Update Algorithms for Software Defined Networks

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

by

Radhika Sukapuram

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
May 2018

To Sajith and Thejas

Thesis submitted on January 5 2018

PhD. viva voce conducted on May 17 2018

© Copyright by Radhika Sukapuram 2018

All Rights Reserved

CERTIFICATE

It is certified that the work contained in the thesis entitled **Update Algorithms for Software Defined Networks** by **Radhika Sukapuram**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Gautam Barua
Professor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Assam, 781039
Email: *gb@iitg.ernet.in*

May 2018

Abstract

In Software Defined Networks, where the network control plane can be programmed by updating switch rules, consistently updating switches is a challenging problem. Updates can cause temporary inconsistencies in the network, leading to packet losses, packet loops, inconsistent application of network policies and corruption of states in stateful nodes, resulting in safety violations, flow failures, measurement errors and lowering of throughput. If a packet (flow) is forwarded either according to the old version of rules or the new version of rules but not a combination of both during an update, the property of Per-Packet Consistency, abbreviated as PPC (Per Flow Consistency, abbreviated as PFC) is preserved. An update that preserves PPC or PFC does not cause temporary inconsistencies. This thesis proposes update algorithms that preserve PPC and PFC.

The ratio of the number of switches where rule updates need to be made to the number of switches actually modified for the update is called Footprint Proportionality (FP). Our solutions progressively increase FP to 1 and the number of concurrent non-conflicting updates to an unlimited value, while always providing an all-or-nothing semantics and supporting wildcarded rules. They work irrespective of the execution speeds of switches or speeds of links, do not require flows in the network for updates to progress and avoid packet buffering. Our PPC algorithm PPCU and PFC algorithm ProFlow use data plane time stamps to decide when switches must move from new to old rules, while accommodating time asynchrony. A proof-of-concept implementation in P4 and Mininet demonstrates the feasibility of the algorithms. In a network with continuous PPCU updates, the throughput and the total number of flows completed are higher compared to a network with continuous random updates, and cause no safety violations. During a ProFlow update, new flows maintain their throughput while old flows undergo a marginal reduction, in comparison with a scenario without an update, where the first affected switch in the flow path has both new and old rules.

Acknowledgements

I would like to acknowledge my PhD. supervisor Prof. Gautam Barua for his guidance throughout the PhD. programme. He gave me the opportunity to work in an area of my choice and a great deal of freedom, yet was always helpful with ideas and suggestions. His clarity of thought, generosity of spirit, sense of technical judgment and patience have contributed a great deal towards this thesis. I am deeply indebted to him.

I would like to thank my doctoral committee members Prof. S.V. Rao, Prof. T. Venkatesh and Prof. Sonali Chouhan for their thoughtful comments, readiness to help and encouragement. I am grateful to my administrative supervisors Prof. Santosh Biswas and Prof. Jatindra Kumar Deka for their support. Thanks are due to Prof. S.V. Rao, Prof. T. Venkatesh, Prof. Hemangee Kapoor, Prof. Sushanta Karmakar and Prof. Arnab Sarkar for re-introducing me to academics through their courses during the programme. I would also like to thank the staff of Computer Science and Engineering Department, especially Mr. Bhriguraj Borah, who was always quick to respond to any requests and was helpful in all interactions.

I am grateful to fellow research scholars Basant Subba, Shounak Chakraborty, Durgesh Kumar and Khushboo Rani for all their help.

I am thankful to my father for his love and encouragement. I would like to thank my mother, who, although no longer with us, continues to inspire by example. I am also grateful to my brother, sister and in-laws for being supportive and understanding. Finally, special thanks to my husband and son, without whose support nothing would have been possible. This thesis is dedicated to them.

Date: _____ Radhika Sukapuram

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis proposes algorithms for consistently updating Software Defined Networks (SDN).

Traditional IP networks, with routers and switches running distributed protocols to control them, are widely and successfully used today. However, managing such networks is a complex task [15]. The interfaces offered by switches and routers are low-level in nature, complex and proprietary, making it difficult for network operators to express high level policies and to configure networks dynamically in response to failures or imbalance of load. Network hardware is complex and hard to manage and therefore creates lock-in with a specific vendor. Innovations in networking are harder because changes to networking equipment take time. Networking equipment is varied, with middle-boxes such as intrusion detectors, firewalls, server load balancers and network address translators, in addition to routers and switches, adding to the complexity [39].

Switches forward incoming packets by matching their headers against rules in a forwarding table and executing the action stored against the matching rule. The *data plane* of a network forwards packets while the *control plane* builds the forwarding table that decides how to forward a packet and the *management plane* remotely monitors and configures the control plane. Traditional networks have the control and data planes integrated, making the network inflexible and reducing the ability for innovation.

1.1 Software Defined Networks

Software Defined Networks (SDN) are characterised by two features: separation of the data and control planes of a network by an open interface and the ability to program the network control plane from a logically centralised controller [115, 72]. Since data and control planes operate at different speeds and need to meet different requirements, their separation allows them to evolve independently, allowing flexibility and room for innovation in network management. The programmatic abstraction provided by the control plane allows applications to be developed using that abstraction, and changed easily, regardless of the network implementation. The logically centralised controller offers a unique vantage point - it has visibility of the entire network and can aid in formulating policies that are beneficial to the entire network. Applications running on the controller program the network control plane by updating the rules in the forwarding tables of switches. A typical SDN application is traffic engineering [128, 56, 2, 60], where, the controller can monitor the network and depending on network conditions, install rules in switches to re-route flows. Kreutz et al. [72] provide a comprehensive survey of Software Defined Networks.

1.1.1 SDN switches

An SDN switch consists of a series of tables, with each table consisting of an ordered set of rules with a match part and an action part. If a packet entering a switch matches the match part of a rule, it executes the action associated with it. This function is called packet classification. An action may be a statement such as "*forward packet to port* 1", "*drop packet*" etc. A series of tables may be chained together.

Hardware based Ternary Content Addressable Memories (TCAM) are the standard devices in network switches for high-speed packet classification. The fixed-width match fields of a TCAM consist of 1, 0 or $*$ (to denote don't cares). Since an incoming packet is matched with many fixed-width match fields in parallel, in hardware, it is possible to achieve high speeds. In general, since TCAM memory space is scarce and consumes a lot of power, it is desirable that the number of rules used is as less as possible. Specific switch solutions that support a large number of rules (up to 1 million wildcard match flow entries) [100], programmable switches [22] which support better usage of switch resources, disaggregated programmable switches which disaggregate memory and computing resources

[29], and use of software rule tables [67] mitigate this issue.

1.1.2 SDN Controllers

A controller runs on commodity servers, providing an abstraction of the network, similar to an operating system. It provides abstractions for the underlying network topology, device management, statistics and notifications, in addition to providing isolation between application programs and enabling their safe execution. The abstractions enable network applications to specify the desired behaviour in high level languages, without being concerned about how that behaviour is implemented.

When a packet reaches a switch, if a rule does not exist on the switch, it forwards the packet (and subsequent packets) to the controller. The controller now installs the desired rule on the switch and forwards the buffered set of packets to the switch. This is known as *reactive* installation of rules. Since this consumes time, rules may be *proactively* installed on switches. A combination of the two methods is also possible and SDN high level languages support the above methods [124]. Rules may be installed such that one rule exists per flow [103, 62], or rules may be compressed, to save space and to prevent sending a message to the controller for every new flow, using wild cards [119, 57, 80, 133].

1.1.3 Middleboxes and Network Functions

Packet headers or payloads are processed for purposes other than forwarding, to ensure safe and optimized use of the network, using specialized equipment called *middleboxes*. Since middleboxes are special-purpose hardware platforms, they are hard to maintain and manage. Introducing new functions requires new middleboxes, additional space and energy, and specialised training to manage them. To alleviate this problem, the same functionality is implemented in software using *Network Functions* (NFs), which are realized as virtual machine instances running on commodity hardware, such as standard storage, servers and switches.

1.2 Rule Updates

Flows using the network are governed by a set of *policies* that control access to hosts, switches or sections of the network. These policies are realised as rules installed on switches

that match certain packet headers and perform actions such as dropping or modifying a packet or forwarding it to a specific port. Policies may require to be changed [120]. Flows may need to be re-routed dynamically in the event of a failure or network congestion [128, 56, 2, 60], to achieve better throughput or to reduce latency or both. Network topology may be altered to save power by switching off nodes that are less used, after re-routing flows from them [17, 54]. Maintenance activities of the network such as upgrading switches, fixing faults etc. must not cause disruptions to network flows [123]. Traffic may need to be steered through specialised nodes in the appropriate order in a network [26]. Heorhiadi et al. [55] enable offline use of specialised network equipment for economical and safe use of network resources using SDNs. These are examples of SDN network applications and all the above result in switch updates. They also illustrate that switch updates are central to the functioning of an SDN.

A *Rule Update* (RU) consists of updates to a set of switches S, called *affected switches*, in the network. RUs cause temporary inconsistencies in the network, leading to packets getting dropped, wrongly routed or packets circulating in loops. If the inconsistency caused by an update is transient, must it be addressed at all ? Networks install policies to comply with safety regulations or to meet commercial requirements [119]. Networks have stringent security demands either due to Service Level Agreements, industry or government regulations (for example, investment and brokerage businesses must be kept separate in a company [119]) and policies must not be violated during a policy or a path update, so that those requirements are met. If updates cause packets to drop, loop, get wrongly routed and miss waypoints (that is, a network node that a packet must always traverse), leading to reduced throughput, flow failures and policy violations, *though transient*, an attacker can use updates themselves to make a network dysfunctional and to steal information [53].

1.3 Rules, packets and RUs

In this section, we discuss the switch model assumed in the thesis, the notion of rule asymmetry in switches and what disjoint RUs are.

1.3.1 Switch model

A switch has an ordered set of rules K (called a table), consisting of n rules, say, $a_1, a_2, ...a_n$. There may be more than one table in a switch. Each rule a has three parts: a priority Q, a match field M and an action field A. A rule is represented as $a = [Q, M, A]$. An action consists of zero or more *primitive actions*. An example of a primitive action is "forward packet to port 1".

The header fields of a packet p are used in matching against rules.

Q is a natural number, with higher values having higher priority. Let a_1 and a_2 be two rules. Let $a_1 = [Q_1, M_1, A_1]$ and $a_2 = [Q_2, M_2, A_2]$. a_1 precedes a_2 if $Q_1 > Q_2$. An incoming packet is matched with the match fields of the rules in a table and the action associated with the highest priority rule that matches it is executed. If the switch has more than one table, the packet is forwarded to the next table. Otherwise it is forwarded to the next switch.

r with suitable subscripts denotes an individual rule, while R with suitable subscripts denotes a set of rules, with $r_o \in R_o$, $r_n \in R_n$ and $r_u \in R_u$, with the suffix o denoting an old rule (a rule to be deleted), n a new rule (a rule to be added) and u an unaffected rule. Thus R_o denotes a set of old rules, R_n a set of new rules and R_u a set of unaffected rules. A packet is denoted by p and a set of packets by P. An update u_i on a switch $s_i \in S$, where S is the set of affected switches, is denoted by $s_i(R_{oi}, R_{ni})$, where R_{oi} denotes the set of old rules and R_{ni} the set of new rules for s_i. A Rule Update U is a set of updates $u_1, u_2, u_3, ..., u_n$. A packet that matches any $r_{oi} \in R_{oi}$ or any $r_{ni} \in R_{ni}$ in any $s_i \in S$ is called an *affected packet*. The controller can specify the values of all the parts of a rule, and install those rules in switches.

1.3.2 Asymmetry of rules

A packet may match more than one rule in a switch. Such rules are said to depend on each other. For example, in Figure 1.1, the rules r_{u1}, r_{o1}, r_{u2} and r_{u5} depend on each other. The solid arrows show the existing dependencies among rules and the dotted arrows, the change in dependencies after an RU.

Figure 1.1 illustrates three types of RUs U_1, U_2 and U_3 on a switch s_1. In U_1, the old rules match exactly the same set of packets that match the new rules and vice versa. In

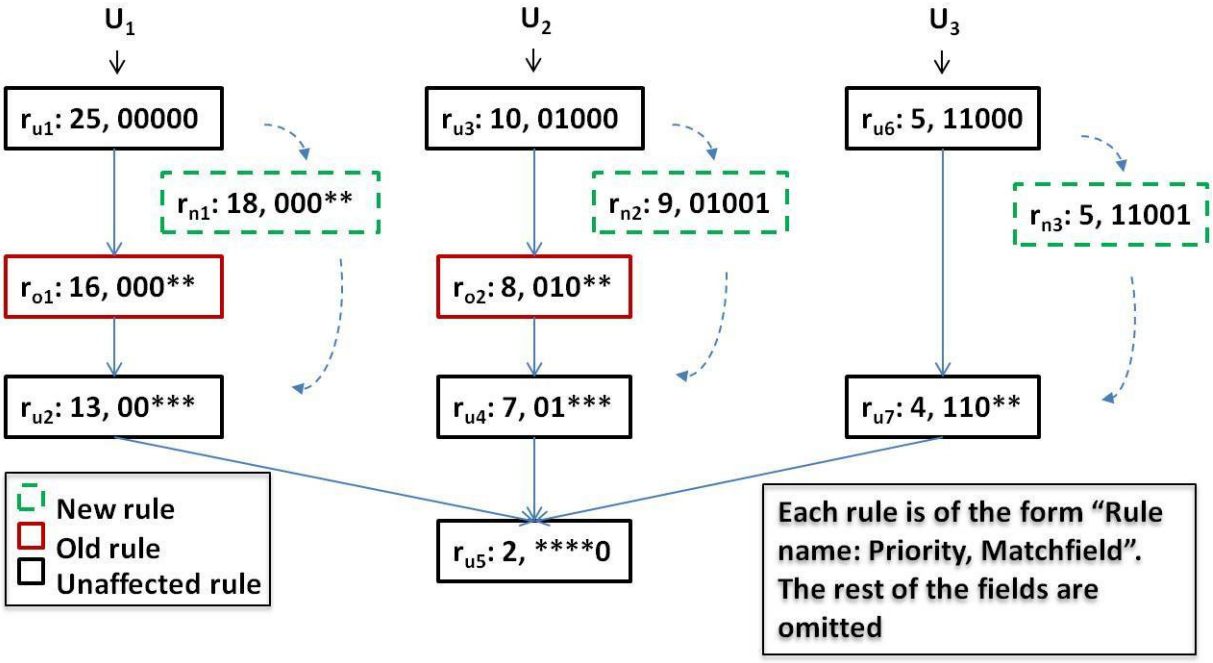


Figure 1.1: Dependencies among rules. In RU U_1, the old rules match exactly the same set of packets that match the new rules and vice versa. In U_2, no new rule matches a packet with the match field 01010 or 01011. In U_3, there is no old rule that matches a packet with the match field 11001.

U_1, let P be the set of packets that match $000**$. Then r_{n1} is the new rule and r_{o1} the old rule that matches P at s_1. Alternately, there may only be an old rule and no new rule that matches P at a switch for an RU U_2. r_{o2} in Figure 1.1 is an example of such an old rule, where P is the set of packets that match $0101*$. Here, as there is no new rule that matches P when r_{o2} is deleted, P will start matching r_{u4}. In the same figure, in another RU U_3, for the set of packets P that match 11001, there are no old rules on s_1 while r_{n3} is the new rule. Thus every affected switch may not have a set of new rules that match all the packets that match a set of old rules, and vice versa. Furthermore, either of old or new rules may not exist at all on an affected switch. In such cases, we say that the new rules and the old rules of the affected switch are asymmetric. Update algorithms that support wild carded rules must take this *asymmetry of rules* into account.

1.3.3 Disjoint RUs

Let P_1 be the set of all packets that match at least one of R_{oi} or R_{ni} of an RU U_1 and let P_2 be the set of all packets that match at least one of R_{oj} or R_{nj} of another RU

U_2, for all $s_i \in S_1$ and $s_j \in S_2$, where S_1 and S_2 denote the set of affected switches for U_1 and U_2 respectively. U_1 and U_2 are said to be disjoint RUs if P_1 and P_2 are disjoint. If U_1 and U_2 are disjoint, they can occur concurrently and every packet in the network will be affected by at most one of the updates U_1 or U_2. For example, let $R_{ni} = \{[5, 1000, forward\ 10]\}$. Let $s_i(null, R_{ni})$ be the only update in U_1. Let $R_{oj} = \{[5, 1 * **, forward\ 5]\}$. Let $s_j(R_{oj}, null)$ be the only update of U_2, where $i \neq j$. U_1 and U_2 conflict. Let $R_{nk} = \{[5, 1111, forward\ 10]\}$. Let $s_i(null, R_{nk})$ be the only update in U_3. U_1 and U_3 are disjoint. Multiple disjoint RUs, such as U_1, U_2 and U_3 in Figure 1.1, can be combined into one. In that case, the set of old (new) rules at an affected switch is the union of the set of old (new) rules of each of the disjoint RUs at that switch. Disjoint RUs are also referred to as non-conflicting RUs and non-disjoint RUs are also referred to as conflicting RUs.

1.4 Properties preserved during RUs

Algorithms that preserve two significant properties during RUs are discussed in the thesis: *per-packet consistency* and *per-flow consistency*. This section discusses in brief, what each of those properties means and the motivations for preserving these properties.

1.4.1 Per-Packet Consistent Updates

In Figure 1.2(a), when switches are updated to change a route from its old path (solid lines), to the new path (dotted lines), if s_f is updated first, packets arriving at s_1 will get dropped, while if s_3 is updated first, packets arriving at s_3 will loop, reducing throughput or causing flow failures. To prevent this, updates to individual switches can be sequenced. In Figure 1.2 (a), if RUs are preformed in the sequence $s_6 - s_5 - s_3 - s_2 - s_1$, there will be no packet drops or loops. However, sequencing updates is not a solution for all update scenarios: in the example in Figure 1.2 (b), s_1, s_3 and s_5 are waypoints and it is not possible to sequence updates such that waypoint invariance is preserved [140] when an RU is performed to change the path of a flow shown in solid lines to the one in dotted lines. In this case, how can waypoint invariance be preserved?

In Figure 1.2(c), s_n has the following policy P installed: *if destination ip = 192.168.*.*, forward the packet to* NF_n. s_n is connected to a stateless Network Function (NF) [64] NF_n,

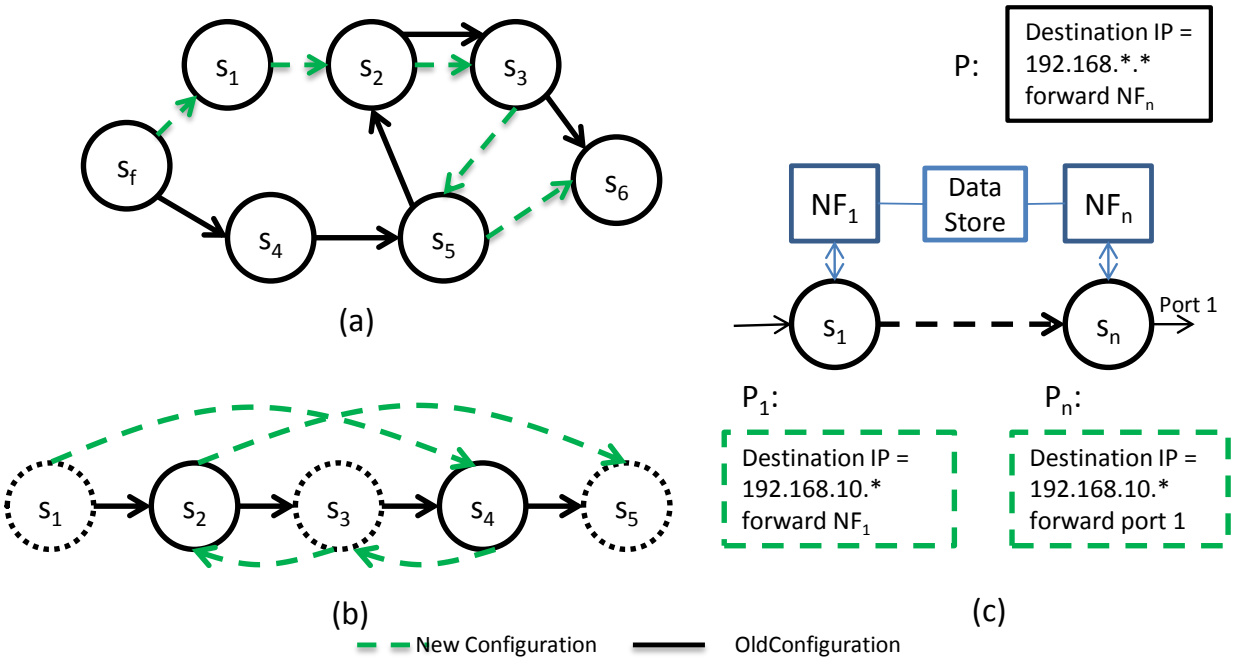


Figure 1.2: Need for PPC updates

that is an Intrusion Prevention System(IPS). When the states associated with all the NFs in a network are stored separately from the NFs in a common data store the NFs are said to be stateless. An IPS detects and prevents attacks on the network. It is desired that a subset of flows is forwarded to another IPS earlier in the path, at NF_1 associated with s_1, to reduce the load on NF_n. This is accomplished by installing P_1: *if destination ip = 192.168.10.*, forward the packet to* NF_1, at s_1, and installing P_n: *if destination ip = 192.168.10.* forward to port* 1, with higher priority than P, at s_n, where forwarding to port 1 results in bypassing NF_n. If P_1 is installed at s_1, before P_n is installed at s_n, some packets will be sent to an IPS twice, once to NF_1 and then to NF_n, which will cause inconsistencies in the functioning of the IPS, whereas if P_n is installed at s_n before P_1 is installed at s_1, some packets will not be sent to any IPS whatsoever, compromising safety. Suppose the IPS forwards packets for further Deep Packet Inspection if there are more than a fixed number of failed connection attempts from a source address [37]. If the SYN of a failed connection is sent twice to such an IPS, it corrupts the state stored by the IPS. Since installing P_1 at s_1 and P_n at s_n simultaneously is impractical, we need them to be installed in a *per-packet consistent (PPC)* [111] manner: every packet must either use the old version of rules (only P at s_n) or the new version of rules (P_1 at s_1 and P_n at s_n) and never a combination of both. Similarly, a PPC RU will solve the problem of preserving

waypoint invariance in Figure 1.2(c).

In addition, languages that enable SDN application developers to specify and use states [11] also require the order of traversal of switches that store those states to be preserved during an update. Algorithms that preserve more relaxed properties use a PPC preserving update as a fall-back option: CCG [140] uses a PPC preserving algorithm if it cannot find an order of updates that preserves waypoint invariance and FLIP [122] falls back to such an algorithm if its Integer Linear Programming algorithms do not converge.

Existing PPC algorithms are presented and compared in detail in Chapter 2, section 2.7.

1.4.2 Per-Flow Consistent Updates

A *flow* is defined as a sequence of packets identified by the source and destination IP addresses, the source and destination ports and the transport protocol (the 5-tuple), with specific starting and ending mechanisms, as defined by the protocol. A flow is assumed to be a TCP connection, unless declared otherwise. The flow from the host that initiates a TCP connection is called a *forward flow*, f_f. The response from the receiving host is called the *reverse flow*, f_r. NFs and middleboxes usually preserve states per connection. Such NFs require both the forward and reverse flows to traverse the same NF instance always, thus preserving *connection affinity*. When the path of a flow in a network is changed, either the sequence of NF instances through which the flow is being routed and the instances themselves are not changed, or if they are changed, states preserved on those instances are moved to the new instances created on the new path of the flow [45]. The former method is said to preserve *waypoint invariance*. The latter method is called *state migration*. In addition to NFs, switches also may have states stored on them.

PPC updates are sufficient if RUs do not involve stateful NFs (that is, NFs that function without storing states anywhere) or middleboxes or stateful switches, or if the RU can be performed preserving waypoint invariance. They are also sufficient if the NFs involved in the RU are stateless [64]. However, this may not be the case for all RUs. Waypoint invariance cannot be preserved if the waypoints are overloaded. State migration is complex, requiring packets to be forwarded and buffered at the controller or other intermediate modules, and requiring additional updates to switches to consistently route packets that get buffered in this manner [130, 45].

Many network applications that use SDNs require RUs to preserve the property of *per-flow consistency* (PFC). If a packet of a flow matches either the new rules or the old rules of an RU, it is called an *affected packet* and the flow is called an *affected flow*. In a *per-flow consistent RU*, all the packets of an affected flow either use the new rules or the old rules, but never a combination of both [111].

If PFC is not preserved during an RU, per-flow states that are maintained on NFs or stateful switches will no longer be accurate. While not maintaining per-flow states on NFs causes safety issues, the impact of not maintaining per-flow states on switches depends on why those states are maintained. If, for instance, a stateful switch maintains per-flow counters, the value of those counters will be incorrect. We propose a PFC RU as a *general* solution for disparate problems that require RUs to stateful switches and NFs, whereas existing work has specific solutions for them. Examples of such problems and existing PFC algorithms are presented, and the algorithms are compared in detail, in Chapter 2.

1.4.3 General Requirements for PPC and PFC RUs

In this section, we discuss the characteristics common to our PPC and PFC algorithms.

All-or-nothing semantics: Updates to switches may fail. However, new rules belonging to an RU must be completely installed on all the affected switches or none of the rules must be installed; rules must never be partially installed.

Footprint Proportionality: Updating only the affected switches will reduce the number of controller-switch messages and the number of individual switch updates that need to be successful for the RU to be successful. We characterize this need as *Footprint Proportionality* (FP), which is the ratio of the number of affected switches of an RU to the number of switches actually modified for the RU. In the best case, FP is 1, and there is no *realistic* general update algorithm that achieves this while maintaining PPC or PFC, as far as we know.

When and how to delete old rules: A rule belonging to an older version can be deleted only when no old packets exist in the network. Otherwise, old packets will be dropped. This needs to be done as early as possible, to reduce the time for which old and new rules co-exist in the network.

Concurrent disjoint RUs: Applications such as traffic engineering [17, 60, 56] and dynamic flow scheduling due to changing workloads [2] require frequent updates to

the network. For tenant management in cloud data centers, there are update bursts of hundreds of changes per minute [105]. To manage a high volume of RUs, it is desirable to perform as many disjoint RUs as possible, concurrently.

Wildcarded and longest prefix match rules: Complex policies require flows to be grouped using wildcards [119] [57]. Support of wildcarded rules will help reduce rule space and enable installation of rules in a proactive manner, thus relieving the burden on switches and the controller for having to install rules for every new flow in the network [133]. Support of longest prefix match rules is required to enable networks to scale.

Varying switch and link speeds: The controller to switch links may have different speeds and the execution speeds of switches may vary. The RU algorithm must perform independent of these factors.

Computing affected paths at the controller: In order to restrict the number of switches that need to be changed, some algorithms require the controller to identify the paths affected by an RU [69, 62, 140]. Since this is time consuming and not always feasible, it must be avoided [140].

The need for meeting these requirements and related work are described in detail in chapter 2.

1.5 Contributions of the Thesis

The thesis describes algorithms for per-packet and per-flow consistent Rule Updates. Our solutions for both PPC and PFC:

1. progressively increase FP to 1
2. progressively increase the number of concurrent non-conflicting updates to an unlimited value
3. provide an all-or-nothing semantics, that is, ensure that rules are installed at all the affected switches or not installed at all
4. support wildcarded/longest-prefix match rules
5. do not assume that affected switches have both old and new rules matching every affected packet - that is, rules may be asymmetric

6. work irrespective of the execution speeds of switches or links

7. do not require flows in the network for updates to progress

8. need no packet buffering at the controller

The following algorithms are presented: The algorithm **E2PU** that preserves PPC, addresses the problem of the means to ensure that old packets are removed from the network before the next RU begins and that an all-or-nothing semantics is preserved for an RU. **CCU**, the next algorithm that preserves PPC, restricts the number of switches modified for the RU to the affected switches and the ingresses and improves concurrency of disjoint RUs, compared to E2PU. The PFC-preserving algorithm **EPCU**, uses a restricted number of exact match flows to perform a PFC RU and relies on the ability of the switch to generate such flows. **E2PU** and **EPCU** also describe how the TCAMs within switches may be combined with a rule table in software to improve the RU performance. Our PPC algorithm **PPCU** and PFC algorithm **ProFlow** achieve an FP of 1, support unlimited concurrent disjoint RUs and use data plane time stamps to decide when switches must move from the new to the old rules, while accommodating time asynchrony. They also exhibit the characteristics from 1 to 8 above.

A proof-of-concept implementation in P4 and Mininet demonstrates the feasibility of PPCU and ProFlow. In a network with continuous PPCU updates, the throughput and the total number of flows completed are higher compared to a network with continuous random updates, and cause no safety violations. During a ProFlow update, new flows maintain their throughput while old flows undergo a marginal reduction, in comparison with a scenario without an update, where the first affected switch in the flow path has both new and old rules.

1.6 Organisation of the Thesis

The thesis is divided into two parts: part 1 has the algorithms for PPC updates and part 2 those for PFC updates. The rest of the thesis is organised into chapters as follows:
Chapter 2 describes the background of the thesis, elaborates the motivations of the problems presented and the related work.
Part 1:

Chapter 3 describes the PPC algorithms E2PU and CCU.

Chapter 4 describes the PPC algorithm PPCU, its implementation and evaluation.

Part 2:

Chapter 5 describes the PFC algorithm EPCU.

Chapter 6 describes the PFC algorithm ProFlow, its implementation and evaluation.

Chapter 7 concludes the thesis and presents future research directions.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter first describes the components of an SDN in detail to provide the background for the algorithms described in the thesis. It then presents the motivation for the algorithms and a literature survey related to the algorithms.

2.2 SDN Switches

SDN separates the data and control planes, making switches simple forwarding devices, the most notable example of which is an Openflow switch [101]. It consists of one or more flow tables, a group table, and a channel that interfaces with the controller. A flow table consists of a set of flow entries or rules, with match fields, priority, counters, timers, and instructions that need to be applied to matching packets, associated with a flow entry. Flow entries are installed in order of priority. Packets are matched with the match fields and the instructions associated with the first matched entry are executed. If there is no matching entry, the instructions in the table-miss flow entry are executed - this entry has wildcards for all match fields and has the lowest priority. The instruction may be an action such as forwarding a packet to a port or modifying a packet header, or a modification of the pipeline. Using the Openflow protocol, the controller can add, delete or modify flow entries. The timer associated with a flow entry is used to specify the maximum amount of time before it is deleted by the switch. The counter is updated when packets are matched with a flow entry. Depending on the rules installed, an Openflow switch can behave as a

router, a switch or a middle-box. However, an Openflow switch does not support states or data plane programmability, and the functions that can be performed at line rate are limited.

Traditional switches are implemented as a sequence of tables, with each table dedicated to a fixed function. If only some of the tables are required for a given functionality, the remaining switch processing and table resources are wasted. If new headers need to be added to packets, it can take years before they are realised in switches. For example, VXLAN, a field for packet encapsulation, took 3 years to be made available on a chip, after the feature was introduced [63]. This inspired the creation of switches with programmable data planes.

2.2.1 Programmable Data Planes

In switches with programmable data planes, the switch data plane behaviour is specified by the data plane program. Since the data plane behaviour and the resources such as table sizes required to implement it are no longer fixed, switch resources can be used according to the needs of the program. Moreover, protocol-independent packet processing is supported at run-time, without sacrificing high switching speed. The abstractions offered by a programmable data plane facilitate the design of the desired forwarding and packet processing behaviour of switches without being restricted by the hardware available at that point in time and support stateful behaviour that is useful to implement many switch functions. P4 [30, 24] and Domino [117] are examples of languages that are offered by programmable data plane switches, RMT [22] and Flexpipe [102] are examples of switch architectures that support programmable data planes and Jose et al. [63] describe a compiler for P4 for these architectures. dRMT [29] disaggregates the memory and computing resources of a programmable switch. It moves memory to a centralized pool and replaces the pipeline stages of an RMT with a cluster of processors that can execute match and action operations in any order, thus improving utilisation of memory and processing power.

P4 exposes a set of APIs to enable the controller to program switches, that is, to provide inputs to the P4 programs running on switches. A P4 program decides what each table in a switch *can* do, while the controller, through the P4 run-time APIs, decide what each switch *must* do. Switches integrating the P4 APIs with Openflow for the controller-

switch interface are currently available in the industry [131].

2.3 Controllers

A controller provides an abstraction of the network. It can control switches through commands to adapt to network changes that are observed by monitoring the network.

Applications may attempt to write conflicting rules into the network or they may attempt to provide functionality that is semantically conflicting. For example, two different applications may attempt to change the route of a flow in a conflicting manner, by installing conflicting rules. A power saving application may try to re-route a flow from a path with the intention of turning off switches along that path, while a load balancing application may attempt to move flows to the same path, creating a conflict. Controllers such as PANE [40] solve these problems by delegating authority to applications, which are hierarchically organised. Meta-controllers such as Athens [12], that operate above SDN controllers, use a voting procedure.

Similar to virtualisation in computer systems, network virtualisation is used in computer networks for optimised use of the resources of a physical network. Network virtualisation may also be variously implemented in a distributed hypervisor implemented across the controller and the network switches [3], an SDN high level language [43], or in a centralised hypervisor positioned between the SDN controller and the physical network [35].

2.4 SDN languages

There are several [124] special purpose and general high level languages such as Frenetic [43] and NetKAT [8] for application writers to write programs for an SDN network. When these programs run, they generate switch rules that need to be installed on switches, using the open interface available between the controller and the switch network.

SDN languages such as Pyretic [109, 95], Procera [126] and Nettle [125] allow composition of programs by the application writer while Redactor [129] automatically achieves such composition by mandating a declarative language to write control programs. Thus semantic conflicts between various SDN applications are resolved before these applications

initiate Rule Updates.

Trois et al. [124] provide a survey of various SDN languages. The compilers of these languages may generate switch rules in the manner expected by the algorithms in the thesis, to implement the algorithms. The run-time modules of these languages are expected to exchange the messages required by the algorithms, with network switches.

2.5 Network Virtualisation

Network virtualisation enables a physical network to be shared by multiple virtual networks [35, 3], allowing tenants to specify their addresses, topologies and control logic, and managed by a network hypervisor. This allows strong isolation between tenants, easy migration of enterprise networks with custom topologies to another infrastructure and optimized use of the physical network.

A hypervisor provides a general interface such as Openflow to its virtual controllers and uses the same general interface to program the physical network. Thus the (virtual) controllers are unaware of whether they interface with a hypervisor or a physical network. Virtualisation allows a physical node to be mapped to one or more virtual nodes, a physical link to be mapped to one or more virtual links and a physical SDN to be mapped to one or more virtual SDNs (vSDN). The hypervisor provides abstractions for physical links and nodes. The data plane, the control plane and the virtual addresses are isolated from those of other virtual networks. Each virtual controller transparently interacts with its virtual network, without being aware of the existence of other vSDNs. Hypervisors may be centralised [35, 116], in which case multiple controllers interface with them or distributed [3], in which case a logical hypervisor implemented in the controller and the switches provides the functions of virtualisation. Blenk et al. [19] provide a survey of network virtualisation hypervisors for SDNs.

2.6 Middleboxes and Network Functions

Network Functions are required to improve network performance, enhance network security or monitor traffic. An example of an NF is an Intrusion Detection System (IDS), which detects attacks on a network. Using Network Functions Virtualization (NFV), NFs

can be virtualized [52] and located anywhere on a path in the network, thus decoupling functionality from location. Packets require a series of NFs to be applied to them in a specific sequence, resulting in a Service Chain (SC) [36]. For example, a packet entering a network may first pass through a Network Address Translator (NAT) that allocates the packet the correct server address, followed by an IDS, which examines if there is an attack, followed by a firewall, which drops packets or allows them to pass.

NFV separates the software of NFs from their hardware, allowing both to evolve independently. It also allows services to grow and shrink depending on network conditions and provides functionality such as inserting and removing NFs onto network paths, depending on needs. NFV and SDN are complementary technologies but can benefit from each other. Mijumbi and et al. [87] provide a survey of NFV.

2.7 General requirements and Per-Packet Consistent Updates

In this section, we examine the general requirements for a PPC or a PFC RU and survey the literature related to these requirements.

Consider an SDN whose switches have a set of rules of version 0 ($v0$) that needs to be updated to version 1 ($v1$). As per the seminal update algorithm that preserves PPC, two-phase update (2PU) [111], the ingress switch always tags packets with the version number of the rule that must be applied on the packet, and the switches other than the ingress, called the *internal switches*, check each packet for the corresponding version tag before applying a rule.

The two-phase update from $v0$ to $v1$ is implemented as follows: The controller updates the internal switches with $v1$ rules while the $v0$ rules remain in the switches. Next, it updates the ingress switches with $v1$ rules. This also results in the ingress switches tagging all matching packets to indicate that they belong to $v1$. After the last $v0$ packet leaves the network, the controller instructs all the ingress and internal switches to delete the $v0$ rules [111].

2.7.1 Under-specification of two-phase update

2PU is under-specified on two matters: how to provide an all-or-nothing semantics during an update and when old rules may be deleted from switches, while preserving PPC.

All or nothing semantics: Updates may fail and failures may be due to any of the following causes during the update: F1) A switch fails and stops. F2) A switch fails and restarts but without any rules in it[1]. F3) A link to a switch fails. F4) A switch operates very slowly. F5) A message is not delivered correctly or not at all. F6) A switch has unsupported features, receives wrong instructions, the flow table is full etc. and the switch responds with an error message during the update. F7) Due to software bugs, a switch exhibits unpredictable behaviour (hangs, does not install updates, sends wrong messages, reboots, corrupts existing rules etc.). F8) A switch exhibits malicious behavior. If the rate at which RUs occur in a network is high, if there is a failure, it is likely that a failure coincides with an update.

If there are failures, we need to limit the effects due to failures on the update process. New rules belonging to an RU must be completely installed on all the switches or none of the rules must be installed; rules must never be partially installed. Those updates where the two-phase algorithm is not fully executed must not change the semantics of those updates where the algorithm is fully executed (after Bernstein et al. [18]). For example, if the link to a switch S1 fails temporarily during a $v1$ update and the $v1$ update did not occur on S1, it must not happen that the rest of the switches upgrade to $v1$ when S1 has $v0$ after the failed link comes up. However, 2PU does not specify a mechanism for the controller to ensure this. If a switch fails to update, the controller does not have the means to know that there has been an update failure. Moreover, in that case, what is to be done with the rest of the updates that have already occurred? Another issue is that if the rules on switches where they are already installed need to be deleted, it wastes a lot of time as writing into and deleting from TCAM is time consuming. The controller must know the list of valid updates active on the network. The use of tagging automatically ensures that the update is versioned and the switch stores the previous version; however the current update algorithm has no mechanism to ensure a rollback.

It is not within the scope of the update process to attempt complete recovery from

[1]Since switches do not have any non-volatile storage, no rules are permanently stored. The controller will need to repopulate the switches.

failures because, if there is an update failure, the next action to take is very application specific. Hence the update process must only limit the effects due to failures. For example, if the $v1$ update was to reroute packets through a switch S2 instead of S1 due to a policy change, and S2 fails during the update, then the application may want to just abort the update. On the other hand, if there is a routing update which changes the ingress to egress path, if a switch fails on the new path while it is being updated to $v1$, the application may want to instruct the controller to suspend the ongoing update and resume it in a modified manner, taking advantage of the switch updates that may have already occurred for $v1$.

There is existing work to provide an abstraction to develop programs that implement fault tolerance in the network [110], algorithms to implement local fast failover [14, 20], re-design of the controller to reduce fate-sharing between apps and the controller [27] etc. but none on failure during an update itself.

In addition to the above, unrelated to failures and recovery from failures, applications may wish to abort an ongoing update. Such an abort is meaningful only if it takes place before the ingress switch updation is complete. Otherwise, the application has to issue a new update to take the network to the desired state.

Time of deletion of old rules is unspecified: A rule belonging to an older version must be deleted only when the switch is certain that no packet belonging to the older version exists either within the switch or upstream from the switch. Otherwise, packets belonging to the older version will be dropped. But deleting a rule cannot be indefinitely delayed because TCAM space is scarce.

An internal switch knows whether there are active flows associated with a rule, but it does not have a mechanism to know if the packet it has received is indeed the last packet belonging to $v0$. Though many networks use load balancing schemes that use the same path for all packets of a flow and distribute different flows over different links, there are schemes that are proposed that send different packets of the same flow over different links [33]. In such cases a switch may not know if the last packet belonging to $v0$ will be sent to it at all. One of the ways to solve this is for the controller to ensure that the last packet has crossed the switch, before it sends a delete command to that switch.

It is possible to associate an inactivity timer with each old rule such that the rule is deleted if it is not accessed in the specified time. The timer value depends on inter-

arrival times between packets, which could vary depending on delays in switches, which perhaps vary due to congestion, or paths that packets traverse, which could vary due to complex load balancing schemes. More than the value of the timer being unpredictable and variable, the issue is that an internal switch can never be certain that there are no packets upstream even if the timer has expired, if the timer is only associated with accessing the rule - the rule may be deleted prematurely. It is desirable to have a mechanism to delete the old rule set independent of the above considerations and only after ascertaining that the new rules are effective.

Katta et al. [69] propose a method that reduces rule space in switches by splitting an RU into several rounds that each transfer a part of the traffic to the new configuration. zUpdate [77] finds a sequence of updates to progressively meet the end requirement of preventing congestion, while SWAN [56] always reserves a percentage of link bandwidth and finds a suitable update sequence to meet the same objective. Mahajan et al. [83] provide a dependency table that denotes at which switch a rule must be modified to guarantee a consistency property, before it uses a new rule. Yuan et al. [134] propose a method of creating a dependency graph between rules such that their topological ordering is a safe update. Dionysus [62] improves the update time by dynamically scheduling a sequence of updates in such a manner that update speeds of different switches are taken into account. CCG [140] provides a mechanism to enforce customize consistency properties by scheduling updates based on the invariant specified for that update by a network operator. None of these describes how and when to delete old rules. McGeer [85] proposes a method to preserve rule space in switches during updation and addresses the problem of identifying when the old rules can be deleted, but it requires installation of intermediate rules and forwarding packets to the controller. No work addresses an all-or-nothing semantics.

Our algorithm, E2PU-SRT (chapter 3) addresses these questions first. All the algorithms in the thesis include the solution provided in E2PU-SRT in various ways.

2.7.2 Need for concurrency

In a large data centre that supports network virtualisation and multiple tenants, each tenant will require multiple VMs to be created [12], all belonging to a virtual network of a certain topology. VMs will need to be assigned to servers depending on what the VM allocator wishes to optimize. To isolate one tenant from another, for rate limiting etc.,

the controller will need to update the virtual switches on servers. The physical switches will need to be updated to implement the virtual network to physical network mapping [3], [13]. We envisage that continuous updates to SDN will be common due to the large number of varied applications available [120, 128, 56, 2, 17, 54, 123, 13, 3, 48, 60, 26, 55] due to the flexibility and programmability of SDN.

Updates to an SDN can be frequent - MicroTE [17] proposes a fine-grained traffic engineering mechanism for data centres that involves updating switches at the granularity of seconds. B4 [60] proposes updates every few minutes and SWAN [56] proposes updates as frequently as possible, to achieve high WAN utilization. If a network is updated based on its current workload, changing workloads would necessitate frequent updates [2], for dynamic flow scheduling. For an L4 load balancer used in a multi-tenant cloud computing environment with 10000 nodes, on an average there are about 12000 updates per day, with bursts of hundreds of changes per minute [105]. According to another work, cloud providers host virtual networks of the order of 10Ks and they need to support millions of concurrent updates of all these networks per day [47].

2PU [111] does not support concurrency as all the switches in the network need to be updated for a single update. Existing algorithms that preserve PPC and allow concurrent RUs limit the maximum number of concurrent RUs [81, 140], with CCG [140] becoming impractical if the number of paths affected by the RU is large. CCG uses fast concurrent update methods but only preserves properties that are not as strict as PPC and falls back to a two-phase update (2PU) [111] for these cases: 1) an RU affects multiple independent paths and therefore calculating the paths affected by the RU becomes time-consuming 2) if all paths must traverse n waypoints, the old and the new paths can be thought of to consist of $n-1$ segments and the update of a new segment may depend on the update of an old segment. 3) properties such as path length constraints need to be met. For these cases, CCG does not support concurrent RUs. Ludwig et al. [79] propose an algorithm that complies with the property of relaxed loop freedom, which ensures that there are only transient loops during the RU, but does not preserve PPC. Dionysus [62] preserves PPC and is concurrent, but works only when any forwarding rule at a switch matches exactly one flow. It does not work where the network uses wildcard rules or longest-prefix matching.

Statesman [123] and other work [93], [40], [129] and [12] deal with conflict resolutions

for concurrent updates. Frenetic [43] supports functions that a program can call to compose non-conflicting rules and ultimately updates the network using 2PU [42]. NetKAT [8] supports a "slicing" [50] abstraction using which conflicting rules can be run on different slices of the network. In these efforts, the emphasis is, variously, on conflict detection, resolution and composition of policies within a program whereas we require algorithms to be able to handle disjoint updates from any source, even from multiple controllers, at a level close to the switches and preserving PPC. While FIXTAG [25] allows fault-tolerant concurrent updates, the tag complexity is exponential in the network size and therefore not practical, whereas REUSETAG [25] reduces tag complexity but allows only sequential updates.

Our algorithm CCU (chapter 3) improves the concurrency of disjoint PPC updates compared to 2PU and our algorithms PPCU (chapter 4) and ProFlow (chapter 6) support unlimited concurrency of disjoint per-packet and per-flow consistent updates, respectively. All the algorithms assume that the controller issues only disjoint updates, using conflict resolution methods discussed in section 2.4.

2.7.3 Footprint Proportionality

A two-phase update(2PU) [111] requires changes to all the switches in the network, even if the update is intended only for one switch. Our algorithm E2PU-SRT (chapter 3) falls in this category. This also causes doubling of all the rules on all the switches and all the tables in a switch while the RU takes place, further reducing its scalability. Concurrent RUs cannot be performed, even if they are disjoint. To improve this, a class of algorithms [69, 62, 140] identify and compute the paths associated with the affected switches, and update *all* the switches along those paths (path aware algorithms); however, they either do not support wildcard rules or longest-prefix match routing, or if they do, discovering affected paths is computation intensive. Another class restricts the changes to the affected switches and the *network ingresses*, without computing paths, namely, our algorithm CCU (chapter 3) and "General Update", referred to as GU [81] from now on (ingress affecting algorithms). These, however, have upper limits on the number of disjoint updates supported and all ingresses need to be updated even if only one switch is affected. TimeFlip [91] expects switches to be synchronized to a clock and schedules updates to occur at specific times (timed updates); however, this does not take into account clock synchro-

nisation inaccuracies, realistic controller-switch delays [140], non-uniform switch speeds and scheduling inaccuracies at switches. A detailed comparison is in Table 2.1.

