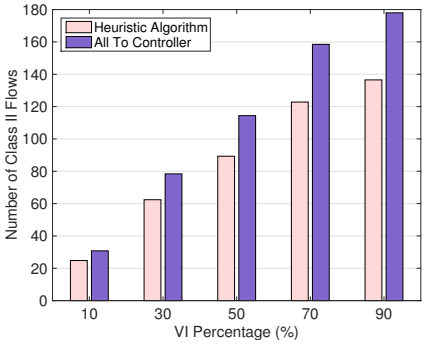


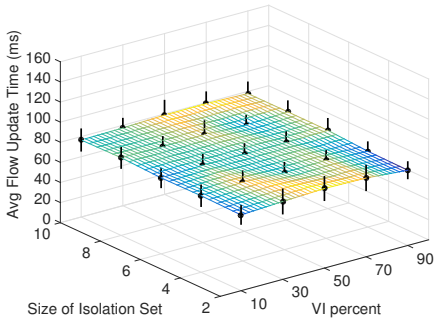
Figure 5.6: Number of Flows of Class II with Heuristic Algorithm and All-to-Controller Method

5.2.4 Classification Algorithm

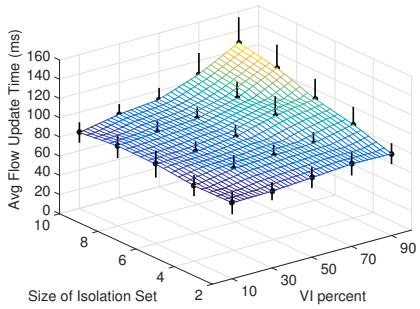
Figure 5.6 shows the number of flows of class II with our heuristic algorithm, Algorithm 1, and the all-to-controller method. The latter is a naive method that forwards all version isolation flows to the controller during a network update. With version isolation percentage in 10%, 30%, 50%, 70% and 90%, the number of flows of Class II with Algorithm 1 is 19.5%, 20.4%, 21.9%, 22.5% and 23.3% less than that with the all-to-controller method, respectively.

5.2.5 Spatial Isolation

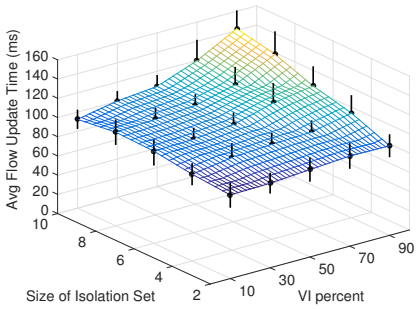
For spatial isolation, there is no version isolation constraints and we change the update percentage in our experiments. In Figure 5.8, we can see that the cumulative distribution curves of non-isolation flows of the two approaches are similar because there is no flow forwarded to the controller, which means zero overhead on the controller side. The differences of average update time for non-isolation flows are all less than 1% shown in Figure 5.9. This indicates that we can utilize alternative approaches (*e.g.*, adding mutex nodes in spatial isolation) instead of the heavyweight forward-to-controller approach to avoid overheads of non-isolation flows. The isolation provided by the lightweight alternative approaches, e.g., spatial isolation in this example, is a weaker variant of version isolation.



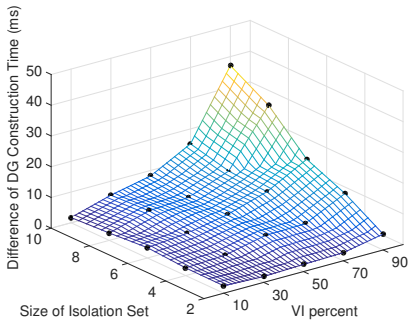
(a) Average Flow Update Time with Basic DG



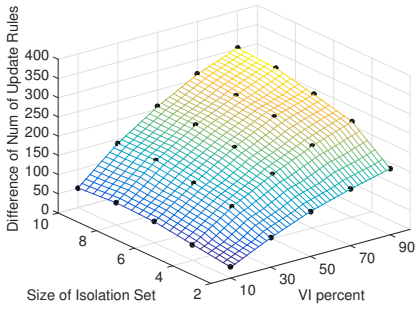
(b) Average Non-isolation Flow Update Time with VI DG