For path aware algorithms [69, 62, 140], every switch in the path, which may be across the network, needs to be changed, even to modify one rule on one switch. If one rule affects a large number of paths, such as "*if TCP port=80, forward to port 1*", computing the paths affected is time consuming [140]. For both 2PU and path-aware methods, the number of rules used in *every* switch modified during the update doubles. The problem is exacerbated in practical switches, as switches supporting RMT [22, 63, 21] Intel's FlexPipe [102] etc. usually use more than one table and every rule in every table in every switch needs to be changed even if the actual change is only for one rule in one table in one switch. Hence the update time is disproportionately large, even if the number of rules updated is small.

In a data centre with a Fat Tree topology with a k-ary tree [1] where the number of ports $k = 48$, the number of ingress switches is about 92% of the total number of switches. Even to modify one rule in a switch, all of the ingresses will need to be modified, for ingress affecting algorithms.

Updating only the affected switches will reduce the number of controller-switch messages and the number of individual switch updates that need to be successful for the RU to be successful. We characterize this need as *footprint proportionality (FP)*, which is the ratio of the number of affected switches of an RU to the number of switches actually modified for the update. In the best case FP is 1, and there is no *realistic* general update algorithm that achieves this while maintaining PPC or PFC, as far as we know.

McGeer [86] proposes an algorithm that has no overheads to achieve a PPC update, but it depends on finding a correct *sequence* of updates. We have already given examples of waypoint invariance and policy installations where it is not possible to find an update sequence such that PPC is satisfied. McGeer [85] provides another general solution; however, it relies on installing an intermediate rule in affected switches that diverts all affected packets to the controller during the course of an update, which is impractical in a real network as it incurs a high overhead on the controller and the controller-switch links, affects packet throughput and is not scalable. Though it preserves PPC, it preserves only a weak form of PFC.

As mentioned earlier, there are update algorithms that use data plane time stamping

Table 2.1: Comparison with prior algorithms

Method	*PPC guaranteed*	*Ratio of affected to modified switches*	*Varying switch, link speeds immaterial*	*No path computing*	*Supports wildcarded and lpm rules*	*All-or-nothing semantics*	*Concurrent updates*	*Tolerates sync. inaccuracies*
Two-phase (2PU) [111]	✓	< 1	X[1]	✓	✓	X	X	-
Two-phase enhanced (**E2PU**) (chapter 3)	✓	< 1	✓	✓	✓	✓	X	-
Path aware [140, 69]	✓	< 1	✓	X	✓	✓	✓	-
Path aware dynamic ordering [62]	✓	< 1	✓	X	X	✓	✓	-
Path unaware ingress affecting - GU [81], **CCU** (chapter 3)	✓	< 1	✓	✓	✓	✓	Limited	-
Timed update [91]	Conditionally [2]	1	X	✓	✓	X	Conditionally [3]	X
PPCU (chapter 4)	✓	1	✓	✓	✓	✓	✓	✓

1 Underspecified

2 If switches are accurately synchronized,scheduling accuracy is high and switches/links have uniform speeds

3 If scheduling accuracy is high regardless of the number of updates

[90, 92, 101, 91] for PPC updates. However, they 1) require accurate and synchronous time stamping [90] 2) depend on an estimate for the time at which the update must take place 3) do not take into consideration variable delays between the controller and the switches [140], leading to messages getting delayed or lost 4) depend on scheduling accuracy at switches, which is not guaranteed as multiple processes run on a switch and therefore a scheduled update may not execute exactly at the time it was scheduled. These practical issues result in them not guaranteeing an all-or-nothing semantics or per-packet consistency for RUs.

Our update algorithms, PPCU (chapter 4) and ProFlow (chapter 6), have an FP of 1 and complete in a finite number of steps regardless of the execution speeds of the switches involved and regardless of the relative speeds of the controller-switch links, while preserving PPC and PFC, respectively.

2.7.4 Need for rules with wildcards/longest prefix match

Complex policies commonly group flows together using wildcarded rules [119] [57]. Installing wildcarded rules for selected flows or rule compression in TCAM for flows or policies [137] reduces both space occupancy in TCAM [80] and prevents sending a message to the controller for every new flow in the network [133]. Installing policies using wildcarded rules is not restricted to ingresses, as ingresses may not have enough space and space may be saved by sharing them across flows by installing them in internal switches [66]. Longest-prefix matching (lpm), where the destination address with the longest matching prefix is chosen in case of a conflict, is commonly used in large networks to enable the network to scale and it is important that updates support such rules. Using switches for traffic splitting, whether for load balancing or multipathing use wild carded rules [128] or prefix-matching rules [65]. Switchreduce [59] provides solutions for the tradeoff between visibility of rules and space occupancy. Thus in a real network, using wildcard or longest prefix match rules regardless of the switch location in the network is inevitable. Many of the existing update algorithms that preserve PPC or PFC do not address rules that cater to more than one flow [62] [140]. Supporting wildcards/lpm also results in updates that require only removing rules and not adding any rule or vice versa, on one or more switches in a network, as in the example given in Figure 1.2 (c), or rule asymmetry. Most SDN application level programming languages already support proactive installation of

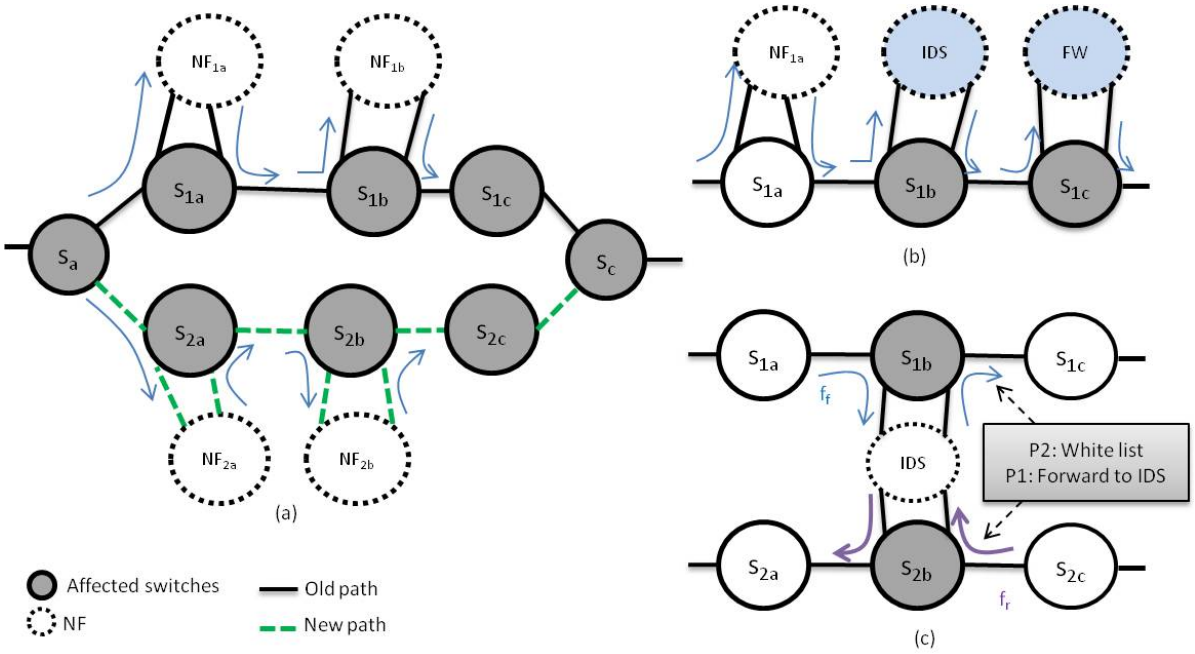


Figure 2.1: PFC Updates to a Service Chain

rules [94, 43, 109, 8, 120, 84] and some of them support aggregating rules using wild cards [43, 94, 109, 8]. All the algorithms in this thesis support wild carded and lpm rules.

2.7.5 Preventing safety violations in the network

There are tools that perform a static verification of the network to detect safety violations [70, 71, 78] and some that generate tests and verify adherence to policies dynamically [37]. There are tools that intercept updates to check if they violate operator specified policies [139] and repair them. Our focus is to prevent safety violations *during* network updates.

2.8 Per-Flow Consistent Updates

A per-flow consistent update is useful to solve problems in several areas such as Service Chaining [36], Server Load Balancing, [128] Network Functions Virtualization [103] and Network Virtualization [35, 3]. In this section, we examine the motivations for using a PFC RU in these areas and describe related work.

2.8.1 Service Chaining:

It is possible that due to network load or other dynamic considerations, some of the flows in a network need to be rerouted [76]. In Figure 2.1(a), let us assume that some of the

flows traversing $s_a - s_{1a} - s_{1b} - s_{1c} - s_c$ need to be re-routed to $s_a - s_{2a} - s_{2b} - s_{2c} - s_c$. At the same time, instead of the Service Chain $NF_{1a} - NF_{1b}$, now they must traverse $NF_{2a} - NF_{2b}$, where NF_{2a} is another instance of NF_{1a} and NF_{2b} is another instance of NF_{1b}. Besides, let us assume that the switch s_{1c} maintains the state associated with a flow, for instance, to check which flows are large [118]. Therefore, if flows are moved from the first path to the second, the states associated with the NFs as well as the switches must be moved. Instead, a PFC update may be performed, where only the flows that begin after the RU is effective (*the new flows*), are moved to the new path, while the rest of the flows (*the old flows*), continue to use the old path. The forward and reverse flows of a given flow, new or old, will thus traverse the same SC. In this example, all the switches are affected - the switches in the new path are affected because new rules for the forward and reverse flows need to be added to them, and in the old path, because rules need to be deleted from them. We shall first examine the existing PFC solutions and why they are inadequate.

Existing PFC solutions: In order to keep track of each individual flow, a *microrule* is added for each flow, where the match for the rule is based on the header fields that uniquely identify a flow. A time-out is associated with each microrule which is triggered when no matches take place for the time-out period and a suitable associated action is taken. The microrule is an exact copy of the original rule, except that its headers match exactly that of a flow.

Devoflow [32] proposes a switch that can support a "clone" command and usage of this to support PFC is examined by Reitblatt et al. [111]. When this command is set for a rule with wildcards in an ingress switch, the switch installs a microrule for each new header seen that signals a new flow. Subsequent packets of that flow match that microrule. Consider an SDN whose switches have a set of rules of version 0 ($v0$) installed first, that needs to be updated to version 1 ($v1$). The algorithm uses clone rules when any update is installed. Thus, for example, all $v0$ rules are clone rules. This results in $v0$ microrules being generated and installed for every flow, right after the $v0$ update. To update with $v1$ rules, they are installed and they too are made clone rules; $v0$ rules are deleted at the same time ($v0$ microrules are not deleted now). Now a new flow will create a new microrule and $v1$ rules will apply while old flows will match their existing microrules and so $v0$ rules will apply. However, this solution has the following issues: 1) Microrules are always present and

the overheads of these microrules will affect the performance of a switch. 2) It is unclear when the microrules must get deleted. 3) Since the rules are versioned, all the tables in all the affected switches need to be modified for one RU, thus making the solution inefficient and limiting the number of disjoint RUs to 1. Alternately, all the paths affected by an RU may be found and rules with the new version installed only in all the tables in all the switches in those paths, which is computation intensive and inefficient, especially so if wildcarded rules are used in switches. 4) This solution does not consider synchronizing forward and reverse flows. 5) Cloning rules is not supported on real switches. Softcell [61] is the only existing solution as far as we know, that performs PFC updates considering forward and reverse flows, but it uses *software switches* at the edge that can generate one rule per flow and is meant only for a specific network architecture for a cellular network.

Existing non-PFC solutions: Instead of a PFC update, a loss-free and order-preserving state migration (LOSM) [130, 45], intended for scaling NFs in and out, where the state of all flows and the flows themselves are moved consistently with the migrated state to the new NFs may be considered. Another solution, TFM, decouples state migration and packet transfer using a TFM Box, developed as a virtual NF implemented on the source and destination NFs, thus impacting scalability. It also incurs additional updates to switches to forward packets to the TFM box. However, these solutions are specified only for individual NF instances and *not for NF chains* [136]. A simple extension of the state of the art mechanism intended for NF migration to be able to support movement of a flow from one service chain to another (such as TFM [130]), is complex and inefficient as it requires movement of packets already in transit involving several NFs, and buffering packets. Another LOSM solution, OpenNF [45], forwards packets to the controller and requires further consistency preserving updates, for steering flows in and out of the controller, besides making the controller a bottleneck. Stratos [44] migrates the affected NF instances themselves to the new path, in the case of a network bottleneck, which is an expensive solution. Besides, it requires one rule per flow. Split/Merge [107] also provides state migration but it is not loss-free and order-preserving. E2, which does not require state migration, routes all flows through the old NF first [103], thus expecting NFs to have high processing capability; additionally, it requires installation of rules other than those warranted by the RU. All the above require changes to NFs themselves [52]. While none of the above migrates *switch* states, SwingState [82] does, but it does not migrate

the states of NFs. Woo et al. [132] use a Distributed State Object to store states associated with flows to facilitate easy migration of flows. However, this requires all NFs to be written using their framework and does not consider stateful switches. Using a PFC RU circumvents the need for migrating the states of switches as well as NFs. Additionally, any number of NFs and stateful switches may be present in the path of the flow.

Session based solutions: Nicutar et al. [98] propose using MPTCP to insert middleboxes into sessions. But this requires a TCP session to be first established before insertion of a middlebox. If a particular session needs to be blocked, the middlebox/NF is not already in place to take that decision, thus preventing the inserted NFs from blocking sessions if required and causing security issues. Dysco [135] uses a session level protocol that requires an agent installed in all hosts, including NFs, and therefore does not consider stateful switches in the path. Also, it requires buffering of packets at the agents during migration, like the LOSM solutions.

Timed update and flow migration solutions: Existing timed update solutions [138, 91, 92] do not preserve PFC or tolerate practical time asynchrony, scheduling inaccuracies of switches or variable controller-switch delays. Ludwig et al. present an algorithm to [79] preserve waypoint enforcement only for *packets*, and do not preserve PFC. A large body of work (Table 3 in [41]) deals with migrating flows while preventing congestion. However, these do not preserve PFC.

In summary, we need PFC RUs to modify only the affected switches and the affected rules, and support wild carded rules. They must be able to perform an unlimited number of concurrent disjoint updates. They must not require changes to NFs or buffering of packets or addition of new rules, yet must be practical. They must be able to support migration of flows from more than one NF (a service chain) or a set of stateful switches or a combination of both. No existing solution addresses all the above.

2.8.2 Adding or deleting NFs:

In Figure 2.1 (b), two NFs - an IDS and a Firewall (FW) - are added to an existing Service Chain. The switches s_{1b} and s_{1c} are the affected switches because rules need to be added to them to forward packets to and from the new NFs added. The IDS, added to detect SYN flooding attacks, examines the number of SYNs and matching FINs (or RSTs) [127] and if there is a mismatch between them that is statistically significant, issues an alert.

Therefore only new flows must pass through the IDS, as an old flow will cause a mismatch. Similarly, removing an NF from an SC requires allowing the old flows traversing the NF to be completed before the NF is removed. Both the cases require PFC RUs.

Slick [9] decomposes middleboxes or NFs into *elements* of fine granularity. For example, an element could be a module calculating a packet checksum. It allows programmatic specification of what elements should operate on a flow. Placing the elements and steering of traffic through them is done by the run-time of Slick. However, it does not support wild carded rules - it installs one rule per flow - and the mechanism to maintain PFC while adding or deleting elements is unspecified. (If the old and new paths have only stateless NFs [64] and switches, inserting a new NF (for example a Deep Packet Inspector that operates at the packet level) will require only a PPC RU.)

2.8.3 Network Virtualization:

Let us assume that a virtual network has a switch s_b connected to an IDS. s_b is mapped to two *physical* switches, s_{1b} and s_{2b}, as shown in Figure 2.1(c). The forward flow of a connection traverses $s_{1a} - s_{1b} - s_{1c}$ and the reverse flow traverses $s_{2c} - s_{2b} - s_{2a}$. Initially, s_b has a policy P_1 that forwards all packets to the IDS. Later, it installs a new policy P_2 on s_{1b} and s_{2b} (the affected switches) that whitelists all packets destined to a port, thus bypassing the IDS. If P_2 is applied to a packet p, say a SYN, in the forward flow f_f, the packet is not forwarded to the IDS. Now the SYN+ACK of f_r reaches s_{2b}. If P_2 is not yet effective there, the packet is forwarded to the IDS, causing an inconsistent state in the IDS. On the other hand, if P_2 is not yet effective at s_{1b} when the SYN of f_f traverses it, the packet is forwarded to the IDS. If P_2 is effective for f_r, when SYN+ACK reaches s_{2b}, the packet bypasses the IDS, causing the IDS to issue an alert erroneously, if there are many such occurrences.

COCONUT [47] solves a subset of the problems arising due to network Virtualization that require *only weak causal consistency* (weaker than PFC) and not PFC. In this solution, both rules and packets are assigned version numbers and if a packet of version v traverses a switch whose highest version of a matching rule is $v + 1$, it applies that rule to the packet. Because of this, packets belonging to existing old flows will also switch to the new version, thus *not* preserving PFC. To preserve connection affinity, a shell, which interfaces with the hypervisor, stores the latest version of a packet as it exits the network

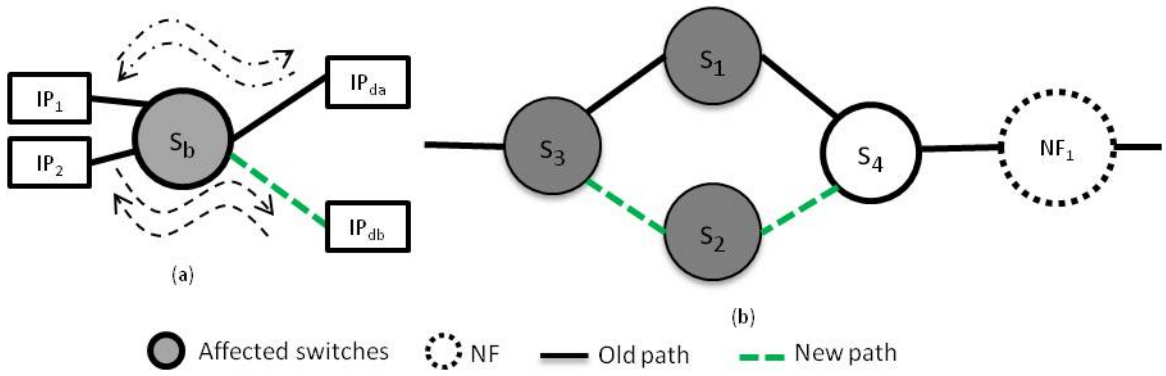


Figure 2.2: Load Balancing switch and re-ordering of packets

and on a packet of the reverse flow, it appends the stored version number. When an update is completed, the shell is instructed to clear the tag associated with a version of the update and also to stop tagging for the time during which packets are getting deleted. Thus every update requires co-ordination with the shell. Besides, since it uses per-RU tag bits, it provides only limited concurrency of disjoint RUs.

Another solution [84] uses packets to carry the states of switches they traverse, updating switches along the way, providing *event-driven consistency*, but not PFC. In an event-driven consistent update, a new set of rules will become effective only after an event occurs in the network. This solution requires flows to be present in the network, in the absence of which (or during their quiescent periods), the controller broadcasting the necessary state information to switches is suggested, an expensive operation. A PFC update will not solve all the problems posed by McClurg et al. [84], and their solution will not solve the problems presented here.

The inconsistency issues during RUs in virtualized networks described above can be solved if the RUs are per-flow consistent. PFC RUs need to progress regardless of whether flows are present or active in the network during the time of the RU.

2.8.4 Load Balancing:

Let a Load Balancing (LB) switch s_b be connected to two servers IP_{da} and IP_{db}, as shown in Figure 2.2 (a). Services offered by IP_{da} and IP_{db} are accessed through a gateway switch (not shown), which offers a single IP address to external clients [128]. The gateway switch and the LB switch are within a data centre network. s_b examines the source IP address of all the packets (IP_1 or IP_2) it receives and stamps the destination IP address as IP_{da}. To

reduce the load on IP_{da}, the controller issues an RU to s_b (the affected switch) to direct a set of flows originating from IP_2, to a new server IP_{db} (shown in dotted lines). If the flows that already exist to IP_{da} are directed to IP_{db}, since the latter does not have an existing connection with the sources of these flows, those flows get terminated abruptly. If a PFC compliant RU is used, the ongoing flows to IP_{da} will naturally complete, while the flows that begin after the RU is effective will be directed to IP_{db}.

Wang et al. [128] provide two solutions for updating a load balancer. In the first solution, assume that all packets from source IP address $0*$ need to be sent to server replica IP_{db}, instead of IP_{da}. During the update, the controller installs transition rules that direct *all* packets with source address as $0*$ to the controller. The controller examines the next packet of all such connections - if it is a SYN, it is a new connection; otherwise, it is an existing connection. The controller then installs microrules with soft timeouts, that direct the new connections to IP_{db} and the old connections to IP_{da}. The controller examines all packets with source address $0*$ for 60 seconds to see if there is any missed connection; if not, it installs the rule that directs packets with source address $0*$ to IP_{db}, at a lower priority than the microrules. In the second solution, the controller first installs the new rule that directs all traffic to IP_{db}, at a lower priority. Next, it installs temporary rules with inactivity timeouts, dividing the address space of 0^* into several parts that direct all the traffic to the old replica, at a higher priority, and deletes the old rule. Upon timeout, the temporary rules are deleted and the new rules take effect. While the first solution has the disadvantage of loading the controller, the second causes the update to be ineffective for a fraction of the flows as long as at least one flow exists in that fraction, either because an old flow is long-lasting or because new flows keep matching this rule.

During an RU, no rule must delay the time at which the RU is effective for any new flow. Moreover, there should be no need to buffer packets or install additional rules.

2.8.5 Packet re-ordering:

In Figure 2.2(b), an RU changes the path $s_3 - s_1 - s_4 - NF_1$ to $s_3 - s_2 - s_4 - NF_1$ of a forward flow from s_3 to NF_1. All the switches except s_4 are affected by the RU - s_3 and s_2 are affected because rules must be added and s_3 and s_1 because old rules must be deleted. A packet p traversing the new path may reach s_4 earlier than the packets sent before p, which are traversing the old path. If NF_1 is a redundancy elimination decoder and p was

encoded with respect to packets that preceded it [7], the RU will cause NF_1 to function incorrectly. Besides, re-ordering of packets may cause TCP to erroneously conclude that there is congestion in the network, thereby degrading application throughput [75, 73]. Switches take a lot of care not to re-order packets [114], but updates could result in packets that are re-ordered and if updates need to be performed frequently, the impact will be high. An RU that preserves PFC prevents re-ordering of packets due to the RU itself.

Foerster et al. [41] provide a survey of SDN update algorithms.

Part I

Per-Packet Consistent Updates

Chapter 3

Two Algorithms for Per-Packet Consistent Updates

3.1 Introduction

We have examined in detail the need to preserve per-packet consistency (PPC) in Chapter 2. Two-phase update (2PU) [111], the seminal algorithm to preserve PPC, requires changes to all the switches in the network, even if the RU is intended only for one switch. It is underspecified on two matters: 1) how to detect when the last packet of the old rule set has left the network and therefore exactly when to delete the old rules 2) preserving all-or-nothing semantics of the RU. We describe an algorithm, **E2PU-SRT** (Enhanced 2 Phase Update with Software Rule Tables), enhancing the two-phase update, to handle 1) and 2) above. It also specifies how to effectively use a software cache to supplement the TCAM, during an RU. However, E2PU-SRT does not support concurrent updates. The second algorithm that we describe in this chapter, **CCU** (Concurrent Consistent Updates), supports concurrent updates that are disjoint. In addition, it requires updates to only the affected switches and the affected rules and all the ingresses of the network.

3.2 Enhanced 2 Phase Update with SRT (E2PU-SRT)

3.2.1 Switch Model

Since TCAM space is scarce, it is desirable to use a software cache to supplement the TCAM such that rules may be installed either in the TCAM or in a rules table implemented in software, called the Software Rules Table (SRT). Since switching in a TCAM is faster, all the rules that are frequently accessed may be periodically moved to the TCAM while the remaining rules are stored in the SRT. Adding (removing) rules to (from) a TCAM incurs high overhead [62], while doing the same to an SRT is faster. For this algorithm, to speed up updates, we assume that there is a Software Rules Table (SRT) associated with the TCAM such that the switch checks if a matching rule exists in the TCAM first and if it does not, it checks the SRT. For moving rules into and out of the TCAM, algorithms such as CacheFlow [67] may be used. Other than this, the switch model in section 1.3.1 of Chapter 1 is used.

3.2.2 Algorithm at the control plane

Consider an SDN whose switches have a set of rules of version 0 ($v0$) that needs to be updated to version 1 ($v1$). E2PU-SRT addresses failures F1 through F6 and the other issues identified in section 2.7.1, while preserving PPC. Figure 3.1 illustrates the algorithm at the control plane, assuming that ingress switches do not act as internal switches for any flow affected by the update. The algorithm at the control plane is as follows:

1. The controller sends "Commit" with the new rules to all the affected switches. All the affected internal switches (this could include ingress switches that receive rules by virtue of them acting as internal switches for other paths that belong to this update) install the new rules into the SRT. The ingress switches do not yet install the rules that tag packets with $v1$ or policy rules, but store them internally.

2. Each switch that processes "Commit" sends back "Ready to Commit".

3. The controller receives "Ready to Commit" from all the switches and then and only then sends "Commit OK" to *all* the affected switches. As soon as the ingress switches receive "Commit OK" they stop sending packets tagged $v0$ and switch over

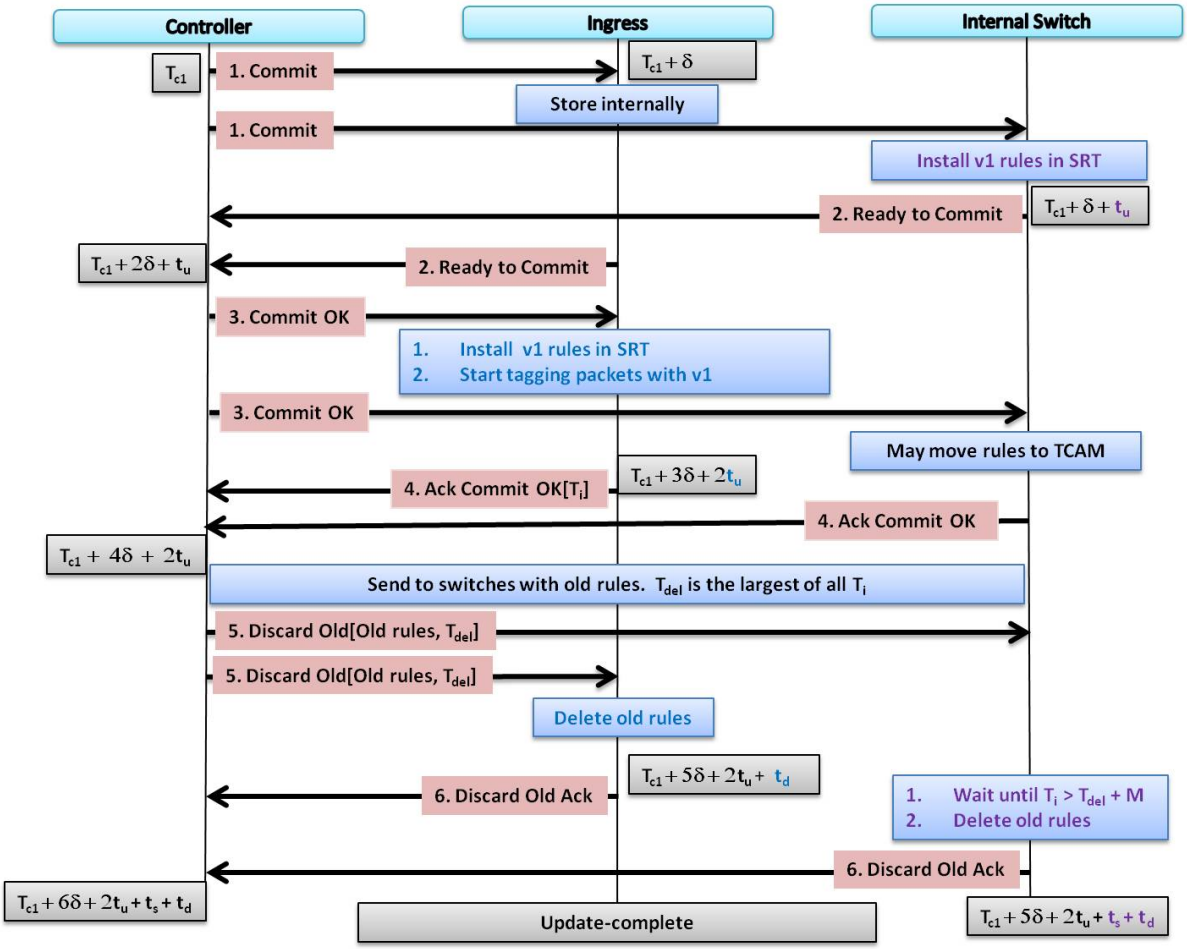


Figure 3.1: E2PU-SRT: Algorithm at the control plane

to $v1$. After receiving "Commit OK", a switch may move the $v1$ rules to TCAM at any time.

4. Each switch that processes "Commit OK" sends "Ack Commit OK" to the controller. Each *ingress* switch sends the current time with this message, called T_i.

5. After receiving "Ack Commit OK" from all the affected *ingresses*, the controller notes the latest value of T_i received and saves it as T_{del}. Now it sends "Discard Old" to all the switches where rules need to be deleted. It sends T_{del} and the $v0$ rules to be deleted as a part of "Discard Old".

6. When each internal switch receives "Discard Old" it deletes the list of rules received in "Discard Old", whenever its current time $T_i > T_{del} + M$, where M is the maximum lifetime of a packet within the network. The ingresses delete immediately.

7. Each switch that processes "Discard Old" sends a "Discard Old Ack". When the controller receives all the "Discard Old Ack" messages the update is complete.

When a packet is tagged $v0$ ($v1$) at the ingress, rules that check for $v0$ ($v1$) are applied on that packet as it traverses the network.

The v_0 rules can be in the TCAM or the SRT, for the internal switches. However, the algorithm assumes that the v_0 rules are in the SRT, for all ingresses. If they are in the TCAM, upon receiving *"Commit"*, those ingresses must install v_1 rules in the SRT, at a lower priority. Then, upon receiving "Commit OK", they must delete the v_0 rules from the TCAM. This variation is referred to as E2PU-SRT′.

It is possible to implement sending and acknowledging a message in Openflow using the barrier command [101].

After T_{del}, no $v0$ packets are injected by any ingress into the network. After M units of time after that, all the $v0$ packets that were in the network would have left it. Therefore deleting $v0$ rules at this point will not cause any packet to be dropped.

3.2.3 Handling Failures and Application Aborts

If one (or more) of the switches is unable to process a "Commit" due to any of the failures F1 through F6 in section 2.7.1, the controller will not receive "Ready to Commit" at all, or on time. Similarly, the controller may not receive "Ack Commit OK" from some ingresses.

The controller now performs the following actions for this update: 1) suspend the update 2) preserve the tag used for the current update 3) inform the application. The application can choose to abort the update, in which case the controller aborts the update. Since $v1$ rules are stored in the SRT, deleting them will not be time consuming. The application, instead of aborting the update, may send some modifications to the update, such as addition or deletion of rules to new or existing switches. This is more likely if the update is partially installed and partially effective, as in the case where the controller has received "Ack Commit OK" from only some of the ingresses. In that case, the controller sends a "Commit" for adding $v1$ rules, using the tag preserved for this application. The existing $v0$ rules and the $v1$ rules that were installed earlier (because they are superseded by a new set of $v1$ rules) may get added to the list of rules to be deleted (later, using "Discard Old"). The switches may thus receive more than one "Commit" and "Commit OK" messages with the same tag, before receiving a "Discard Old". If there is a failure after the controller receives "Ack Commit OK" from all ingresses, the application must issue a new update, if required, as all the new rules have now been successfully installed and are effective. If the application requests the controller to abort the update before the controller has sent any "Commit OK", the controller aborts the update. If the controller aborts the update, all $v1$ rules are deleted.

All subsequent algorithms in this thesis use the same set of messages. The nodes to which the messages are sent and the contents of the messages vary, depending on the algorithm. In all cases, to provide an all-or-nothing semantics, timers are required to be started at the controller. This and other error-handling semantics are not presented to simplify the explanation.

Alternate Designs for Deleting the Old Rules: It is possible for the *controller* to wait for an additional time of M after receiving the acknowledgement from the last ingress switch and then send a message to all the switches requesting them to delete the old rules *immediately*. However, the last ingress switch may take some time to send "Ack Commit OK" to the controller, because it is far away from the controller or because the switch is just slow. Each internal switch only needs to wait until its time exceeds $T_{del} + M$, which is faster than the controller waiting for M units of time *after* it receives the last "Ack Commit OK". It is possible to further optimize this by the controller sending "Discard Old" messages as soon as it receives "Ack Commit OK" from an ingress, to all

the internal switches that accept $v0$ packets only from that ingress.

3.2.4 Switches without an SRT

If the switches in a network do not have an SRT, when an internal switch receives a "Commit" with new rules, it installs the rules in the *TCAM* and sends back "Ready to Commit". The controller, on receiving "Ready to Commit", needs to send "Commit OK" to *only the ingresses*. The ingresses install the new $v1$ rules in the TCAM. The rest of the algorithm remains the same. If the controller wishes to abort the update, the switches that have already installed the rules need to incur the overhead of deleting them from the TCAM. Section 3.2.5 gives further comparisons.

3.2.5 Analysis of E2PU-SRT

The parameters of interest during a PPC are: 1) **Parameter 1**: Duration for which the old and new rules exist at each type of switch 2) **Parameter 2**: Duration within which new rules become usable 3) **Parameter 3**: Message complexity - the number of messages required to complete the protocol and 4) **Parameter 4**: Time complexity - the total update time. Parameters 3 and 4 are along the lines of complexity measures in distributed systems [18].

The purpose of the analysis is to understand what the above depend upon.

The symbols used in the analysis is as per Table 3.1. It is assumed that the time taken for the sum of the propagation times and switch delays between the controller and the switches is uniform (δ). Let the time taken for all insertions (t_u) and deletions (t_d) be uniform and let the processing time at each switch be negligible. The time at which the last switch sends "Ready to Commit" is $T_{c_1}+\delta+t_u$, assuming the sum denotes the longest time taken. Let the number of rules that need to be removed (n_o), added to switches in general (n_n) and added to the ingress to meet its ingress functions (n_i) be uniform across switches.

We need to consider different kinds of switches while evaluating various parameters: Case 1) ingress switches, with Case 1.1 where the ingress switch is not an internal switch and Case 1.2 where the ingress switch is also an internal switch, Case 2) internal switches where new rules do not need to be installed but old rules need to be removed, Case 3)

Table 3.1: Symbols Used in the Analysis of E2PU

Symbol	**Meaning**
T_{c_1}	Time at which the controller sends "Commit"
δ	The message transmission time between the controller and a switch
t_u	The time taken to insert rules in a switch (TCAM or SRT)
t_d	The time taken to delete rules from the SRT
t_{dt}	The time taken to delete rules from the TCAM
t_s	The time for which a switch waits after it receives "Discard Old" and before it deletes rules
n_o	The number of old rules that need to be removed
n_n	The number of new rules that need to be added
n_i	The number of new rules that need to be added to the ingress, to meet its ingress functions
k_o	The number of switches where new rules do not need to be installed but old rules need to be removed
k_n	The number of switches where only new rules need to be installed (this includes such ingresses too)
k_c	The number of switches where old rules need to be removed and new rules need to be added (this includes such ingresses too)
k_i	The number of ingress switches where new rules need to be installed

internal switches where only new rules need to be installed and Case 4) internal switches where old rules need to be removed and new rules need to be added. For Case 1.2, to simplify the presentation, it is assumed that there are no old internal rules to be deleted. For all ingresses, it is assumed that old ingress rules need to be removed and new ingress rules added.

The time elapsed at each stage of the RU is shown in Figure 3.1. The results of the

Table 3.2: Analysis of E2PU

Parameter	**E2PU-SRT**	**E2PU-SRT′**
1, Case 1.1	$2\delta + t_d$	$2\delta + t_u + t_{dt}$
1, Case 1.2	$4\delta + t_u + t_d$	$2\delta + t_{dt}$
1, Cases 2,3	NA	NA
1, Case 4	$4\delta + t_u + t_s + t_d$	$4\delta + t_u + t_s + t_d + t_{dt}$
2	$3\delta + 2t_u$	$3\delta + 2t_u + t_{dt}$
3	$2k_o + 6k_c + 4k_n$	$2k_o + 6k_c + 4k_n$
4	$6\delta + 2t_u + t_s + t_d$	$6\delta + 2t_u + t_s + t_d + t_{dt}$

analysis are shown in Table 3.2.

Parameter 1: Duration of overlap

Case 1.1: When the ingress switch receives a "Commit", there are n_o rules in the switch. At $T_{c_1}+3\delta+2t_u$, there are n_o+n_i rules. At $T_{c_1}+5\delta+2t_u+t_d$, there are n_n+n_i rules. Therefore the time for which the old and the new rules coexist in the switch is $T_{c_1}+5\delta+2t_u+t_d$ - $(T_{c_1}+3\delta+2t_u) = 2\delta+t_d$.

Case 1.2: At $T_{c_1}+\delta+t_u$, there are n_o+n_n rules. At $T_{c_1} + 3\delta + 2t_u$, there are $n_o+n_n+n_i$ rules. At $T_{c_1} + 5\delta + 2t_u + t_d$, there are n_n+n_i rules. Therefore the time for which the old and the new rules coexist in the switch is $4\delta+t_u+t_d$.

For cases 2 and 3, there is no overlap of rules.

Case 4: At $T_{c_1}+\delta+t_u$, there are $n_o + n_n$ rules and at $T_{c_1}+5\delta+2t_u+t_s+t_d$ there are n_n rules. The duration of overlap is $4\delta+t_u+t_s+t_d$.

Since the internal switches have a larger overlap duration, they need to have bigger SRTs or TCAMs than the ingresses.

Parameter 2: Duration within which new rules become usable: After $3\delta+2t_u$ units from the beginning of the update, the ingress switches to the new rules and they are usable.

Parameter 3: Message Complexity: For E2PU-SRT, k_n+k_c messages of type "Commit", "Ready to Commit", "Commit OK" and "Ack Commit OK", and k_o+k_c messages of type"Discard Old" and "Ack Discard Old" are sent, totalling to $2k_o+6k_c+4k_n$.

Parameter 4: Total Update Time: The total update time is $6\delta + 2t_u + t_s + t_d$.

If v_0 rules of an ingress are in the TCAM, they need to be deleted from the TCAM, taking time t_{dt} and updated in the SRT, taking time t_u. Therefore Parameter 1 changes to $T_{c_1} + 3\delta + 2t_u + t_{dt}$ - $(T_{c_1} + \delta + t_u) = 2\delta + t_u + t_{dt}$ for case 1.1, $T_{c_1} + 3\delta + 2t_u + t_{dt}$ - $(T_{c_1} + \delta + 2t_u) = 2\delta + t_{dt}$ for case 1.2 and for case 4, it increases by t_{dt}. Parameters 2 and 4 increase by t_{dt}. The parameters if v_0 rules of an ingress are in the TCAM are shown under E2PU-SRT′ in Table 3.2.

3.2.5.1 Observations:

1. For networks without an SRT, parameters 1,2 and 4 worsen because the value of t_u increases significantly and $t_{dt} > t_d$, though the formulae remain the same. However, the number of messages used reduces to $2k_o + 4k_c + 2k_n + 2k_i$, as there is no need to send Commit OK and Ack Commit OK to the internal switches.

2. If v_0 rules for all the switches are in the SRT, the parameter values reduce as the time to delete rules reduces. If some of the internal switches (but not ingresses) have v_0 rules in the TCAM, their deletion times will dominate parameters 1 and 4.

3. Since a mix of rules in TCAM and SRT is going to be a practical scenario, to reduce these parameters, mechanisms for fast deletions, such as to disregard a rule from a switch by quickly marking it as deleted and physically removing it later, must be considered.

4. For small networks the update and delete times will dominate δ and t_s (the network diameter). For large networks, with all rules in the SRT, δ and t_s will have a larger influence on the parameters.

5. For PPC with SRT, the new rules get installed fairly quickly and the old rules are removed as soon as is feasible.

6. 2PU [111] does not specify a method to delete old rules and it does not specify whether every message that the controller sends to switches is acknowledged. Therefore a comparison of E2PU with 2PU is not straightforward. Due to these reasons, Parameters 1 (duration of overlap of old and new rules), 3 (message complexity) and 4 (total update time) of E2PU cannot be compared with that of 2PU. In fact, the total update time of 2PU cannot be analytically determined, as it is not fully

specified. However, assuming that 2PU expects an acknowledgement from all the switches, E2PU-*SRT* fares better than 2PU for Parameter 2 (duration within which new rules become usable), as the value of t_u is lesser for E2PU-SRT.

A qualitative comparison of E2PU with other algorithms that preserve PPC is given in Table 2.1 of Chapter 2.

3.2.6 Summary of E2PU

In E2PU-SRT, we identified areas where the basic update algorithm for SDNs is under-specified and described enhancements for update algorithms for PPC, exploiting the availability of an SRT. We also analyzed the algorithm quantitatively.

For real implementations, it is desirable that rules in every switch in the network are not modified for every update. One method to accomplish this is for the controller to identify the exact paths affected by every rule change [71], whenever practically possible, and modify switches only along those paths. Another method is to modify *only* those switches where there is a genuine rule change, by installing $v1$ rules always in the SRT at a higher priority compared to the $v0$ rules and matching with the rules in the SRT first. All ingresses tag all incoming packets with $v1$. $v0$ rules do not check the version numbers of packets. The version number field of $v1$ rules is set to don't cares only when $v0$ rules are deleted. This is the basic idea of the algorithm CCU, described in the next section. Moreover, CCU also addresses how concurrent disjoint updates from a controller can be executed on a network, ensuring an all-or-nothing semantics.

3.3 Concurrent Consistent Updates

This section describes a general per-packet consistent update algorithm that supports concurrent *disjoint* updates.

E2PU-SRT (section 3.2) has two properties: 1) it does not make assumptions about the topological properties of the update or the nature of flows and therefore works for all situations (only deletion, only insertion, or both, of flows using wildcarded rules) 2) it preserves PPC. However, it does not allow concurrent disjoint updates. This is because every update affects every rule in every switch: thus effectively, there are no disjoint updates, even though, in reality, the updates may be disjoint.

If all the paths affected by an update are known in advance, using the mechanisms identified in Veriflow [71] or CCG [140], then concurrent updates preserving PPC are possible, using E2PU-SRT. Suppose for an update, an internal switch requires this rule to be inserted: "*tcp_port=80, forward2*". This rule affects a very large number of paths in a network. Identifying the number of paths that affect a large number of paths is not possible in a manner that is fast enough for real implementations [71]. Only for the scenarios where the paths can be identified, disjoint updates can be installed in parallel. Moreover, every switch in the path needs to be modified, whether that switch is affected or not.

CCU accomplishes the following: C1) it preserves PPC C2) it allows concurrent disjoint updates C3) it makes no assumptions on the sequence of updates or the nature of rules and is therefore general C4) it minimises the number of internal switches to be updated by restricting the update to only those internal switches that require a genuine rule change. C5) it provides a trade off between concurrency and packet header overhead.

3.3.1 Switch Model

An internal switch has a rule table implemented in an SRT, "in series" with a rule table implemented in a TCAM. A packet, on entering an internal switch, is first matched with the rules in the SRT. If there is no match, it is forwarded to the TCAM. In implementations that do not have an SRT, a TCAM may be used, without any change to the algorithm. Every rule has two fields associated with it, a *version number*, initialised to don't cares, and *limit*, initialised to the maximum value that the field can accommodate. Other than this, the switch model in section 1.3.1 of Chapter 1 is used.

3.3.2 Concurrency Requirements

We envisage a network model where applications from either the same controller or different controllers issue *disjoint* updates. In E2PU-SRT (section 3.2), each update gets a unique tag from the controller (or a central entity in the case of multiple controllers). After the controller receives "Ready to Commit" from all the switches that process "Commit" the update is said to be *stage1-complete*. After the controller receives "Ack Commit OK" from all the switches that process "Commit OK", the update is said to be *stage2-complete*.

After the controller receives all the "Discard Old Ack" messages, the update is *complete*.

The controller sends a "Commit" to the switches as soon as a tag is allocated. Let there be several "Commit" messages sent, belonging to different versions. If an update $n+1$ is stage1-complete before an update n, it must be possible to proceed with the rest of the update for $n+1$, without waiting for n to be stage1-complete. Ideally, this must be possible in all situations. Practically, this depends on the level of concurrency.

An RU RU_1 that conflicts with an update RU_2 may begin as soon as RU_1 is complete and not earlier. An application will issue RU_2 only after RU_1 is complete.

3.3.3 Rule installation in internal switches

All new rules are first inserted into the SRT, at a higher priority than the old version of the rules. New rules check packets for a specific version number, which must always be less than *limit*. Whenever the old version of rules, if any, is deleted, the version field of the current version of rules is changed to don't cares so that they cease checking packets for a version number. The old rules will typically be in the TCAM (alternately they can be in the SRT) but always at a lower priority than the new rules.

3.3.4 Version tagging of packets

Each packet has a version tag, followed by 4 (this can be any number) bits, called the *status* bits. The status bit denotes the status of the update, with the most significant bit denoting the status of the update whose tag is 1 less than the value of the version tag, the next significant bit 2 less than the value of the version tag and so on. When the switch sees a packet with the status bit set to 1, the update corresponding to that version is in "Commit OK sent". This means that rules belonging to that version are effective and that packets, if any, are getting switched according to those rules. If the version tag of a packet is n, all updates with versions less than or equal to $n-4$ are in the state "Commit OK sent", or later. For example, $10 : 0110$ indicates that update versions 10, 8 and 7, and 5 and below, are in the state "Commit OK sent" or later. Update versions 9 and 6 are not stage1-complete. The updates for which the status bit is set are called *companion updates*. The number of status bits may be increased to improve concurrency or decreased to reduce the overhead.