(c) Average Isolation Flow Update Time with VI DG

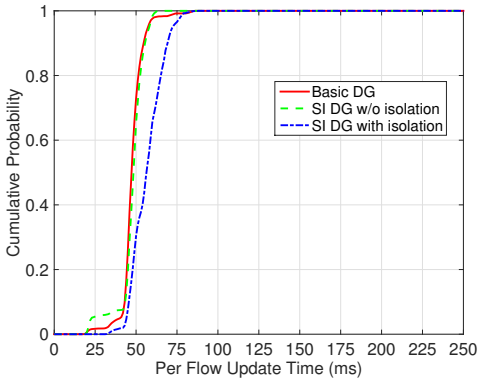


(d) Difference of Construction DG Time

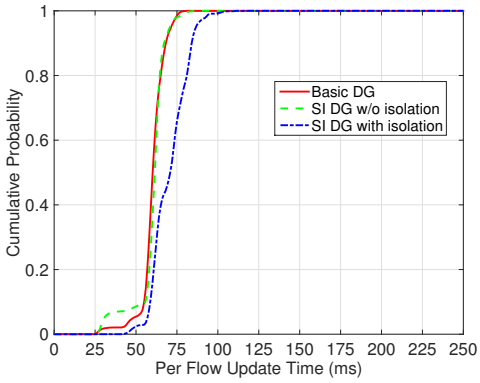


(e) Difference of Number of Update Rules

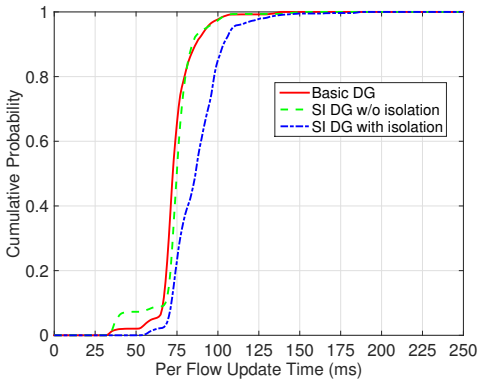
Figure 5.7: Experimental Results with Different Values of VI Set Size and VI Percentage



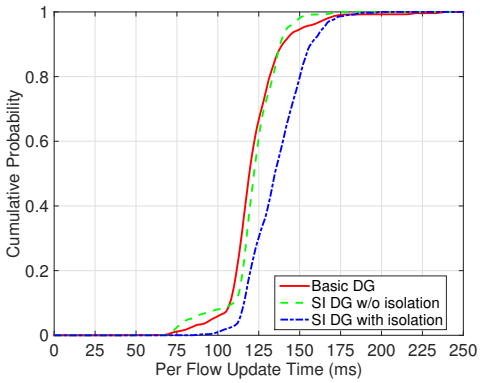
(a) Update Percentage = 10%



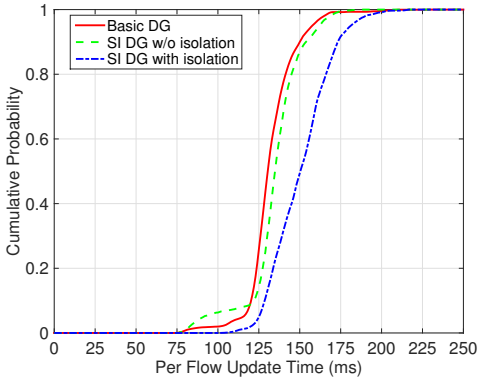
(b) Update Percentage = 15%



(c) Update Percentage = 20%



(d) Update Percentage = 25%



(e) Update Percentage = 30%

Figure 5.8: Cumulative Distribution of Per Flow Update Time with Different Update Percentages and Spatial Isolation

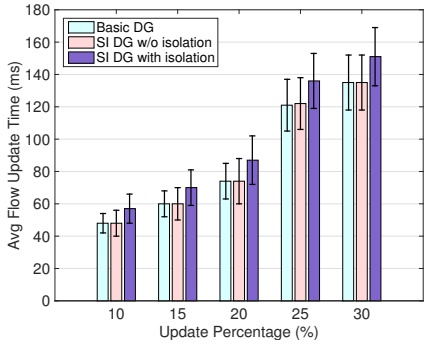


Figure 5.9: Average Update Time (standard error) with Different Update Percentages and Spatial Isolation