3.3.5 Algorithm at the control plane

Assume that a set of rules with version number n need to be installed. *current* denotes the version tag currently sent in packets and *status* is the half-byte that denotes the status bits that the packets are tagged with.

The controller queues update requests from all applications in *app_queue*. The controller also maintains a table, *update_table*, which has *the state of the update*, for each update. The update states stored are **stage1-complete**, **"Commit OK" sent**, **stage2-complete** and **"Discard Old" sent**. The updates that are queued are disjoint.

The messages exchanged are the same as in E2PU-SRT. Alterations are made for generating and processing packets with version number and status fields, for concurrent updates, as shown in Figure 3.2.

0. The controller checks if there are messages from the switches, associated with an ongoing update. If there are messages, it goes to the appropriate step below, depending on the state of the update. If not, it checks whether there are any update requests in *app_queue*. If there are any requests, it goes to step 1.

1. The controller retrieves the first update request from *app_queue*, gets the next available tag for this update, say n, and sends "Commit" with the new and old rules to all the affected switches. All the affected internal switches (this could include the ingress switches that receive rules by virtue of them acting as internal switches for the other paths that belong to this update) install the new rules into the SRT. The new rules are such that they have higher priority than any version lesser than n. The affected internal switches set $limit = n$, for every *old* rule received in Commit. The ingress switches do not yet install either the rules that tag packets with n or the policy rules of version n, but store them internally.

2. Each switch that processes "Commit" sends back "Ready to Commit".

3. After the controller receives "Ready to Commit" from all the switches, it marks n as stage1-complete in *update_table*. It calls the procedure resume_update() in Algorithm 1 to resume updates, which is described in detail in section 3.3.6. The procedure sends "Commit OK", if required, to *all the ingress switches*. The controller also sends the value of the status bits to be sent, along with the correct tag, in

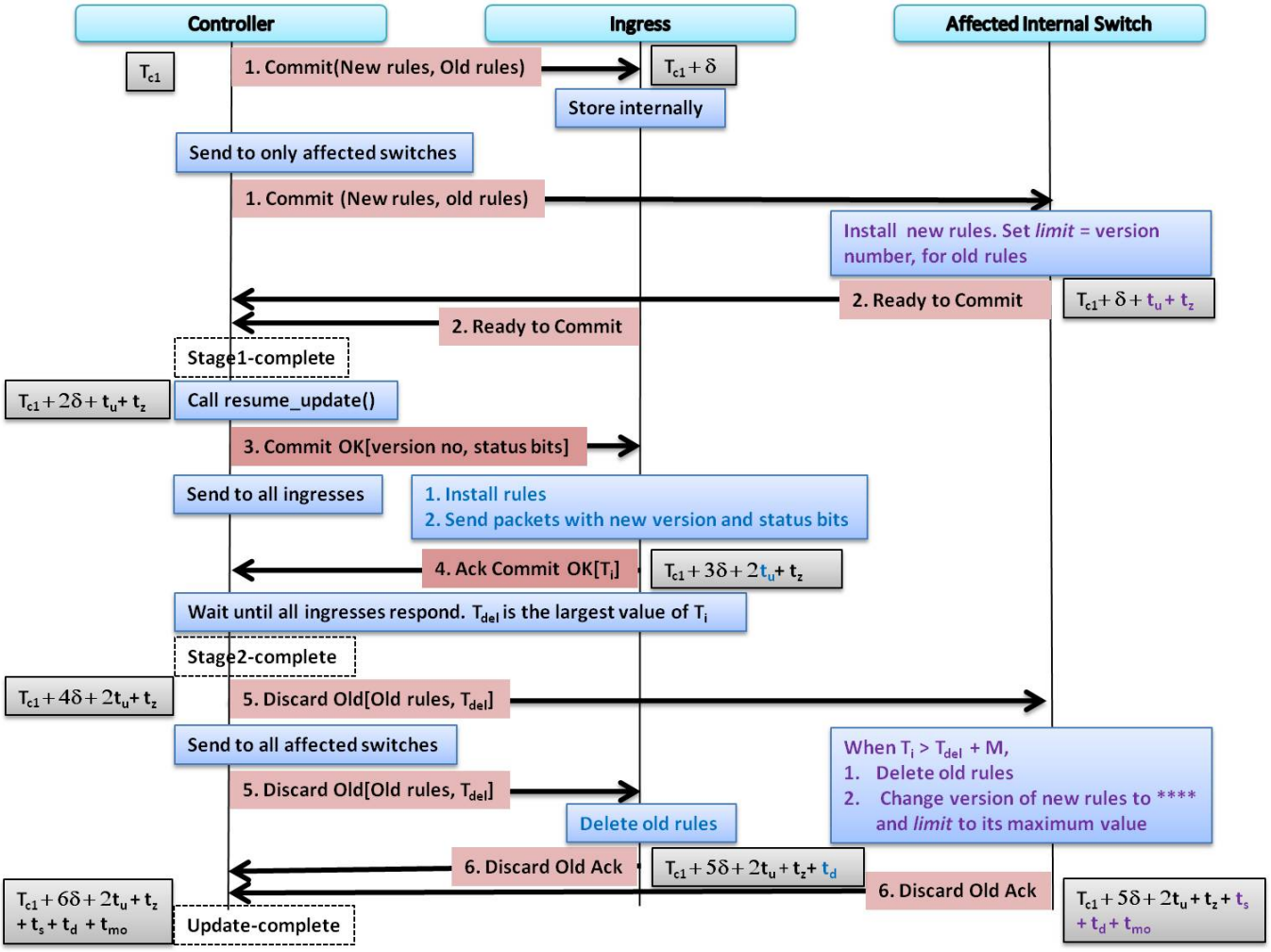


Figure 3.2: CCU: Algorithm at the control plane

"Commit OK". As soon as the ingress switches receive "Commit OK", they install new rules of version n, if any, stop sending packets tagged with the previous version and switch over to the new version. They also modify the status bits in the packet header. Let us assume that a "Commit OK" with n as the version is sent at this stage. (It is possible that due to previous updates not being stage1-complete, the update n is stalled, in which case, "Commit OK" is not sent for version n).

4. Each ingress that processes "Commit OK" sends "Ack Commit OK" to the controller. Each ingress sends the current time with this message, called T_i.

5. After receiving "Ack Commit OK" from all the ingresses, the controller notes the latest value of T_i received and saves it as T_{del}. It marks n and its companion updates that are not yet marked stage2-complete, as stage2-complete, in *update_table*. Now it sends "Discard Old" to *all* the switches where rules were either inserted or deleted or both, for update n and its companion updates, unless already sent, as indicated in *update_table*. It sends T_{del} and the rules to be deleted (old version) as a part of "Discard Old".

6. After each internal switch receives "Discard Old", when the current time of the switch $T_i > T_{del} + M$, where M is the maximum lifetime of a packet within the network, it does the following: a) it deletes the list of old rules received in "Discard Old" 2) it sets the version number field of the version n and its companion updates to don't cares 3) it sets *limit* to its maximum value. These rules may be moved to the TCAM any time from now. The ingresses delete the old rules as soon as they receive "Discard Old".

7. Each switch that processes "Discard Old" sends a "Discard Old Ack". When the controller receives all the "Discard Old Ack" messages the update is complete. The controller deletes the entries belonging to the completed updates from *update_table*. The procedure continues from step 0.

3.3.6 Resuming an update

Algorithm 1 shows the algorithm to send "Commit OK" for updates that are stage1-complete and to move the status window, if it is appropriate to do so.

Algorithm 1 Algorithm to resume updates

1: **procedure** RESUME_UPDATE() ▷ Resumes the next set of pending updates in `update_table` by sending "Commit OK".
2: $size$ = Get number of consecutive updates from $current - 4$, in increasing order, whose bits are set to 1, by checking *update_table*
3: **if** $size \neq 0$ **then** ▷ The new window position is $current + size$
4: $temp$ = Position where the first update less than or equal to $current + size$ is stage1-complete
5: **if** $temp > current$ **then**
6: $current = temp$
7: **end if** ▷ Otherwise no change to *current*
8: **end if**
9: Update *status* by reading *update_table*
10: **if** "Commit OK" not already sent for at least one of *current* or companion updates **then**
11: Send "Commit OK" with *current* as the version and *status* as the status bits
12: State of *current* and companion updates for which the state is "stage1-complete" in *update_table* = *"Commit OK" sent*
13: **end if** ▷ If "Commit OK" sent, do nothing
14: **end procedure**

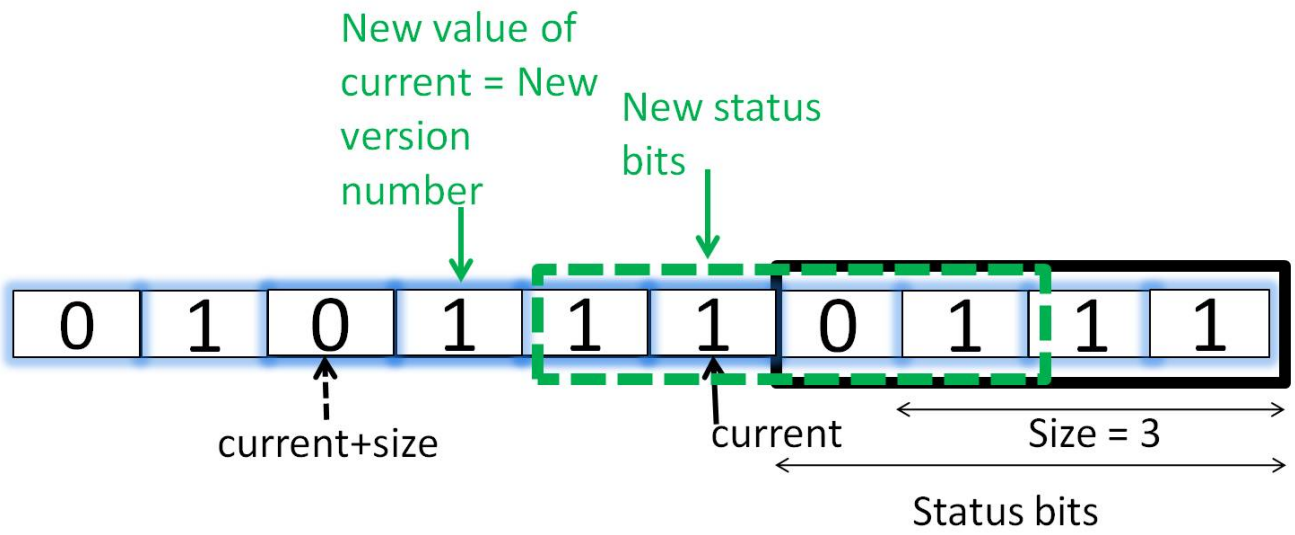


Figure 3.3: Adjusting the status window

The procedure first checks whether the status window can be moved. For that, it determines the number of consecutive updates in *update_table*, starting from $current - 4$, that are stage1-complete (line 2). Let the number be *size*. (*size* can be more than the size of the status field). The status window cannot be moved *size* bits because the version $current + size$ may not be stage1-complete, as illustrated in Fig.3.3. The updates marked "0" in the figure are not stage1-complete and the updates marked "1" are stage1-complete. So the algorithm finds the first version less than or equal to $current + size$ that is stage1-complete (line 4). Now it sets *current* to this value, as long as this is greater than *current*. If there are no updates later than *current* that are stage1-complete and *size* is non-zero, we just set the correct *status* bits to 1, with *current* remaining the same.

When there are no consecutive updates that are stage1-complete, *size* is 0 and the window cannot be moved at all, it is possible that some updates currently within the window become stage1-complete and "Commit OK" needs to be sent for those. The controller sets bits in *status* by determining the status from *update_table*. If, for at least one of the updates in *current* or its companion updates, a "Commit OK" has not been sent, it sends "Commit OK" (line 10).

3.3.7 Algorithm at the data plane

The version number of a packet is denoted as n and that of a rule is denoted as v. When an internal switch receives a packet with version number n, the internal switch attempts

Algorithm 2 Algorithm at the data plane

```
1: procedure PROCESS_PACKET(packet)
2:     ▷ n is the version number of the packet. v is the version number of the rule, if any.
3:     if matched rule has a version number then
4:         ▷ This rule is new.
5:         if (v = n) then
6:             ▷ New rules are installed in all the affected switches
7:             Execute actions
8:         else if (v < n) AND ((status bit in packet = 1) OR (v < n − 4)) then
9:             ▷ New rules are installed in all the affected switches. RUs subsequent to v
   are in progress.
10:            Execute actions
11:        else
12:            ▷ v > n or status bit in packet = 0 for this v
13:            ▷ New rules are installed in this switch but not yet installed in all the
   affected switches
14:            Skip this rule and apply the next matching rule
15:        end if
16:    else
17:        ▷ This rule is not new. It is old or unaffected.
18:        if n ≥ limit then
19:            ▷ New rules have been installed in all the affected switches. However, this
   switch does not have a new rule. The matching rule is an old rule that needs to be
   deleted. Therefore, skip this rule.
20:            Skip this rule and apply the next matching rule
21:        else
22:            ▷ If limit has its maximum value, this is an unaffected rule. Otherwise it is
   an old rule. In either case, the rule can be applied to the packet.
23:            Execute actions
24:        end if
25:    end if
26: end procedure
```

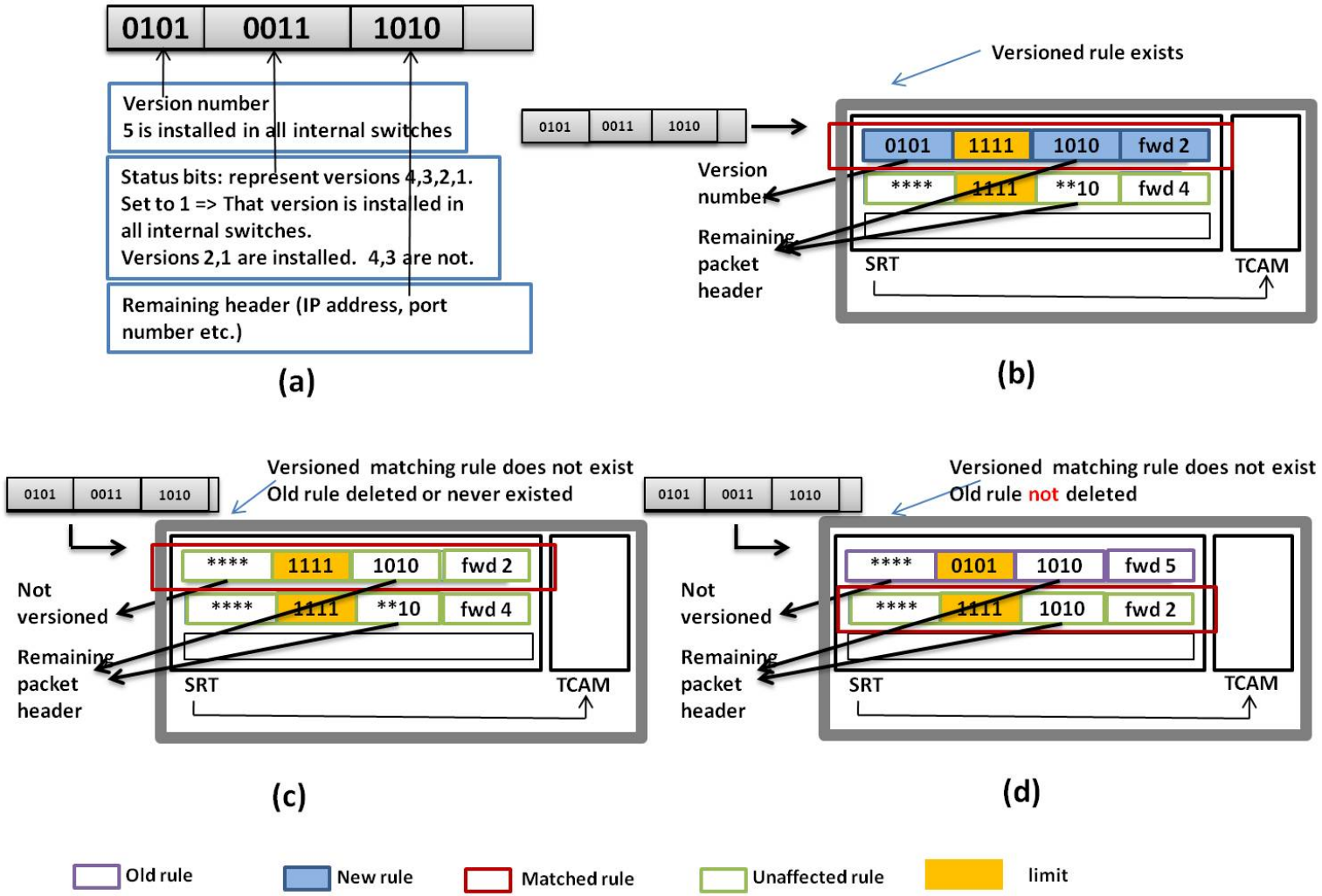


Figure 3.4: (a): A packet with its labels and (b),(c),(d): how a packet is matched

to match the packet with a *valid* rule present in the SRT that has a version number less than or equal to n and applies that rule. A rule is valid if the status bit in the packet associated with the rule version is set to 1 or its version is less than $n-4$. A packet header with $10:0110$ as its version and status, indicates that the rules with versions 10, 8 and 7, and 5 and below are valid for that packet.

When an RU begins, suppose a switch has both new and old rules. Suppose the new rules have been installed and the old rules have their *limit* set to k, where k is the version number of the packet. As the RU for version k begins, an affected packet has a version n, which is less than k. Now the packet skips matching a new rule (line 14 of Algorithm 2) and matches an old rule (line 23 of Algorithm 2), as expected. Suppose $n = k$. This implies that new rules have been installed in all the affected switches. The packet now matches a new rule (line 7 of Algorithm 2, Figure 3.4(b)). As the RU progresses, suppose $n > k$ for another packet. Since all the new rules have been installed for version k, the packet matches a new rule again (line 10 of Algorithm 2, Figure 3.4(b)) and continues to match new rules.

Suppose a switch has only old rules and no new rules (section 1.3.2 of Chapter 1). When $n < k$, n is also less than *limit*. Therefore it matches an old rule, as expected (line 23 of Algorithm 2). A packet whose $n = k$ matches an old rule (line 20 of Algorithm 2, Figure 3.4(d)), but skips it, as it must either match a new rule or an unaffected rule. Now it matches an unaffected rule (line 23 of Algorithm 2). As the RU progresses and n becomes greater than k, this continues to happen until the old rule is deleted. Now the packet matches an unaffected rule (line 23 of Algorithm 2), Figure 3.4(c).

The ingresses tag all the packets with the same version and set the same status bits, upon receiving "Commit OK". For ease of implementation, the instruction for tagging packets with a version number and status bits may be separated from the rest of the instructions for the packet (such as forwarding to a port). A single rule tagging all packets with the desired version number and status bits may be installed in a flow table, implemented in software - thus only this instruction needs change and can be changed quickly when a "Commit OK" is received. All packets to an ingress may match the rule in this table first. The packet may then be forwarded to the next flow table in the ingress, which performs the rest of the actions as required.

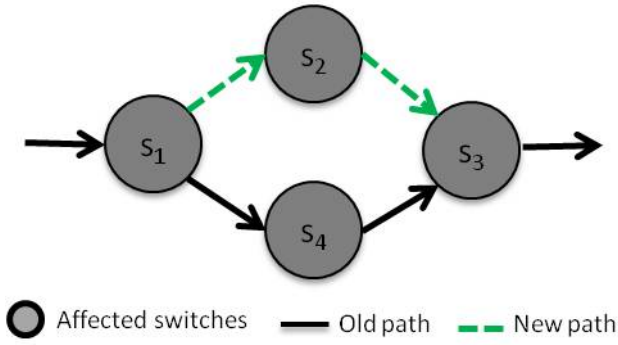


Figure 3.5: Updating only the affected internal switches

3.3.8 Handling Failures in CCU

The controller must start a timer after sending "Commit", for each version number n. If the timer times out without receiving a "Ready to Commit", the application must be informed. The application must instruct the controller whether to delete the installed rules or to proceed with additional updates (that is, modifying the current update) with the same version tag. This must be done to enable later stage1-complete updates to proceed. This can be easily accommodated in the algorithm, but for ease of exposition it is not included.

The error handling for timeouts at the controller while expecting "Ack Commit OK" and "Discard Old Ack" are the same as for E2PU, described in section 3.2.3, and are not further described.

3.3.9 Updating only the affected internal switches

As an example, a rule "*tcp_port=80, forward 2*" needs to be inserted in an internal switch s_1, to reduce the load on another switch s_4, as shown in Fig.3.5. Let two adjacent switches s_2 and s_3 also have new rules on account of this. The new path is shown in broken lines and the old path in solid lines. Let the version associated with this change be n. After the update is stage1-complete, the switches between the ingresses and s_1 (not shown in the figure) match each packet with a rule whose version field has don't cares, that is, an unaffected rule. When a packet with tag n reaches s_1, s_2 and then s_3, it matches a rule with version n, which is the newly inserted rule. Subsequent switches from s_3 to the egress match this packet with unaffected rules, whose version fields are set to don't cares. Thus switches without genuine rule changes are not affected.

3.3.10 How the data and control plane algorithms work together

Suppose a packet p is stamped with a version number n by an ingress switch. Let the new path of the packet as prescribed by the update n be D_n and let the old path be D_o. Only the switches that need a genuine rule change are updated with rules that check for version n. On the rest of the switches unaffected rules (which do not check for version numbers) are used.

Suppose the controller has decided that an update n, which is stage1-complete, can proceed to the next stage. Now all the ingresses start tagging packets with version number n. Suppose the first internal switch that the packet p traverses is A and it has a rule that matches version n. That rule is applied to the packet. Similarly, all other switches along D_n that have rules of version n will match that rule with p. If the next switch on D_n is B and that does not have a rule of version n, p will match a rule on B that does not check for a version. (A or B cannot have a rule matching p that will check for a version less than n, as such a rule would indicate a conflicting update in progress, which the algorithm does not support.)

Now let us examine what happens when the next update $n+1$ is stage1-complete. Let the set of packets affected by update n be S. Packets belonging to S have the same set of forwarding actions from the ingress to the egress. Let update $n+1$ be such that it does not affect S. All the ingresses now start stamping all packets with the version number $n+1$ and with the status bit for n set to 1. A packet p belonging to S will reach switch A. Let A have a rule that checks for version $n+1$ installed. Since p will not match the rule that checks for version $n+1$ (since update $n+1$ does not affect p) and will match only the rule that checks for version n, the switch will check if the status bit associated with n is set to 1 in p. Since it is, the switch A continues to match p with the rule that checks for n. On switches similar to A, the same behaviour will follow. In switch B, there may be a rule that checks for version $n+1$, but again, p will not match that rule. It will continue to match the rule that does not check for a version number, on B and switches similar to B. The behaviour is similar when the ingresses start stamping packets with version numbers greater than $n+1$.

What happens to the packets belonging to S, *before* the update n is stage1-complete? Since they have version numbers less than n, even after version n rules are installed, they will not match those rules. These are the old packets that take the path D_o and they get

switched along D_o using the old rules that do not check for a version number and continue to get switched along that path even after update n is installed, thus preserving PPC. These old rules get deleted only after $T_c > T_{del} + M$, by which time, all the old packets would have exited the network - therefore no old packet is dropped.

Thus the new packets belonging to S, regardless of their version numbers, get switched exclusively along the path D_n and the old packets exclusively along the path D_o, preserving PPC. If the paths overlap, the same rules are used by the switches in the overlapping region, but that does not violate PPC.

The above explanation holds good if a switch has both new and old rules to be installed, or only new rules to be installed. Suppose a switch has only old rules to be deleted and suppose the update n is not stage1-complete. *limit* of the old rule is set to n. When this switch receives a packet with version $n-1$, it matches the old rule and since $(n-1) < limit$, it applies the old rule, as required. Suppose the update is stage1-complete and the ingresses start sending packets with version number n. Since all the affected switches are now ready to switch to new rules, on this particular switch, since there are no new rules, an unaffected rule must be applied to the packet. When the packet matches the old rule, since $n \geq limit$ now, the old rule is skipped, and the next matching rule is applied to the packet, as required. The next matching rule has $n < limit$, as *limit* is initialised to the maximum value the field can hold.

The status bits indicate to the internal switches the versions of the rules that are not yet stage1-complete and therefore must not be used. Since only 4 bits are used, there can be a gap of utmost 4 updates that are not stage1-complete between two stage1-complete updates $n+5$ and n. If updates $n+1$ through $n+4$ are not stage1-complete and update $n+5$ is, update $n+5$ can proceed to the next stage, by setting all the four status bits to 0. However, if update $n+6$ becomes stage1-complete next, then it cannot proceed to the next stage unless update n is stage1-complete, thus limiting concurrency. If the size of the status window is set to 5 bits, then update $n+6$ can proceed. Thus by changing the size of the status window, concurrency can be improved.

Subsequent to the publication of this work, we discovered that GU [81] employs a mechanism similar to the one described in section 3.3.9, to update only the affected switches, but with a different mechanism for achieving concurrency. In the case of CCU, with the same overhead bits, more updates can begin in parallel and can proceed simul-

Table 3.3: Additional Symbols Used in the Analysis of CCU

Symbol	**Meaning**
t_z	The time taken to update the stateful value *limit* for all old rules
t_{mo}	The time required to modify a set of rules
k_a	The total number of ingress switches in the network

taneously, if, by the time the controller checks for stage1-completion, all updates reach stage1-completion simultaneously or in the order of their version numbers. Deletion of old rules does not require further changes to the installed rules in CCU, whereas in GU [81], old rules need to be changed at the beginning of the update to have a label of 0, which is a time consuming activity if the rule is in the TCAM. CCU judiciously uses an SRT to improve the update time. CCU is also specified in greater detail.

3.3.11 Analysis of CCU

We assume an implementation of CCU where the version number of a rule is a part of the match part of a rule and *limit* is a stateful value checked in the action part. Thus changing the version of a rule involves modifying the TCAM of a switch, which typically takes more time than modifying a stateful value. We also assume that switches do not have an SRT. The analysis of CCU uses the same symbols as those of E2PU in Table 3.1 of section 3.2.5. The symbols in Table 3.3 are used, in addition. The time taken by the controller to calculate the version and status bits (Algorithm 1) is assumed to be negligible.

In addition to analysing the parameters identified in section 3.2.5 for CCU, we also compare CCU with E2PU. For a fair comparison of both, we assume that no SRT is used for both E2PU and CCU. We also make the following assumptions, that are common with E2PU: 1) at an ingress switch, old rules need to be removed, but new rules do not need to be added (Figure 3.2), 2) the sum of the propagation times and switch delays between the controller and the switches is uniform (δ), 3) the time taken for all insertions (t_u) and deletions (t_d) is uniform and the processing time at each switch is negligible, 4) the time at which the last switch sends Ready to Commit is $T_{c_1}+\delta+t_u + t_z$ and it denotes

the longest time taken and 5) the number of rules that need to be deleted and added to switches is uniform across switches, at each stage of the update, wherever applicable.

The time taken at each stage of an update is indicated in Figure 3.2. An update begins at T_{c1}. At an affected internal switch, after receiving Commit, new rules are installed, taking time t_u, and the value of *limit* for each old rule is changed, taking time t_z. When an ingress receives Commit OK, it installs its rules (the ones that change the version number and status bits of packets), which takes time t_u. The ingress does not have new rules that need to be installed, with respect to this update. Upon receiving Discard Old, an ingress deletes its old rules (the old ones affected by the update and the old ones that change the version number and status bits of packets) immediately, taking time t_d, whereas an internal affected switch deletes its old rules after t_s units of time (waiting for the timer to expire), again taking time t_d for the deletion. It also incurs a time t_{mo} to modify its new rules.

The major differences from E2PU from the standpoint of a timing analysis are: 1) In E2PU, all switches require rule changes, as the controller does not compute the paths affected for an update, whereas in CCU, only the affected switches require rule changes, along with ingresses. 2) CCU requires modifying a stateful value (*limit*) associated with each old rule. 3) CCU requires modifying the version field of a new rule so that it accepts all packets, at the end of an update.

Parameter 1: Duration of overlap of old and new rules: This is unaffected by the changes in CCU with respect to E2PU, and is the same as those for E2PU (Section 3.2.5).

Parameter 2: Duration within which new rules become usable: After $3\delta+2t_u+t_z$ units from the beginning of an update, all ingress switches instruct packets to use new version numbers and from that point, new rules are usable.

Parameter 3: Message Complexity: k_n+k_c messages of type Commit and Ready to Commit are required. Since Commit OK is sent to *all ingresses* and Ack Commit OK is received from all ingresses and only ingresses are involved in this stage, $2 * k_a$ messages of this type are required. k_o+k_c messages of type Discard Old and Discard Old Ack are sent, totalling to $2k_o + 4k_c + 2k_n + 2k_a$.

Parameter 4: Total Update Time: The total update time is $6\delta + 2t_u + t_z + t_s + t_d + t_{mo}$.

Table 3.4: Analysis of CCU

Parameter	**CCU**	**Comparison**
1, Case 1.1	$2\delta + t_d$	The same as E2PU
1, Case 1.2	$4\delta + t_u + t_d$	The same as E2PU
1, Cases 2,3	NA	NA
1, Case 4	$4\delta + t_u + t_s + t_d$	The same as E2PU
2	$3\delta + 2t_u + t_z$	t_z units more than E2PU, which is negligible
3	$2k_n + 4k_c + 2k_o + 2k_a$	Less than E2PU in the best case and equal to E2PU in the worst case
4	$6\delta + 2t_u + t_z + t_s + t_d + t_{mo}$	$t_{mo} + t_z$ units more than E2PU

The time taken for various parameters and a comparison with E2PU are given in Table 3.4. Parameter 2 is more than that of E2PU by t_z units, which is the time to update *limit*, associated with each rule. Since these values are stored in SRAM, the update times are negligible (in nanoseconds), compared to rule update times (in milliseconds), if an RMT switch is used for implementation. The message complexity for E2PU, if an SRT is not used is $2k_n + 4k_c + 2k_o + 2k_i$ (section 3.2.5.1). Since the affected paths are not computed, all the ingresses are affected for E2PU too. In CCU, since the changes are confined to only the affected switches, the values of k_n, k_c and k_o are lesser than that of E2PU, as long as all the switches in the network are not affected. Therefore, CCU has lesser message complexity compared to E2PU, in the best case and equal message complexity, in the worst case. The update time (Parameter 4) is higher, compared to E2PU.

For a CCU implementation where all switches have an SRT, the values of t_u and t_{mo} will be lesser, thus reducing the values of parameters 1, 2 and 4.

3.4 Conclusions

This chapter described two algorithms that preserve per-packet consistency. The first algorithm, E2PU, identified areas where the basic update algorithm for SDNs, that is, 2PU, is under-specified and described enhancements for that, exploiting the availability

of an SRT. We also analyzed the algorithm quantitatively.

The second algorithm, CCU, performs concurrent disjoint updates that preserve per-packet consistency and it works for all scenarios - whether only inserting rules, or only deleting rules, or a combination of both, on all the affected switches. It confines changes required for an RU to the affected switches and the ingresses of a network. CCU was quantitatively analyzed and compared with E2PU.

We observe that increasing one or more of these overheads improves concurrency: overhead bits in the packet, processing in the switch or the number of messages. We note that the algorithm presented will work well if each concurrent update takes roughly the same amount of time. Therefore it must be examined whether updates can be sized in that manner. Also, is there a way to achieve unlimited concurrency and increase FP to 1 at the same time ? How can the need to change all the ingresses of a network be eliminated? That is the subject of the next chapter.

Chapter 4

Proportional Per-Packet Consistent Updates

4.1 Introduction

This chapter describes the update algorithm Proportional Per-packet Consistent Updates (PPCU) that preserves PPC, confines changes to only the affected switches (FP=1), supports wild-carded rules and rules that have longest-prefix matches, provides an all-or-nothing semantics for a Rule Update and allows any number of concurrent non-conflicting updates, regardless of the execution speeds of switches and links. The algorithm does not require flows in the network for the RU to progress and needs no packet buffering at the controller.

4.2 Our contributions

In addition to describing the algorithm, we analyze its significant parameters and find them to be better than comparable algorithms. We illustrate that the algorithm can be implemented at line rate. A prototype of the algorithm is implemented on a simulated data centre network that uses realistic routing, and flow arrival, inter-packet delay and controller- switch delay distributions. The simulator also uses a realistic topology, and switch code that can be used directly in production environments. Our results demonstrate that using continuous PPCU updates provides better throughput for large flows and completes a larger number of small flows, compared to using random updates. Moreover,

PPCU updates do not violate safety requirements, unlike random updates.

Assumptions: 1) All switches use a synchronized real time clock using protocols such as PTP or ReversePTP [89]. If there is time asynchrony between switches, it has a known maximum value. 2) It is possible to add a time stamp to a packet header, which is the case with programmable switches [22].

Summary of the algorithm: Let the latest time at which all the switches in S install the new rules be T_{last}. Each affected switch examines the time stamp, set to the current time by the ingresses, in each data packet. If its value is less than T_{last}, it is switched according to the old rules while if it is greater than or equal to T_{last}, it is switched according to the new rules, thus preserving PPC.

4.3 PPCU: Algorithm for concurrent PPC updates

4.3.1 Switch Model

The switch model described in chapter 1 section 1.3 is used. For PPCU, optionally, a time stamp T and a rule type *rule_type* are associated with each rule. These optional values are not used while matching a packet but actions may read from them. PPCU requires two match fields, *a label* and *a time stamp*, to be added to every packet. Why they are added is described in the next section.

Applications that wish to control the behavior of the network issue RUs to the Controller. The Controller exchanges a series of messages with the affected switches to install new rules and delete old rules. The switches act on data packets using the rules installed. The sections below describe the changes required for concurrent PPC updates for the controller and the switches.

4.3.2 Algorithm at the data plane of the ingress and egress switches

Each packet p entering the network has *a time stamp field* TS_p added to it at the ingress and removed from it at the egress, as shown in Table 4.1. All ingresses set TS_p to the current time at the switch for all the packets entering it *from outside the network*. Once TS_p is set on a packet, its value is never altered. Each packet has a label that is set to

Variable	Values	Initial value	Description
TS_p	0 to T_{max}	Current time at the ingress	Stored when the packet enters the network
label	NEW_p, OLD_p, U_p	U_p	The rule type that must be applied to the packet

Table 4.1: Fields added by ingress switches

any of the values NEW_p, OLD_p or U_p, indicating if that packet is new, old or unaffected, respectively. All the ingress switches set the packet label to U_p and the egresses remove the packet label. Throughput the chapter, a suffix of p indicates that the entity being described belongs to a packet.

4.3.3 Algorithm at the control plane

Figure 4.1 shows the algorithm for PPCU at the control plane. The message exchanges are the same as in chapter 3; the parameters in them and actions upon receiving them have been modified to suit PPCU.

T_i denotes the current time at $s_i \in S$. The value of T associated with each rule is used by the rule to compare packet time stamp values and is initialised to T_{max}, 1 less than the maximum value that T can hold, for every rule. The value of *rule_type* associated with each rule is initialised to U, indicating that the rule is unaffected. These values are listed in Table 4.2.

Variable [index]	Written by	Read by	Values	Initial value	Purpose
T [n]	Controller	DP	0 to T_{max}	T_{max}	To compare with the packet time stamp to decide *label* of a packet
rule_type [n]	Controller	DP	NEW, OLD or U	U	Indicates the type of rule

DP: Switch Data Plane, n: rule number, T_{max}: 1 less than the maximum value the item can hold

Table 4.2: Stateful lists used by the affected switches

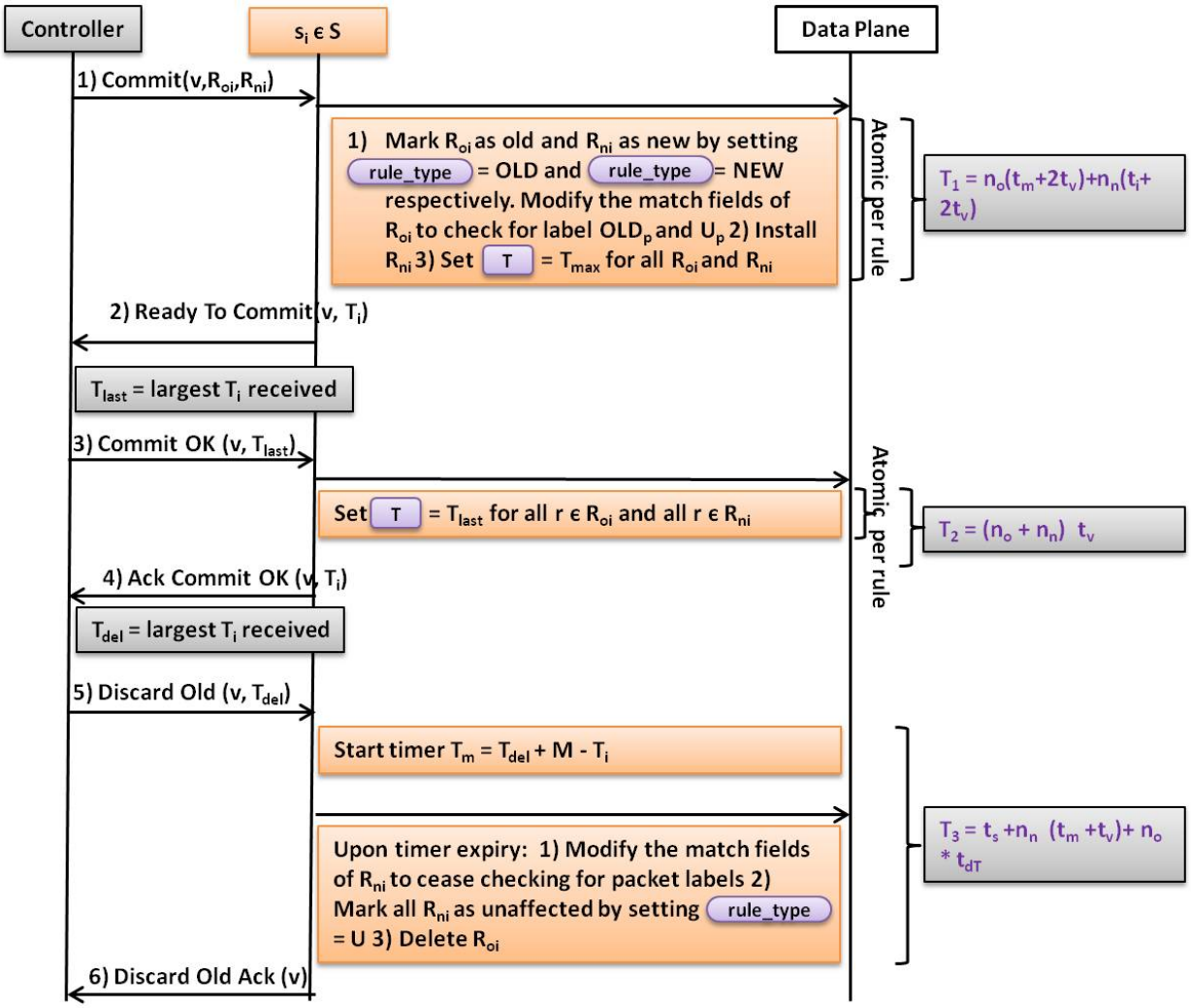


Figure 4.1: PPCU: Algorithm at the control plane

1. The Controller, upon receiving an RU from the application, with S, and R_{oi} and R_{ni} for every $s_i \in S$, sends a "Commit" to every $s_i \in S$ with v, R_{oi} and R_{ni}. v is a unique identifier associated with the RU.

2. A switch $s_i \in S$ upon receiving the "Commit" message, a) marks the R_{oi} rules as old and the R_{ni} rules as new by setting their *rule_type* to OLD and NEW respectively b) sets T associated with R_{oi} and R_{ni} rules to T_{max} c) changes the installed R_{oi} rules to check if a packet is labelled OLD_p or U_p (R_{ni} rules check if an incoming packet is labelled NEW_p or U_p) by changing their match fields and d) installs the R_{ni} rules. All the changes to *each rule* must be done atomically. Now it sends "Ready to Commit" with T_i.

3. The controller, after receiving "Ready To Commit" from all $s_i \in S$ sends "Commit OK" to all $s_i \in S$ with T_{last}, where T_{last} is the largest value of T_i received in "Ready To Commit", from the switches.

4. Upon receiving "Commit OK", the switch sets $T = T_{last}$ in all the rules in R_{oi} and R_{ni}, atomically per rule, and sends T_i in "Ack Commit OK".

5. The controller sends "Discard Old" to all the switches S with T_{del}, after receiving "Ack Commit OK" from all S, where T_{del} is the largest value of time received in "Ack Commit OK".

6. Each $s_i \in S$ starts a timer whose value is $T_{del} + M - T_i$, where M is the maximum lifetime of a packet within the network and T_{del} the time received in "Discard Old". When the timer expires, as all the packets that were switched using the rules R_{oi} are no longer in the network, the switch deletes R_{oi}. It marks every R_{ni} rule as unaffected by setting its *rule_type* to U and modifies its match fields such that it ceases to check for packet labels. Now it sends "Discard Old Ack" to the controller. With this, the RU at the switch is complete.

7. After the controller receives "Discard Old Ack" messages from all $s_i \in S$, the RU is complete at the controller.

After M units after the timer expiry at the last affected switch, the last packet tagged NEW_p will no longer be in the network. Now the next RU not disjoint with the current one may begin.

4.3.4 Algorithm at the data plane of the affected switches

Algorithms 3 and 4 specify how the data plane of an affected switch processes an incoming packet. Algorithm 3, MATCH-PACKET, specifies the algorithm to match packets and Algorithm 4, EXECUTE-ACTIONS, specifies the algorithm to process a packet within the action part of a rule. EXECUTE-ACTIONS provides the *template* for the action part of a rule. The primitive actions invoked from the action part will vary depending on the intention of the action.

New rules are always installed in the switch with a priority higher than the old and unaffected rules. MATCH-PACKET examines whether an incoming packet is already labelled NEW_p (OLD_p) or U_p. If labelled NEW_p (OLD_p) or U_p and the remaining match fields of the packet match a new (old) rule, as shown in line 4 of Algorithm 3 (line 6 of Algorithm 3) the action corresponding to the new(old) rule is executed.

rule_type is initialised to U and set to NEW (OLD) for new(old) rules at the beginning of the RU, by the controller. T is initialised to T_{max} for all the existing rules and set to T_{last} by the controller after the new rules have been installed in all the switches, in the course of an RU.

In EXECUTE-ACTIONS, when a packet arrives, first *rule_type* associated with that rule is checked (line 4). If *rule_type* is NEW (OLD) and if the packet is labelled NEW_p (OLD_p), the packet is switched using the new (old) rules, as shown in line 9 (line 19). If *rule_type* is NEW (OLD), the packet is labelled U_p and if its time stamp $TS_p \geq T$ ($TS_p < T$), it is labelled NEW_p (OLD_p) and it is also switched using new (old) rules. If the packet matches a new rule and its $TS_p < T$, the packet is labelled OLD_p. MATCH-ACTIONS is again called to match the newly labelled packet, in line 13, resulting in the packet subsequently getting switched out using old rules, as explained above.

An unaffected packet gets switched using the unaffected rules (line 10 of Algorithm 3 and line 26 of Algorithm 4). If and only if either of new or old rules do not exist in a table, that is, *when the rules are asymmetric*, an *affected packet* gets switched using unaffected rules (line 10 of Algorithm 3). Suppose an affected switch does not have an old (new) rule for an affected packet and the packet is labelled U_p. In that case, it first gets matched by a new (old) rule as shown in line 4 (6) of Algorithm 3, then if its $TS_p < T$ ($TS_p \geq T$), it gets labelled OLD_p (NEW_p). Subsequently, MATCH-PACKET is called again, as shown in line 13 (23) of Algorithm 4 and the packet gets switched by an unaffected rule , shown in

Algorithm 3 Match a packet

1: **procedure** MATCH-PACKET(*packet*)

2: ▷ This algorithm is for the match part of the match-action table. New rules are always installed with a priority higher than the old and unaffected rules. New rules check if the label of the incoming packet is equal to NEW_p or U_p and old rules check if it is equal to OLD_p or U_p. Unaffected rules do not check for a packet label.

3: ▷ *label* is the label of *packet*

4: **if** $(((label = NEW_p)$**OR**$(label = U_p))$ **AND** (*packet* matches fields of a NEW rule)) **then** ▷ This is the match part of a new rule

5: EXECUTE-ACTIONS(*packet*) ▷ Actions associated with the new rule

6: **else if** $(((label = OLD_p)$**OR**$(label = U_p))$ **AND** (*packet* matches fields of an OLD rule)) **then** ▷ This is the match part of an old rule

7: EXECUTE-ACTIONS(*packet*) ▷ Actions associated with the old rule

8: **else**

9: ▷ The packet does not match a new or old rule (unaffected packet) or no new or old rules exist on that switch to match the affected packet, regardless of the packet label

10: EXECUTE-ACTIONS(*packet*) ▷ Actions associated with the unaffected rule

11: **end if**

12: **end procedure**

Algorithm 4 Execute actions of the appropriate rule type

1: **procedure** EXECUTE-ACTIONS(*packet*)

2: ▷ This algorithm is for the action part of the match-action table. *rule_type* indicates if the *rule* is OLD (old), NEW (new) or U(unaffected). This value is associated with every rule and is initialised to U (unaffected). It is set to $NEW(OLD)$ by the controller for the new(old) rules in the beginning of the RU. T, associated with each rule, is initialised to T_{max} and is modified to T_{last} when the switch receives Commit OK. T_{last} is the latest time at which all the affected switches have installed the new rules.

3: Extract the label of *packet* into *label*

4: **if** $rule\_type = NEW$ **then**

5: ▷ TS_p is the time stamp of the packet

6: **if** $(TS_p \geq T)$ **OR** $(label = NEW_p)$ **then**

7: ▷ If the packet is already labelled NEW_p, the value of TS_p is immaterial

8: Set label of $packet = NEW_p$

9: Execute primitive actions of NEW rule ▷ Matches r_{ni}. Primitive actions such as forwarding the packet

10: **else**

11: Set label of $packet = OLD_p$

12: ▷ Needs to be matched with old rules, call MATCH-PACKET again

13: MATCH-PACKET(*packet*)

14: **end if**

15: **else if** $rule\_type = OLD$ **then**

16: **if** $(TS_p < T)$ **OR** $(label = OLD_p)$ **then**

17: ▷ If the packet is already labelled OLD_p, the value of TS_p is immaterial

18: Set label of $packet = OLD_p$

19: Execute primitive actions of OLD rule ▷ Matches r_{oi}

20: **else**

21: Set label of $packet = NEW_p$

22: ▷ Needs to be matched with new rules, call MATCH-PACKET again

23: MATCH-PACKET(*packet*)

24: **end if**

25: **else** ▷ Matches r_{ui}

26: Execute primitive actions of unaffected rule

27: **end if**

28: **end procedure**

10 of Algorithm 3 and line 26 of Algorithm 4, as there is no old (new) rule that matches the packet. If a packet is labelled NEW_p (OLD_p) and no new (old) rule exists on an affected switch, it will be switched using an unaffected rule.

4.3.5 How the algorithms at the ingresses, the affected switches and the control plane work together

The first switch $s_i \in S$ that changes the label of a packet from U_p to NEW_p is called the *first affected switch* s_f [1] for that packet. An RU can have more than one s_f.

This is the chronological sequence of events during an RU: 1) A switch that has received a Commit message, but not subsequent messages (Figure 4.2 a), changes the label of an incoming affected packet to OLD_p, if it was U_p, and uses old rules to process the packet. This label change ensures that all the switches subsequently visited by the packet also treat the packet as old irrespective of the current state of the switches. 2) When s_f receives Commit (Figure 4.2 b), it labels the affected packets that are labelled as U_p as OLD_p and continues to do so, until it receives the value of T_{last} in Commit OK (Figure 4.2 d). *At T_{last}, all the affected switches have completed installing new rules. Therefore any affected packet entering the ingress at a time later than T_{last} may be switched using new rules.* 3) When s_f receives the value of T_{last} in Commit OK, it starts labelling the affected packets whose $TS_p \geq T_{last}$ as NEW_p and from then on, the affected packets are switched using new rules (Figure 4.2 d). If s_f has not yet received T_{last} and a subsequent switch s_1 has received T_{last}, since s_f has already labelled the packet as OLD_p, the packet gets switched only using old rules (Figure 4.2 c). Thus a packet is either switched using old rules or using new rules, maintaining PPC. The algorithm is unaffected by the relative execution speeds of switches as the controller waits for a response to each message sent to the affected switches.

[1] Neither the controller nor any $s_i \in S$ knows which switch s_f is and that is immaterial to the algorithm. It is defined for ease of exposition.

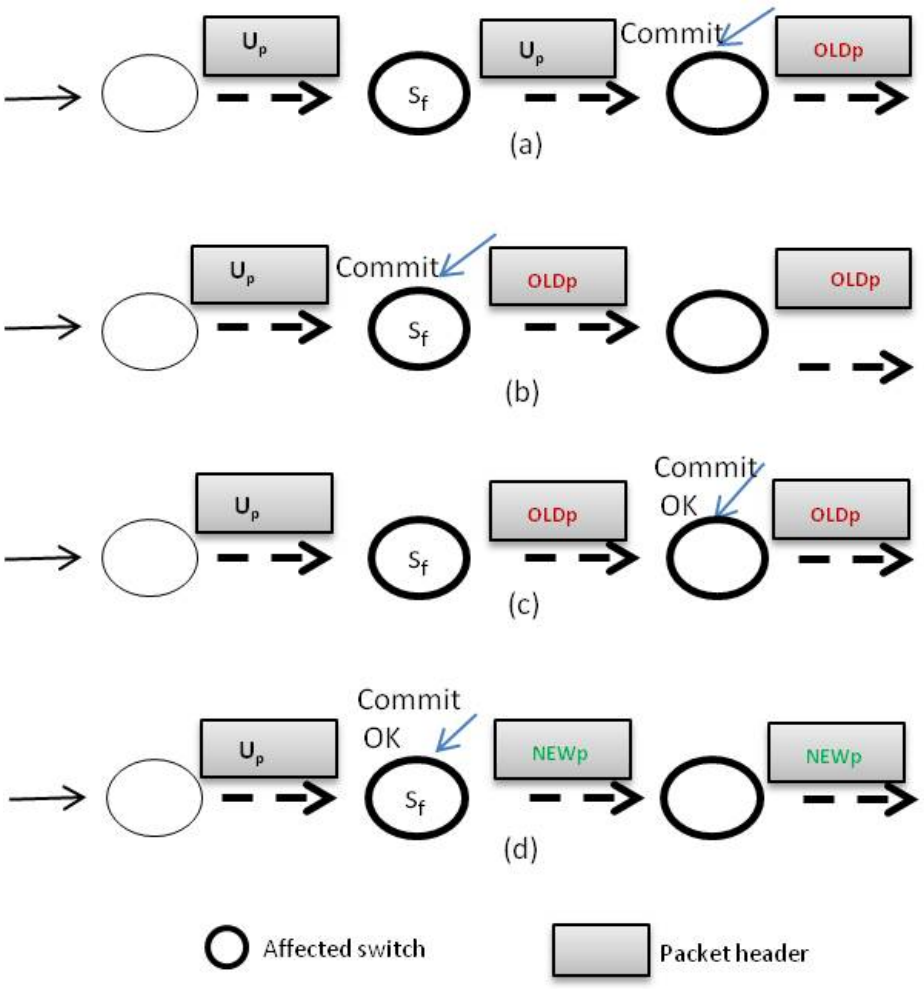


Figure 4.2: How the data and control planes work together

The recursive call (lines 13 and 23 of Algorithm 4) is executed only at s_f, only for two cases: 1) between receiving "Commit OK" and until the timer T_m expires, if s_f has no new rules for an RU 2) between s_f receiving"Commit" and "Commit OK", otherwise. If a switch supports multiple tables [22, 63, 21, 102] and an RU requires updates to multiple tables concurrently, the algorithm supports that, as the first affected table in the first affected switch will decide whether the packet must be switched by the new or old rules.

4.4 Handling Asynchronous time at each switch

Since a single rule update in a TCAM is of the order of ms (milliseconds) [62], a time stamp granularity of ms is sufficient for RUs. PPCU uses comparisons with time for two cases: TS_p with T and T_i with $T_{del} + M$.

Let us assume that the time stamp value is in ms and that the clock at an ingress s_g

is *faster* than an affected internal switch . Let γ be the maximum time drift of switches from each other (at the most $1\mu sec$, if the network supports PTP [90]). Let $TS_p = t_1$ of a packet p in the network that is stamped by s_g, even before the RU begins. After the RU begins, let the (temporally) last affected switch send its current time stamp T_{last} in "Ready To Commit". Let $t_1 > T_{last}$, since the clock at the ingress is faster. Let p_1 cross s_f before the RU begins and reach $s_j \in S$, where $j \neq f$. The label of p_1 is still set to U_p. Let us assume that all the affected switches have received Commit OK. Now s_j will switch p with new rules, violating PPC. To prevent this, each switch may set $T = \lceil T_{last} + \gamma + 1 \rceil$, instead of $T = T_{last}$. If an ingress clock is slow, there will be no PPC violations. Suppose an affected switch has a clock faster than the rest of the switches. That switch will conclude earlier than M units from T_{del} that old packets have exited the network. Therefore, each affected switch, upon receiving T_{del}, must set it to $\lceil T_{del} + \gamma + 1 \rceil$. Thus PPCU tolerates known inaccuracies in time synchronisation if the maximum allowed drift is known.

4.5 Concurrent Disjoint RUs

Each disjoint RU requires a unique update identifier v for the duration of the update, to track the update states at the affected switches and the controller. The number of disjoint RUs that can be simultaneously executed is limited only by the size of v. v is exchanged only between the controller and the switches and hence is not dependent on the size of a field in any data packet. Therefore as many disjoint updates as the size of the update identifier or the processing power of switches would allow can be executed concurrently.

4.6 Providing an all-or-nothing semantics

If an RU provides an all-or-nothing semantics, either all the new rules are *installed* or none of the new rules is. For this, the response to each message sent from the controller in Figure 4.1 may be guarded by a timer. 1) If the timer waiting for Ready To Commit expires, the new rules that have been installed, if any, may be deleted. Alternately, the application may issue a new update taking advantage of the already installed new rules. In either case, the affected packets get switched only using the old rules, as Commit OK is not sent to any switch at this point. 2) If Ack Commit OK does not arrive from all

the affected switches while the timer is alive, then Commit OK is resent to the switch(es) from where Ack Commit OK has not come. If, after a number of retries, there is still no acknowledgement, then the controller has to read any of the affected switches with the new rules, by means external to PPCU, using the monitoring constructs or run-time systems supported by application level programming languages [124], to know if the new rules are effective. If it receives Ack Commit OK from s_f, it will know that the new rules are effective (see Figure 4.2 d), but it does know which switch s_f is. Hence it needs to read the network. If the new rules are effective, the algorithm can proceed to the next step. If they are not, the behaviour is the same as in 1) above. If the controller is unable to read any of the affected switches where the new rules are installed, the application must decide what needs to be done when the outcome of the RU is uncertain, from its perspective. Regardless of whether the controller is able to read the affected switches, PPCU is preserved because, if s_f has received Commit OK, the affected packets are switched using new rules, whenever $TS_p \geq T_{last}$ (Figure 4.2 d) and if it has not, they are switched using old rules, even after $TS_p \geq T_{last}$ (Figure 4.2 c). 3) If the timer waiting for Discard Old Ack does not arrive, Discard Old must be resent for a specified number of tries. If there is still no acknowledgement, the application must ensure that the message reaches the switches from where it was not received, before beginning the next conflicting RU. PPC is preserved as the new rules are already effective and the affected packets are switched using the new rules.

4.7 Implementation

We have implemented a proof-of-concept prototype of the algorithm for the data plane (Algorithm 3) in P4 [30] and the control plane (in Figure 4.1) in our own controller and the switch simulator available with a P4 switch implementation on Linux [121]. The P4 open source switch implementation available is integrated with Mininet [88].

4.7.1 A Brief Introduction to P4

The abstract forwarding model advocated by protocol independent programmable switches [30] consists of a parser that parses the packet headers, sends them to a pipeline of ingress match-action tables (the *ingress pipeline*) that in turn consist of a set of match-fields and

associated actions, then a queue or a buffer, followed by a pipeline of egress match-action tables (the *egress pipeline*). In this section, we describe only the features of P4 that are relevant to PPCU.

A P4 program consists of definitions of 1) packet header fields 2) parser functions for the packet headers 3) a series of *match-action tables* 4) *compound actions*, made of a series of *primitive* actions and 5) a *control flow*, which imperatively specifies the order in which tables must be applied to a packet. Each match-action table specifies the input fields to match against; the input fields may contain packet headers and *metadata*. The match-action table also contains the actions to apply, which may use metadata and *registers*. Metadata is memory that is specific to each packet, which may be set by the switch on its own (example: value of ingress port) or by the actions. Upon entering the switch for the first time, the metadata associated with a packet is initialised to 0 by default. A register is a stateful resource and it may be associated with each entry in a table (not with a packet). A register may be written to and read in actions.

P4 provides a set of primitive actions such as $modify\_field$ and *add_header* and allows passing *parameters* to these actions, that may be metadata, packet headers, registers etc. When the primitive action *resubmit* (analogously *recirculate* for the egress pipeline) is applied to the ingress pipeline, a packet completes its ingress pipeline and then resubmits the *original* packet header and the possibly modified metadata associated with the packet, to the parser. If there are multiple *resubmit* actions, the metadata associated with each of them must be made available to the parser when the packet is resubmitted. Conditional operators and statements are available for use in the control flow to process expressions.

While this specifies the *definition* of the programmable regions of the switch, actual rules (table entries) and the parameters to be passed to actions, called *inputs*, need to be populated by an entity external to this model - the controller, through the switch CPU. This is facilitated by a run-time API available with the P4 switch [121].

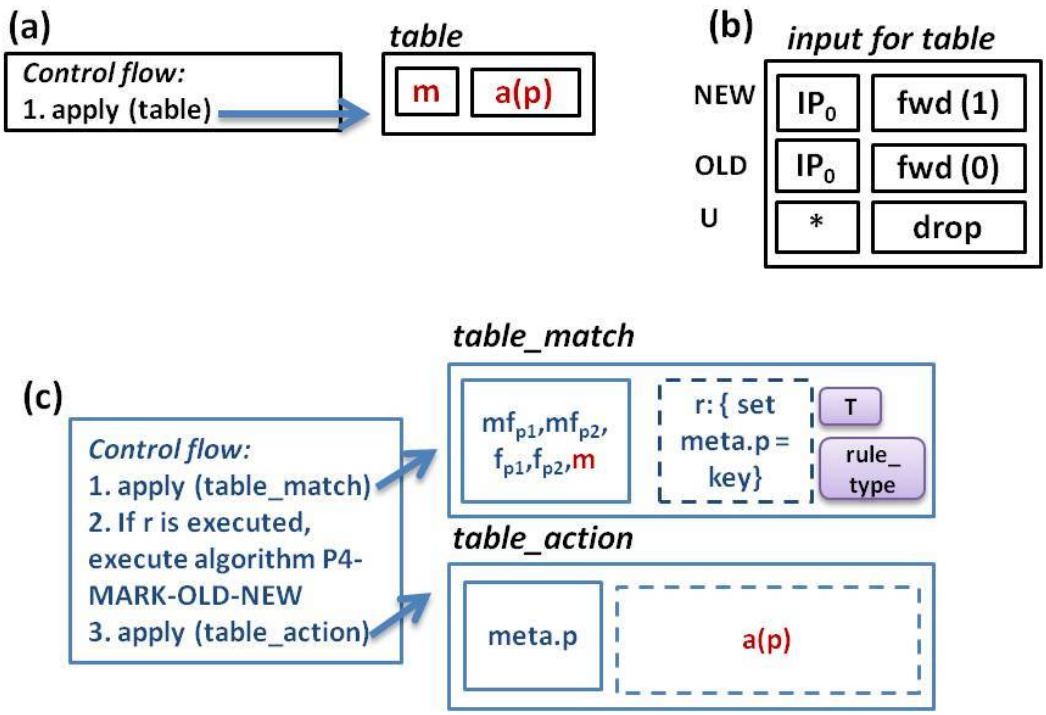


Figure 4.3: Inputs to tables and changes to table definitions. Figure (a) shows the original table and (c) its inputs. Figure (b) shows the changes required to *table* as per the implementation.

4.7.2 Implementation of the data plane algorithm in P4

Algorithm 3 is implemented in the *match part* of the P4 table and Algorithm 4 in the *control flow* [2] of the table that needs to be updated. To implement the ingress switches labelling packets as U_p, NEW_p and OLD_p, two bits f_{p1} and f_{p2} are used and are set to 0 and 0, 0 and 1, and 1 and 0 respectively (Table 4.3).

4.7.2.1 Algorithm at the data plane of the affected switches - changing the table definitions and control flow

Let us assume that a table called *table* in an affected switch needs to undergo an update, with match field m and action a, as shown in Figure 4.3(a). The *inputs* for this table are shown in Figure 4.3(b), where IP_0, IP_0 and $*$ indicate the match parts and $fwd(1)$, $fwd(0)$ and *drop* the action parts of new, old and unaffected rules, in that sequence. First, *table*

[2] This is because, the version of P4 [30] at the time of this implementation does not support expressions and conditionals in actions. In the latest version of P4, $P4_{16}$ [31], Algorithm 4 may be implemented in actions, simplifying the implementation.

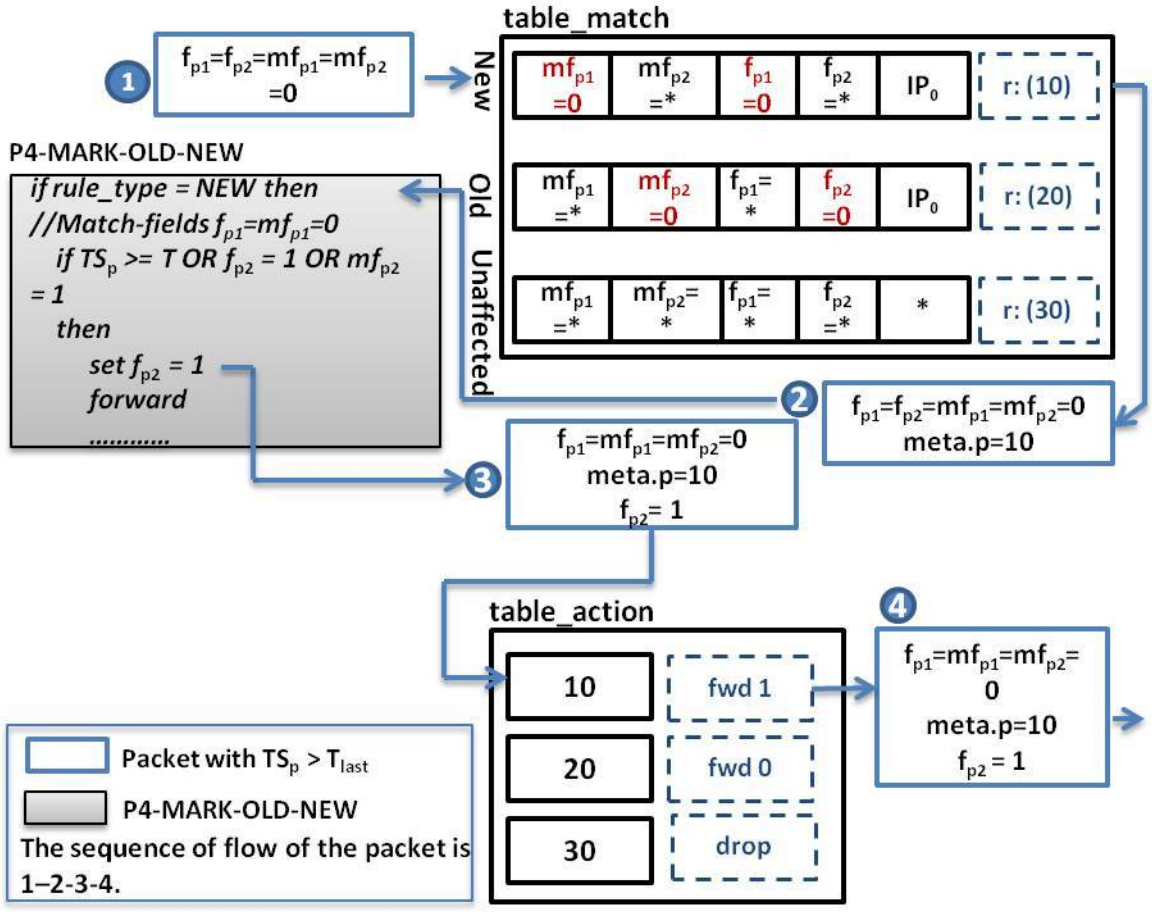


Figure 4.4: An example of changes to the inputs in Figure 4.3 (c) and the trace of an affected packet.

must be split into two tables, *table_match* and *table_action*, with the former containing only the match-fields of *table* and the latter containing only the actions of *table*, as shown in Figure 4.3(c). The following match fields must be added to *table_match*: f_{p1}, f_{p2}, mf_{p1} and mf_{p2} (mf_{p1} and mf_{p2} are needed due to the limitations of P4, as explained later). *table_match* will have an action r, that accepts a unique key *key* from the input and sets it to a metadata field associated with the packet, called *meta.p*, as shown in the action part of *table_match*. *table_action* will match a packet coming from *table_match* with the key set in *meta.p*; if it matches, the action specified in the input is executed, with the parameters, if any. These are the changes to the *definition* of tables. Splitting *table* in this manner is required only because P4 [30] does not support conditionals in actions. If the new version of P4 [31] is used, this will not be required.

The *input* to *table_match* must also be accordingly changed, as shown in Figure 4.3(c). *The match fields in table_match are created such that packets are forwarded as per*

Table 4.3: Packet labels

f_{p1}, mf_{p1}	f_{p2}, mf_{p2}	**label value**
0	0	U_p
0	1	NEW_p
1	0	OLD_p

Algorithm 3. The match fields of new (old) rules for f_{p1} and mf_{p1} (f_{p2} and mf_{p2}) are set to 0 and f_{p2} and mf_{p2} (f_{p1} and mf_{p1}) set to $*$, enabling the rule to check if the packet label is NEW_p (OLD_p) or U_p respectively, as may be inferred from Table 4.3.

If a packet matches an entry in *table_match* and the action r is executed, *the subsequent control flow* is as per Algorithm 5, P4-Mark-Old-New, which is an implementation of Algorithm 4. P4-Mark-Old-New marks the packet as OLD_p or NEW_p, unless it is already marked so. *table_action* is applied next, as shown in Figure 4.3(c).

To give an example of an *input*, in Figure 4.3(b), if the incoming packet has the destination address IP_0, it is forwarded to port 0 (the rule marked OLD). Now this rule needs to be changed to send such packets to port 1 (the rule marked NEW). In Figure 4.4, for a PPCU compliant RU, the new rule is installed with the match fields for f_{p1} and mf_{p1} set to 0 and f_{p2} and mf_{p2} set to $*$, indicating that it checks whether the packet label is NEW_p or U_p. The associated action is r and the value of *key* is 10. Due to the control flow, a packet (marked 1), whose $TS_p \geq T_{last}$ entering *table_match*, has its *meta.p* set to 10 (marked 2). In the control flow, P4-Mark-Old-New is applied next, which sets is f_{p2} to 1 (marked 3). Next, in the table *table_action*, it matches the entry with a key value 10 and the associated action to forward the packet to port 1 (marked 4) is executed, as desired.

4.7.2.2 Implementation of Algorithm 4 - P4-Mark-Old-New

Algorithm 5 specifies the *template* for a control flow associated with a switch table and is an implementation of Algorithm 4, in P4. The recursive calls of Algorithm 4 - Match-Packets in lines 13 and 23 - are implemented by resubmitting packets (lines 11 and 20 of Algorithm 5). If the fields of a packet (f_{p1} and f_{p2}) are modified, the modifications are not retained when the packet is resubmitted, while modifications to metadata are.

Algorithm 5 P4 implementation: Mark as old or new

1: **procedure** P4-MARK-OLD-NEW

2: ▷ *rule_type* indicates if a rule is new, old or unaffected. For a given value of *rule_type*, the match-field values for the rule are shown against it.

3: ▷ f_{p1}, mf_{p1}, f_{p2} and mf_{p2} refer to the values in the matched packet (labels).

4: **if** $rule\_type = NEW$ **then** ▷ After matching $mf_{p1} = f_{p1} = 0$

5: ▷ The rule is new. TS_p is the time stamp of the incoming packet and T is the time stored with the rule. $f_{p2} = 1$ indicates that the packet is labelled NEW_p

6: **if** $TS_p \geq T$ **OR** $f_{p2} = 1$ **then**

7: Set $f_{p2} = 1$

8: Execute primitive actions ▷ New rule

9: **else**

10: Set $mf_{p1} = 1$ ▷ Do not match this rule again

11: resubmit ▷ Use *recirculate* for egress tables

12: **end if**

13: **else if** $rule\_type = OLD$ **then** ▷ After matching $mf_{p2} = f_{p2} = 0$

14: ▷ An update is in progress and the rule is old. TS_p is the time stamp of the incoming packet and T is the time stored with the rule. $f_{p1} = 1$ indicates that the packet is labelled OLD_p

15: **if** $TS_p < T$ **OR** $f_{p1} = 1$ **OR** $mf_{p1} = 1$ **then**

16: Set $f_{p1} = 1$

17: Execute primitive actions ▷ Old rule

18: **else**

19: Set $mf_{p2} = 1$ ▷ Do not match this rule again

20: resubmit

21: **end if**

22: **else** ▷ After matching $mf_{p1} = mf_{p2} = f_{p1} = f_{p2} = *$

23: **if** $mf_{p1} = 1$ **then**

24: Set $f_{p1} = 1$ ▷ Only new rules exist

25: **else if** $mf_{p2} = 1$ **then**

26: Set $f_{p2} = 1$ ▷ Only old rules exist

27: **end if**

28: Execute primitive actions ▷ Unaffected rule

29: **end if**

30: **end procedure**

Therefore, two metadata bits, mf_{p1} and mf_{p2} are modified instead, as shown in lines 10 and 19 of Algorithm 5, before resubmitting a packet. Checking for the label of NEW_p (OLD_p) is implemented by checking for f_{p2} (f_{p1}) in line 6 (15). If the new and old rules are asymmetric, a resubmitted packet will match an unaffected rule, in which case also, the packet needs to be labelled OLD_p or NEW_p, as is done in lines 24 and 26 of Algorithm 5. T and *rule_type* are implemented as direct registers, with one entry associated with each rule. [3] The P4 code for the tables that are anticipated to be updated must be written in the above way. The inputs to the tables may be generated by any of the high level SDN programming languages [124].

4.7.2.3 Algorithm at the control plane

Commit is implemented as the union of two messages: one to the old switches to modify the old rules and another to the new switches to install new rules. *If a switch has both old and new rules, the old rules are installed first, that is, the existing unaffected rules are modified to be old first, before installing the new rules.* Similarly, to implement Discard Old, a message is sent to the old switches to delete the old rules and to the new switches to modify the new rules to unaffected.

4.7.2.4 Handling concurrency issues

In the version of P4 used [30] at the time of this implementation, it is not possible to atomically set the entries (one entry per rule) of a direct register, with respect to a packet. This is an issue if more than one rule needs to be changed for an RU, in which case, more than one entry may not be changed atomically. In light of this, as shown in Figure 4.5 (a), suppose the first affected switch s_f of an RU has both new and old rules that match a

[3] As the P4 compiler does not support multiple invocations of the same table, the code is structured differently. The actions in the control flow, such as setting packet fields or metadata or resubmitting packets, are invoked from further tables, as actions cannot be directly invoked from the control flow. Registers such as *rule_type* and T are not directly read in the control flow, but read from packet metadata, where they are copied into by the action r of *table_match*. We omit these details to improve readability.

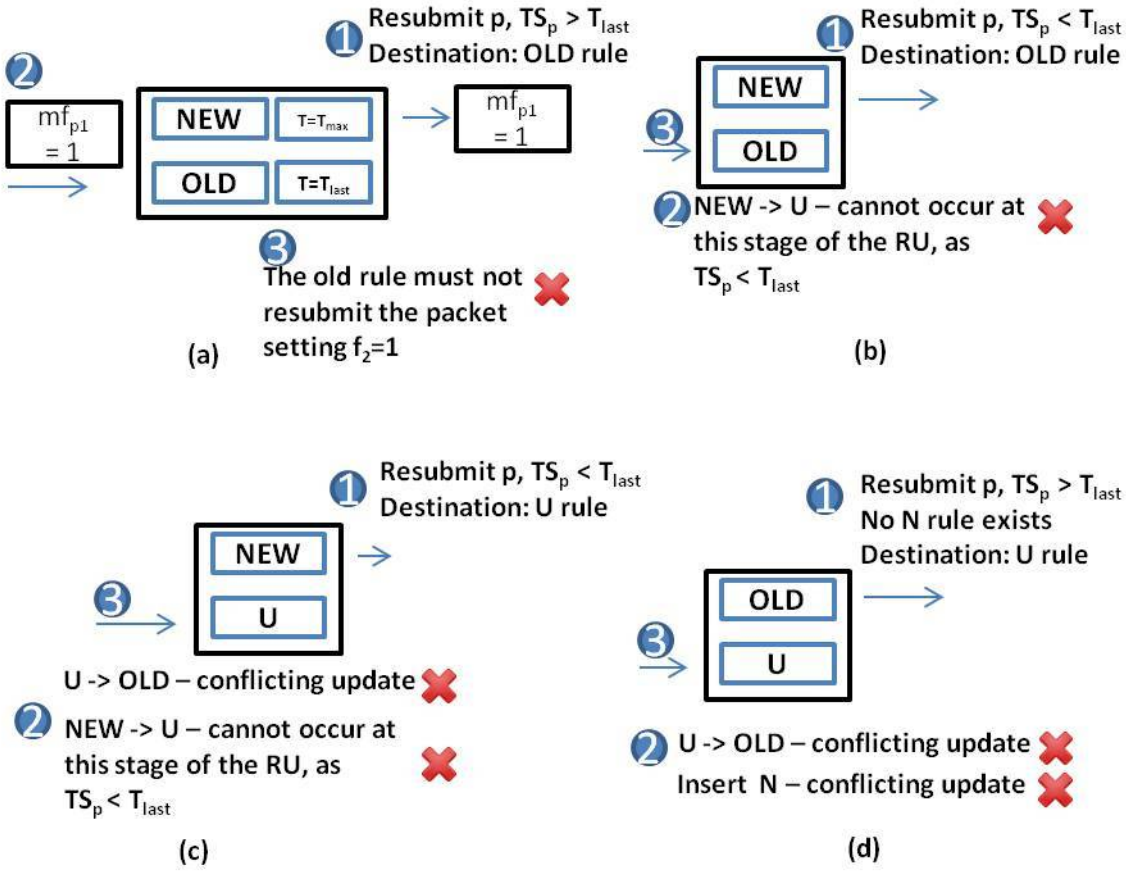


Figure 4.5: Value of *rule_type* changing after a resubmit

packet p and they are r_{nf} and r_{of} respectively. Let TS_p be greater than T_{last}. Assume the controller has not yet set the value of T for r_{nf} to T_{last} but has set the value of T for r_{of} to T_{last}. When a packet p traverses s_f, it matches the new rule first, and since $TS_p < T_{max}$, sets $mf_{p1} = 1$ and resubmits the packet, expecting it to match r_{of} and execute the actions of r_{of}. The resubmitted packet matches r_{of}. Suppose "$mf_{p1} = 1$" is not checked in line 15 of Algorithm 5. Since $TS_p > T_{last}$ and T of r_{nf} is set to T_{last}, r_{of} will resubmit the packet once more, in line 20, which is incorrect. Hence $mf_{p1} = 1$ must be checked in line 15.

Another issue is that of the change of state of a switch *after a packet is resubmitted, but before the next match takes place*. Consider a packet p that is resubmitted, assuming that it will match a rule r with a certain *rule_type*, after the resubmission. Suppose *rule_type* of r changes after p is resubmitted. Alternately, r may be superseded by another rule r_{async}, due to *rule_type* of r_{async} changing, or due to r_{async} getting installed during the RU, both after p is resubmitted.

The only valid type changes for a rule are from U to OLD, at the beginning of the RU

or from NEW to U, at the end of the RU. Figure 4.5 shows all the possible resubmission scenarios, with the numbers 1, 2 and 3 indicating the sequence of occurrence of events. As shown in Figure 4.5 (b), *rule_type* may change after a packet p is resubmitted by a new rule, to match an old rule after resubmission. If *rule_type* of the new rule is modified to U, it will supersede the old rule. However, this cannot occur at this stage of the RU. If the intended destination of the resubmitted packet from a new rule is an unaffected rule, as shown in Figure 4.5 (c), that means $TS_p < T_{last}$ and the switch does not have an old rule. The unaffected rule cannot be changed to an old rule, as it would imply a conflicting update (In a switch with old and new rules, since the old rules are always installed first, the possibility of an unaffected rule being changed to an old rule legitimately is ruled out). If the new rule is changed to an unaffected rule, it will supersede the unaffected rule. But this cannot happen at this stage of the RU. In Figure 4.5 (d), $TS_p > T_{last}$ and p is resubmitted by an old rule. The destination of p cannot be a new rule as new rules are always inserted with a higher priority and p would have matched the new rule to begin with. So the destination is an unaffected rule, which cannot be modified to be an old rule nor can a new rule be inserted, as both would imply a conflicting update. Thus *rule_type* changing its value asynchronously or rules getting asynchronously installed will not cause problems.

The value of TS_p will not change after a resubmission even if the resubmission occurs at an ingress, as the ingress inserts its time stamp only once on a packet.

In summary, once a table in s_f labels a packet as old or new, Algorithm 5 ensures that that label is never changed.

4.7.2.5 Supporting concurrent updates to multiple tables:

To support updates to more than one ingress or egress table in a switch, instead of defining mf_{p1} and mf_{p2} specific to a *table*, they must be defined as *global* to a switch, by declaring mf_{p1} and mf_{p2} as *intrinsic metadata*, a construct supported by P4. Thus, even if updates to one table in a switch are not atomic with respect to those in another table in the same switch, the value of mf_{p1} or mf_{p2} will be set to indicate if the packet is old or new by the *first table* t_f in that switch that is undergoing an update and subsequent tables will act according to this decision, if the packet is decided to be resubmitted by the first table. If the packet is just forwarded by t_f by setting f_{p1} or f_{p2}, subsequent affected tables will

use f_{p1} or f_{p2} to switch the packet.

4.7.3 Practical considerations

4.7.3.1 Feasibility of implementation at line rate:

Adding and removing headers such as TS_p, f_{p1} and f_{p2} are feasible at line rate; so are setting and checking metadata, such as mf_{p1} and mf_{p2} and header fields f_{p1} and f_{p2} in actions, as per table 1 in RMT [22]. While compiling the action part, Domino [117], another data plane programming language, checks if operations on stateful variables in actions can run at line rate by mapping those operations to its instruction set - PPCU requires only reading the state variables *rule_type* and T and this can be achieved using the "Read/Write atom" (name of instruction) in Domino. This demonstrates the feasibility of PPCU running at line rate.

4.7.3.2 Feasibility of adding TS_p at the ingress

We assume that all switches have their clocks synchronized and the maximum time drift γ of switches from each other is known. If the network supports Precision Time Protocol, $\gamma = 1\mu sec$ [90]. Intel FM6000, a *programmable* SDN capable switch, supports PTP, its $\gamma < 1\mu sec$ and the time stamp is accessible in software [58]. The size of the register used to store a packet time stamp in FM6000 is 31 bits. The feature of "intrinsic metadata" has target specific semantics and may be used to access the packet time stamp. Using the *add_header* and *remove_header* actions and intrinsic metadata, the TS_p field may be added at the ingress and removed at the egress for every packet, for targets that support protocols such as PTP.

4.7.3.3 Modifying rules in hardware switches:

PPCU requires modifying the match fields of unaffected rules to that of old rules at the beginning of an RU and modifying the match fields of new rules to that of unaffected rules at the end of an RU. Modification of rules in TCAM without changing their priorities is faster than rule insertion or deletion [28], with Tango [74] stating that "that modifying 5000 entries could be six times faster" than for inserting a rule, for a hardware switch.

4.7.3.4 Size of TS_p:

To improve throughput the size of TS_p may be reduced, for instance, by using the time relative to a recent date instead of POSIX time, thus reducing the granularity, in which case s_f will need to wait longer to receive packets whose $TS_p \geq T_{last}$, thereby lengthening the RU time. Since the size of TS_p is programmable in the field, the operator may be choose it according to the nature of the network.

4.7.4 Analysis of the Algorithm

The symbols used in the analysis are: δ: the propagation time between the controller and a switch, t_i: the time taken to insert *each* rule in a switch TCAM, t_{dT}: the time taken to delete each rule from a TCAM, t_m: the time taken to modify each rule in the TCAM, t_v: the time taken to modify each register associated with a rule, t_s: the time for which a switch waits after it receives "Discard Old" and before it deletes rules, n_o: the number of old rules that need to be removed, n_n: the number of new rules that need to be added, n: the maximum number of rules in a switch, k_a: the number of affected switches, k_i: the number of ingresses, k_t: the total number of switches, T_1: The time between the switch receiving "Commit" and sending "Ready To Commit", T_2: The time between the switch receiving "Commit OK" and sending "Ack Commit OK", T_3: The time between the switch receiving "Discard Old" and the switch performing its functions after timer expiry and T_4: Time taken to modify the new rules (for GU). It is assumed that the value of δ is uniform for all switches and all rounds, the values of time are the worst for that round, the number of rules, the highest for that round and that the processing time at the controller is negligible. Since unaffected rules have ternary matches for f_{p1} and f_{p2}, we assume that all the match fields are stored in TCAM and the corresponding actions in SRAM [22] [63], for PPCU, and for other algorithms that it is compared with. The values of T_1, T_2 and T_3 for PPCU are shown in Figure 4.1.

We add to and evaluate using the parameters of interest identified for a PPC in chapter 3, called *control plane parameters*: 1) **Overlap**: Duration for which the old and new rules exist at each type of switch 2) **Transition time**: Duration within which new rules become usable from the beginning of the update at the controller 3) **Message complexity:** the number of messages required to complete the protocol 4) **Time complexity:**

Table 4.4: Comparison with E2PU and CCU

Parameter	**PPCU**	**E2PU (chapter 3)**	**GU [81]**
Message complexity	$6k_a$	$4k_t$	$4k_a + 4k_i$
FP	1	k_a/k_t	$k_a/(k_a + k_i)$
Round 1 (T_1)	$n_o(t_m + 2 * t_v) + n_n(t_i + 2 * t_v)$	$(n - n_o + n_n)t_i$	$(n_o * t_m) + (n_n * t_i)$
Round 2 (T_2)	$(n_o + n_n) * t_v$	$(n - n_o)t_m + n_n * t_i$	$n * t_m$
Round 3 (T_3)	$t_s + n_n * (t_m + t_v) + n_o * t_{dT}$	$t_s + n * t_{dT}$	$t_s + n_n * t_m + n_o * t_{dT}$
Round 4 (T_4)	Not applicable	Not applicable	$n * t_m$
Propagation Time P	6δ	6δ	8δ
Time Complexity	$P + T_1 + T_2 + T_3$	$P + T_1 + T_2 + T_3$	$P + T_1 + T_2 + T_3 + T_4$
Concurrency	Unlimited	0	Number of bits in version field

the total update time[4]. 5) **Footprint Proportionality:** the ratio of the number of affected switches of an RU to the number of switches actually modified for the update. 6) **Concurrency**: Number of disjoint concurrent updates.

To quantify the effectiveness of PPCU on the data plane, we identify the following *data plane parameters*: 1) **Safety of the update**: For a given flow, this is the ratio of the number of packets that violate a policy/ the total number of packets sent.

In a data center network, there is always a mix of small flows (of size $128KB$) and large flows, of size varying from $1MB$ to $20MB$ [6]. In such a network, since the time to complete a small flow is small compared to the update time, what is of significance is whether small flows complete or not and whether large flows complete at the required throughput. For a given flow arrival rate and mean of inter-packet interval, we define the *usable duration* of the network (similar to ProjecToR [46]) as the the time for which the ratio of the total number of failed flows to the total number of flows completed is less

[4]Excludes the time for current packets to be removed from the network at the end of the update, where applicable

than R, where R is defined by network administrator in a network where there are no load balancing and flow scheduling mechanisms [2] [4] [17] used. We include the following additional data plane parameter: 2) **the number of small flows that successfully complete during the usable duration of the network**.

In a network with a large number of affected switches, it is possible that a small number of large flows complete with high throughput or a large number of large flows complete with a small throughput or it may be a mix of both. To account for this variation, we include 3) **the sum of the throughputs of the large flows that complete during the usable duration of the network** as another data plane parameter.

4.7.4.1 Analysis of control plane parameters:

The purpose of the analysis is to understand what the control plane parameters depend upon and to compare with a single update in E2PU (chapter 3) (which updates switches using 2PU with 3 rounds of message exchanges while taking into account when to delete the old rules and where the updates are to TCAMs), and with the algorithm GU in [81]. We show the evaluation results of the data plane parameters in the next section. We assume that GU [81] uses acknowledgements for each message sent. We also assume that only one rule is being modified/installed per affected switch during the update. Since all three have similar rounds, it is meaningful to compare the time taken by each round as given in table 4.4. The Overlap is $4\delta + T_1 + T_2 + T_3$ and Transition time is $3\delta + T_1 + T_2$ for all the algorithms under consideration.

n_o old rules are modified for PPCU, with each modification taking t_m units, n_n new rules are inserted, with each insertion taking t_i units and two variables, T and *rule_type*, are changed, taking time t_v each, for each affected rule, for Round 1. In Round 2, since only T is modified for the affected rules, it takes $(n_o + n_n) * t_v$ units of time. In Round 3, after waiting for t_s units, n_o rules are deleted and n_n rules are modified. *rule_type* of each new rule is also modified.

Since the values associated with the action part of a rule (T and *rule_type*) are stored in SRAM, we assume that update times of these values (in nanoseconds) will be negligible compared to TCAM update times and ignore the terms that involve t_v. We find in table 4.4 that 1) PPCU has lower message complexity, assuming $k_i > k_a/2$ (as is likely: the number of ingress switches is more than half the number of affected switches)

and better FP 2) PPCU and GU [81] have comparable times for Rounds 1 and 3. For Round 2, PPCU fares better. Therefore, PPCU fares better for Overlap, Transition time and Time complexity. 3) PPCU has better concurrency. Packets need to be resubmitted either between s_f receiving Commit and Commit OK or between Commit OK and expiry of T_m, that is, for a duration of $T_1 + 2\delta + T_2$ or $T_2 + 2\delta + T_3$, respectively, only at s_f (as explained in section 4.3.5). Since *resubmit* is an action supported by line rate switches, we assume that the delay due to a resubmission at s_f for this time frame during an update is tolerable, which is borne out by the evaluation in the next section.

4.8 Evaluation

4.8.1 Goals

The algorithms that meet most of the objectives of PPCU are 2PU [111] and E2PU (chapter 3). Since implementing 2PU or E2PU requires changes to all the rules in all the tables in all the switches of the network for every update, it is inefficient, as illustrated in 4.7.4. Therefore we do not compare PPCU with 2PU, and instead, compare it with a *random* update, which does not preserve PPC, for all the data plane parameters identified in 4.7.4. In a random update, commands to install the new rules are issued from the controller and without waiting for them to complete, commands are issued for the old rules to be deleted, to all the affected switches.

4.8.2 Implementation

The algorithm as described may be used for updating switches for any SDN. We have implemented a proof-of-concept prototype for a FatTree [1] network, common in data centres and enterprise networks. We implemented the data plane of the algorithm (Algorithm 5) in P4 (version 1.0.2 [30]), the changes to the control plane (Figure 4.1) of switches, and, a minimal controller. The P4 software switch is available integrated with Mininet [88] and Docker [34], to enable creation of a network of nodes, with each node running the data plane described by P4, on a Linux platform [121]. The controller creates and initialises the network using the Mininet and the target-independent Switch Abstraction Interface (SAI) APIs [121], starts the flows on the network and performs updates. The P4 code compiler

also generates low level Switch APIs [121] which, similar to SAI, controls the behaviour of the switches. Updates use the (suitably enhanced) Switch APIs to alter table entries, values of registers, metadata and action parameters and to insert, modify and delete rules, all at run-time. Updates to individual switches are performed in as concurrent a manner as possible, using threads. The controller and the P4 switch code, modified for PPCU, can be used as they exist, in a real environment.

4.8.2.1 Configuration:

We simulated a FatTree network [1], with realistic flowsizes [6] approximating a web-search workload, and flow arrival [6] (Poisson), inter-packet arrival [5] [16] (lognormal) and controller-switch delay [140] (normal) distributions. Table entries for routing in the network are implemented as per the routing scheme by Al-Fares et al. [1], but with one rule per flow, to enable the controller to perform a variety of updates, such as updates that affect only one flow or disjoint updates that affect more than one flow. The network is initially configured using rules installed from the controller on all the switches, using the SAI APIs. The maximum queue size within each switch, the switch and link delays and the percentage of CPU accorded to a host in the network are configurable in Mininet. Our simulation allows configuring the number of ports k of the FatTree, the mean of all the required distributions, and by providing the source-destination host-pairs and the maximum number of flows between them, the traffic flow within the network, in a *configuration file*.

The value of M (in Figure 4.1) can be configured by the administrator before compiling the P4 code; in our simulation, we account for the time asynchrony between Docker switches (section 4.4), in addition to the maximum time it takes for a packet to traverse the network. The implementation was tested on a Linux server using a 16-core Intel(R) Xeon(R) E5-2630 v3 CPU running at 2.40 GHz, and Ubuntu 16.04, with Mininet version 2.2.1 and Docker version 17.04.0-ce. All the experiments below are conducted on a FatTree network with $k = 4$, where k is the number of ports per switch.

Routing: The switches in the network are numbered from s_0 to $s_{<5k^2/4>}$, with the edge switches numbered first, followed by the aggregate switches and then the core

[5] Only inter-arrival time between packets in the ON period is simulated, it is assumed that there are no OFF periods in the flow

switches, as shown in Figure 4.6. For k=4, the edge switches are s_0 to s_7, the aggregate switches are s_8 to s_{15} and the core switches are s_{16} to s_{19}. The hosts are numbered $h_{<sn>0}$, $h_{<sn>1}$, ..., $h_{<sn>k/2}$, where sn is the switch suffix number, for every edge switch in the network. The addressing and routing scheme implemented by Al-Fares et al. [1] is briefly explained below. The IP address of each host has an address of the scheme $10.<pod-number>.<edge-switch-number>.<suffix>$, where the suffix is unique to each host attached to that edge switch. For a k=4 network, in our implementation, all host IPs have either their addresses ending with .3 or with .67. The edge switches examine the suffix of the destination host IP and route a packet to the appropriate aggregate. For example, in a k=4 network, all packets with the *destination* IP addresses $*.*.*.3$ arriving at s_0 get routed to s_8 and from s_8, they get routed to s_{16}, while packets with the destination IP addresses $*.*.*.67$ get routed to s_9 and subsequently to s_{18}. For diversity, it is packets with a different destination suffix that get routed to s_8 from s_1; in the above example, it would be the packets with the destination suffix .67. A similar strategy is followed to allow for diversity from the aggregate to the core layer. The core switches examine the pod number of the packet from the IP address and route the packet to the appropriate aggregate switch; the aggregate switch examines the switch number and routes to the appropriate edge switch. When the network comes up and the switches are initialized, the routing tables have one forwarding rule per destination IP address (which is different from the one described by Al-Fares et al. [1]) and all the hosts are reachable from all the other hosts.

Default configuration values: The value of M as mentioned in Algorithm Figure 4.1 is $100ms$, which is configured in the P4 switch simulator. In Mininet, the maximum bandwidth of a link is configured to be 200 Mbps and the switch does not artificially introduce a delay and causes no losses. The maximum switch queue size is 2000 packets and it uses a Hierarchical Token Bucket scheme. Each host gets 50% of the CPU time. No delay is artificially introduced between the controller and the switches.