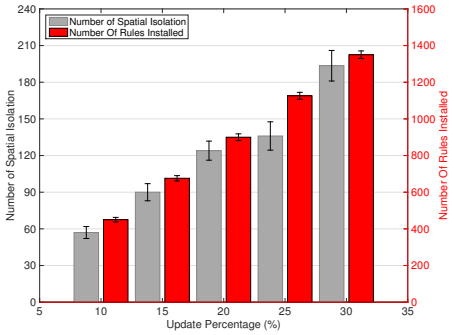


Figure 5.10: Number of Update Operations and Spatial Isolation Pairs

CHAPTER 6

RELATED WORK

6.1 Update Consistency of Traditional Networks

Francois *et al.* [1] uses an encapsulation scheme to reduce the loss of connectivity when network maintenance shuts down the peer links of eBGP. Raza *et al.* [4] introduces Graceful Network State Migration (GNSM) problems, where the goal is to find out a sequence of network update operations progressively changing the initial network configuration to the final network configuration such that the overall performance disruption is minimized. They propose a dynamic programming method to find out the optimal solution of the small-scale GNSM problem and utilize Ants Colony Optimization to solve the large-scale GNSM problem. Vanbever *et al.* [2] studies several update scenarios of the commonly used link-state Interior Gateway Protocols (IGP) and creates an algorithm to find out a strict operation sequence that can avoid IP transit service outages. Francois *et al.* [23] incrementally modify the metric of links to transit the original network configuration to the targeted network configuration so that each step of the updates is loop-free. All of these works focus on a single routing protocol, *e.g.*, BGP and IGP and simple properties, *e.g.*, loop freedom and connectivity. Also, the distributed algorithms in these works are no longer suitable for the centralized control plane in the SDN world.

6.2 Update Consistency Theory of SDN

SDN opens a new era of networking by decoupling the control plane and data plane [5]. With the advent of SDN, many works provide solutions for network verification. NetPlumber in [24] presents an efficient verification tool based on Header Space Analysis (HSA) for incremental compliance checking. Veriflow in [9] also provides fast network-wide invariants checking based on a concept of

equivalence class. However,they focus on verification in the configured network states without considering the transitional states caused by SDN updates.

Reitblatt *et al.* in the seminal work [14] provide the first formal foundation for network updates through the abstractions of per-packet and per-flow consistency in SDNs and propose a two-phase method to enforce such consistency abstractions. It also reveals security vulnerabilities in SDN due to inconsistent updates. The scope of our work is to show that the abstractions proposed in [14] are not sufficient to address many common security and reliability needs. We propose a new complementary abstraction for network updates and our solution naturally guarantees the per-packet consistency.

Noyes *et al.* [25] proposed a tool for synthesizing network updates automatically while satisfying a specified collection of invariants during the transition. The constraints can be specified as linear temporal logic formulas (LTL) formulas. 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. Further, that work does not specifically focus on inter-flow constraints but rather on basic reachability constraints such loop freedom.

McGeer [26] proposed an update protocol that provides per-packet consistency and a weak form of per-flow consistency with the additional goal of conserving switch rule space. Our proposed update approach for version isolation is very similar to the update approach proposed in [26] albeit with very different goals. McGeer's approach is for per-packet consistency and our approach considers the constraints across different flows. Another important distinction is that we re-transmit the original packets back into the network at the source whereas the approach in [26] sends the stored packets directly to the destinations. That approach of sending directly to the destination may break application functionality, especially related to security. For instance, if there was a requirement that certain flows should be routed through a middlebox such as a firewall than this requirement would be violated.

Katta *et al.* [27] point out the high space overhead in the two-phase method and propose to divide the update schedule into multiple rounds. This method trades update time for switch memory space. However, it is just a multi-round version of [14] and doesn't consider other factors such as network link bandwidths and constraints across different flows. Our solution has the same advantage of [27] to divide the overall update operation sequence into several

rounds in order to save the switch memory space. Also, our solution considers the bandwidths and inter-flow constraints.

Mahajan and Wattenhofer [13] highlight the dependency among update operations at different switches and introduces Directed Acyclic Graph (DAG) as the data structure to represent the dependency among update operations. It also develops an algorithm for loop-free guarantees during SDN updates. Its DAG method can be considered as the theoretical foundation of [16] and our solution.

Ghorbani and Godfrey [15] 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. Our proposed inter-flow consistency provides a framework to account for end application semantics in a generic way. Our version isolation consistency in particular addresses the needs for some of the applications identified in [15]. However, the coverage and limitations of the proposed update abstraction in terms of addressing the needs of various application classes need to be further explored.

6.3 Update Consistency of Software Defined DCN and WAN

ZUpdate by Liu *et al.* [28] studies the congestion problem during Date Center Networks (DCNs) updates. It models this problem as a linear programming problem and develops a novel algorithm with the help of a linear programming solver to apply updates in an effective congestion-free manner without any assumptions about the timing and sequence of updates at individual switch. Both of physical switch experiments and large-scale simulations present that zUpdate can effectively perform zero-loss updates of DCNs. However, this mechanism only considers congestion during network updates. It fails to guarantee other significant update properties, *e.g.*, connectivity, loop-freedom and inter-flow constraints.

Hong *et al.* [29] deals with the packet loss problem in current Wide Area Networks (WANs). It develops a system called SWAN that may rate limit the rate of flows in intermediate phase if the paths in the network cannot carry all the traffic. Between consecutive phases, it utilizes linearly programming technique to compute a congestion-free update plan. Both real-world experi-

ments and simulations indicates that SWAN can carry 60% more traffic than the current mechanisms. However, SWAN only considers the static metrics of the networks and doesn't take into consideration the dynamic performance of the switches. Therefore, it fails to effectively handle the practical scenarios with switches of unknown dynamic performance fluctuation.