The values of large flows vary from 1 to $19MB$ and the the values of small flows is fixed to be $128KB$. About 1/3rd of the flows are large and the remaining are small, to simulate a web-search workload [6]. The network has flows configured between the 10 host pairs $h_{00}-h_{20}$, $h_{31}-h_{50}$, $h_{40}-h_{60}$, $h_{40}-h_{21}$, $h_{60}-h_{01}$,$h_{20}-h_{51}$,$h_{60}-h_{31}$,$h_{21}-h_{71}$,$h_{30}-h_{61}$ and $h_{61}-h_{11}$, with a maximum of 2 flows between each host pair, running a web-search

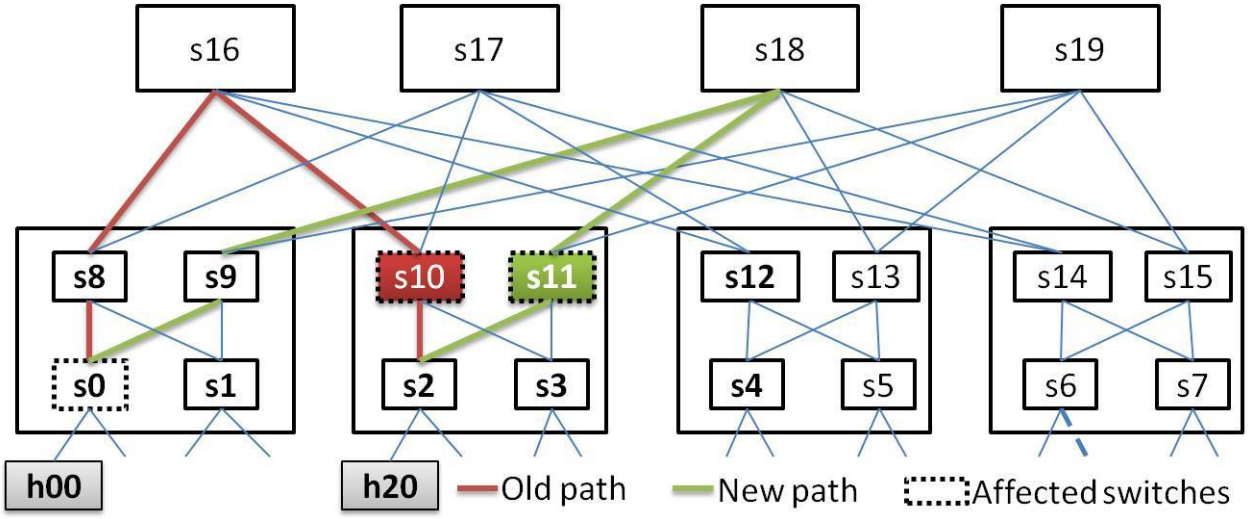


Figure 4.6: Experiment 1: FatTree with k=4

workload. The maximum length of a message sent from the clients running on hosts is 1400 *bytes*. The application timeout at each client is 60*s* wherever there is a client-server communication over TCP.

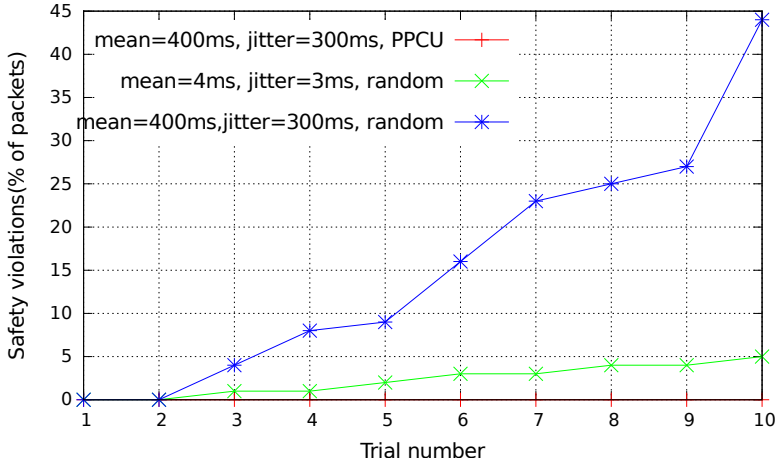
4.8.2.2 Data plane implementation:

The registers *rule_type* and T are assumed to have the well known names *table_name_register* and *table_name_T_register* respectively where *table_name* is the name of the table. They are initialised to U and T_{max} respectively when the switches come up and when each rule is installed. We use a 32-bit field for TS_p, with a time relative to a recent date, in milliseconds. The implementation in the data plane is only for two tables in a switch: the Access Control List (ACL) table and the routing table, which is sufficient for the experiments conducted. It may easily be extended to any other table in the switch.

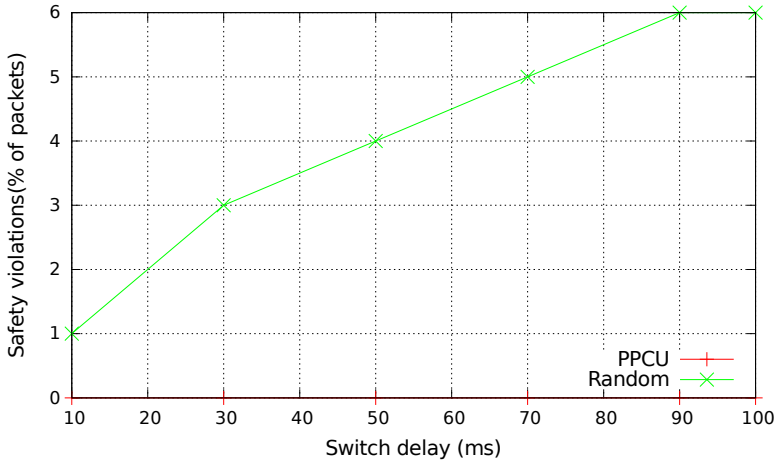
The experiments conducted are described below. In each of them, the switch logs and port logs are examined to ensure correctness of the algorithm. Also, updates occur continuously unless otherwise stated, as we expect to be the case in real networks.

4.8.3 Experiment 1

The goal of this experiment is to compare the safety (section 4.7.4) of PPCU and random updates. The network has a flow from h_{00} to h_{20}, using the route shown in Figure 4.6, as "old path". Let us assume that the packets on this flow need special processing - for



(a) Varying controller-switch delay



(b) Varying switch delay

Figure 4.7: Safety violations with varying controller-switch and switch delays

example, all packets destined to h_{20} must be dropped. This policy is implemented in the ACL table at s_{10} and not at the ingress, to simulate the condition of the switches in the path having exceeded their table limits [97]. When the route of this flow is changed to the one shown in Figure 4.6 as "new path", the policy also must be installed at s_{11} and removed from s_{10}. We alter the path by updating the exact match routing table of s_0 and install the policy by updating the ACL tables of s_{10} and s_{11}, all within the same update. If the update is safe, no packet from h_{00} must reach h_{20}, during and after the update, thus adhering to the policy.

Before the update begins, no packet from h_{00} reaches h_{20}. We start a ping flood from h_{00} to h_{20}, begin a PPCU update, stop the ping flood after the update ends and examine the number of packets that reach h_{20}. We repeat the experiment with a random

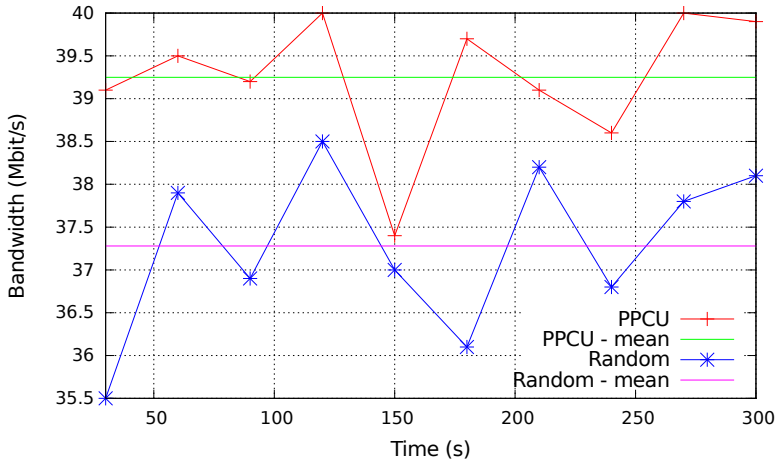


Figure 4.8: Bandwidth using iperf over one link

update, instead of a PPCU update. We run these experiments for different values of controller-switch delays (Figure 4.7a) and switch delays (Figure 4.7b) at the data plane. Each message sent from the controller to a switch is assumed to be delayed with the given mean and standard deviation ((4ms ,3ms) and (400ms,300ms)), for both random and PPCU updates. The results show that if the update is PPCU, there are no policy violations, regardless of switch-controller delays or switch delays, whereas with a random update, policy violations increase with switch-controller and switch delays. In a random update, if the update to s_0 happens to occur first and the update to s_{11} is delayed, packets routed through the new path reach h_{20}, violating the policy. Similarly, if some of the switches happen to be slow and packets are delayed, the old rule in s_{10} may get deleted before all the packets that matched the (old) rules in the s_0 to s_{10} path have been removed from the network, leading to a policy violation.

4.8.4 Experiment 2

The goal of this experiment is to compare the throughput of a link that undergoes PPCU updates with those of a random update, using iperf [99]. An iperf client running on h_{00} sends the maximum possible amount of data, with a message size of 1400 bytes, over a TCP connection to an iperf server running on h_{20}. We start a series of PPCU updates to change the h_{00} to h_{20} path from old to new and back to old, as shown in Figure 4.6, without any delay between updates, for more than 5 minutes. Then we start an iperf client on h_{00} and an iperf server on h_{20} and send data between them for 5 minutes, measuring

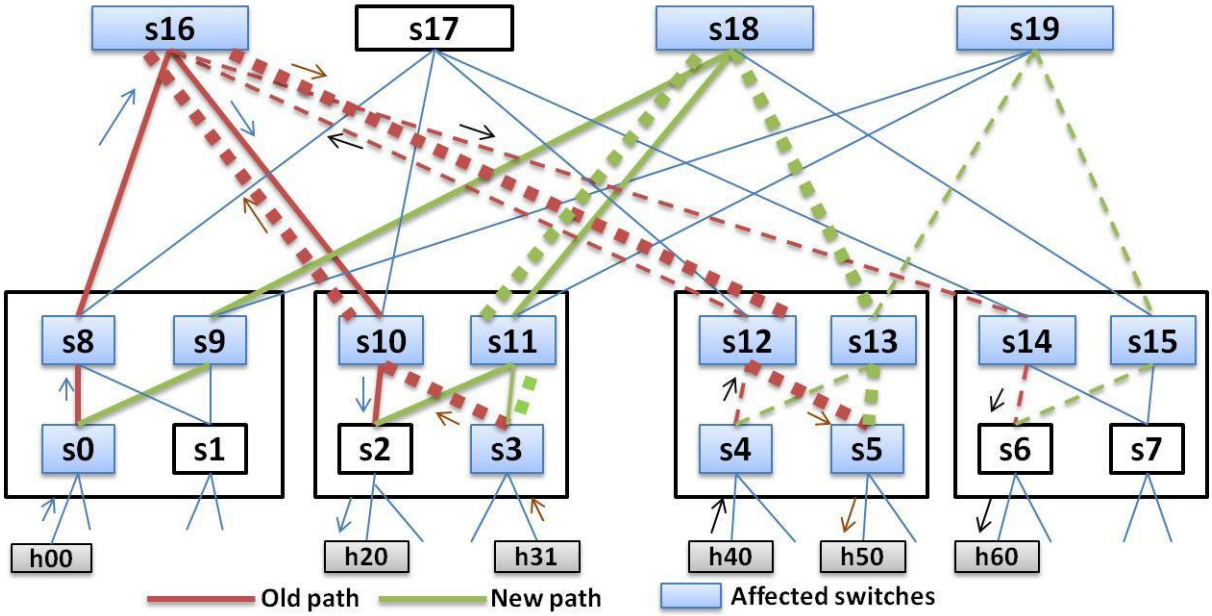


Figure 4.9: Experiment 3: Changing edge to aggregate links

the average throughput every 30s. The results in Figure 4.8 show that the throughput while using PPCU updates is better than while using random updates. In a random update, since rules are deleted from affected switches arbitrarily, there may not be a rule to forward a packet matching old rules, arriving at an affected switch with deleted old rules. Also, since rules are installed at affected switches in an arbitrary order, there may not be a rule to forward a packet already matched with a new rule arriving at an affected switch, as new rules are not yet installed there. In PPCU, both these situations do not arise, leading to a better throughput, compared to a random update.

4.8.5 Experiment 3

In real networks, a large number of updates take place simultaneously, affecting multiple flows. A switch may undergo more than one update simultaneously and a single update may result in modifying multiple flows on the same set of switches. The delay between updates for a single flow may be more than that in Experiment 2. Realistic flow arrival, inter-packet arrival and controller-switch delay distributions also need to be simulated. The goal of this experiment is to find out how PPCU affects the throughput for large flows and the number of small flows and total flows completed, compared to random updates, during the usable duration (defined in Section 4.7.4) of the network. To increase

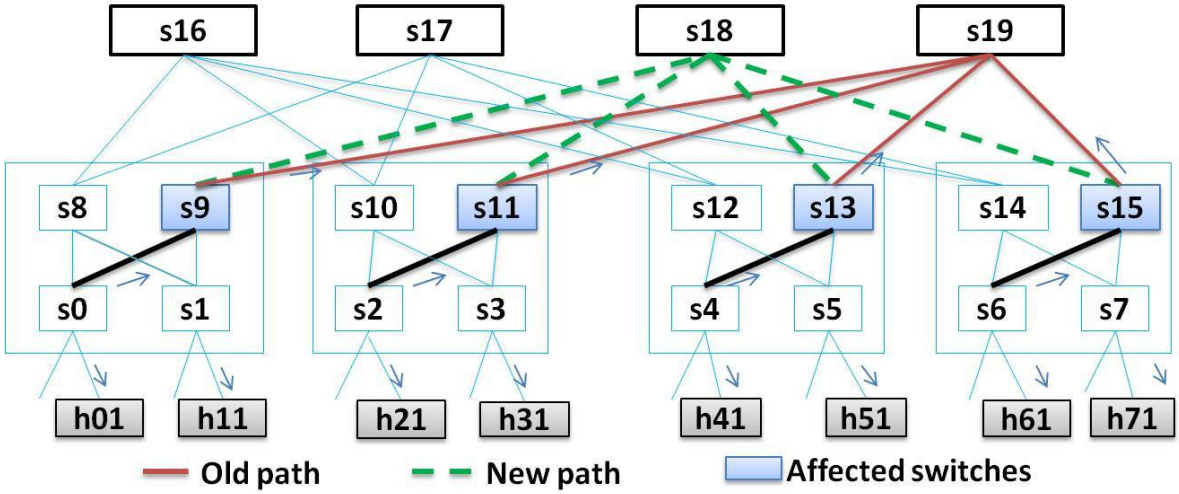
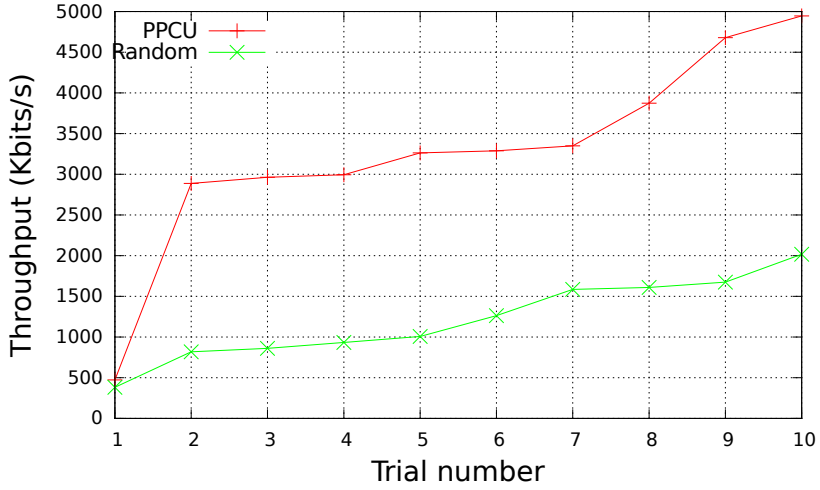


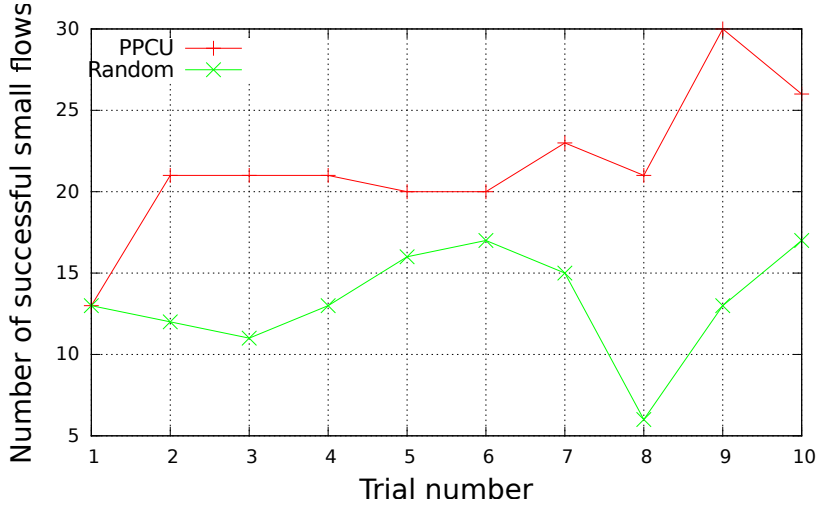
Figure 4.10: Experiment 3: Changing core to aggregate links

predictability of results, all the flows in the network are affected by an update. 75% of the switches are affected by at least 1 update and 25% of the switches are affected by 2 disjoint updates, with the latter causing updates of different entries in the same P4 table for two different RUs. The updates performed are : 1) edge to aggregate link changes: The flows from h_{00} to h_{20}, h_{31} to h_{50} and h_{40} to h_{60} are switched from the old path to the new path, as shown in Figure 4.9. Paths are switched from the old to the new and back, in the given sequence, in a loop, with the sequence repeating forever. 2) aggregate to core link changes: In this, all traffic destined to *.*.*.67 on the set of red links is moved to the set of green links in a single update and then back, forever, as shown in Figure 4.10. The affected destination hosts are shown. The affected traffic from edges to aggregates is shown in black. All the four updates are disjoint, but occur simultaneously, within one PPCU.

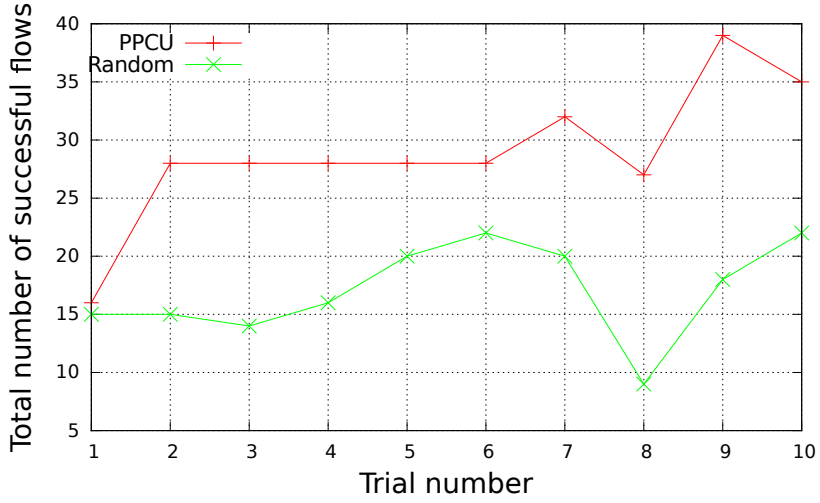
After the network is initialised by the controller, a flow generator starts multi-threaded servers on the destination hosts, as per the configuration file described in section 4.8.2.1. Next, it starts flows one by one, as per the configured flow rate, by starting clients on the desired hosts until the maximum number of flows of each host pair is reached, after which flows begin on the next host pair. To increase predictability of results, we include only affected flows and the maximum number of flows is kept the same, across host pairs. The clients start connections to the multi-threaded servers started on the destination hosts. The flow generator sends a token to each client according to the packet inter-arrival time and distribution. Upon receiving the token, the client sends a message



(a) Throughput

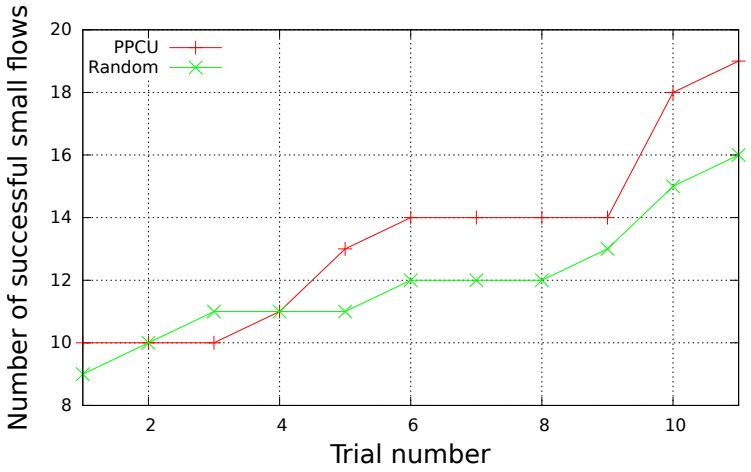


(b) Successful small flows

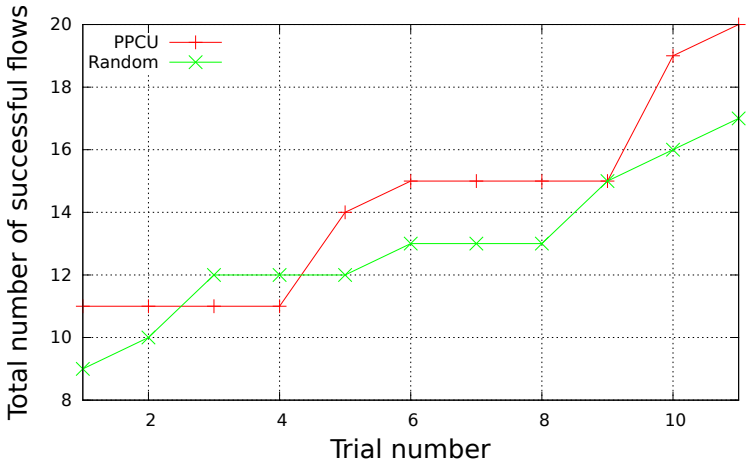


(c) Total successful flows

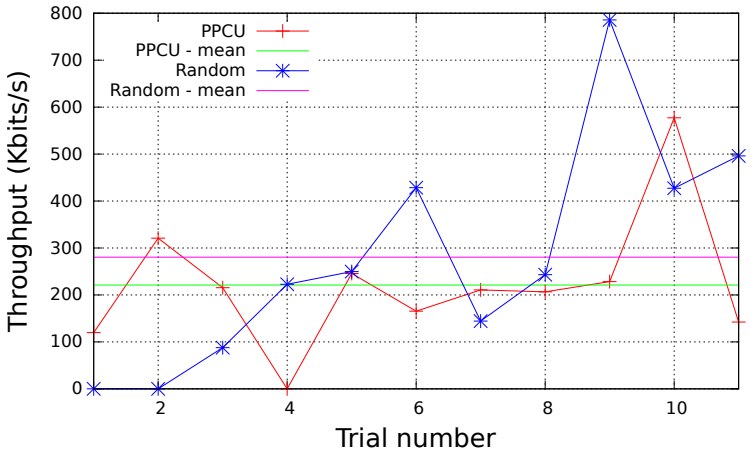
Figure 4.11: Throughput, small and total flows completed for flow rate=0.033 flows/s



(a) Successful small flows

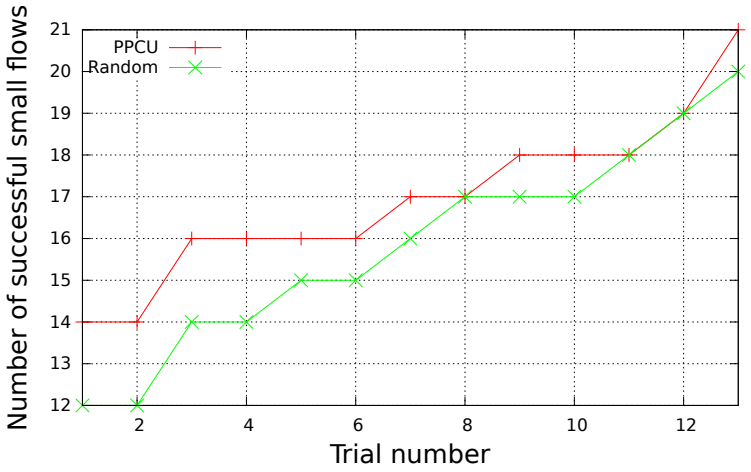


(b) Total successful flows

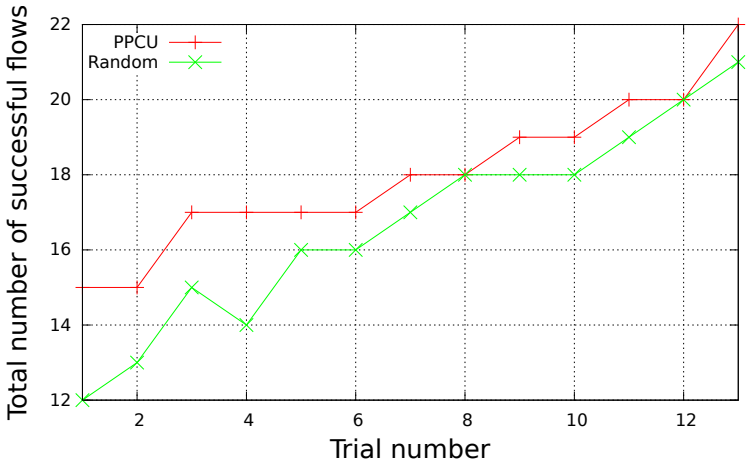


(c) Throughput

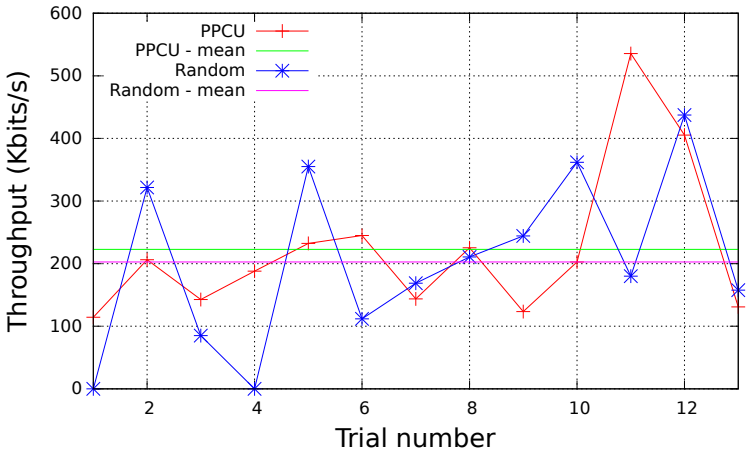
Figure 4.12: Throughput, small and total successful flows for flow rate=0.33 flows/s, maximum flows per host pair=2



(a) Successful small flows

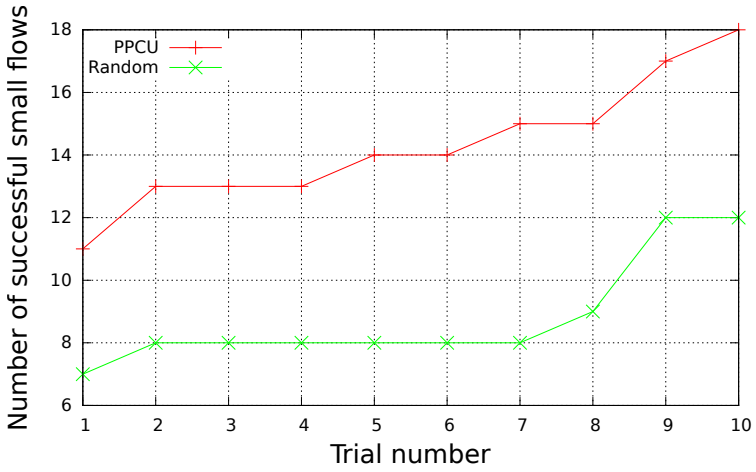


(b) Total successful flows

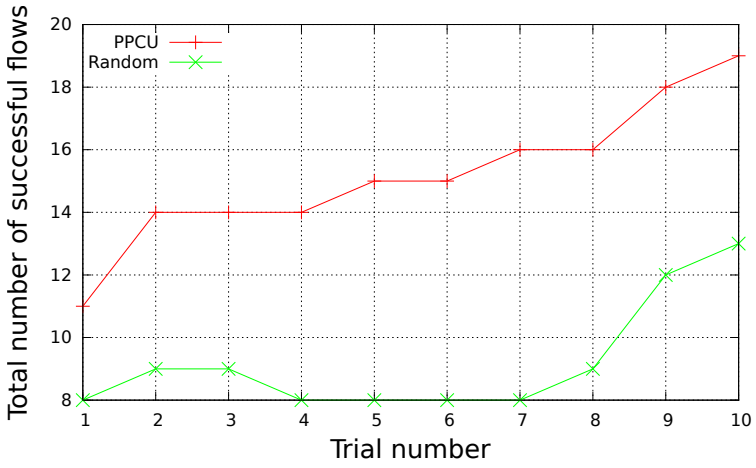


(c) Throughput

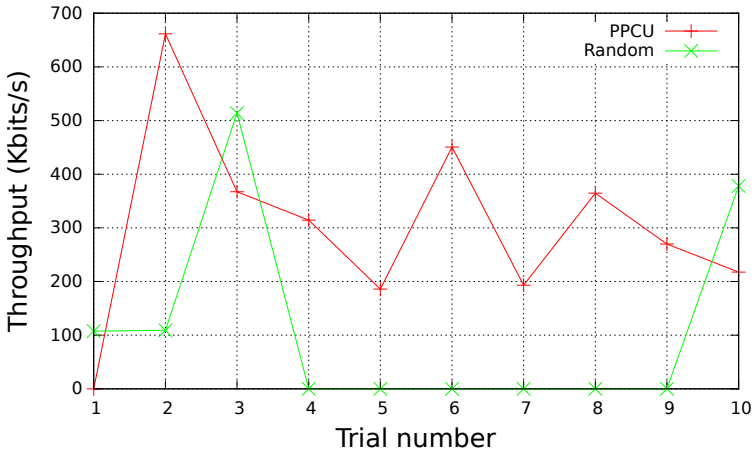
Figure 4.13: Throughput, small and total successful flows for flow rate=0.33 flows/s, maximum flows per host pair=4



(a) Successful small flows



(b) Total successful flows



(c) Throughput

Figure 4.14: Throughput, small and total successful flows for flow rate=0.33 flows/s, maximum flows per host pair=2, controller-switch delay: mean=400ms, s.d=300ms

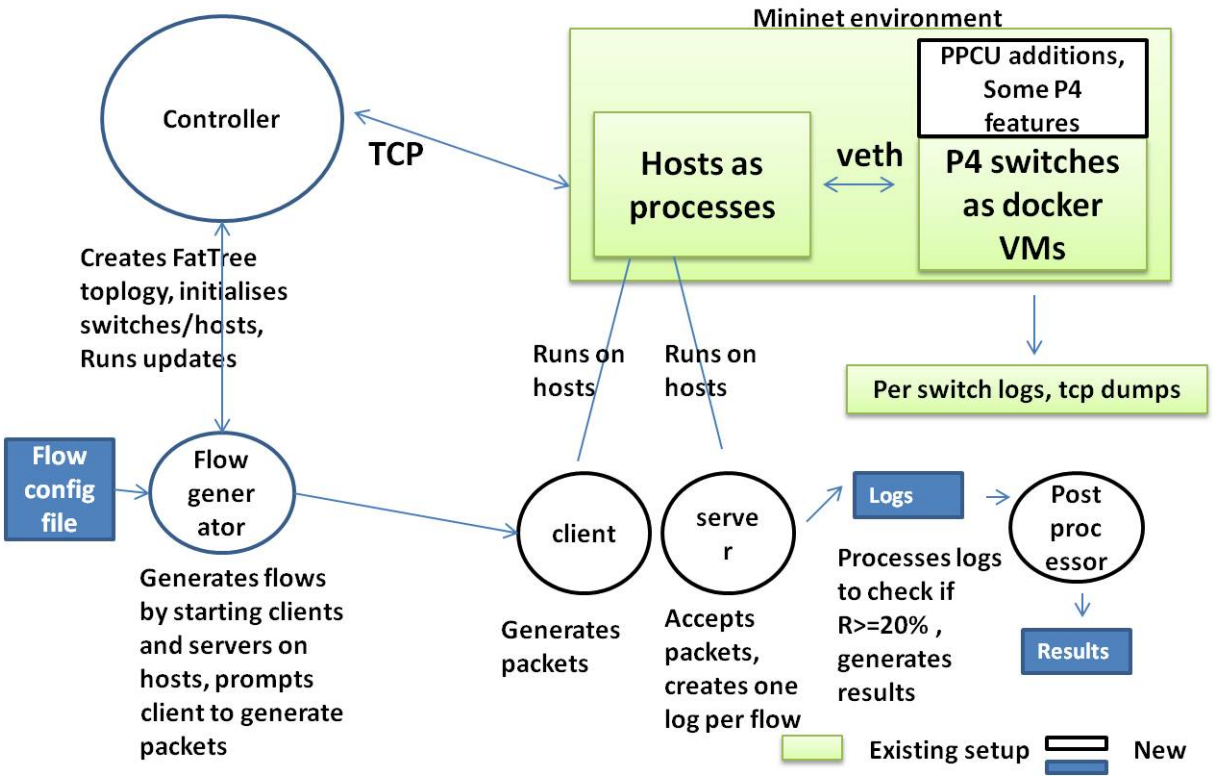


Figure 4.15: Experimental setup

of fixed size to the destination server. The server running on the destination host creates one log per flow. An online post-processor operates on the logs to identify failed flows, checks if the usable duration R has exceeded its upper limit, and generates test results. If R has exceeded the upper limit, the simulation is stopped. This is illustrated in Figure 4.15.

Before the experiment begins, we adjust the values of flow arrival rate and inter-packet delay and find the values at which at least one large flow completes, during the usable duration of the network, in the presence of continuous PPCU updates. The values are 0.033 flows per second, and 15 ms with a standard deviation of $0.1ms$, respectively. We initialise the network and start all the updates simultaneously, using PPCUs for all the updates, at the flow and packet arrival rates discovered above. We stop simulation when $R \geq 20\%$ and measure the number of small flows completed, the number of large flows completed, their source-destination pairs and individual throughputs. The experiment is repeated for random updates.

For a low flow arrival rate, since more number of large flows or larger flows are expected to complete successfully, the collected data is sorted on the sum of throughputs

of successful large flows and is shown in Figure 4.11. For PPCU updates, the sum of throughputs, the number of successful small flows and the total number of successful flows is higher than that of random updates.

The experiment is repeated with a controller-switch delay and the results are shown in Figure 4.14. Since no large flow completes for most of the cases, the output is sorted with the number of small flows completed. We find that PPCU updates result in a higher number of small flows getting completed, in all the trials. The total number of flows and the throughput are higher too.

It is not sufficient for the network to *operate* in the usable duration because "a reasonable number" of large flows must also complete at a "reasonably high" throughput. If we increase the flow arrival rate with the network still operating in the usable duration, there are two issues: no large flow completes and/or the simulation cannot maintain the flow rate as enough flows do not finish. (Even though flows must be scheduled [2] [4] [17] to avoid this situation, we conduct two experiments to check the behaviour of the network in this situation also.) In the first case, instead of comparing the throughputs of large flows, we compare the number of small flows completed. In the second case, we discard the reading. Besides, the throughputs of large flows that manage to complete, if any, are much smaller than the throughputs of large flows when a lower flow rate is used. We first conduct the experiment with a flow rate of 0.33/s and the number of maximum flows per host-destination pair set to 2. The results in Figure 4.12 demonstrate that with PPCU updates, the number of small flows and total flows successfully completed is higher. The same is the case when the maximum number of flows per host-destination pair is increased to 4, as shown in Figure 4.13. For both random and PPCU updates, the throughputs of the large flows that complete are comparable.

4.8.6 Summary of experiments

In the experiments conducted above, we demonstrated that PPCU does not cause safety violations, regardless of the execution speeds of switches and controller-switch link delay variations. We also illustrated that the throughput of the network using iperf in a realistic network setting is superior if PPCU updates are used, compared to similar use of random updates. PPCU updates result in a higher number of small flows completing successfully and a higher total throughput for large flows, during **normal** operation of the network,

that is, when small flows complete and large flows have a reasonably high throughput. Even during higher loads, using PPCU updates ensures completion of more flows compared to random updates. The total throughput and the number of flows completed during the usable duration is more significant than the duration for which the network remains usable. Experiment 3 also illustrates that more than one non-conflicting update can be performed on the network in the same RU, affecting the same set of switches. As seen in all the experiments, realistic variations in switch speeds and controller-switch delays cause PPCU updates to have better throughput and higher number of flows completed, compared to random updates. We expect that flow rates are maintained using tools such as MicroTE [17], Hedera [2], and CONGA [4] such that the network completes large flows at a high enough throughput, as shown in Figure 4.11.

4.9 Discussion

Among application level languages for SDN programming, only Frenetic [42] addresses consistency of updates. Mechanisms to generate flow table rules in the manner required for PPCU may be incorporated into application level programming languages [124] as per the specifications of the target switch. The target switch need not necessarily use P4, but may use any data plane programming language that is sufficiently expressive, such as Domino [117]. Integration of PPCU into an application level programming language will also enable experimentation with a wider variety of scenarios.

PPCU describes an algorithm that can be implemented in software for PPC updates. There may be changes required in hardware in switches to implement consistency within the switch. Blueswitch [53] is research in that direction.

To reduce the number of rules in the affected switches during an update, techniques explored in E2PU (chapter 3) to install and move flow table rules to software may be integrated with PPCU. E2PU (chapter 3) uses the same set of message exchanges as PPCU; when a Commit is received, new rules may be installed in a Software Rules Table, instead of the TCAM, thus not affecting the TCAM space.

To improve the update time, it is recommended that register names for T and *rule_type* are associated by default with every P4 table and have well known names, as explained in Section 4.7.4, with these registers initialised to the correct values by the

P4 implementation. Initialising T to T_{max} can be done while a rule is installed in the switch, without an explicit instruction from the controller, to reduce update time. The new version of P4 [31] supports modifying registers atomically from the point of view of incoming packets, using the "@atomic" annotation. Hence the extra checks introduced in section 4.7.1 to handle concurrency issues with respect to registers may be removed for targets that support newer versions of P4.

4.10 Conclusions

This chapter has described for the first time a per-packet consistent update algorithm which is efficient, applicable for any kind of update, which is not affected by timing issues such as switch and network delays and time asynchrony in the network, and works at line rate. Efficiency is achieved by restricting the interaction to the switches where updates are to take place and to the rules that are being changed. The implementation in P4 and evaluation in Mininet demonstrate that under realistic network conditions, continuous PPCU updates provide better throughput and complete more flows compared to random updates, without violating safety.

4.11 Proof of the Algorithm

We need to prove that the algorithms in section 4.3 together provide PPC updates.

p is an affected packet with a label *label*, that can take the values U_p, NEW_p or OLD_p, indicating if the packet is unaffected, new or old respectively.

Let us assume that the individual algorithms at the data and control planes are correct. Let us assume that switches are synchronized, to make descriptions easier and that no conflicting RUs occur. However, the controller-switch links may have variable delays and the switches may have varying execution speeds.

Case 1: s_f has not received Commit and so it is unaware of an update. At time T_f, packet p arrives with time stamp TS_p. $TS_p \leq T_f$ by definition (TS_p was the time at the ingress switch when the packet left it and s_f may be an ingress). It will match with some existing (old) rule, and the packet will be forwarded. *label* will remain U_p. So p has been handled by old rules. Now, is it possible that some switch downstream of s_f uses new

rules on p? This is not possible.

Suppose a new rule has been installed in a downstream switch, say s_p, and either Commit has been received (T=T_{max} for all affected rules), or both Commit and Commit OK have been received (T=T_{last} for all affected rules). Now when p matches a new rule, the check in line 4 of Algorithm 4 will find that TS_p<T of the new rule. If only Commit has been received, T=T_{max}, and TS_p<T_{max} by definition, for any TS_p. At time T_f, since s_f has not received Commit, it has not received Commit OK. T_{last} is the largest of the values of time received in Ready To Commit sent by all the affected switches, including s_f. Hence T_{last}>T_f. Since $T_f \geq TS_p$, it follows that TS_p<T_{last}. So *label* of packet p will be made OLD_p (line 11 of Algorithm 4) and a match with an old or unaffected rule will take place. Once a packet is labelled OLD_p, it is always handled by old rules (line 16 of Algorithm 4) as long as old rules exist, and so all subsequent switches will also handle p with old rules.

Case 2: s_f has received Commit but not Commit OK. s_f will have new rules, but in all of them T=T_{max}. So, as discussed above, s_f will find that TS_p<T in any new rule, and old rules will apply. *label* of p will be made OLD_p (line 11 of Algorithm 4). Once so labelled, all subsequent switches will use old rules on p.

Case 3: s_f has received both Commit and Commit OK. s_f will match p with a new rule, and if $TS_p \geq T$, it will apply the new rule and set the label to NEW_p (line 21 of Algorithm 4). Once a packet is labelled NEW_p, all subsequent switches will use new rules irrespective of the value of T and TS_p (line 21). All subsequent switches will have new rules already installed since Commit OK has been received by s_f, and this can happen only if all affected switches have received Commit and have sent Ready to Commit (steps 2 and 3 of Figure 4.1) to the Controller.

If s_f finds that TS_p<T, it will use old rules, and change the label to OLD_p. Similar to earlier cases, all subsequent switches will now use old rules.

The above cases will hold as long as both old rules and new rules co-exist in the affected switches. Old rules are discarded and checking for rule type ceases once a timer with time T_m expires at a switch. Once s_f receives Commit OK, it sends Ack Commit OK with the current time to the controller (step 4 of Figure 4.1). Once the controller receives Ack Commit OK from all switches, it sends a Discard Old message with T_{del} set to the time of the last Ack Commit OK sent (step 5 of Figure 4.1). The timer value T_m

is then set by each switch as $T_{del} + M - T_i$, to account for the elapsed time from T_{del} to T_i. After time T_{del}, no packet with time stamp greater than T_{del} will be switched with old rules. So the last packet to be using old rules will have $TS_p \leq T_{del}$. By definition the maximum lifetime of a packet in the network is M. So after time $T_{del} + M$, no old packet will be there in the network and so the old rules can be deleted. This ensures that old rules are not discarded prematurely. Packets with label NEW_p will still be there in the network, but since switches will have converted the new rules to type U, the new rules will continue to apply.

It has to be ensured though, that the next conflicting update does not start too soon so that a packet with label NEW_p (but of the previous update) is not erroneously handled. To take care of this, the next update is not started until after another time period of M has elapsed. No packet will have its *label* set to NEW_p if its time stamp $TS_p > T_{del} + M$, since new rules will no longer exist at any switch. Waiting for time M will ensure that all such new packets leave the network before the next update begins.

Part II

Per-Flow Consistent Updates

Chapter 5

EPCU-SRT: Enhanced Per-Flow Consistent Updates

5.1 Introduction

We have examined in detail in chapter 2 the need to preserve per-flow consistency (PFC). In this chapter, an algorithm, EPCU-SRT, to preserve PFC to address the concerns during Server Load Balancing (section 2.8.4 chapter 2) and to prevent packet re-ordering (section 2.8.5 in chapter 2) is presented. This algorithm assumes that when the path of a flow in a network is changed, the sequence of stateful switches and the NF instances (middleboxes) through which the flow is being routed, and the instances themselves, are not changed (waypoint invariance is maintained).

5.2 Algorithm for Per-Flow Consistent Updates

5.2.1 Switch model

The same switch model used in chapter 1 in section 1.3 is used. There is a Software Rules Table (SRT) associated with the TCAM such that the switch checks if a matching rule exists in the TCAM first and if it does not, it checks the SRT. For moving rules into and out of the TCAM algorithms such as CacheFlow [67] may be used.

5.2.2 Enhanced Per-Flow Consistent Update with SRT (EPCU-SRT)

EPCU-SRT relies on generating microrules, in which the match for the rule is based on the header fields that uniquely identify a flow. We restrict microrule generation between the switch receiving a "Commit" and "Commit OK". We also specify a method to delete the microrules. The solutions by Wang et al. [128] explained in section 2.8.4 of chapter 2, either require buffering packets at the controller or fractional processing of rules, leading to installation of additional rules, and the likelihood of the RU not completing and not being immediately effective. The solution by Reitblatt et al. [111] requires generating a microrule for *every* flow. Our solution does not require buffering packets at the controller. We generate exact rules and take advantage of the SRT to reduce concerns on rule space.

In switches with an SRT, the algorithm is as given below, assuming one flow table (Figure 5.1). The figure illustrates the case where an ingress is not an internal switch.

1. The controller sends "Commit" with the new rules to all the affected switches. All the affected *internal* switches install the new rules into the SRT. The *ingress* switches do not yet install the rules that tag packets with $v1$ or policy rules but store the rules internally. An ingress switch also allows microrule creation.

   When an *ingress* receives a "Commit", the ingress demotes the set of $v0$ rules to the SRT, if the rules currently exist in TCAM. Next, if a packet matches an old $v0$ rule, it installs a microrule for that flow in the SRT, if one does not already exist, and associates a timer with it that will time out and delete the rule after T_g units of inactivity. These microrules have higher priority than the old $v0$ rules. This process continues until $v1$ rules are installed later.

   If flows are assumed to be implemented by TCP, then the following is added to each microrule. Along with a match for the header fields associated with a flow, FIN, ACK and RST are also matched. Two variables are kept for each microrule: FINreceived, and FINACKreceived. When there is a FIN match, FINreceived is set to true and if FINACKreceived is found to be set, the microrule is removed as the flow has ended. If a FIN and ACK both match, then FINACKreceived is set and if FINreceived is already set, the microrule is removed. Finally, if there is an RST match, the microrule is deleted. If a microrule is not removed by the above matches,

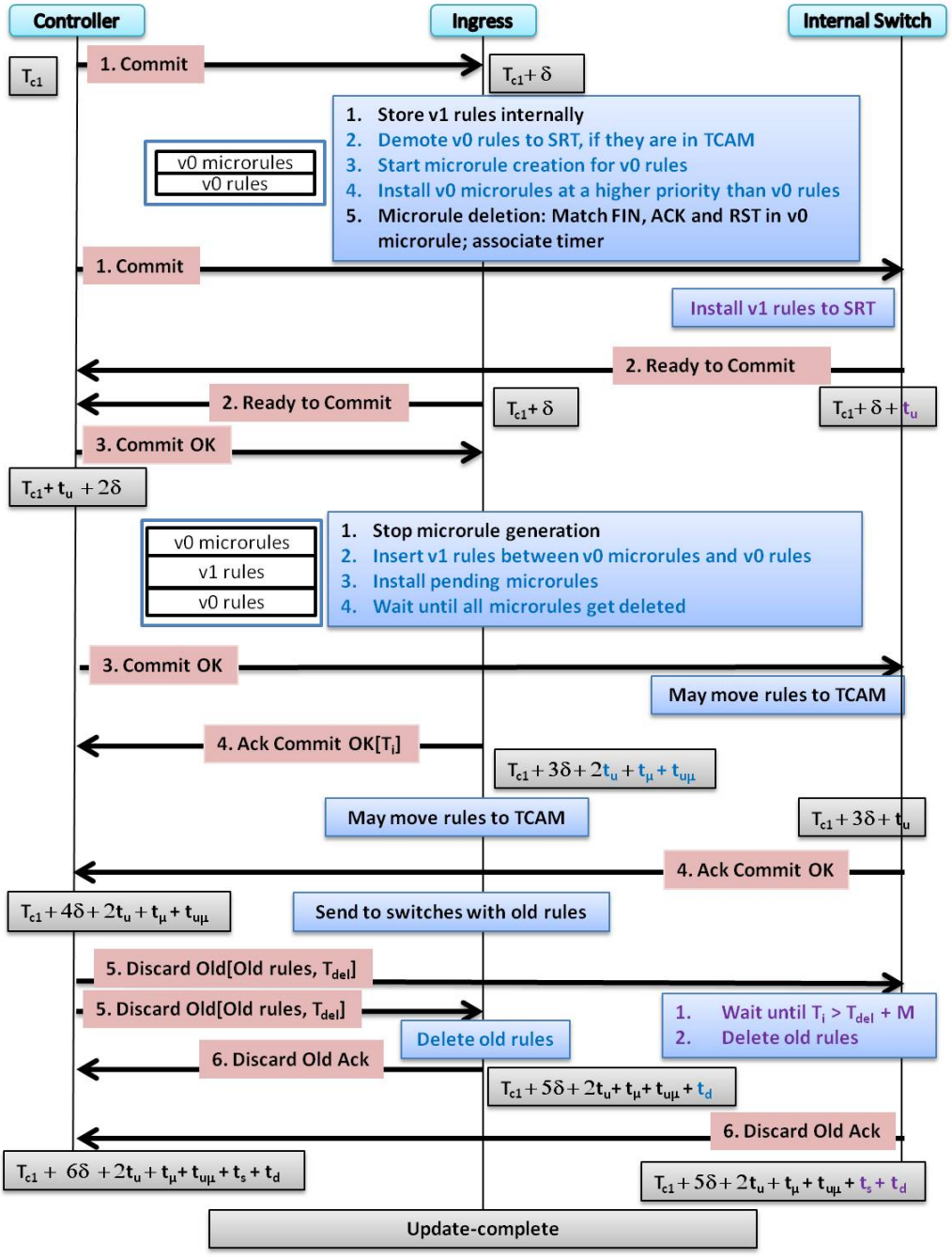


Figure 5.1: EPCU-SRT: Algorithm at the control plane

then it is removed when the timeout associated with it is triggered. This is discussed further in 5.2.3.

2. All the ingress switches and the switches that have acted upon "Commit" send back "Ready to Commit".

3. The controller receives "Ready to Commit" from all the switches and then and only then sends "Commit OK" to *all* the switches. As soon as the ingress switches receive "Commit OK", they stop generating further microrules. They insert $v1$ rules in between the $v0$ microrules (which have the highest priority) and the old $v0$ rules (which have the lowest priority). The above actions are atomic. After receiving "Commit OK", an *internal* switch may move the $v1$ rules to TCAM at any time. All new flows now use $v1$ rules.

4. Each ingress switch sends "Ack Commit OK" only *after all its* $v0$ *microrules are removed*. The internal switches send "Ack Commit OK" soon after receiving "Commit OK". Each ingress switch sends "Ack Commit OK" to the controller with the current time, called T_i. After all the $v0$ microrules are deleted, an *ingress* switch may move the $v1$ rules to TCAM at any time; however, it is not necessary to do so.

5. After receiving "Ack Commit OK" from all the affected *ingresses*, the controller notes the latest value of T_i received and saves it as T_{del}. Now it sends "Discard Old" to all the switches where rules need to be deleted. It sends T_{del} and the $v0$ rules to be deleted as a part of "Discard Old".

6. When each *internal* switch receives "Discard Old", it deletes the list of rules received in "Discard Old", whenever its current time $T_i > T_{del} + M$, where M is the maximum lifetime of a packet within the network. The ingresses may delete immediately.

7. Each switch that processes "Discard Old" sends a "Discard Old Ack". When the controller receives all the "Discard Old Ack" messages the update is complete.

5.2.2.1 Updates to a load balancing switch

Suppose, in section 2.8.4, Figure 2.2 of Chapter 2, all flows originating from both IP_1 and IP_2 are given a destination IP address of IP_{da} by the load balancing switch s_b. All switches

have $v0$ rules. All flows originating from IP_2 must be now diverted to IP_{db}, instead of IP_{da}, as explained in section 2.8.4 of Chapter 2, to reduce the load on IP_{da}. Upon receiving Commit, the ingress of IP_2, s_g, (not shown), starts generating microrules for those flows for which it receives packets and tags them as $v0$. Upon receiving Commit OK, it installs the $v1$ rules that tag packets with the version number $v1$, at a lower priority than the microrules. Similarly, s_b, which is the egress for the flow, upon receiving Commit, starts generating $v0$ microrules and upon receiving Commit OK, installs $v1$ rules with a lower priority than those microrules. Let us assume that a flow is in existence from IP_2 when the RU began. The microrules of s_b tag packets as $v0$ and s_b therefore sends even those with their source address as IP_2 to IP_{da}. The packets originating from IP_{da} to IP_2 traverse s_b and are switched, again, using $v0$ microrules. When s_g (s_b) receives Commit OK, it stops microrule generation, waits until the existing $IP_2 - IP_{da}$ ($IP_{da} - IP_2$) flows cease and then sends Ack Commit OK. If a new flow arrives from IP_2 after s_g receives Commit OK, s_g sends it to s_b, with a $v1$ tag. s_b applies the new rule to it, which sends the packets of the flow to IP_{db}. The packets from IP_{db} to IP_2 reach s_b and are sent through s_g to IP_2, as the destination IP address of the reverse flow does not change. Thus flows existing at the time of the update with their source address as IP_2 continue to be terminated at IP_{da}, while flows after s_g receives Commit OK are terminated at IP_{db}. In this description, s_b may also be an internal switch and the algorithm would still apply.

5.2.3 Value of the Microrule Expiry Timer T_g

This value will be protocol dependent. However, if we assume TCP, as per the TCP protocol definition ([106] and [23]), a connection can remain alive without any upper limit on idle time. However, in practice, most implementations have time-outs, which if triggered, results in the connection being dropped. So the microrule timer should expire when there are *no packets on the connections for this amount of time*. Since such time-outs are large compared to network time to live (Linux implementations have a maximum keep-alive duration of 75 seconds, implying that a connection will get closed if there is no activity for 75 seconds), problems of re-ordering of packets and of broken connections due to load balancing will be taken care of.

Table 5.1: Symbols Used in the Analysis of EPCU

$t_{u\mu}$	The time required after receiving a "Commit OK" to install (and delete) the microrules pending installation (and deletion), if any
t_{μ}	The time that the ingress waits until all its remaining microrules get deleted, after installing $v1$ rules

Table 5.2: Analysis of EPCU

Parameter	**EPCU-SRT**	**EPCU-SRT′**
1, Case 1.1	$2\delta + t_{\mu} + t_d$	$2\delta + t_{\mu} + t_d$
1, Case 1.2	$4\delta + t_u + t_{u\mu} + t_{\mu} + t_d$	$4\delta + t_u + t_{u\mu} + t_{\mu} + t_d + t_{dt}$
1, Cases 2,3	NA	NA
1, Case 4	$4\delta + t_u + t_{u\mu} + t_{\mu} + t_s + t_d$	$4\delta + t_u + t_{u\mu} + t_{\mu} + t_s + t_d + t_{dt}$
2	$3\delta + 2t_u + t_{u\mu}$	$3\delta + 2t_u + t_{u\mu} + t_{dt}$
3	$2k_o + 6k_c + 4k_n$	$2k_o + 6k_c + 4k_n$
4	$6\delta + 2t_u + t_s + t_d + t_{u\mu} + t_{\mu}$	$6\delta + 2t_u + t_s + t_d + t_{u\mu} + t_{\mu} + t_{dt}$
5	$2\delta + 2t_u + t_{u\mu} + t_{\mu}$	$2\delta + 2t_u + t_{u\mu} + t_{\mu} + t_{dt}$

5.3 Switches without an SRT

If the switches in a network do not have an SRT, when an internal switch receives a "Commit" with the new rules, it installs the rules in the *TCAM* and sends back "Ready to Commit". The controller, on receiving "Ready to Commit", needs to send "Commit OK" to *only the ingresses*. The ingresses install the new $v1$ rules in the TCAM. This behaviour is the same as in section 3.2.4. The rest of the algorithm remains the same. Since the microrules also get installed in the TCAM, the usage of TCAM becomes very high and the update becomes slow. Also, if the controller wishes to abort the update, the switches that have already installed the rules need to incur the overhead of deleting them from the TCAM. Section 5.4 mentions further comparisons.

5.4 Analysis of the Algorithm

We use and add to the parameters of interest identified in chapter 3, section 3.2.5, for a PFC update: 1) **Parameter 1**: Duration for which the old and new rules exist at each type of switch 2) **Parameter 2**: Duration within which new rules become usable 3) **Parameter 3**: Message complexity - the number of messages required to complete the protocol 4) **Parameter 4**: Time complexity - the total update time and 5) **Parameter 5**: Duration for which microrules are present at the ingress.

The purpose of the analysis is to understand what the above depend upon.

The symbols used in the analysis is as per Table 5.1 of this chapter and Table 3.2 of chapter 3. It is assumed that the time taken for the sum of the propagation times and switch delays between the controller and the switches is uniform (δ). Let the time taken for all insertions (t_u) and deletions (t_d) be uniform and let the processing time at each switch be negligible. The time at which the last switch sends "Ready to Commit" is $T_{c_1}+\delta+t_u$, assuming the sum denotes the longest time taken. Let the number of rules that need to be removed (n_o), added to switches in general (n_n) and added to the ingress to meet its ingress functions (n_i) be uniform across switches.

We need to consider different kinds of switches while evaluating various parameters: Case 1) ingress switches, with Case 1.1 where the ingress switch is not an internal switch and Case 1.2 where the ingress switch is also an internal switch for update affecting a flow for which this is not the ingress, Case 2) internal switches where new rules do not need to be installed but old rules need to be removed, Case 3) internal switches where only new rules need to be installed and Case 4) internal switches where old rules need to be removed and new rules need to be added. For Case 1.2, to simplify the presentation, it is assumed that there are no old internal rules to be deleted. For all ingresses, it is assumed that old ingress rules need to be removed and new ingress rules added.

For PFC, the ingress starts generating and updating the SRT with microrules from when it receives a "Commit" until it installs $v1$ rules. During this interval, it is possible that some microrules get deleted too. Let us assume that it takes $t_{u\mu}$ more units of time after receiving "Commit OK" to install (and delete) the microrules pending installation (and deletion), if any, in the SRT, before it starts installing $v1$ rules. t_u is the time that the ingress takes to install $v1$ rules. Let t_μ be the time that the ingress waits until all

its remaining microrules get deleted, after installing $v1$ rules. So there is an additional amount of time $t_{u\mu} + t_\mu$ that the ingress incurs between it receiving "Commit OK" and before it sends "Ack Commit OK", compared to E2PU-SRT. Let there be n_μ microrules in the switch, just before the $v1$ rules are installed.

The time elapsed at each stage of the RU is shown in Figure 5.1. Let us assume that the v_0 rules of an ingress are in the SRT and the ingress has no new rules to install. Then at step 2, when an ingress sends Ready To Commit, the time is $T_{c_1} + \delta$. Since an internal switch has installed the v_1 rules, it sends Ready To Commit at $T_{c_1} + \delta + t_u$. Before the ingress sends Ack Commit OK, it takes t_u units of time to install the v_1 rules and $t_{u\mu}$ units to install pending microrules, if any. It takes an additional t_μ units of time until all microrules get deleted, either due to a TCP connection closing our due to a rule timeout. Thus the time when an ingress sends Ack Commit OK is $T_{c_1} + 3\delta + 2t_u + t_{u\mu} + t_\mu$. An internal switch sends Ack Commit OK at $T_{c_1} + 3\delta + t_u$. An ingress sends Discard Old Ack after it deletes the v_0 rules, at $T_{c_1} + 5\delta + 2t_u + t_{u\mu} + t_\mu + t_d$, whereas an internal switch waits for an additional amount of time, t_s. An internal switch therefore sends the Discard Old Ack at $T_{c_1} + 5\delta + 2t_u + t_{u\mu} + t_\mu + t_d + t_s$.

Parameter 1: Duration of overlap

Case 1.1, the ingress switch is not an internal switch: When the ingress switch receives a Commit, there are n_o rules in the switch. At $T_{c_1} + 3\delta + 2t_u + t_{u\mu}$, there are $n_o + n_\mu + n_i$ rules (t_μ is not included as the ingress microrules are not yet deleted). At $T_{c_1} + 5\delta + 2t_u + t_{u\mu} + t_\mu + t_d$, there are $n_n + n_i$ rules. Therefore the time for which the old and the new rules coexist in the switch is $T_{c_1} + 5\delta + 2t_u + t_{u\mu} + t_\mu + t_d - (T_{c_1} + 3\delta + 2t_u + t_{u\mu}) = 2\delta + t_\mu + t_d$.

Case 1.2, the ingress switch is also an internal switch: When the ingress switch receives a Commit, there are n_o rules in the switch. At $T_{c_1} + \delta + t_u$, there are $n_o + n_n$ rules. At $T_{c_1} + 3\delta + 2t_u + t_{u\mu}$, there are $n_o + n_n + n_i + n_\mu$ rules. At $T_{c_1} + 5\delta + 2t_u + t_{u\mu} + t_\mu + t_d$, there are $n_n + n_i$ rules. Therefore the time for which the old and the new rules coexist in the switch is $T_{c_1} + 5\delta + 2t_u + t_{u\mu} + t_\mu + t_d - (T_{c_1} + \delta + t_u) = 4\delta + t_u + t_{u\mu} + t_\mu + t_d$.

For cases 2 and 3, where either only new rules need to be installed or only old rules to be removed, there is no overlap of rules.

Case 4, switches where old rules need to be removed and new rules need to be added: At $T_{c_1} + \delta + t_u$, there are $n_o + n_n$ rules and at $T_{c_1} + 5\delta + 2t_u + t_{u\mu} + t_\mu + t_s + t_d$ there are n_n rules. The duration of overlap is $4\delta + t_u + t_{u\mu} + t_\mu + t_s + t_d$.

Parameter 2: Duration within which new rules become usable

At $T_{c_1}+3\delta+2t_u+t_{u\mu}$, the ingress switches to the new rules and they are immediately usable. From the beginning of the update at T_{c1}, after $3\delta + 2t_u + t_{u\mu}$ units, the new rules are usable.

Parameter 3: Message Complexity A total of $2k_o+4k_c+2k_n+2k_i$ messages are sent, just as in E2PU-SRT in chapter 3, if no SRT is present.

For EPCU-SRT, a total of $2k_o+6k_c+4k_n$ messages are sent, which is more than the case when an SRT is not used, just as in per-packet consistent updates.

Parameter 4: Total Update Time The total update time is $6\delta + 2t_u + t_s + t_d + t_{u\mu} + t_\mu$.

When an SRT is used, parameters 1, 2 and 4 improve because the values of t_u, $t_{u\mu}$ and t_d reduce significantly.

If, at the ingress, the deletion time of rules is lesser than the insertion time, then rules can be installed at the ingress when "Commit" is received, at a lower priority. The old rules can be deleted when "Commit OK" is received. Thus new rules can be used faster, within a time of $3\delta+t_u+t_d+t_{u\mu}$. The total update time reduces to $6\delta+t_u+t_s+2t_d+t_{u\mu}+t_\mu$.

Parameter 4: Duration for which Microrules are Present at the Ingress Microrules are present from $T_{c_1} + \delta$ to $T_{c_1} + 3\delta + 2t_u + t_{u\mu} + t_\mu$, for a duration of $2\delta + 2t_u + t_{u\mu} + t_\mu$.

Table 5.2 captures all the parameters for a PFC update. The new rules are considered usable even when microrules are present (parameter 2).

If v_0 rules of an ingress are in the TCAM, all the timing related parameters worsen and are shown under EPCU-SRT′. In this case, before sending Ready To Commit, an ingress must delete the v_0 rules from TCAM and install it in the SRT, taking an additional time of $t_u + t_{dt}$, before sending Ready To Commit.

Observations: The observations are the same as in chapter 3 section 3.2.5. Additionally, if there are long-lived connections, t_μ (this is the time-out for a flow to be closed if there is no traffic) will dominate all parameters - either they must be excluded from updates (further discussed in section 6.3.5.8) or applications must use EPCU only short connections. For switches with SRT, the new rules get installed fairly quickly and the old rules are removed as soon as is feasible.

5.5 Conclusions

This chapter described an update algorithm for PFC, exploiting the availability of an SRT, suitable for PFC updates for Load Balancing switches and cases where waypoint invariance is preserved and stateless switches are not used. We also analyzed the algorithm quantitatively.

EPCU, like the solution by Reitblatt et al. [111], requires the creation of rules that exactly match the old flows, dynamically, on the affected switches, which is not supported on real switches. Besides, all the paths affected by the RU must be found and rules with the new version installed only in all the tables in all the switches in those paths, which is computation intensive and inefficient, especially so if wildcarded rules are used in switches. Moreover, these solutions do not consider synchronizing the forward and reverse flows, as the time at which the ingress for a forward connection receives Commit OK is not synchronized with the time at which the ingress of the reverse connection receives Commit OK. Synchronising forward and reverse flows is required if stateful switches or NFs exist on the packet path. For real implementations, it is desirable that rules in every switch in the network or every switch in the affected path are not modified for every update. Our algorithm ProFlow, discussed in the next chapter, solves all the above problems.

Chapter 6

Proportional Per-Flow Consistent Updates

6.1 Introduction

The algorithm ProFlow presented in this chapter provides per-flow consistency for any kind of update, be it to a Service Chain, to a Server Load Balancing switch or to switches in a virtualised network, as explained in Chapter 2. The network may contain stateful NFs or switches.

6.2 Our contributions

ProFlow has the following characteristics in addition to the ones mentioned in section 1.5 of chapter 1: 1) **Efficiency**: The algorithm confines changes to the affected switches and the affected rules. Thus the number of switches actually modified for the RU is *proportional* to the number of affected switches. It does not require forwarding packets to temporary storage or does not require the affected paths to be computed. 2) **Immediate effectiveness:** A long duration old flow will not prevent a new version of a rule r from being installed and used by new flows, while it will continue to use the old version of rule r. 3) **Causal synchrony:** We use a mechanism to *causally* synchronize the reverse flow with the forward flow to preserve PFC. 4) **Tolerates asynchrony:** The algorithm requires data plane time stamps, but tolerates practical asynchrony of time across the network. 5) **Requires no changes to NFs.**

Due to the general nature of the algorithm, it solves a number of problems for some of which specific solutions have been proposed in the literature. In particular, it solves the following problems: 1) providing connection affinity when a) flows are re-routed from one SC with switches that may maintain states, to another, in a non-virtualized [61] [1] and virtualized network and b) an NF is added to or removed from an SC in a non-virtualized and virtualized network 2) providing consistent flows in a load balancing switch update [128]. 3) preventing packet-re-ordering during updates [111].

Assumptions: 1) All the switches use a synchronized real time clock using protocols such as PTP or ReversePTP [89]. If there is time asynchrony between switches, it has a known maximum value. 2) It is possible to add a time stamp to a packet header, which is the case with programmable switches [22]. 3) Flows use TCP. 4) Flows are assumed to be completed when they are quiescent for a configurable amount of time (denoted as T_p). [2] 5) The network does not run applications that require a peer to peer connection, which, in turn, requires a simultaneous TCP open, a reasonable assumption in data centres. 6) The egress of a forward flow must be the same as the ingress of a reverse flow.

Summary of the algorithm: Let the latest time at which all the switches in S install new rules be T_{last}. Each affected switch examines the time stamp TS_p, set to the current time by the ingresses, in each data packet. If its value is less than T_{last}, the flow and the packet are marked old and the packet is switched according to old rules. If it is greater than or equal to T_{last}, 1) if it is a SYN (with ACK not set) the flow and the packet are labelled new and 2) if it belongs to a flow that started before T_{last}, the flow and the packet itself are labelled old. The packets marked old are subsequently switched according to old rules, on all the affected switches and the packets marked new are switched according to new rules. Once a flow is marked new (old), all its packets are subsequently switched only according to new (old) rules, regardless of the value of its time stamp. Upon receiving a SYN labelled new, the egress through which the flow exits stores the state of the flow as new and uses a different label to label the SYN+ACK of *its* reverse flow as new. The first affected switch that receives the reverse flow labels such a packet as new, regardless of the state of that switch, thus *causally* synchronizing the reverse flow with the forward

[1] Softcell [61] is supported only for a specific wireless network architecture

[2] If there are long running flows with inter-packet delays larger than the configured value, during an RU (or a series of RUs) that affects them, packets belonging to them will get switched preserving *PPC*.

flow.

6.3 Algorithm for concurrent consistent per-flow updates

6.3.1 The switch model

The switch model used is the same as in section 1.3 of Chapter 1. Optionally, a time stamp T, a rule type *rule_type* and a version V are associated with each rule. These optional values are not used while matching a packet but actions may read from them. A rule is represented as $a{=}[Q, M, A, T, rule\_type, V]$. T, *rule_type* and V are mentioned henceforth only if they are relevant to the context of the discussion.

6.3.2 Challenges in achieving PFC

It is not sufficient to check the value of TS_p in the packet header, as explained in chapter 4 for PPCU, and additionally the state of a flow (stored as old, new or unaffected) for every packet, as presented in the simplified summary in section 6.2, to maintain PFC. The challenges are listed below.

6.3.2.1 Detecting the end of a flow

It is not possible for an affected switch $s_i{\in}S$ to keep track of when a flow has ended: 1) An application may time out and consider the connection closed without sending any packets to close the connection. 2) If an ACK in response to a FIN is lost by the time it reaches s_i, the sender of the FIN will timeout and consider the connection closed. 3) If s_i is on the path of only the forward flow and the host generates an RST, it may not traverse s_i, in which case, s_i may consider the flow to be merely quiescent. Therefore, switch s_i assumes that a forward (or reverse) flow has ended if it does not receive any packet in the flow for time T_p since the receipt of the last packet in that flow, instead of trying to infer the state of the flow from the canonical signalling headers that end the flow [128]. [3]

[3]In EPCU-SRT (chapter 5), it was sufficient for ingresses to detect the end of a flow and delete the corresponding microrule based on a timeout if no packet matched the microrule for T_g units of time. Here, a *rule* detects the end of a flow and no changes are made to ingresses unless they are affected. However,

6.3.2.2 Synchronising forward and reverse flows

If and only if a forward flow is marked new, its corresponding reverse flow must be marked new. To begin a flow, both ends of the connection must send a SYN. Even if the SYN is lost, it will be resent. Therefore, to detect the beginning of a flow and to synchronize the forward and reverse flows of a connection, we make use of the fact that a SYN must cross the egress of the forward flow and its response must cross the same switch that its SYN traversed on the way out of the network.

6.3.2.3 Flows not completing before an RU ends or multiple instances of the same flow during an RU

All old flows must complete before an RU ends, but new flows will continue to exist. Both old and new flows can be shorter than the RU that affects them and new flows can be longer than the RU that affects them. Examples of some situations that need to be handled on account of this are given below:

E1: Forward flow and reverse flow belonging to different RUs: An RU, RU_1, affecting a forward flow that is deemed new may complete before the first packet p_r of its reverse flow originates from the host. A conflicting RU, RU_2, may start next and p_r may reach the network during this RU. Now p_r must not be considered to belong to a new flow by RU_2.

E2: Flow spanning RUs: An RU, RU_1, affecting a forward flow f that is deemed new, may complete before the flow is complete. The state of f is stored as new at an affected switch. If a conflicting RU, RU_2, begins after RU_1 is complete, f must be considered old, as it is already in existence when RU_2 began.

E3: Multiple instances of the same flow in an RU: A forward flow f that is deemed old by an RU RU_1 completes much earlier than RU_1. There are other old flows that are ongoing. If another instance of the same flow f (that is, matching the same 5-tuple) starts before RU_1 ends, f may be required to be considered new, depending on what stage of RU_1 it began.

---

the rule itself can be deleted only when it is certain that there are no packets matching that rule in the network.

Variable	Values	Initial value	Description
TS_p	0 to T_{max}	Current time at the ingress	Stored when the packet enters the network
label	NEW_p, OLD_p, U_p	U_p	The rule type that must be applied to the packet
t_label	NEW_p, U_p	U_p	The rule type that must be applied to the packet of the reverse flow

Table 6.1: Fields added by ingress switches

6.3.3 Addition of headers at the ingress:

As shown in Table 6.1, each packet p entering the network has *a time stamp field* TS_p and two labels, *label* and *t_label*, added to it at the ingress and removed from it at the egress, programmatically. *label* can take the values NEW_p, OLD_p or U_p depending on whether the new, old or unaffected rules need to be applied to the packet and *t_label* can take the values NEW_p or U_p. All ingresses set TS_p to the current time at the switch and *label* and *t_label* to U_p, for all packets entering it *from outside the network*, unless mentioned otherwise. In addition to the above, the ingress and egress further process packets, and this is explained later in Algorithm 6.

6.3.4 Algorithm at the control plane

The control plane (CP) of the switch exchanges messages with the controller and configures the data plane (DP) of the switch, as shown in Figure 6.1. The stateful lists that are used by the CP and the DP to realize Proflow, the indices used for those lists, the values each item may take and their purpose, are shown in Table 6.2. In the description of algorithms, these lists are not always shown with their indices. The table also shows which entity writes values into each item and which entity reads the values. The value of T associated with each rule is used by the rule to compare packet time stamp values. It is initialized to T_{max}, for every rule. The value of *rule_type* associated with each rule is initialized to U, indicating that the rule is unaffected. This section specifies the algorithm for the control plane, at the controller and at each $s_i \in S$ in Figure 6.1. The message exchanges below are the same as in the algorithms in previous chapters ; the parameters in

Table 6.2: Stateful lists used by the affected and ingress switches

Variable [index]	**Used by**	**Written by**	**Read by**	**Values**	**Initial value**	**Purpose**
T $[n]$	s_i	Controller	DP	0 to T_{max}	T_{max}	To compare with the packet time stamp to decide *label* of a packet
$rule\_type$ $[n]$	s_i	Controller	DP	NEW, OLD or U	U	Indicates the type of rule
$state$ $[d\_hash]$	s_i	DP	DP	NEW_FL, OLD_FL	OLD_FL	Indicates the type of flow
$prev\_t$ $[d\_hash]$	s_i	DP	DP	0 to T_{max}	0	Stores the time stamp of a packet as it exits a switch
$live\_fl$ $[V]$	s_i	DP	CP	0 to T_{max}	0	Indicates if there are any old flows alive, using a packet time stamp.
ev $[V]$	s_i	CP	DP	$START$, $STOP$	$STOP$	Indicates if DP must update $live\_fl$
T_p $[V]$	s_i	Controller	CP	0 to T_{max}	T_{max}	If the inter-packet delay is less than this value, an old flow is considered alive
T_{RC} $[V]$	s_i	Controller	CP	0 to T_{max}	T_i	Time at which Ready To Commit was sent. Used to distinguish new and old flows
V $[n]$	s_i	Controller	CP	0 to K	0	Version number of an RU

$istate$ $[hash]$	ingress	DP	DP	NEW_FL, U_FL	U_FL	Indicates type of flow
$itstamp$ $[hash]$	ingress	DP	DP	0 to T_{max}	0	The time stamp of SYN

DP: Switch Data Plane, CP: Switch Control Plane, n: rule number, v: RU identifier, T_{max}: 1 less than the maximum value the item can hold, T_i: Current time at switch i, $hash$: Hash of the 5-tuple (with the same value for forward and reverse flows), d_hash: Hash of the 5-tuple (with a different value for forward and reverse flows), s_i: an affected switch, K: Maximum number of simultaneous RUs

them and the actions upon receiving them have been modified for a PFC update.

1. The Controller, upon receiving an RU from the application, with S, and R_{oi} and R_{ni} for every $s_i \in S$, sends a "Commit" to every $s_i \in S$ with v, R_{oi} and R_{ni}. v is a unique RU identifier, used only between the controller and the switches.

2. The CP of every switch $s_i \in S$ receives "Commit", a) installs the R_{ni} rules with a higher priority than R_{oi} rules, b) changes the installed R_{oi} rules to check if a packet is labelled OLD_p or U_p by changing their match fields (R_{ni} rules check if an incoming packet is labelled NEW_p or U_p), c) sets $rule\_type$ of the R_{oi} rules as OLD and R_{ni} rules as NEW, d) sets V of each affected rule to v, denoted by $V[n]{=}v$. $V[n]$ is henceforth referred to as V. e) sets T of each affected rule to T_{max}, f) sends "Ready to Commit" with T_i and g) sets $T_{RC}[V]{=}T_i$. The above actions, per changed rule, must be atomic.

3. The controller, upon receiving "Ready to Commit" from all $s_i \in S$, updates T_{last} to reflect the largest T_i received and sends "Commit OK" to all $s_i \in S$, with T_{last} and t_p. As long as the inter-packet delay of an old flow is less than t_p, the flow is considered alive.

4. Upon receiving "Commit OK", the switch CP a) sets $T_p[V]{=}t_p$ b) sets $T{=}T_{last}$ in R_{oi} and R_{ni} c) sets ev to $START$, to inform the DP that it has to begin updating $live\_fl[V]$ (explained subsequently) to indicate that there are live old flows and d) starts a timer of value $2 * T_p$. All the above actions after receiving Commit OK are atomic, per rule. Upon expiry of the timer, if $live\_fl[V]$ has increased from the previous time it was checked, there are live old flows, in which case the timer is restarted. Otherwise, it considers all the old flows complete, sets ev to $STOP$ and sends "Ack Commit OK" to the controller, with T_i.

5. Upon receiving "Ack Commit OK" from all $s_i \in S$, the controller sets the largest of T_i received in "Ack Commit OK" to T_{del}. It sends "Discard Old" to all $s_i \in S$ with T_{del}.

6. Upon receiving "Discard Old", the switch CP starts a timer $T_{del} + M - T_i$, where M is the maximum lifetime of a packet within the network. When the timer expires, as

all the packets that were switched using R_{oi} are no longer in the network, the switch deletes R_{oi}. It marks all the rules in R_{ni} as unaffected by setting their *rule_type* to U and modifies their match fields such that they cease to check for the packet label NEW_p. T need not be initialised to T_{max} as packets matching unaffected rules do not check for T. Next, it sends "Discard Old Ack" to the controller. the update at the switch is complete.

7. After the controller receives "Discard Old Ack" from all $s_i \in S$, the RU is complete at the controller. After M units after timer expiry at the last $s_i \in S$, the last of packets that have *label* set to NEW_p are no longer in the network. Now the next update not disjoint with the current one may begin.

6.3.5 Algorithm at the data plane

6.3.5.1 Introduction to the data plane algorithms

As defined in chapter 4 section 4.3.5, the first switch $s_i \in S$ that changes the label of a packet from U_p to NEW_p is called the *first affected switch*, s_f, of that packet. An RU can have more than one s_f. s_f of the forward flow is called s_{ff} and that of the reverse flow is called s_{fr}. [4]

When s_{ff} (s_{fr}) receives Commit OK, it becomes aware that all the affected switches have installed new rules. Therefore, the packets belonging to flows whose SYN cross s_{ff} from now on may be switched using new rules without violating PFC. However, all the packets of all the flows that cross s_{ff} (s_{fr}) currently, other than those of new flows, that is, the old flows, must continue to get switched using old rules, to preserve PFC. If a packet matches an affected rule, s_{ff} (s_{fr}) must set *label* of the packet as NEW_p, for the packet to be switched by new rules, and OLD_p, for the packet to be switched by old rules, by the rest of the affected switches. In addition, it is possible that s_{fr} receives the first packet of the reverse flow before it receives Commit OK, while the corresponding forward flow is switched using new rules. In this case, s_{fr} needs to be informed that it must switch the reverse flow using new rules, in spite of not receiving Commit OK. The

[4] The algorithms are the same for all the affected switches. The controller and the affected switches do not know which switch s_{ff} or s_{fr} is. This is defined only for ease of exposition.

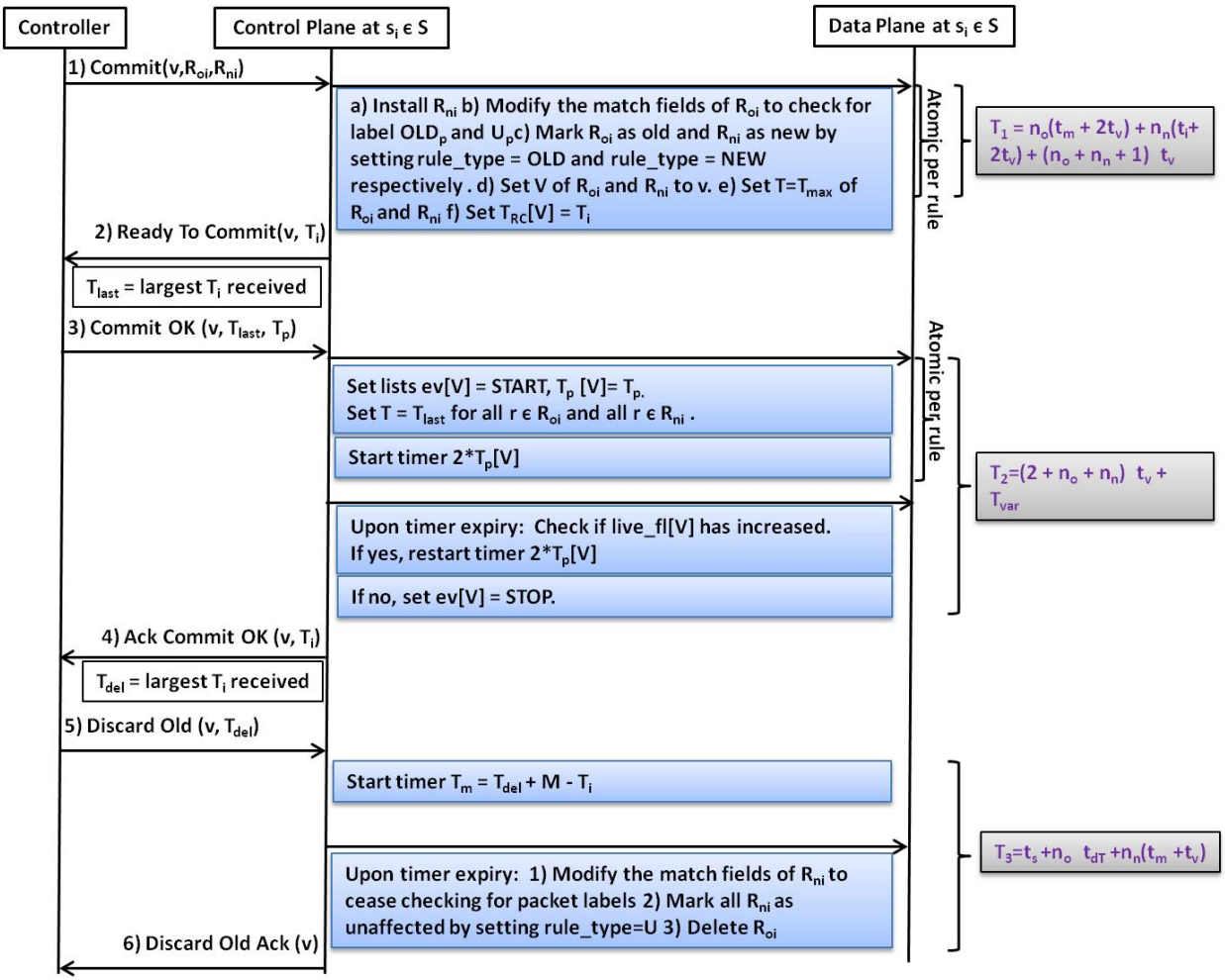


Figure 6.1: ProFlow: Algorithm at the switch and the controller

data plane algorithms are all about how to label every affected packet in light of the above observations so that PFC is preserved.

6.3.5.2 Synchronizing forward and reverse flows

For the forward and reverse flows to use the same version of rules, the packets that exit the *egress of the forward flow* are processed according to Algorithm 6. Every egress (or ingress) [5] maintains a table *istate*, indexed by *hash* (hash of the 5-tuple calculated such that it is the same value for both forward and reverse flows), that stores the state of the required flows, as shown in Table 6.2.

If a packet is exiting the network and the packet has *label* set to NEW_p, the reverse flow corresponding to this forward flow must also use new rules, that is, rules designated as new by the *current* RU. Therefore, if the packet is a SYN and its *label*=NEW_p, *istate* for that flow is set to NEW_FL (line 7) and *itstamp* to the value of TS_p in the packet (line 9), whereas if the packet is exiting the network and has *label* set to U_p or OLD_p, *istate* is reset to U_FL (line 12).

If a SYN+ACK is entering the network and *istate* of that flow is set to NEW_FL, *t_label* of that packet is set to NEW_p (line 16), and TS_p of that packet is set to the value in *itstamp*[*hash*] (line 19). If s_{fr} sees that *t_label* of a packet is set to NEW_p, it applies new rules to the packet and sets *label* of the packet to NEW_p, whether it has received Commit OK or not, as long as the value of TS_p in the packet is greater than or equal to $T_{RC}[V]$ of s_{fr}, thus synchronizing the reverse flow with the forward flow. $T_{RC}[V]$ is the time at which all the new rules are installed at s_{fr} and Ready To Commit is sent. If $TS_p \geq T_{RC}[V]$, it means that the forward flow of this SYN+ACK was designated as new in the *current* RU. This is explained in detail later.

6.3.5.3 Deciding if a packet is new or old

Table 6.3 lists the conditions used at s_{ff} or s_{fr} to decide if a new label or an old label must be applied to an affected packet that has *label* set to U_p. *label* of a packet must be set to NEW_p if and only if s_{ff} has received Commit OK (now $TS_p \geq (T = T_{last})$) and: 1) the packet is a SYN, or 2) the data plane no longer wants to check the status of any flow, indicated by setting ev=$STOP$. Additionally, whether s_{ff} has received Commit OK or

[5] The egress and ingress switches are the same. They are distinguished for clarity.

Algorithm 6 Algorithm at the ingress/egress data plane

1: **procedure** INGRESS DATA PLANE
2: ▷ Sets *t_label* of reverse flow to NEW_p only if the forward flow uses new rules. Store the state of the forward flow as NEW_FL in $istate[hash]$.
3: $hash$ = hash of packet header
4: **if** (packet is exiting network) **then**
5: **if** ($label{=}NEW_p$) **then**
6: **if** (Control bit for SYN is set but not for ACK) **then**
7: $istate[hash]{=}NEW\_FL$
8: ▷ Store the time stamp of SYN exiting the network
9: $itstamp[hash]{=}TS_p$
10: **end if**
11: **else** ▷ Reset istate
12: $istate[hash]{=}U\_FL$
13: **end if**
14: **end if**
15: **if** (packet is entering network) **then**
16: **if** ($istate[hash]{=}NEW\_FL$) **AND** (Control bits for both SYN and ACK are set) **then**
17: ▷ Label SYN+ACK as NEW_p and set its time stamp to the time stamp value received in the corresponding SYN of the forward flow
18: Set $t\_label{=}NEW_p$ of incoming packet
19: Set $TS_p{=}itstamp[hash]$
20: **end if**
21: **end if**
22: **end procedure**

Case	Value
C_{old1}	$(TS_p{<}T[n])$ **AND** $(label{=}U_p)$ **AND** $((t\_label{\neq}NEW_p)$**OR** $(TS_p{<}T_{RC}[V]))$ **AND** $((prev\_t[d\_hash]{<}T_{RC}[V])$ **OR** $(state[d\_hash]{=}OLD\_FL))$
C_{old2}	$(TS_p{\geq}T[n])$ **AND** $(label{=}U_p)$ **AND** $(t\_label{\neq}NEW_p)$ **AND** $((prev\_t[d\_hash]{<}T_{RC}[V])$ **OR** $(state[d\_hash]{=}OLD\_FL))$ **AND** $(flags{\neq}SYN)$ **AND** $(ev[V]{\neq}STOP)$
C_{new1}	$(TS_p{\geq}T[n])$ **AND** $(label{=}U_p)$ **AND** [$(flags{=}SYN)$ **OR** $(ev[V]{=}STOP)$]
C_{new2}	$((t\_label{=}NEW_p)$ **AND** $(TS_p{\geq}T_{RC}[V]))$ **OR** $((prev\_t[d\_hash]{\geq}T_{RC}[V])$ **AND** $(state[d\_hash]{=}NEW\_FL))$

Table 6.3: Conditions checked in Algorithm 7

not, if 3) the previous packet of the flow traversed s_{ff} after T_{RC} (when Ready to Commit was sent) and if the state associated with the flow indexed by the direction-sensitive hash of the 5-tuple (d_hash) is set to NEW_FL (which means this is a packet of a flow already deemed new), or, 4) t_label of a packet is set to NEW_p and $TS_p{\geq}T_{RC}[V]$ (which means this is SYN+ACK of a reverse flow belonging to the *current* RU), *label* is set to NEW_p. 1) and 2) are covered by C_{new1} and 3) and 4) by C_{new2} of Table 6.3. The affected packets that do not meet these conditions are labelled OLD_p (C_{old1} and C_{old2}). How the conditions in Table 6.3 are used to determine the label to be applied to a packet are explained below in cases 1, 2, 3 and 4.

6.3.5.4 Algorithms at the affected switches

We explain ProFlow in the chronological order of arrival of packets in the data plane. To begin with, let us assume that s_f has both NEW and OLD rules (is symmetric, as explained in section 1.3.2 of chapter 1).

Algorithm 3 of chapter 4 specifies how to match a packet with a set of rules (that is, using *match fields* as explained in section 6.3.1) and Algorithm 7, *the template* for executing the *action* associated with the rule that matches the packet. Algorithm 3 is reproduced one page 134 again, for easy reference. Let us consider three cases, to illustrate how the algorithms work: 1) a packet belonging to a flow f_1 reaches s_{ff} before it receives Commit OK (Figure 6.2(b)), 2) The first packet belonging to a flow f_2 (SYN) reaches

Algorithm 3 Match a packet

1: **procedure** MATCH-PACKET(*packet*)
2: ▷ This algorithm is for the match part of the match-action table. New rules are always installed with a priority higher than the old and unaffected rules. New (old) rules check if the label *label* of the incoming packet is equal to NEW_p (OLD_p) or U_p. Unaffected rules do not check for a packet label.
3: ▷ *label* is the label of *packet*
4: **if** ((*label*=NEW_p)**OR**(*label*=U_p)) **AND** (*packet* matches fields of a NEW rule) **then** ▷ This is the match part of a new rule
5: EXECUTE-ACTIONS(*packet*) ▷ Actions associated with the new rule
6: **else if** (((*label*=OLD_p)**OR**(*label*=U_p)) **AND** (*packet* matches fields of an OLD rule)) **then** ▷ This is the match part of an old rule
7: EXECUTE-ACTIONS(*packet*) ▷ Actions associated with the old rule
8: **else**
9: ▷ The packet does not match a new or old rule (unaffected packet) or no new or old rules exist on that switch to match the affected packet, regardless of the packet label
10: EXECUTE-ACTIONS(*packet*) ▷ Actions associated with the unaffected rule
11: **end if**
12: **end procedure**

s_{ff} after s_{ff} receives Commit OK (Figure 6.2(d)) 3) The first packet belonging to a reverse flow (SYN+ACK), whose forward flow uses new rules, reaches s_{fr} before it receives Commit OK (Figure 6.2(d)). 4) SYN of flow f_3 reaches s_{ff} after it receives Commit OK for an RU RU_1 (f_3 is deemed new). RU_1 ends and an RU RU_2, that conflicts with RU_1, begins. Now SYN+ACK of f_3 (must be deemed old) reaches s_{fr} of RU_2 before s_{fr} receives Commit OK. This is the same as example E1 in section 6.3.2.