Jin *et al.* [16] first presents a solution for dynamic update scheduling and builds a concrete system, Dionysus, using a greedy topological-sorting schedule algorithm based on dependency graphs and critical path length (CPL). Our solution is based on Dionysus. Dionysus dynamically calculates the congestion-free update plan and limits the rate of flows when it encounters deadlocks in the topological-sorting algorithm. But Dionysus only considers the per-packet constraints. Our works further extends Dionysus to support update constraints across multiple flows.

CHAPTER 7

CONCLUSION

Network updates may take a network into a transitional state that consists of a mix of old and new network configurations. Such transitional states may result in inconsistency of forwarding behaviors and consequently lead to security vulnerabilities. Decoupling control plane and data plane, Software Defined Networks (SDN) bring great capacities to network operator for more convenient and faster network management. Lots of research effort has gone into devising consistent SDN updates. However, existing network update abstractions for SDNs are not sufficient to meet security and reliability requirements that impose constraints across multiple flows.

To address this, we argue for a novel update abstraction, inter-flow consistency, that accounts for relationships and constraints among flows during network updates. We focus on two specific types of inter-flow consistency, *spatial isolation* and *version isolation*. To enforce the two isolation properties, we build a dynamic update scheduling algorithm with dependency graphs. We use the controller to buffer certain flow packets and create a greedy algorithm to minimize the buffer overheads. A prototype system on top of Ryu controller is developed and a Mininet OpenFlow network is utilized for performance evaluation.

Experimental results show that the update time of non-isolation flows in our approach is similar to that in the basic dependency graph method. Moreover, with straggling switches or spatial isolation constraints, the update time of non-isolation flows in the two approaches are very close. For isolation flows, our approach guarantees their isolation constraints with the price of larger update time and additional forwarding rules installed. Furthermore, we change the number of flows in a version isolation constraint and the percentage of flows with version isolation to study their impacts on update performance. The results of update performance show linear growth over the parameter range tested.

REFERENCES

[1] P. Francois, O. Bonaventure, B. Decraene, and P.-A. Coste, "Avoiding disruptions during maintenance operations on bgp sessions," *Network and Service Management, IEEE Transactions on*, vol. 4, no. 3, pp. 1–11, 2007.

[2] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide igp migrations," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 314–325, 2011.

[3] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed internet routing convergence," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 175–187, 2000.

[4] S. Raza, Y. Zhu, and C.-N. Chuah, "Graceful network state migrations," *IEEE/ACM Transactions on Networking (TON)*, vol. 19, no. 4, pp. 1097–1110, 2011.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[6] D. Erickson, G. Gibb, B. Heller, D. Underhill, J. Naous, G. Appenzeller, G. Parulkar, N. McKeown, M. Rosenblum, M. Lam et al., "A demonstration of virtual machine mobility in an openflow network," 2008.

[7] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simplefying middlebox policy enforcement using sdn," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 27–38.

[8] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 1–6.

[9] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[10] D. Kreutz, F. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 55–60.

[11] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 121–126.

[12] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks." in *NDSS*, 2013.

[13] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 20.

[14] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 323–334.

[15] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2014.

[16] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 539–550.

[17] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker, "Transparent, live migration of a software-defined network," Technical report, CS UIUC, 2013. www. cs. illinois. edu/˜ ghorban2/papers/lime, Tech. Rep.

[18] "Ryu controller," http://osrg.github.io/ryu/, accessed: 2014-11-01.

[19] B. Lantz, B. Heller, and N. 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*. ACM, 2010, p. 19.

[20] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, vol. 10, 2010, pp. 19–19.

[21] "Estimating the number of tcp sessions per host," http://blog.ipspace.net/2013/10/estimating-number-of-tcp-sessions-per.html, accessed: 2015-4-22.

[22] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu et al., "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[23] P. Francois, M. Shand, and O. Bonaventure, "Disruption free topology reconfiguration in ospf networks," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*. IEEE, 2007, pp. 89–97.

[24] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *NSDI*, 2013, pp. 99–111.

[25] A. Noyes, T. War, P. Černý, and N. Foster, "Toward synthesis of network updates." in *Proceedings of Workshop on Synthesis (SYNT)*, July 2013.

[26] R. McGeer, "A safe, efficient update protocol for OpenFlow networks," in *Proceedings of HotSDN*, 2012.

[27] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 49–54.

[28] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: updating data center networks with zero loss," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 411–422.

[29] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 15–26.