Case 1: Let p_1 be the first packet of f_1 (Figure 6.2(b)) for this case that arrives at s_{ff} after s_{ff} receives Commit and sends Ready To Commit. Since *label* of p_1 is U_p and it is an affected packet, it matches the rest of the fields of a new rule, as shown in line 4 of Algorithm 3 and the corresponding action is executed (line 5 of Algorithm 3), which calls

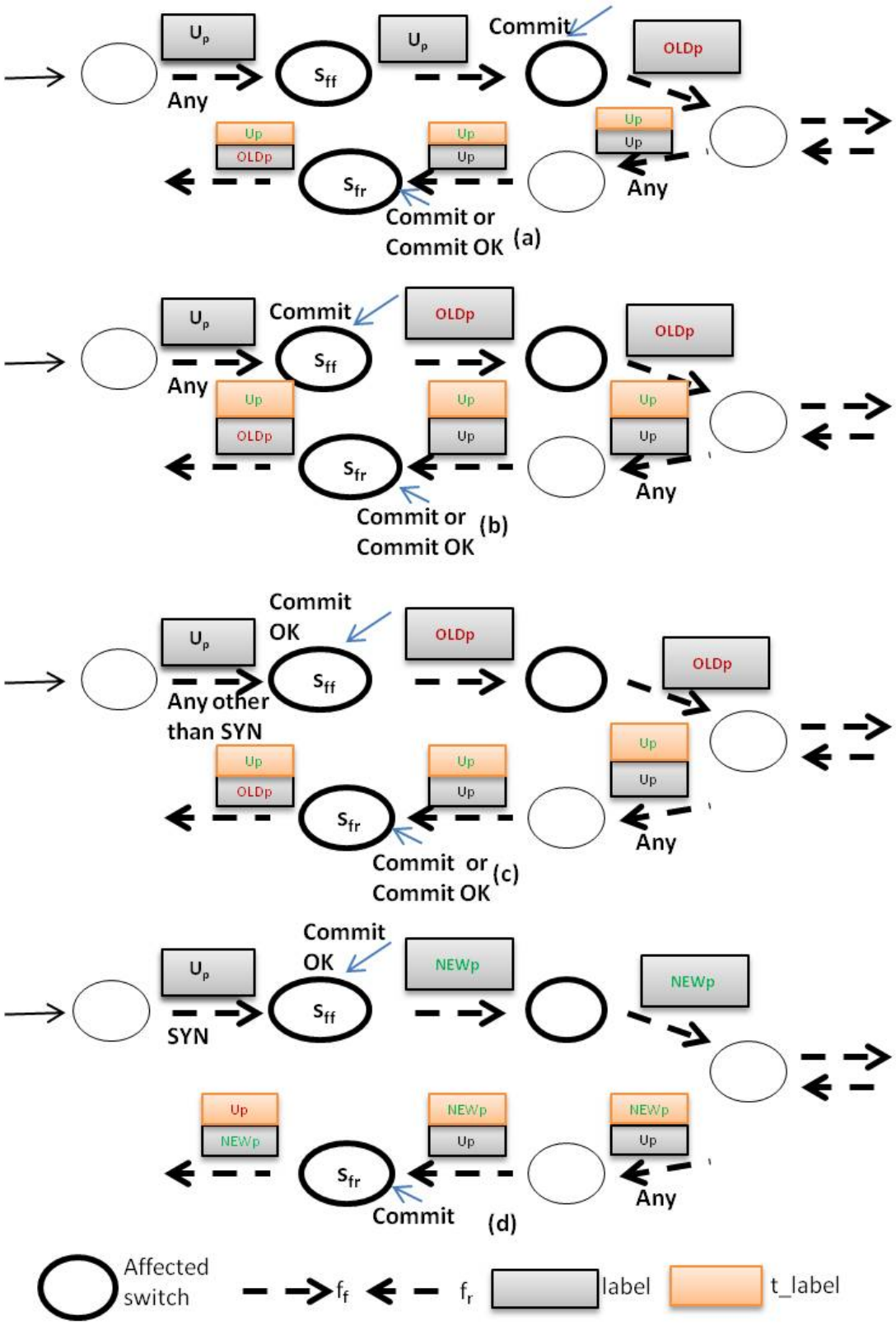


Figure 6.2: PFC for forward and reverse flows

Algorithm 7.

When p_1 arrives at s_{ff}, since s_{ff} has not received Commit OK yet, all the affected

switches may not have completed installing new rules. Therefore T, associated with each rule, is still T_{max}. s_{ff} can ascertain this by checking if $TS_p<T$. But, in general, checking this is insufficient to know if the packet belongs to an old flow, as *t_label* may have been set to NEW_p, for a packet belonging to a reverse flow. Therefore the value of *t_label* is checked. Every switch sets the value of *prev_t*[*d_hash*] (Table 6.2) to the time stamp TS_p of a packet when the packet exits the switch (line 32 of Algorithm 7), unless otherwise stated. *prev_t*[*d_hash*] is either 0 or the time at which the previous packet of this flow or the last packet belonging to the previous instance of this flow matching the same 5-tuple traversed s_{ff}. In either case, it is less than T_{RC}. If the packet is already labelled NEW_p or OLD_p, these conditions need not be checked - therefore whether its label is U_p is checked. Thus p_1 meets C_{old1} and therefore it belongs to an old flow and line 3 of Algorithm 7 is executed. This sets *state* to *OLD_FL* and *label* of the packet to OLD_p. Since the rule that must be matched is the corresponding old rule and the current rule is new, MATCH-PACKET is called recursively (line 7). Updating *live_fl* to indicate the existence of old flows is explained later.

p_1 now matches an *OLD* rule (line 6, Algorithm 3) and executes the corresponding action in line 7. This action is different from the action associated with line 5 in Algorithm 3. However, since EXECUTE-ACTIONS is only to show the *template* of actions, we use the same function name as in line 5 of Algorithm 3. Since the packet is labelled OLD_p, it matches line 17 of Algorithm 7, and the primitive actions (such as forwarding a packet to a port) associated with the matched *OLD* rule are executed (line 21 of Algorithm 7). When subsequent switches observe a packet labelled OLD_p, they directly apply old rules to the packet (lines 6, Algorithm 3 and 21 of Algorithm 7). Suppose a switch other than s_{ff} receives Commit before s_{ff} receives Commit (Figure 6.2 (a)). The behaviour at that switch is the same as described above and a packet labelled U_p entering the switch exits the switch with the label OLD_p.

Next, let us assume that any switch $s_i \in S$, other than s_{ff} receives Commit OK. Since an incoming affected packet is already labelled OLD_p by s_{ff}, the other affected switches directly apply old rules to it, as before.

Suppose s_{ff} receives Commit OK and it receives packet p_2, belonging to f_1 (Figure 6.2(c)). This packet is labelled U_p and its $TS_p \geq T$ as $T=T_{last}$. *state*[*d_hash*] is already set to *OLD_FL*. p_2 matches a new rule (line 4, Algorithm 3) and executes the corresponding

action in line 3, Algorithm 7, meeting the condition C_{old2}. When MATCH-PACKET is called again, it executes line 17 of Algorithm 7 as C_{old2} is satisfied. Now it gets switched using the old rules (line 21 of Algorithm 7).

Thus all the packets of a flow that begins before s_{ff} receives Commit OK are switched according to the old rules.

Case 2: Suppose s_{ff} receives Commit OK and it receives a packet p_1 of f_2 (a SYN) labelled U_p. This matches a new rule (line 4 of Algorithm 3) and subsequently executes line 12 of Algorithm 7, as it satisfies C_{new1}. This sets *state* to NEW_FL and *label* to NEW_p and executes the actions associated with the new rule. All subsequent affected switches switch p_1 with new rules. The next packet p_2 of f_2 gets switched using new rules at s_{ff} because *state* associated with this flow is already set to NEW_FL and the previous packet arrived after T_{RC}, satisfying C_{new2}.

Case 3: This case refers to the reverse flow of Case 2. The ingress of the reverse flow sets *t_label* to NEW_p of p_{r1}, that is, SYN+ACK of the reverse flow. TS_p of p_{r1} is the TS_p of the corresponding SYN, which is always greater than T_{last}, as otherwise SYN cannot have its *label* set to NEW_p. However, $T_{last} \geq T_{RC}[V]$, by definition. Therefore, at s_{fr}, p_{r1} matches a new rule (line 4 of Algorithm 3) and subsequently executes line 12 of Algorithm 7, since it satisfies C_{new2}. It sets *state* to NEW_FL and *label* to NEW_p. Subsequent packets use new rules as *state* is set to NEW_FL.

Case 4: When SYN of f_3 crosses s_{ff} of RU_1, it is labelled NEW_p (line 4 of Algorithm 3 and line 12 of Algorithm 7). When it reaches its egress, the *istate* of f_3 is set to NEW_FL and *itstamp* is set to TS_p of the received packet (lines 7 and 9 of Algorithm 6). Suppose SYN+ACK arrives at the ingress after RU_2 has begun. Its *t_label* is set to NEW_p and TS_p to the time stamp of the SYN of f_3 (lines 18 and 19 of Algorithm 6). Even though f_3 was considered a new flow for RU_1, it must be identified as an old flow for RU_2. Suppose s_{fr} of RU_2 has not received Commit OK. When s_{fr} of RU_2 processes this SYN+ACK, its $TS_p < T_{RC}[V_2]$, where V_2 is the update identifier of RU_2. Unless TS_p is compared with $T_{RC}[V_2]$, it will assume that this packet belongs to a new flow of RU_2, which is incorrect. If $TS_p < T_{RC}[V_2]$, it means that the SYN corresponding to this SYN+ACK crossed s_{ff} of some RU before new rules were installed for RU_2 and therefore it cannot belong to RU_2. The SYN+ACK satisfies C_{old1}, is labelled OLD_p and $state[d\_hash]$ is set to OLD_FL. When the next packet of the reverse flow arrives, again before s_{fr} receives Commit OK,

Algorithm 7 Execute actions of the appropriate rule type

1: **procedure** EXECUTE-ACTIONS(*packet*)
2: ▷ Executes actions of rule n with which *packet* has matched.
3: **if** [($rule\_type[n]$=NEW) **AND** (C_{old1} **OR** C_{old2})] **then** ▷ Packet matches a new rule, but belongs to an old flow
4: UPDATE_LIVE_FL(*packet*)
5: $state[d\_hash]$=OLD_FL
6: $label$=OLD_p
7: MATCH-PACKET(*packet*) ▷ Packet is old, match it again
8: **else if** [($rule\_type[n]$=OLD) **AND** (C_{new1} **OR** C_{new2})] **then** ▷ Packet matches an old rule, but belongs to a new flow
9: $state[d\_hash]$=NEW_FL
10: $label$=NEW_p
11: MATCH-PACKET(*packet*) Packet is new, match it again
12: **else if** [($rule\_type[n]$=NEW) **AND** (($label$=NEW_p) **OR** C_{new1})] **OR** [($rule\_type[n]$=NEW) **AND** C_{new2}] **OR** [($rule\_type[n]$=U) **AND** ($label$=NEW_p)] **then** ▷ Packet matches a new/unaffected rule and belongs to a new flow
13: $state[d\_hash]$=NEW_FL
14: $label$=NEW_p
15: t_label=U_p
16: Execute primitive actions
17: **else if** [($rule\_type[n]$=OLD) **AND** (($label$=OLD_p) **OR** C_{old1})] **OR** [($rule\_type[n]$=OLD) **AND** C_{old2}] **OR** [($rule\_type[n]$=U) **AND** ($label$=OLD_p)] **then** ▷ Packet matches an old/unaffected rule and belongs to an old flow
18: UPDATE_LIVE_FL(*packet*)
19: $state[d\_hash]$=OLD_FL
20: $label$=OLD_p
21: Execute primitive actions
22: **end if**

23: **if** [($rule\_type[n]$=U) **AND** ($label$=U_p)] **then** ▷ Packet is unaffected. Reset *state*

24: $state[d\_hash]$=OLD_FL

25: Execute primitive actions

26: **end if**

27: ▷ Executed right before the packet exits

28: **if** (Control bits for SYN+ACK are set) **AND** (t_label=NEW_p) **then**

29: $prev\_t[d\_hash]$=$current\,time\,of\,switch$

30: t_label=U_p

31: **else**

32: $prev\_t[d\_hash]$=TS_p

33: **end if**

34: **end procedure**

its t_label is not NEW_p. However, since $state[d\_hash]$ is set to OLD_FL, it satisfies C_{old1} again and the packet is labelled OLD_p.

In the example E3 in section 6.3.2, let us assume that the first packet of the second instance of flow f arrives at s_{ff} after it receives Commit OK. TS_p>T, $label$=U_p and $flags$=SYN. Therefore this packet satisfies C_{new1} and is labelled new, even though *state* is OLD_FL.

6.3.5.5 Need to check *prev_t*

When an affected packet exits an affected switch, the switch sets $prev\_t$ to the TS_p of the packet (line 32 of Algorithm 7), unless the packet is a SYN+ACK (line 29 of Algorithm 7). Suppose a flow f begins after an RU RU_1 begins and ends before RU_1 ends and suppose it sets *state* of f to NEW_FL at s_{fr}. Now RU_2 begins. f begins again. SYN of f traverses s_{ff} of RU_2, but this time before s_{ff} receives Commit OK, designating the flow as OLD_FL and the packet as OLD_p. When SYN+ACK of f traverses s_{fr} of RU_2, t_label is not NEW_p, but *state* is set to NEW_FL by the previous instance of the flow. Checking if $prev\_t[d\_hash]$ is less than T_{RC} ensures that the packet is correctly identified as belonging to an old flow.

Suppose a new flow f_n begins after an RU RU_1 begins and does not end before RU_1 ends. $state[d\_hash]$ is set to NEW_FL of this flow, at s_{ff}. Now another RU, RU_2, that

affects f_n, begins after RU_1 ends. Suppose only after s_{ff} of RU_2 receives Commit OK does the next packet p of f_n arrive at s_{ff}, and that packet is not a SYN. If $prev\_t[d\_hash]$ is not checked in C_{old2}, f_n will be incorrectly classified as a new flow by RU_2. Since $prev\_t[d\_hash]$ of f_n is less than T_{RC}, checking it will correctly classify f_n during RU_2 as old.

In the example E2 in section 6.3.2, let us assume that the first packet of flow f in RU_2 arrives at s_{ff}, after s_{ff} receives Commit OK. $TS_p{>}T$, $label{=}U_p$, $t\_label{\neq}NEW_p$. Even though *state* of f is NEW_FL, since $prev\_t{<}T_{RC}[V2]$ where $V2$ is the version number of RU_2, C_{old2} is satisfied and the packet is labelled old.

In summary, to ascertain if a flow is old from the first packet of that flow (which may not be a SYN) that traverses s_{ff} after an RU has begun, checking only the value of *state* is insufficient, as *state* may not be initialised to the right value whereas for subsequent packets, checking only the value of $prev\_t$ is insufficient to know the state of the flow. Hence both need to be checked.

6.3.5.6 Dealing with asymmetry of rules

Let us assume that s_{ff} has only OLD rules. When an affected packet, a SYN, whose time stamp $TS_p{\geq}T_{last}$ arrives, it matches an OLD rule (line 6 of Algorithm 3). In the corresponding action, since $TS_p{\geq}T_{last}$ and the packet is a SYN, it is labelled NEW_p and Algorithm 3 is recursively called (line 11 of Algorithm 7). This packet next matches an unaffected rule as intended (line 10 of Algorithm 3). (If there are no unaffected rules, the RU is malformed, as after deleting the old rule, the flow will have no rules to match at s_{ff}.) Subsequently, it matches line 12 of Algorithm 7, as $rule\_type{=}U$ and $label{=}NEW_p$. If s_{ff} has only NEW rules, an old packet is handled in a similar manner. Any affected switch other than s_{ff} applies an unaffected rule on a packet labelled NEW_p (OLD_p) if a NEW (OLD) rule is not present on the switch. If there are only new rules to be inserted at every affected switch and there are no unaffected or old rules that match all the packets that match the new rules (that is, flows matching these rules never existed in the network before), starting flows matching such rules will be meaningful only after all the rules are installed, as otherwise, switches will drop packets which do not match any rule.

Algorithm 8 Indicate that old flows exist

1: **procedure** UPDATE_LIVE_FL(*packet*)

2: ▷ If the inter-packet delay is less than T_p or the packet is a SYN or SYN+ACK, the flow is alive. If the flow is alive and $live\_fl$ was updated T_p units before, update it now. *label* is checked as only s_f needs to update $live\_fl$.

3: **if** $(label{=}U_p)$ **AND** $((TS_p - prev\_t[d\_hash] \leq T_p[V])$ **OR** $(flags{=}SYN)$ **OR** $(flags{=}SYN + ACK))$ **AND** $((TS_p - live\_fl[V]) \geq T_p[V])$ **then**

4: $live\_fl[V]{=}TS_p$ of current packet

5: **end if**

6: **end procedure**

6.3.5.7 Checking if an old flow is live

If the inter-packet delay of an old flow is less than T_p or if the packet is a SYN or SYN+ACK (in which case the inter-packet delay is not meaningful), it is deemed live. Every *old* flow updates $live\_fl[V]$ if *label* of a given packet is U_p, the flow is live and if there is a delay greater than or equal to T_p from when it previously updated $live\_fl$, as shown in Algorithm 8. If the inter-packet delay is greater than T_p, the old flow is deemed new and $live\_fl$ is no longer updated.

Since TS_p of a SYN+ACK whose t_label is NEW_p is the TS_p of its SYN, for s_{fr} to calculate a meaningful inter-packet delay for a packet arriving after SYN+ACK, $prev\_t$ is set to the current time of s_{fr}, as an approximation (line 29 of Algorithm 7). t_label is reset to U_p. The rest of the affected switches will not update $live\_fl$ as *label* of the packet is set to NEW_p. Therefore they will not require the correct value of TS_p.

Suppose (Figure 6.5a(c)), the switch CP polls $live\_fl$ immediately after $live\_fl$ was updated (t_1). Next, the last packet of the last old flow (denoted as p_f) arrives a little before T_p units of time from t_1. The switch CP must wait for T_p units of time from t_2 before it reads $live\_fl$ again. $live\_fl$ will not be updated for p_f, as T_p units have not elapsed from t_1. Therefore the switch CP considers all the old flows completed at t_3, T_p *units from* p_f*, in the best case*. In the worst case (Figure 6.5a(d)), after the switch CP polls $live\_fl$, it is immediately updated by p_f, at t_1. Therefore the switch CP must poll twice more before it finds that the value of $live\_fl$ has not changed, thus taking $2 * 2 * T_p$ *units after* p_f. Alternately, $live\_fl$ may be updated for every packet and the switch CP

can poll this every T_p, to reduce the RU duration. If so, after p_f, the switch CP must wait for T_p in the best case (Figure 6.5a(a)) and $2T_p$ in the worst case (Figure 6.5a(b)), from p_f, before the RU can be considered completed.

6.3.5.8 Handling long flows

In a PFC compliant RU, if there are long running *old* flows, though new rules become effective immediately for new flows, old rules cannot be deleted and an RU cannot be completed until the old long running flows cease, also preventing a *conflicting* RU from occurring. Therefore, it is desirable that an RU with a long old flow is not combined with any other RU. Servers discourage persistent connections with large HTTP keep-alive timers to prevent idle clients from consuming server resources [10], putting an upper limit on T_p. But flows could be long-lived and *continuous*. However, the median number of concurrent continuous large flows ($>1MB$) is only 1 and the $75th$ percentile is 2 [5] in a real data centre. There are means to identify such flows [118, 2] and to enable dynamic routing at small enough time scales [17, 108] by isolating them. If such flows are isolated and RUs performed separately on them, ProFlow will not cause an RU to be delayed by such flows. Installing separate rules for such flows to isolate them does *not* require a PFC RU, as the rule actions are not changed, and will not cause PFC violations.

6.3.6 ProFlow as a general solution

In a virtualized network, since the ingress switch that the reverse flow traverses to enter the network may be different from the egress switch that the forward flow traverses to exit the network, Algorithm 6 must be executed in a module that intercepts the traffic between the ingress/egress and the host and not the ingress/egress. This module can be a part of the network hypervisor and need not use the same clock as the rest of the network. This solves the problem discussed in section 2.8.3.

6.4 Discussion

6.4.1 Overheads

Let the maximum number of flows expected at a switch be $10,000/s$ [16], with a maximum number of 100 concurrent RUs, 10000 rules per switch and with the sizes of stateful variables in bits as: $T(32)$, $rule\_type(2)$, $ev(1)$, $T_p(32)$, $state(1)$, $istate(1)$, $prev\_t(32)$, $live\_fl(32)$, $T_{RC}(32)$ and $itstamp(32)$, $V(7)$. In a 32 staged RMT switch [22], there are 106 blocks, with each block having $1000 * 80$ bits of memory available per stage, in the SRAM, to store both actions and overhead bits [63]. Roughly, less than 13% of the SRAM memory of a stage is sufficient to store the stateful memory. $prev\_t[d\_hash]$ may be reused across applications unchanged [68].

6.4.2 Concurrent RUs and other claims

As v is used only in the control plane and is not a part of the packet header, the number of disjoint concurrent RUs is limited only by the processing power and memory of switches, similar to PPCU, section 4.5. The RU will progress to completion even if there are no flows in the system, as flows carry no information regarding the progress. If multiple tables require changes for the same flow, it can be done as described in section 4.7.2.5 for PPCU.

6.4.3 Handling Asynchronous time at each switch

The DP algorithm relies on synchronous time for three comparisons: 1) TS_p with $T_{RC}[V]$ 2) TS_p with $T[n]$ and 3) $prev\_t[d\_hash]$ with $T_{RC}[V]$.

Case 1: Suppose the SYN of a flow f deemed new by an RU RU_1, traverses s_{ff} and RU_1 completes. A conflicting RU RU_2 begins. Now SYN+ACK of f reaches s_{fr}. Let us assume that the clock of the ingress that stamped the SYN of f is faster than that of s_{fr}. When TS_p of SYN+ACK is compared with T_{RC} by s_{fr}, it may so happen that $TS_p > T_{RC}$, causing s_{fr} to incorrectly conclude that SYN+ACK belongs to a new flow of RU_2. Therefore, when every switch sends its current time in Ready to Commit, instead of T_i, it must send $\lceil T_i + \gamma + 1 \rceil$, and store the latter value in T_{RC}, to solve this problem, where γ is the maximum time drift of switches from each other. If the network supports PTP, γ=1μsec [90].

One of the conditions for C_{new2} to be satisfied is that $TS_p \geq T_{RC}[V]$. Is there a situation where this comparison can fail when it must not? Suppose the clock at the ingress that stamped the SYN of f is *slower* than that of s_{fr}. Only a SYN with its $TS_p \geq T_{last}$ will be given a label of NEW_p, causing t_label of SYN+ACK to be set to NEW_p. This is the only case of t_label being set to NEW_p. Since $T_{last} \geq T_{RC}$, TS_p will be greater than or equal to T_{RC}, when the packet is labelled NEW_p. Therefore if the ingress clock is slower than that of s_{fr}, it does not matter.

Case 2: Suppose a packet p with label U_p enters the network and crosses s_{ff} before an RU RU_1 begins. Let $TS_p > T_{last}$ (T_{last} of RU_1) as the ingress clock that stamped p is faster than the rest of the affected switches. After s_{ff} receives Commit OK, it is possible that p is labelled incorrectly as NEW_p by an affected switch other than s_{ff}, violating PFC. However, since T_{last} is the largest of all values of T_{RC}, which is already offset to account for time asynchrony, this issue will not arise. Also, if the ingress clock is slower than rest of the affected switches, again, this issue will not arise.

Case 3: C_{old1} and C_{old2} check if $prev\_t[d\_hash] < T_{RC}$. If the ingress clock is faster than the clock at s_{ff}, it is possible that $prev\_t \geq T_{RC}$ due to time asynchrony, which will not occur if $\lceil T_i + \gamma + 1 \rceil$ is stored in T_{RC}.

C_{new2} checks if $prev\_t \geq T_{RC}$. Can this condition be violated for a new flow due to time asynchrony ? Let us assume that the ingress clock is slower than the clock at s_{ff} making $prev\_t$ less than T_{RC}. Let the first packet of a new flow be a SYN that arrives after Commit OK is received, satisfying C_{new1}, or with $t\_label = NEW_p$ and $TS_p \geq T_{RC}$. The second packet must satisfy C_{new2}. Since this is a new flow, TS_p of the first packet is greater than or equal to T_{last}. Therefore $TS_p \geq T_{RC}$ and $prev\_t \geq T_{RC}$. Therefore this condition will not be violated due to time asynchrony. At s_{fr}, for the second packet of a new flow, $prev\_t$ is derived from the time at s_{fr} itself, as per Algorithm 7, due to which this condition will not be violated.

As for the control plane, since the switch CP examines $live\_fl$ only to check if there is a *change* in its value, variation in ingress clock speeds will not affect it. Similar to PPCU, in section 4.4 of chapter 4, each affected switch, upon receiving Discard Old, must set it to $\lceil T_{del} + \gamma + 1 \rceil$.

In summary, if the values of T_{RC} and T_{del} are offset as described above, time asynchrony between switches will be taken care of.

6.4.4 Simultaneous TCP Open

If two hosts simultaneously open a TCP connection it may require SYN messages from both the hosts to traverse the same NF instance [49], which cannot be guaranteed by ProFlow. TCP simultaneous-open with NAT traversal is a problem still being researched [49] [113] [112]. A strong case for simultaneous-open TCP connections is only for peer-to-peer applications (such as VoIP).Therefore the algorithm is suitable for data centers where these are not used.

6.4.5 Execution time of ProFlow

The symbols used in the analysis are: δ: the propagation time between the controller and a switch, t_i: the time taken to insert each rule in a switch TCAM, t_{dT}: the time taken to delete each rule from a TCAM, t_m: the time taken to modify each rule in a TCAM, t_v: the time taken to modify a stateful list entry, t_s: the time for which a switch waits after it receives "Discard Old" and before it deletes rules, n_o: the number of old rules that need to be removed, n_n: the number of new rules that need to be added, T_1: The time between the switch receiving "Commit" and sending "Ready To Commit", T_2: The time between the switch receiving "Commit OK" and sending "Ack Commit OK" and T_3: The time between the switch receiving "Discard Old" and the switch performing its functions after timer expiry. It is assumed that the value of δ is uniform for all switches and all rounds, the values of time are the worst for that round, the number of rules, the highest for that round and that the processing time at the controller is negligible. The values of T_1, T_2 and T_3 are shown in Figure 6.1.

T_1=$n_o(t_m+2t_v)+n_n(t_i+2t_v)+(n_0+n_n+1)*t_v$. T, $rule\_type$ and V of each affected rule will get updated in time t_v. $T_{RC}[V]$ is updated in time t_v. T_2=$(2+n_o+n_n)*t_v+T_{var}$. T is updated for every affected rule, with each update taking time t_v. ev and T_p are updated in time t_v. T_{var} depends on the type of flows in existence at the time of the RU. T_{var} is the poll time, $2*T_p$, if there are only new flows. If there is at least 1 old flow, let t_{old} be the remaining length of the longest such flow, after $T_1+(2+n_o+n_n)*t_v$ units have elapsed after the RU begins. Reading $live\_fl$ occurs in parallel with t_{old} and setting ev to $STOP$ occurs in parallel with T_{var} and are therefore ignored. T_3=$t_s+n_o*t_{dT}+n_n(t_m+t_v)$.

T_{var} will vary from $t_{old}+T_p$ to $t_{old}+4T_p$, as described in section 6.3.5.7. The total

propagation time is $6 * \delta$. Thus the total RU time from the perspective of the controller is $T_1 + T_2 + T_3 + 6 * \delta$ (M units must elapse from this time before a conflicting RU begins). Since T_p will dominate this value if there are no old flows or only short old flows, it must be chosen judiciously.

Updating $live\_fl$ for *every* packet and the switch CP polling it every T_p units will reduce the range of T_{var}. It will now vary between $t_{old} + T_p$ and $t_{old} + 2 * T_p$, as described in section 6.3.5.7.

6.4.6 Anomalous flows

If there are long old flows with inter-packet delays larger than a given value of T_p (anomalous flows), an RU with a suitably larger value of T_p may be used, specifically for such flows. Alternately, if a PPC update is sufficient only for the anomalous flows while the rest of the flows in the RU are updated preserving PFC, Proflow will update them in a per-packet consistent manner, without any change. They may be isolated and separately managed, as described for long old flows (section 6.3.5.8). New flows may be long or may be anomalous or both and they do not affect an RU in any way.

Suppose f is a flow in an RU RU_1 and it is anomalous. Suppose its SYN traverses s_{ff} when RU_1 affecting it is in progress and it is labelled NEW_p. $state$ is set to NEW_FL and $prev\_t$ to TS_p of SYN, as the packet exits s_{ff}. Suppose this RU completes and another RU RU_2, conflicting with RU_1, begins. Let p be the next packet of f_1 after SYN, that arrives after $T_p + 1$ units of time, which traverses s_{ff} of RU_2. p is now labelled OLD_p and $state$ set to OLD_FL, as $prev\_t$ for f is set to a value of time before RU_2 began. This packet and subsequent packets until ev is set to $STOP$ will be switched using old rules. Suppose further packets arrive with an inter-packet delay less than T_p. RU_1 and RU_2 preserve PFC. Suppose f_1 is an old anomalous flow, that is deemed completed before its RU, RU_3, completes. Let Ack Commit OK be sent from all the affected switches. Any subsequent packet of f_1 has $ev{=}STOP$, its $TS_p{\geq}T$ and $label{=}U_p$, and therefore it satisfies C_{new1}. It is marked NEW_p until this RU ends. RU_3 will not preserve PFC, but will preserve PPC. Thus RUs will be at least PPC compliant, and no packets will be dropped, for anomalous flows. There may be old flows in the network with inter-packet delays larger than T_p. But if there are packets arriving due to *any* old flow at a rate sufficient to update $live\_fl$ for the control plane to conclude that there are live old flows, all old flows will continue to

get switched using old rules. Thus an anomalous flow will be switched preserving PFC in the best case and PPC in the worst case.

6.4.7 Header modification by NFs

Certain NFs such as NATs dynamically modify packet headers. A solution such as FlowTags [38] can be employed, without affecting the functioning of the algorithm. The controller allocates a unique tag to a flow as it enters a middlebox, a subsequent middlebox consumes this tag and possibly gets another tag generated by the controller and so on. Since this set of tags is uniquely associated with a flow and is valid as long as the flow is, in addition to the direction sensitive 5-tuple (d_hash), the tag available at an affected switch may be used to uniquely and consistently identify a flow. ProFlow does not require all the affected switches of a forward or reverse flow to use the same hash - the hash only needs to identify that flow uniquely and consistently on that switch. Ingresses and egresses can continue to use the direction insensitive hashes of 5-tuples ($hash$), as no NFs or middleboxes will alter the 5-tuple after a packet exits an egress and before it enters an ingress.

6.4.8 Data Plane Algorithms run at line rate

To examine if the data plane algorithms can run at line rate, the statements in the algorithms may be expressed in three-address codes, as is done in Domino [117] and other prior work [68]. Such an instruction is of the form $f1{=}f2\ op\ f3$, where $f1$, $f2$ and $f3$ are stateful variables or packet headers or both, and op is an operation. Complex expressions are converted to such instructions using temporary variables. After conversion, the most complex instruction supported is "Pairs", which allows updates to two stateful variables, with predicates that can use the same two stateful variables. In the worst case, the data plane algorithms of ProFlow update $state$ and $live\_fl$, while using both in predicates. Hence it may be concluded that they can run at line rate.

6.4.9 Concurrent access

$live\_fl$ needs to be incremented at an interval greater than or equal to T_p units by every live old flow. If there is contention between flows while accessing this counter, it is sufficient

that a packet belonging to one of the old flows succeeds in incrementing it - that is, an atomic write is not expected. This is because the switch CPU that polls this list only requires the list value to increment by at least 1 to conclude that there are old flows still alive.

6.5 Implementation and evaluation

A network of switches using a FatTree [1] topology, with the routing scheme by Al-Fares et al. [1] was implemented, using Mininet [88] (version 2.2.1), with one rule per destination host. The data plane algorithms were implemented in P4 (version 1.0.2 [30]). The algorithm for the control plane was implemented in our own elementary controller and the P4 switch simulation available on Linux [121]. Algorithm 3 is implemented in the match part of a table and Algorithm 7 is implemented in the control flow of a table, as the version of P4 used does not support conditional statements in actions [6]. The Proflow implementation in the controller uses threads to send messages to the affected switches in parallel, using the Switch API provided with the P4 implementation.

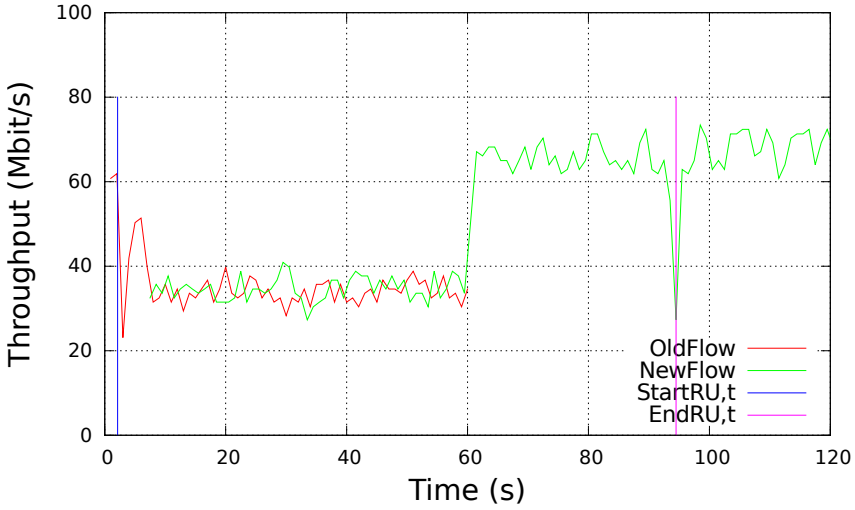
The implementation in P4 enhances the code for PPCU in section 4.7.2 of chapter 4 and is not further described.

Experiments were carried out on a Linux server (16-core Intel(R) Xeon(R) E5-2630 v3 CPU running at 2.40 GHz, and Ubuntu 16.04), with Mininet version 2.2.1. For all experiments, the individual link speeds have their upper limit set to 200Mbps, the switch queue size to 6000 packets and the delay and loss parameters to 0. A Hierarchical Token Bucket scheme is used for queueing. T_p is set as $15s$. Logs are examined to to ensure that PFC is preserved.

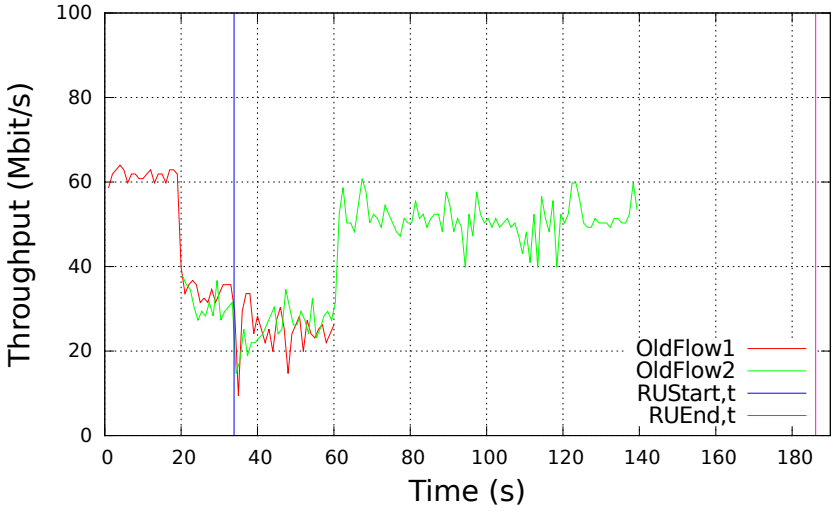
6.5.1 Experiment 1

The purpose of this experiment is to measure the impact of a PFC RU upon the throughput of a mix of old and new flows. A flow $OldFlow$ is started from h_{00} to h_{20} in Figure 6.4, using iperf [99], which sends as many packets as possible into the network (MSS = 1350 bytes), for a duration of $60s$, followed by a PFC RU, where both the forward flows

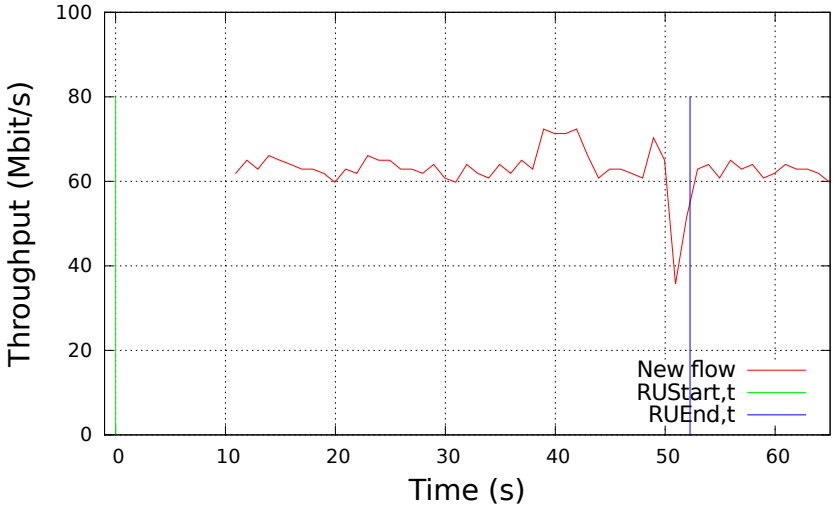
[6] Implementation in a newer version of P4, $P4_{16}$ [31], will be simpler, since it supports conditional statements in actions.



(a) Throughput-new flow



(b) Throughput-old flow



(c) Throughput-s_{fr} receives Commit OK late

Figure 6.3: Throughput during an RU

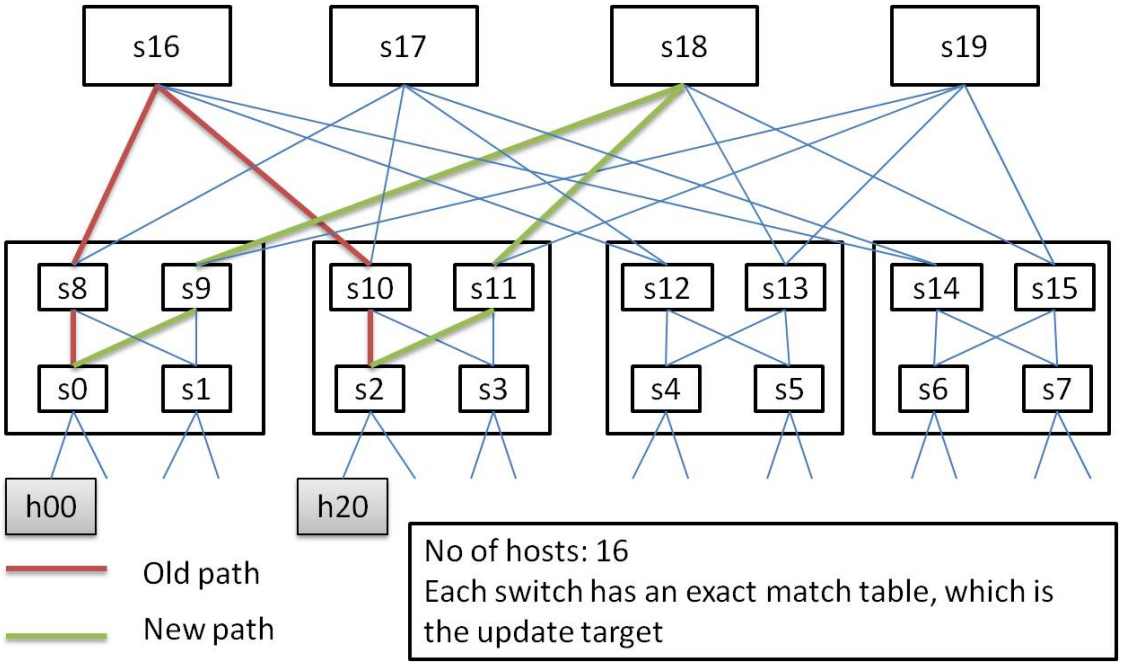


Figure 6.4: A PFC RU on a FatTree network

from h_{00} to h_{20} and the reverse flows from h_{20} to h_{00} are required to change from the old path to the new path (Figure 6.4). Next, a new flow $NewFlow$ is started between the same host pairs, for a duration of $120s$. From Figure 6.3a, it can be observed that there is an instantaneous drop in throughput at the beginning ("StartRU") and at the end of the RU ("EndRU"), because rules are inserted and deleted, while during the RU itself, there is no change in the throughput of the new flow in spite of the additional processing. This is clear when the throughput of the new flow after the old flow ceases at $60s$ is compared with the throughput of the new flow after the RU ends. During the course of the RU, the switch CP polls the switch DP every $2*T_p$=$30s$ to decide if the old flow is complete, and at the fourth poll, it finds that $live\_fl$ has not incremented. It concludes that the old flow no longer exists and completes the RU. This also illustrates immediate effectiveness of the RU (section 6.2) - $OldFlow$ and $NewFlow$ coexist, using different versions of the same rule.

6.5.2 Experiment 2

This experiment is to check the impact of ProFlow on the throughput of flows where each packet needs to be resubmitted at s_f. Here, two old flows, $OldFlow1$ of duration $60s$ and $OldFlow2$, of duration $120s$, are started one after the other, using iperf, from h_{00} to h_{20}

and subsequently, an RU is started to change the path from old to new (Figure 6.4). The throughput of each flow is observed every $1s$. As shown in Figure 6.3b, there is a drop in the throughput of $OldFlow2$ during the RU, compared to $OldFlow1$ before the RU started, as s_{ff} and s_{fr} (and no other affected switch) need to resubmit each packet, as each packet first matches a new rule.

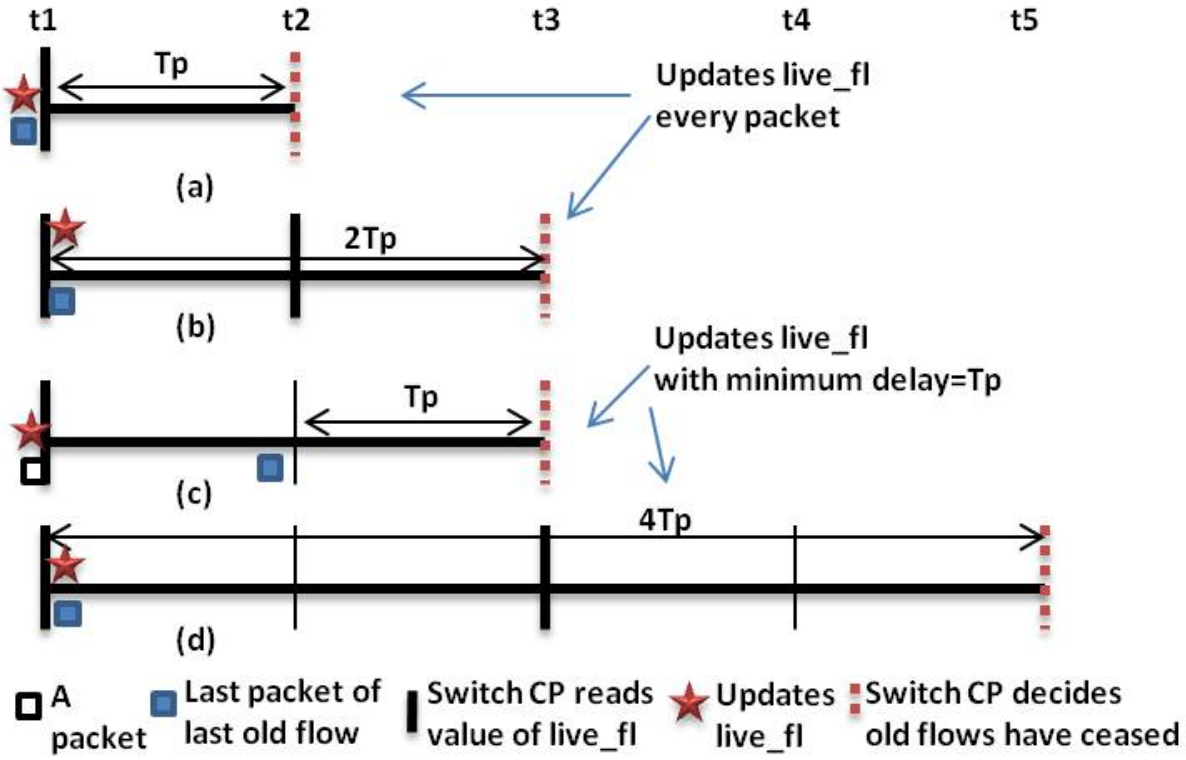
6.5.3 Experiment 3

Here, sending Commit OK to $s2$ (s_{fr} of $NewFlow$ from h_{00} to h_{20}) in Figure 6.4 is delayed by $20s$ to ensure that in spite of this, if the forward flow follows the new path, the reverse flow from h_{20} to h_{00} also follows the new path, and to measure the impact on throughput. At the end of the RU, there is an instantaneous drop in throughput; otherwise, the throughput during and after the RU appears similar, in Figure 6.3c.

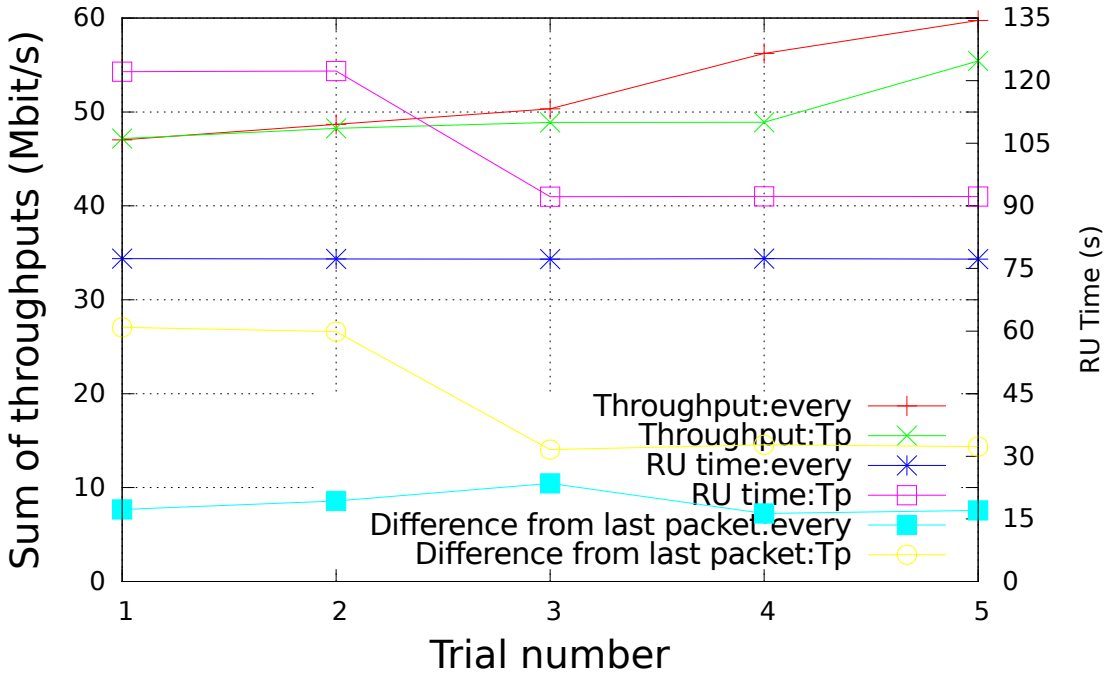
In summary, if a flow does not undergo resubmission, in a simulated setup, there is no observable drop in throughput during the RU, in spite of the additional processing. Otherwise, there is a drop in throughput during the RU, *compared to the throughput when there is no update affecting the flow.*

6.5.4 Experiment 4

The goal of this experiment is to check if updating $live\_fl$ for every packet is feasible, as discussed in section 6.4. 25 flows are started simultaneously from h_{00} to h_{20}, of duration $60s$, using iperf. If the number of flows is further increased, due to high load on the switches, flows start getting slower and the inter-packet delay increases beyond T_p, causing some flows to start using new rules. An RU begins next, to change the path of the flows. The throughput and the difference between the time of arrival of the last packet and completion of the RU are plotted in Figure 6.5b. The experiment is repeated after $live\_fl$ is set to be updated for every packet and DP is polled every T_p s. The throughput when $live\_fl$ is updated for every packet (denoted as "Throughput:every") is comparable with that when updated with a minimum delay of T_p (denoted as "Throughput:Tp"), as shown in Figure 6.5b. Also, the difference in time from when the RU ends (denoted as "RU time:every") to when the last packet of any flow arrives (denoted as "Difference from last packet:every") lies between T_p and $2 * T_p$. (It may also be observed that the



(a) Time to wait after the last packet



(b) Update *live_fl* every packet - 25 flows

Figure 6.5: Frequency of updating *live_fl*

same quantity varies between T_p and $4T_p$ when *live_fl* is updated with a minimum delay

of T_ps.) Thus updating $live\_fl$ for every packet is a practical solution, as explained in section 6.4. While the version of P4 used in the implementation automatically assumes that a register operation is atomic, $P4_{16}$ [31] supports explicitly specifying this. This may be left unspecified for $live\_fl$, as explained in section 6.4, to further improve throughput, when $P4_{16}$ is used.

6.6 Conclusions

This chapter has described a general algorithm for a per-flow consistent update, to solve problems in Service Chaining, Network Virtualization and Load Balancing, for updates that involve stateful network elements. Updates are confined to the affected switches and the affected rules and are immediately effective for new flows. The algorithm tolerates switch timing asynchronies and varying execution speeds of switches and links. It progresses even in the absence of flows and maintains line rate. The prototype implemented in P4 and evaluated on Mininet demonstrates that the algorithm is feasible and causes no impact on throughput for new flows, if the first affected switch in the course of the flow has both new and old rules, compared to a situation with no updates.

6.7 Proof

p is an affected packet with a label $label$, that can take the values U_p, NEW_p or OLD_p, indicating if the packet is unaffected, new or old respectively. We consider an RU RU_1, with its version number denoted as V_1.

Let us assume that the individual algorithms at the data and control planes are correct. Let us assume that switches are synchronized, to make descriptions easier and that no conflicting RUs occur. However, the controller-switch links may have variable delays and the switches may have varying execution speeds. We assume that no packets arrive for an old flow if T_p units of time elapse since the last packet of the flow.

Property 1: All forward flows preserve PFC.

Case 1 - s_{ff} has not received Commit and so it is unaware of an update: At time T_f, packet p arrives with time stamp TS_p. $TS_p \leq T_f$ by definition (TS_p was the time at the ingress switch when the packet left it and s_{ff} may be an ingress). It will match

some existing (soon to be declared old) rule, and the packet will be forwarded. *label* will remain U_p. p and packets subsequent to p will also be handled by old rules.

Suppose a new rule has been installed in a downstream switch, say s_p, and either (1.1) Commit has been received and Ready To Commit has been sent (T=T_{max} for all affected rules), or (1.2) both Commit and Commit OK have been received (T=T_{last} for all affected rules).

(1.1) From Table 6.3, C_{old1} = (TS_p<$T[n]$) **AND** (*label*=U_p) **AND** ((*t_label*≠NEW_p) **OR** (TS_p<$T_{RC}[V]$)) **AND** ((*prev_t*[*d_hash*]<$T_{RC}[V]$) **OR** (*state*[*d_hash*]=OLD_FL)). C_{old1} will be true because:

(a) If only Commit has been received, T=T_{max}, and TS_p<T_{max} by definition, for any TS_p.

(b) *label*=U_p because this is the first affected switch which has received Commit and therefore earlier switches will behave like s_{ff}.

(c) *t_label*≠NEW_p as *t_label* is set to U_p at the ingress and no subsequent switch has installed affected rules yet.

(d) ((*prev_t*[*d_hash*]<$T_{RC}[V]$) **OR** (*state*[*d_hash*]=OLD_FL)). Let p be the first packet that arrives after Ready To Commit is sent from this switch for RU_1. So *prev_t*[*d_hash*] < $T_{RC}[V_1]$ by this assumption. As a consequence of C_{old1} being true due to this, (a), (b) and (c), p matches a new rule, sets *label* to OLD_p and *state*[*d_hash*] to OLD_FL (line 5 of Algorithm 7) and is resubmitted. Then it matches an old rule (line 17 of Algorithm 7) and *state* remains unchanged. When the next packet arrives, *prev_t*[*d_hash*] may be greater than or equal to $T_{RC}[V_1]$, but *state*[*d_hash*] is already set to OLD_FL, thus satisfying C_{old1}. If p is not the first packet after Ready To Commit is sent, then an earlier packet, say p_y, will have *prev_t*[*d_hash*]<$T_{RC}[V_1]$, and (a), (b) and (c) above will be applicable to p_y too, satisfying C_{old1}, regardless of the value of *state*[*d_hash*] then, which may be NEW_FL. (Packets earlier than p_y may belong to the current RU RU_1 or a previous conflicting RU, RU_0, and it may have set *state*[*d_hash*] to NEW_FL.) p_y will set *state*[*d_hash*] to OLD_FL. In that case, p will satisfy (*state*[*d_hash*]=OLD_FL), and C_{old1} will be true. Thus p and subsequent packets of this flow continue to get switched using old rules.

(1.2) (a) If s_p has received both Commit and Commit OK, T=T_{last} for all affected rules. At time T_f, since s_{ff} has not received Commit, it has not sent Ready To Commit.

T_{last} is the largest of the values of time received in Ready To Commit sent by all the affected switches, including s_{ff}. Hence $T_{last}>T_f$. Since $T_f \geq TS_p$, it follows that $TS_p<T_{last}$.

(b) $label=U_p$, as in case (1.1)

(c) $t\_label \neq NEW_p$, as in case (1.1).

(d) Suppose p_x is the last packet of this flow (or a previous flow with the same value of d_hash) that crossed s_p before it sent Ready To Commit. Suppose p is the next packet. p will find $prev\_t[d\_hash]<T_{RC}[V_1]$. Because of this and (a), (b) and (c), it satisfies C_{old1}. It matches line 3 of Algorithm 7 and sets $state[d\_hash]$ to OLD_FL. p is resubmitted and matches an old rule. $state$ and $label$ are unchanged. If p is not the first packet after p_x, as discussed for case (1.1), it will still get switched using old rules, as an earlier packet would have set $state[d\_hash]$ to OLD_FL. Packets subsequent to p also get switched using old rules.

A packet labelled OLD_p gets switched using old rules, in the rest of the affected switches, wherever old rules exist (line 17 of Algorithm 7).

Case 2 - s_{ff} has received Commit and sent Ready To Commit. It has not received Commit OK:

(a) s_{ff} will have new rules, but in all of them $T=T_{max}$. So p will find that $TS_p<T$ in any new rule, as $T=T_{max}$.

(b) $label=U_p$ as this is the first affected switch in the path of p

(c) $t\_label \neq NEW_p$ and

(d) $prev\_t[d\_hash]<T_{RC}[V_1]$ or $state[d\_hash]=OLD\_FL$, using the same arguments in Case 1, (1.1), since the behaviour of s_{ff} will be the same as s_p, for this case.

Since p satisfies C_{old1}, $label$ of p will be made OLD_p (line 20 of Algorithm 7). Once so labelled, all subsequent switches will use old rules on p.

Case 3 - s_{ff} has received both Commit and Commit OK: s_{ff} will match p with a new rule, and if $TS_p \geq T$, it will apply the new rule, if C_{new1} is satisfied.

(3.1) Suppose $TS_p<T_{last}$ for p.

(a) $TS_p<T$ due to the above assumption.

(b) $label=U_p$ as this is the first affected switch in the path of p

(c) $t\_label \neq NEW_p$.

(d) An earlier packet of the flow to which p belongs or to a previous instance of the flow with the same value of d_hash has set $prev\_t[d\_hash]$ to a value less than $T_{RC}[V_1]$ or

$state[d\_hash]$ to OLD_FL, as explained for Case 1, (1.1). Therefore, due to this condition and (a) through (c), C_{old1} is satisfied, the label of p is set to OLD_p and p gets switched by old rules.

(3.2) Suppose $TS_p \geq T_{last}$ and p is not a SYN. C_{old2} = $(TS_p \geq T[n])$ **AND** $(label=U_p)$ **AND** $(t\_label \neq NEW_p)$ **AND** $((prev\_t[d\_hash] < T_{RC}[V])$ **OR** $(state[d\_hash]=OLD\_FL))$ **AND** $(flags \neq SYN)$ **AND** $(ev[V] \neq STOP)$, from Table 6.3.

(a) $TS_p \geq T$ due to the above assumption.

(b) $label=U_p$ as this is the first affected switch in the path of p.

(c) $t\_label \neq NEW_p$.

(d) $flags \neq SYN$ as p is not a SYN

(e) $ev[V] \neq STOP$, as when Commit OK is received, $ev[V]$ is set to $START$, by step 3 of Figure 6.1.

Suppose p_1 was the first packet of the flow (a SYN).

(3.2.1) If p_1 reached s_{ff} before s_{ff} received Commit OK, $TS_p < T_{last}$. An earlier packet, of the flow to which p belongs or a previous instance of the flow to which p belongs, has set (f) $prev\_t[d\_hash]$ to a value less than $T_{RC}[V_1]$ or $state[d\_hash]=OLD\_FL$, as explained for Case 1, (1.1). Therefore, for p, due to (a) through (f), C_{old2} is satisfied and its $label$ is set to OLD_p.

(3.2.2) (a′) If p_1 reached s_{ff} after s_{ff} received Commit OK, TS_p of p_1 is greater than or equal to T_{last}.

(b′) Its $label=U_p$ as this is the first affected switch in the path of p.

(c′) Due to p_1 being a SYN, $flags=SYN$.

C_{new1} = $(TS_p \geq T[n])$ **AND** $(label=U_p)$ **AND** [$(flags=SYN)$ **OR** $(ev[V]=STOP)$] as per Table 6.3. Therefore, due to (a′), (b′) and (c′), p_1 satisfies C_{new1}.

Therefore (a″) $state[d\_hash]=NEW\_FL$ (line 12 Algorithm 7). This also sets $prev\_t[d\_hash]$ to the current time stamp TS_p of the packet.

Since $TS_p \geq T_{last}$ and $T_{last} \geq T_{RC}[V_1]$ by definition, for all subsequent packets of this flow, including p, (b″) $prev\_t[d\_hash] \geq T_{RC}[V_1]$. C_{new2} = $((t\_label=NEW_p)$ **AND** $(TS_p \geq T_{RC}[V]))$ **OR** $((prev\_t[d\_hash] \geq T_{RC}[V])$ **AND** $(state[d\_hash]=NEW\_FL))$ as per Table 6.3. Therefore due to (a″) and (b″), those packets satisfy C_{new2} and get switched using new rules (line 12 of Algorithm 7).

(3.3) Suppose (a) $TS_p \geq T_{last}$

(b) p is a SYN.

(c) $label{=}U_p$ as s_{ff} is the first affected switch.

Due to (a), (b) and (c), p satisfies C_{new1}. Therefore, $label$ of p is set to NEW_p (line 12 of Algorithm 7) and that packet gets switched subsequently by new rules. This also sets $state[d\_hash]$ to NEW_FL. Subsequent packets of this flow use new rules, due to C_{new2} being satisfied, as explained for (3.2.2).

In all the above cases except (3.2.2) and (3.3), any packet p gets switched using old rules, thus preserving PFC in the forward direction. In (3.2.2) and (3.3), all the packets belonging to the flow to which p belongs get switched using new rules, due to the reason explained for (3.2.2), thus preserving PFC in the forward direction. This proves Property 1.

Property 2: If all the packets of a forward flow are switched using new (old) rules, all the packets of its reverse flow are also switched using new (old) rules.

Case R1 - s_{fr} has not received Commit and so it is unaware of an update:

Let p_r be a packet of a reverse flow. When p_r arrives at s_{fr}, it switches p_r using unaffected (soon to be declared old) rules.

(R1.1) A subsequent switch s_{pr} has received Commit and sent Ready To Commit.

(a) t_label of p_r is not NEW_p, for forward flows belonging to this RU, as every packet of the forward flow is switched using old rules, due to cases 1 and 2 above, for forward flows, and Algorithm 6. Algorithm 6 sets t_label to NEW_p for a SYN+ACK if and only if the SYN of the forward flow has its $label$ set to NEW_p. Case 3 cannot be true, as Commit OK can be sent from the controller only after all the affected switches have sent Ready To Commit (as per Figure 6.1, step 3). t_label is not set to NEW_p for any packet other than SYN+ACK, as per Algorithm 6. Therefore, whether p_r is SYN+ACK or a subsequent packet, its $t\_label{\neq}NEW_p$.

Now all the arguments for case 1 above hold good - at s_{pr}, $TS_p{<}T$ as $T{=}T_{max}$, $t\_label{\neq}NEW_p$. p_r is switched using old rules, using the same arguments in case (1.1) and switched using old rules in all the affected switches of the reverse flow.

(R1.2) Suppose s_{pr} has received Commit OK. Now all the arguments of case (1.2) hold good and the packet exits s_{pr} labelled OLD_p.

(R1.3) Suppose a flow belonging to a previous RU, RU_0, has left $istate$ as NEW_FL and $itstamp$ set to the TS_p of its SYN, at the egress for that flow. Now suppose SYN+ACK

arrives at s_{fr} only during RU_1. Suppose s_{pr} has received Commit but not Commit OK.

(a) $TS_p<T$ as $T=T_{max}$

(b) $label=U_p$ as this is the first affected switch for p.

(c) $t\_label=NEW_p$. However, at s_{pr}, $TS_p<T_{RC}[V_1]$, as TS_p is the time stamp of a SYN before RU_1 began. The packet previous to this SYN+ACK, say p_x, with a matching value of d_hash, has crossed s_{pr} before $T_{RC}[V_1]$, as SYN+ACK is the first packet of this flow during RU_1.

(d) Due to p_x, p finds $prev\_t[d\_hash]<T_{RC}[V_1]$. Subsequent packets, including SYN+ACK and later, satisfy C_{old1} due to (a) through (d) and get switched using old rules.

(R1.4) In (R1.3), suppose s_{pr} has received Commit OK.

(a) Now for SYN+ACK, $TS_p<T$ as the TS_p of this packet belongs to the SYN of RU_0, which completed before RU_1 began.

(b) $label=U_p$ as this is the first affected switch that p_r crosses.

(c) $t\_label=NEW_p$. However, at s_{pr}, $TS_p<T_{RC}[V_1]$, because TS_p was assigned its value even before RU_1 began.

(d) Suppose p_x is the last packet of this flow (or a previous flow) that crossed s_{pr} before it sent Ready To Commit, with the same value of d_hash. Suppose p_r is the next packet. p_r will find $prev\_t[d\_hash]<T_{RC}[V_1]$. Therefore, it satisfies C_{old1}. It matches line 3 of Algorithm 7 and sets $state[d\_hash]$ to OLD_FL. p_r is resubmitted and matches an old rule. $state$ and $label$ are unchanged. If p_r is not the first packet after p_x, as discussed for case (1.1), it will still get switched using old rules. Packets subsequent to p_r also get switched using old rules.

Case R2 - s_{fr} has received Commit and sent Ready To Commit. It has not received Commit OK:

(R2.1) In the forward path, if s_{ff} has not received Commit OK, due to cases 1 and 2 above and Algorithm 6, t_label of no affected packet, for which s_{ff} was the first affected switch for the forward direction of its flow, will be set to NEW_p. p_r and subsequent packets of the flow to which p_r belongs will be labelled OLD_p and get switched using old rules, as they satisfy C_{old1}, as explained for Case 2.

(R2.2) In the forward path, suppose s_{ff} has received Commit OK.

(R2.2.1) For cases other than (3.2.2) and (3.3), for all forward flows, all packets get

switched using old rules. Therefore $t\_label \neq NEW_p$ and as described in cases 1 and 2 above, p_r and subsequent packets belonging to its flow get switched using old rules.

(R2.2.2) (a) In cases (3.2.2) and (3.3), p of the forward flow gets switched using new rules. Therefore, the SYN of such a flow causes $istate[hash]$ to be set to NEW_FL and $itstamp$ to TS_p of that SYN (lines 7 and 9 of Algorithm 6). The SYN+ACK of the corresponding reverse flow has t_label set to NEW_p and its TS_p to the value of $itstamp[hash]$ (lines 18 and 19 of Algorithm 6). This SYN+ACK is the first packet of the flow to which p_r belongs, and is called p_{r1}. For p_{r1}, $TS_p \geq T_{last}$, as the SYN from which TS_p was taken, was labelled as NEW_p by s_{ff}.

(b) But T_{last} is the largest of all $T_{RC}[V_1]$. Therefore, $TS_p \geq T_{RC}[V_1]$ at s_{fr}.

Thus due to (a) and (b), p_{r1} satisfies C_{new2}, matches a new rule (line 12 of Algorithm 7), sets $label$ to NEW_p, $state[d\_hash]$ to NEW_FL and gets switched using a new rule. Subsequent switches apply new rules to this packet. Subsequent packets such as p_r do not have t_label set to NEW_p. However, for the first packet after p_{r1}, (a′) $prev\_t[d\_hash] \geq T_{RC}[V_1]$ as $prev\_t$ has the time of the switch stored when p_{r1} exited s_{fr} (line 29 of Algorithm 7), which is greater than or equal to $T_{RC}[V_1]$ because s_{fr} has already sent Ready To Commit. For packets after that, $prev\_t$ will have TS_p of the previous packet, which is greater than or equal to $T_{RC}[V_1]$ (line 32 of Algorithm 7) (A SYN+ACK can arrive at s_{fr} only after a SYN exits an egress, at a time greater than T_{last}, as $TS_p \geq T_{last}$ for that SYN. A response to this from a host can enter the network through the same egress only after time T_{last}. $T_{last} \geq T_{RC}[V_1]$, by definition of T_{last}. Therefore, for packets after p_{r1}, $prev\_t[d\_hash] \geq T_{RC}[V_1]$).

(b′) SYN+ACK has already set $state[d\_hash]$ to NEW_FL.

Therefore, every packet , due to (a′) and (b′), p_r satisfies C_{new2} and gets switched using new rules (line 12 Algorithm 7).

(R2.3) Suppose a flow belonging to a previous RU, RU_0, has left $istate$ as NEW_FL and $itstamp$ set to the TS_p of its SYN, at the egress for that flow. Now suppose SYN+ACK arrives at s_{fr} only during RU_1, but with $t\_label = NEW_p$. s_{fr} has received Commit but not Commit OK. Using the arguments in (R1.3), this SYN+ACK and subsequent packets of this flow get switched using old rules.

Case R3 - s_{fr} has received both Commit and Commit OK :

(R3.1) s_{ff} of the forward flow has not received Commit OK.

(a) In this case, all packets are labelled OLD_p and p_{r1} (a SYN+ACK) has its $t\_label \neq NEW_p$.

(b) $label = U_p$ as this is the first affected switch.

(c) If its $TS_p < T$, it is the same as case (R 2.1) above.

(c′) If its $TS_p \geq T$, suppose the last packet of a previous flow with the same d_hash as p_{r1} that crossed s_{fr} after it sent Ready To Commit, is p_y. For p_y, $prev\_t[d\_hash] < T_{RC}[V_1]$ and it sets $state[d\_hash]$ to OLD_FL.

(d′) $flags \neq SYN$ as p and subsequent packets belong to a reverse flow.

(e′) $ev[V] = START$, as is set in step (3) of Figure 6.1.

Packets subsequent to p_y, including p_{r1}, satisfy C_{old1} if their $TS_p < T_{last}$, due to (a),(b) and (c). They satisfy C_{old2} if their $TS_p \geq T_{last}$, due to (a), (b), (c′) , (d′) and (e′). All packets subsequent to p_{r1} get switched using old rules.

(R3.2) s_{ff} of the forward flow has received Commit OK. If SYN of a forward flow has its $TS_p < T_{last}$, all packets of the forward flow get switched using old rules. $t\_label \neq NEW_p$ of the SYN+ACK (p_{r1}) of its reverse flow. This is the same as case (R3.1) above.

(R3.3) If the SYN of a forward flow has its $TS_p \geq T_{last}$, it gets switched using new rules and sets $t\_label = NEW_p$ for p_{r1}. This is the same as (R2.2.2) above.

In the above discussion, it is assumed that rules at s_{ff}, s_{fr} , s_p and s_{pr} are symmetric. Suppose new (old) rules do not exist at one switch. The label of the packet will be decided by an old (a new) rule, as shown in line 8 (line 3) of Algorithm 7. If it is NEW_p (OLD_p), it will be re-matched, as shown in line 11 (line 7) of Algorithm 7.

The above cases will hold as long as both old rules and new rules co-exist in the affected switches. Old rules are discarded and checking for rule type ceases once a timer with time T_m expires at an affected switch. After an affected switch receives Commit OK, it polls $live\_fl$ every $2 * T_p$ units of time. When each affected packet of an old flow is received, an affected switch checks if the time at which it received the previous packet of the same flow is less than or equal to T_p, or whether the packet is a SYN or a SYN+ACK. If either of these conditions is satisfied, it checks if it updated $live\_fl$ more than T_p units ago. If so, it stores TS_p of that packet in $live\_fl$ (Algorithm 8). If the control plane of the affected switch finds that $live\_fl$ has increased from its previous value, it restarts its poll timer. If not, all the old flows have ceased and it stops the timer, sets $ev[V]$ to $STOP$ and sends sends Ack Commit OK with the current time to the controller (step

4 of Figure 6.1). The objective is to ensure that no old rule gets deleted while old flows with an inter-packet delay less than or equal to T_p are in the network.

Suppose at time t_1, $live\_fl$ was updated. In the worst case, packets arrive belonging to various old flows, up to $t_1+T_p-\epsilon$, where ϵ is a small value of time. $live\_fl$ is not updated by the last packet at $t_1 + T_p - \epsilon$, as T_p units have not elapsed since it was updated at t_1 before. Suppose no packet arrives for any old flow from $t_1+T_p-\epsilon$ to t_1+2T_p. Then $live\_fl$ will show no increase when polled at t_1+2T_p and it can be correctly concluded that there are no old flows, as at least one packet belonging to an old flow must have arrived, in the worst case, at $t_1 + T_p - \epsilon + T_p$=$t_1 + 2T_p - \epsilon$. However, if, for at least one old flow another packet arrives with an inter-packet delay less than or equal to T_p, it will arrive before or at $t_1 + 2 * T_p - \epsilon$, thus leading to an update of $live\_fl$. In that case, when $live\_fl$ is read at $t_1 + 2 * T_p$, the control plane of the affected switch correctly concludes that old flows exist.

Once the controller receives Ack Commit OK from all affected switches, it sends a Discard Old message with T_{del} set to the time of the last Ack Commit OK sent (step 5 of Figure 6.1). The timer value T_m is then set by each switch as $T_{del} + M - T_i$, to account for the elapsed time from T_{del} to T_i. After time T_{del}, no packet with time stamp greater than T_{del} will be switched with old rules. So the last packet to be using old rules will have $TS_p \leq T_{del}$. By definition the maximum lifetime of a packet in the network is M. So after time $T_{del} + M$, no old packet will be there in the network and so the old rules can be deleted. This ensures that old rules are not discarded prematurely. Packets with label NEW_p will still be there in the network, but since switches will have converted the new rules to type U, the new rules will continue to apply.

It has to be ensured though, that the next conflicting update does not start too soon so that a packet with label NEW_p (but of the previous update) is not erroneously handled. To take care of this, the next update is not started until after another time period of M has elapsed. No packet will have its *label* set to NEW_p if its time stamp $TS_p > T_{del} + M$, since new rules will no longer exist at any switch. Waiting for time M will ensure that all such new packets leave the network before the next update begins.

This proves Property 2. Therefore PFC is preserved for all flows during an RU.

Chapter 7

Conclusions

Software Defined Networks can be managed better than traditional networks using applications that are written using the abstractions provided by the control plane of the network. These applications update the switches of the network to achieve the desired functions. For reasons of safety and efficiency, updates need to be performed meeting the properties of per-packet and per-flow consistency.

7.1 Summary of Contributions

This thesis has presented general algorithms that preserve per-packet and per-flow consistency to update SDNs.

The algorithms progressively improve efficiency and concurrency of non-conflicting updates. The algorithms are applicable for any kind of updates, not affected by switch and network delays, do not rely on flows to exist for the update to progress, support wild carded rules and provide an all-or-nothing semantics. Efficiency is progressively achieved by restricting the interaction to the switches where updates are to take place and to the rules that are being changed. Unlimited concurrency of disjoint updates is eventually achieved by not using labels in the data plane to distinguish updates. While the first PPC-preserving algorithm E2PU affects all the switches and does not support concurrency, CCU requires changes to only the affected switches and the ingresses and supports concurrent disjoint updates, limited by the size of a packet header field. PPCU further improves upon CCU and requires changes to only the affected switches and rules and supports an unlimited number of disjoint updates. Similarly, ProFlow improves upon EPCU

and requires changes to only the affected switches and rules and achieves practically unlimited concurrency of disjoint updates that are per-flow consistent.

Our algorithm PPCU, for per-packet consistent updates, and ProFlow, for per-flow consistent updates, use data plane time stamps to decide when new rules must be applied to a packet. They assume that network switches have access to a synchronous clock, while tolerating time drifts bounded by a limit. In a theoretical analysis of control plane parameters, PPCU fares better than comparable algorithms. The implementation of PPCU in P4 and evaluation in Mininet demonstrate that under realistic network conditions, continuous PPCU updates provide better throughput and complete more flows compared to random updates, without violating safety. The prototype of ProFlow implemented in P4 and evaluated on Mininet demonstrates that the algorithm is feasible and causes no impact on the throughput of new flows, compared to a situation with no updates, if the first affected switch in the course of the flow has both new and old rules. Additionally, since ProFlow is immediately effective for new flows, supports connection affinity and requires no changes to NFs, it is a viable alternative for disparate update problems in SDNs. Moreover, since both the algorithms work at line rate and tolerate time drifts, they are practical. In order to use these algorithms, the high level languages that program an SDN must generate the rules to be installed on switches, as per the expectations of these algorithms.

PPCU and ProFlow also take advantage of the increased processing power and programmability of programmable data plane switches. The ability of these switches to process packet headers in the data plane, to maintain states and to resubmit packets enable updates to be confined to the affected switches and rules, to manage any kind of update, be it only insertion or only deletion or both, of wild carded rules, and to manage rule asymmetry, that is, an affected switch not having an old and a new rule to match every affected packet. Restricting resubmission to only the first affected switch in the path of the flow and the resubmission duration to the duration of overlap of the old and new rules reduce the impact on throughput. Since programmable switches manage resources including switch tables better, both the new and old rules co-existing on an affected switch for some time during the RU, the duration of which we have quantified, becomes tolerable. Later improvements in P4 such as atomic constructs and support of expressions in actions will improve throughput and make implementation of the algorithms easier.

To conclude, PPCU and ProFlow form the ***first*** set of general, efficient and practical update algorithms that implements the strict consistency requirements of per-packet and per-flow consistency. The algorithms are general as no assumptions are made about the nature of updates (supporting insertion of rules, deletion of rules or both) or rules (may be wild carded and asymmetric) or network topology, efficient because updates are confined to the affected switches and rules and practical because they operate at line rate and tolerate time drifts.

7.2 Future Work

7.2.1 Update Algorithms

It is possible to include other consistency requirements such as event-driven consistency [84] and general requirements such as concurrent *conflicting* updates into the ambit of the algorithm, and is being done. At the same time, since per-packet and per-flow consistency are strict consistency requirements, the reaction time to network events is high. How can this be reduced?

An affected switch needs to know two things for the update to progress: 1) whether all the affected switches have completed some event (such as completion of installation of new rules or started application of new rules) and 2) the latest time at which that event occurred (the latest rule installation occurred or the latest time at which it started applying new rules). It uses this information a) to instruct a packet to use a certain rule version, by reading, for instance, the time at which the packet crossed the ingress and the state of the packet or b) to instruct itself, for instance, to delete rules at some time, by reading its own time. The above require messages to be sent to the controller, increasing the update time. Using specially constructed packets in the data plane to inform the affected switches, in addition to involving the controller, may speed up the update, at the cost of increased processing of the switches and network bandwidth and is worthy of exploration. Special probes in a data centre network used by HULA [68] for load balancing purposes can perhaps be used for this purpose.

7.2.2 Deriving abstractions

"Informally, a fact is *common knowledge* if it is true, everyone knows it, everyone knows that everyone knows it, and so on ad infinitum" [96, 51]. The algorithms PPCU and ProFlow fundamentally use an algorithm for affected switches to attain a level of knowledge weaker than common knowledge. The knowledge that other affected switches have installed a new version of rules (or they have not) and the value of the time stamp of an affected packet are sufficient for a first affected switch to decide which version of rules to apply. The fact that affected switches other than the first affected switch may or may not know that other affected switches have installed new rules (or not) is immaterial. Providing the packet with the correct label (new or old) is sufficient for the packet to traverse the network in a per-packet consistent manner. We believe that formalising the notion of weak common knowledge, and the problems solved by PPCU and ProFlow and their solutions for synchronous systems with reliable communication and time drifts with an upper limit, will be of interest to the distributed systems community.

7.2.3 Distributed systems theory

In general, the tradeoff between strictness of consistency, processing power and memory availability at switches, processing power at the controller, network bandwidth and update time, is an interesting and useful area for further exploration. What are the impossibility results in this area ? What are the parameters that can be tuned for the algorithms to be of use ? Is it possible to construct general algorithms with tunable parameters? Additionally, the interaction of the above parameters with network availability and partition tolerance [104] may be explored.

Appendix A

Publications

A.1 Publications in Conferences

- Radhika Sukapuram and Gautam Barua, "**Enhanced Algorithms for Consistent Network Updates**," IEEE Network Function Virtualization and Software Defined Networks (IEEE NFV-SDN 2015), San Francisco
- Radhika Sukapuram and Gautam Barua, "**CCU: Algorithm for Concurrent Consistent Updates for a Software Defined Network**," Twenty Second National Conference on Communications (NCC 2016), Guwahati
- Radhika Sukapuram and Gautam Barua, "**PPCU: Proportional Per-Packet Consistent Updates for Software Defined Networks**," Poster, IEEE 24th International Conference on Network Protocols (ICNP 2016), Singapore

A.2 Manuscripts Submitted

- Radhika Sukapuram and Gautam Barua, "**PPCU: Proportional Per-Packet Consistent Updates for SDNs using Data Plane Time Stamps**," August 2017
- Radhika Sukapuram and Gautam Barua, "**ProFlow: Proportional Per-Flow Consistent Updates**", December 2017

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM CCR*, 38(4):63–74, August 2008.

[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, page 19, 2010.

[3] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. OpenVirtex: Make your virtual SDNs programmable. In *HotSDN*, pages 25–30. ACM, 2014.

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM CCR*, volume 44, pages 503–514. ACM, 2014.

[5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM CCR*, volume 40, pages 63–74. ACM, 2010.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM CCR*, volume 43, pages 435–446. ACM, 2013.

[7] Ashok Anand, Vyas Sekar, and Aditya Akella. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM CCR*, volume 39, pages 87–98. ACM, 2009.

[8] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. *POPL*, 49(1):113–126, 2014.

[9] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. Programming slick network functions. In *SOSR*, page 14. ACM, 2015.

[10] Apache. Keepalivetimeout directive. https://httpd.apache.org/docs/2.4/mod/core.html#keepalivetimeout.

[11] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *SIGCOMM*, pages 29–43. ACM, 2016.

[12] Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang, and Jeffrey C Mogul. Democratic resolution of resource conflicts between SDN control programs. In *CoNEXT 2014*, pages 391–402. ACM, 2014.

[13] Mohammad Banikazemi, David Olshefski, Ali Shaikh, John Tracey, and Guohui Wang. Meridian: an SDN platform for cloud network services. *Communications Magazine, IEEE*, 51(2):120–127, 2013.

[14] Neda Beheshti and Ying Zhang. Fast failover for control traffic in software-defined networks. In *GLOBECOM*, pages 2665–2670. IEEE, 2012.

[15] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the complexity of network management. In *NSDI*, pages 335–348, 2009.

[16] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, pages 267–280. ACM, 2010.

[17] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, page 8. ACM, 2011.

[18] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[19] Andreas Blenk, Arsany Basta, Martin Reisslein, and Wolfgang Kellerer. Survey on network virtualization hypervisors for software defined networking. *IEEE Communications Surveys & Tutorials*, 18(1):655–685, 2016.

[20] Michael Borokhovich, Liron Schiff, and Stefan Schmid. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *HotSDN*, pages 121–126. ACM, 2014.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3):87–95, 2014.

[22] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM CCR*, volume 43, pages 99–110, 2013.

[23] R. Braden. *RFC 1122 Requirements for Internet Hosts - Communication Layers*. Internet Engineering Task Force, October 1989.

[24] Mihai Budiu and Chris Dodd. The P416 Programming Language. *ACM SIGOPS Operating Systems Review*, 51(1):5–14, 2017.

[25] Marco Canini, Petr Kuznetsov, Dan Levin, Stefan Schmid, et al. A distributed and robust SDN control plane for transactional network updates. In *INFOCOM 2015*. IEEE, 2015.

[26] Zizhong Cao, Murali Kodialam, and TV Lakshman. Traffic steering in software defined networks: Planning and online routing. In *ACM SIGCOMM CCR*, volume 44, pages 65–70. ACM, 2014.

[27] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating SDN application failures with LegoSDN. In *HotSDN*, page 22. ACM, 2014.

[28] Huan Chen and Theophilus Benson. The case for making tight control plane latency guarantees in sdn switches. In *SOSR*, pages 150–156. ACM, 2017.

[29] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dRMT: Disaggregated Programmable Switching. In *SIGCOMM*, pages 1–14. ACM, 2017.

[30] The P4 Language Consortium. The P4 language specification, version 1.1.0, January 27, 2016.

[31] The P4 Language Consortium. The $P4_{16}$ Language Specification, Version 1.0.0, 16 May 2017.

[32] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM*, volume 41, pages 254–265. ACM, 2011.

[33] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *INFOCOM 2013*, pages 2130–2138. IEEE, 2013.

[34] Docker. https://www.docker.com/, 2017.

[35] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. Scalable network virtualization in software-defined networks. *IEEE Internet Computing*, 17(2):20–27, 2013.

[36] Abhishek Dwaraki and Tilman Wolf. Adaptive service-chain routing for virtual network functions in software-defined networks. In *HotMiddleBox*, pages 32–37. ACM, 2016.

[37] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. BUZZ: Testing context-dependent policies in stateful networks. In *NSDI*, pages 275–289, Santa Clara, CA, 2016. USENIX Association.

[38] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Flowtags: enforcing network-wide policies in the presence of dynamic middlebox actions. In *HotSDN*, pages 19–24. ACM, 2013.

[39] Nick Feamster, Jennifer Rexford, and Ellen W. Zegura. The road to SDN: an intellectual history of programmable networks. *SIGCOMM CCR*, 44(2):87–98, 2014.

[40] Andrew D Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM CCR*, volume 43, pages 327–338. ACM, 2013.

[41] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. Survey of consistent network updates. *CoRR*, abs/1609.02305, 2016.

[42] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, et al. Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134, 2013.

[43] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, volume 46, pages 279–291. ACM, 2011.

[44] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Vyas Sekar, and Aditya Akella. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. *arXiv preprint arXiv:1305.0209*, 2013.

[45] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. *ACM SIGCOMM CCR*, 44(4):163–174, 2015.

[46] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM*. ACM, 2016.

[47] Soudeh Ghorbani and Philip Brighten Godfrey. COCONUT: Seamless Scale-out of Network Elements. In *EuroSys*, pages 32–47, 2017.

[48] Soudeh Ghorbani, Cole Schlesinger, Matthew Monaco, Eric Keller, Matthew Caesar, Jennifer Rexford, and David Walker. Transparent, Live Migration of a Software-Defined Network. In *SOCC*, pages 1–14. ACM, 2014.

[49] Saikat Guha, Kaushik Biswas, Bryan Ford, Senthil Sivakumar, and Pyda Srisuresh. NAT Behavioral requirements for TCP. Technical report, 2008.

[50] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, pages 79–84. ACM, 2012.

[51] Joseph Y Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)*, 37(3):549–587, 1990.

[52] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[53] Jong Hun Han, Prashanth Mundkur, Charalampos Rotsos, Gianni Antichi, Nirav H Dave, Andrew William Moore, and Peter G Neumann. Blueswitch: Enabling provably consistent configuration of network switches. In *ANCS*, pages 17–27. IEEE Computer Society, 2015.

[54] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[55] Victor Heorhiadi, Seyed Kaveh Fayaz, Michael K Reiter, and Vyas Sekar. Snips: A software-defined approach for scaling intrusion prevention systems via offloading. In *ICISS*, pages 9–29. Springer, 2014.

[56] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM 2013*, volume 43, pages 15–26. ACM, 2013.

[57] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. FLOWGUARD: building robust firewalls for software-defined networks. In *HotSDN*, pages 97–102. ACM, 2014.

[58] Intel. Intel®Ethernet Switch FM5000/FM6000 1Gb/2.5Gb/10Gb/40Gb Ethernet (GbE) L2/L3/L4 Chip Datasheet.

[59] Aakash S Iyer, Vijay Mann, and Naga Rohit Samineni. Switchreduce: Reducing switch state and controller involvement in openflow networks. In *IFIP Networking Conference, 2013*, pages 1–9. IEEE.

[60] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Expe-

rience with a globally-deployed software defined WAN. In *SIGCOMM*, volume 43, pages 3–14. ACM, 2013.

[61] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. Softcell: Scalable and flexible cellular core network architecture. In *CoNEXT*, pages 163–174. ACM, 2013.

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

[63] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, pages 103–115, 2015.

[64] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*, pages 97–112, 2017.

[65] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. Efficient traffic splitting on commodity switches. In *CoNEXT*. ACM, 2015.

[66] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *INFOCOM*, pages 545–549. IEEE, 2013.

[67] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *SOSR*, 2016.

[68] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *SOSR*, 2016.

[69] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *HotSDN*, pages 49–54. ACM, 2013.

[70] 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.

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

[72] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[73] Michael Laor and Lior Gendel. The effect of packet reordering in a backbone link on application throughput. *IEEE Network*, 16(5):28–36, 2002.

[74] Aggelos Lazaris, Daniel Tahara, Xin Huang, Erran Li, Andreas Voellmy, Y Richard Yang, and Minlan Yu. Tango: Simplifying sdn control with automatic switch property inference, abstraction, and optimization. In *CoNEXT*, pages 199–212. ACM, 2014.

[75] Ka-Cheong Leung, Daiqin Yang, et al. An overview of packet reordering in transmission control protocol (TCP): problems, solutions, and challenges. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):522–535, 2007.

[76] Xin Li and Chen Qian. An NFV Orchestration Framework for Interference-Free Policy Enforcement. In *ICDCS*, pages 649–658. IEEE, 2016.

[77] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating data center networks with zero loss. *ACM SIGCOMM CCR*, 43(4):411–422, 2013.

[78] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *NSDI*, pages 499–512, 2015.

[79] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *HotNets*, page 15. ACM, 2014.

[80] Shouxi Luo, Hongfang Yu, et al. Fast incremental flow table aggregation in SDN. In *ICCCN*, pages 1–8. IEEE, 2014.

[81] Shouxi Luo, Hongfang Yu, and Lemin Li. Consistency is not easy: How to use two-phase update for wildcard rules? *Communications Letters, IEEE*, 19(3):347–350, 2015.

[82] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing state: Consistent updates for stateful and programmable data planes. In *SOSR*, pages 115–121. ACM, 2017.

[83] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *HotNets*, page 20. ACM, 2013.

[84] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černỳ. Event-driven network programming. In *PLDI*, pages 369–385. ACM, 2016.

[85] Rick McGeer. A safe, efficient update protocol for openflow networks. In *HotSDN*, pages 61–66. ACM, 2012.

[86] Rick McGeer. A correct, zero-overhead protocol for network updates. In *HotSDN*, pages 161–162. ACM, 2013.

[87] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network Function Virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2016.

[88] Mininet. http://mininet.org, 2017.

[89] Tal Mizrahi and Yoram Moses. Using REVERSEPTP to distribute time in software defined networks. In *ISPCS*, pages 112–117. IEEE, 2014.

[90] Tal Mizrahi and Yoram Moses. Software defined networks: Its about time. In *IEEE INFOCOM*, 2016.

[91] Tal Mizrahi, Ori Rottenstreich, and Yoram Moses. TimeFlip: Scheduling network updates with timestamp-based TCAM ranges. In *INFOCOM*, pages 2551–2559. IEEE, 2015.

[92] Tal Mizrahi, Efi Saat, and Yoram Moses. Timed consistent network updates. In *SOSR*, 2015.

[93] Jeffrey C Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. Corybantic: Towards the modular composition of SDN control programs. In *HotNets*, page 1. ACM, 2013.

[94] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, volume 47, pages 217–230. ACM, 2012.

[95] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *NSDI*, volume 13, pages 1–13, 2013.

[96] Yoram Moses and Mark R Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3(1-4):121–169, 1988.

[97] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Rules placement problem in openflow networks: a survey. *IEEE Communications Surveys & Tutorials*, 18(2):1273–1286, 2016.

[98] Catalin Nicutar, Christoph Paasch, Marcelo Bagnulo, and Costin Raiciu. Evolving the internet with connection acrobatics. In *HotMiddleBox*, pages 7–12. ACM, 2013.

[99] NLANR/DAST. https://iperf.fr/.

[100] Noviflow. https://noviflow.com/.

[101] Openflow switch specification version 1.5.0. 2014.

[102] Recep Ozdag. White paper:intel® Ethernet Switch FM6000 Series-Software Defined Networking. 2012.

[103] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for NFV applications. In *SOSP*, pages 121–136. ACM, 2015.

[104] Aurojit Panda, Colin Scott, Ali Ghodsi, Teemu Koponen, and Scott Shenker. Cap for networks. In *HotSDN*, pages 91–96. ACM, 2013.

[105] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al.

Ananta: Cloud scale load balancing. In *ACM SIGCOMM CCR*, volume 43, pages 207–218. ACM, 2013.

[106] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[107] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI 2013*, 2013.

[108] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM CCR*, volume 44, pages 407–418. ACM, 2014.

[109] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN programming with Pyretic. *Technical Reprot of USENIX*, 2013.

[110] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *HotSDN*, pages 109–114. ACM, 2013.

[111] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *SIGCOMM*, pages 323–334, 2012.

[112] Jonathan Rosenberg. Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal for offer/answer protocols, RFC 5245. Technical report, 2010.

[113] Jonathan Rosenberg. TCP candidates with interactive connectivity establishment (ICE) RFC 6544. 2012.

[114] Ori Rottenstreich, Pu Li, Inbal Horev, Isaac Keslassy, and Shivkumar Kalyanaraman. The switch reordering contagion: Preventing a few late packets from ruining the whole party. *IEEE Transactions on Computers*, 63(5):1262–1276, 2014.

[115] Scott Shenker, Martn Casado, Teemu Koponen, and Nick McKeown. A gentle introduction to SDN. http://tce.technion.ac.il/files/2012/06/Scott-shenker.pdf, 2012.

[116] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 1:132, 2009.

[117] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM*, pages 15–28. ACM, 2016.

[118] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *SOSR*, pages 164–176. ACM, 2017.

[119] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the network with Merlin. In *HotNets*, page 24. ACM, 2013.

[120] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, pages 213–226. ACM, 2014.

[121] Open source. https://github.com/p4lang.

[122] Luca Cittadini Stefano Vissicchio. FLIP the (flow) table: Fast lightweight policy-preserving SDN updates. In *INFOCOM 2016*. IEEE.

[123] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. In *SIGCOMM*, pages 563–574. ACM, 2014.

[124] Celio Trois, Marcos D Del Fabro, Luis CE de Bona, and Magnos Martinello. A Survey on SDN Programming Languages: Toward a Taxonomy. *IEEE Communications Surveys & Tutorials*, 18(4):2687–2712, 2016.

[125] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. *Practical Aspects of Declarative Languages*, pages 235–249, 2011.

[126] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *HotSDN*, pages 43–48. ACM, 2012.

[127] Haining Wang, Danlu Zhang, and Kang G Shin. Detecting SYN flooding attacks. In *INFOCOM*, volume 3, pages 1530–1539. IEEE, 2002.

[128] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *Hot-ICE*. USENIX Association, 2011.

[129] Wen Wang, Wenbo He, and Jinshu Su. Redactor: Reconcile network control with declarative control programs in SDN. In *ICNP*, pages 1–10. IEEE, 2016.

[130] Yang Wang, Gaogang Xie, Zhenyu Li, Peng He, and Kavé Salamatian. Transparent flow migration for NFV. In *ICNP*, pages 1–10. IEEE, 2016.

[131] Business Wire. http://www.businesswire.com/news/home/20171010005970/en/NoviFlow-Deliver-WorldE28099s-Highest-Performance-SDN-Switch-Solution.

[132] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *NSDI*, pages 299–312. USENIX Association, 2018.

[133] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *Communications magazine, IEEE*, 51(2):136–141, 2013.

[134] Yifei Yuan, Franjo Ivančić, Cristian Lumezanu, Shuyuan Zhang, and Aarti Gupta. Generating consistent updates for software-defined network configurations. In *HotSDN*, pages 221–222. ACM, 2014.

[135] Pamela Zave, Ronaldo A Ferreira, Xuan Kelvin Zou, Masaharu Morimoto, and Jennifer Rexford. Dynamic service chaining with dysco. In *SIGCOMM*, pages 57–70. ACM, 2017.

[136] Bin Zhang, Pengfei Zhang, Yusu Zhao, Yongkun Wang, Xuan Luo, and Yaohui Jin. Co-scaler: Cooperative scaling of software-defined nfv service function chain. In *IEEE NFV-SDN*, pages 33–38. IEEE, 2016.

[137] Shuyuan Zhang, Franjo Ivancic, Cristian Lumezanu, Yuan Yuan, Arpan Gupta, and Sharad Malik. An adaptable rule placement for software-defined networks. In *DSN*, pages 88–99. IEEE.

[138] Jiaqi Zheng, Guihai Chen, Stefan Schmid, Haipeng Dai, and Jie Wu. Chronus: Consistent data plane updates in timed SDNs. In *ICDCS*, pages 319–327. IEEE, 2017.

[139] Wenxuan Zhou, Jason Croft, Bingzhe Liu, and Matthew Caesar. NEAt: Network Error Auto-Correct. In *SOSR*, pages 157–163. ACM, 2017.

[140] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and Brighten P. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *NSDI*, pages 73–85, May 2015.