Enhanced Algorithms for Consistent Network Updates

Radhika Sukapuram
Dept. of Computer Science and Engineering
Indian Institute of Technology Guwahati
Assam, India 781039
Email: r.sukapuram@iitg.ernet.in

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

***Abstract*—The basic algorithm that consistently updates the switches of a Software Defined Network while preserving the update property of per-packet consistency is the two-phase update. The two-phase update 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) *recoverability* of the update. Recoverability ensures that those updates for which the two-phase algorithm is not completely executed do not change the semantics of those updates for which the algorithm is completely executed. This paper examines the failures that can occur during an update and how to handle a subset of those failures. It proposes an algorithm, enhancing the two-phase update to handle deletion of old rules and recoverability. It also specifies how to effectively use a software cache to supplement the TCAM, *during an update*. The paper extends the same algorithm for per-flow consistent updates, using a restricted number of exact-match rules for existing flows and specifying when to delete the exact-match rules. The two algorithms are also specified for switches that do not support a software cache. The paper also analyzes the algorithms quantitatively, identifying the parameters of interest and what they depend upon.**

I. Introduction

Dynamically updating the rules installed on a switch is central to a Software Defined Network (SDN). The two-phase update [1] is a basic algorithm that updates the switches of an SDN, while maintaining the property of *per-packet consistency*. In a per-packet consistent update (PPC), any packet entering the network is either forwarded using the old version of the rules or the new version and never a mixture of both. Similarly, in a per-flow consistent update (PFC), all packets belonging to a flow are either forwarded using the old version of rules or the new version of rules, but not a mixture of both. A flow is a sequence of packets where each packet has the same values for the following headers: source IP address, destination IP address, source port, destination port, ingress input port and protocol and where each packet in the sequence is not separated by more than n seconds. The value of n depends on the protocol [1]. We describe the problems of existing algorithms for PPC and PFC and propose enhancements to those algorithms that solves those problems. We also provide a quantitative analysis on the performance of the enhanced algorithms, identifying the parameters of interest and what they depend upon. We assume Openflow switches [2] to keep the descriptions concise.

II. Per-packet consistent updates

A. Two-phase update

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 two-phase update process, 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 has left the network, the controller instructs all the ingress and internal switches to delete the $v0$ rules [1].

If an update is not PPC, packets may be dropped or they may loop. Manually finding an update sequence of switches that does not cause packet drops or loops and programming the network in that sequence is a difficult task [1].

B. Under-specification of two-phase update

Lack of update recoverability: 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.

Updates to an SDN can be frequent - [3] proposes a fine-grained traffic engineering mechanism for data centres that involves updating switches at the granularity of seconds. [4] proposes updates every few minutes and SWAN [5] proposes updates as frequently as possible, to achieve high WAN utilization. If the network is updated based on the current workload,

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

changing workloads would necessitate frequent updates in the data center [6], for dynamic flow scheduling. *Since the update rate 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. The two-phase update does not address *recoverability of an update*. By recoverability of an update, we mean that either an update must be completely installed on all the switches or none of the updates must be installed; the update must never be partially installed. Recoverability ensures that those updates where the two-phase algorithm is not fully executed do not change the semantics of those updates where the algorithm is fully executed (after [7]). 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, the algorithm 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 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 rereoute 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 [8], algorithms to implement local fast failover [9], [10], re-design of the controller to reduce fate-sharing between apps and the controller [11] etc. but none on failure during an update itself. In the absence of actual data on the frequency of failures, *software must try to provide as much protection as possible*. This helps *hide some of the failure complexity from applications* and architects can choose hardware that improves overall cost, in the context of a data center [12]. It may also be possible to use *hardware from different vendors more easily*, which is one of the goals of SDN. Also, as argued for server failures in [12], it *"simplifies a broad class of operational procedures"*. For example, switch software upgrades or shutting down a switch can happen during normal operation, without special procedures; handling failures during updates is a movement in this direction.

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, violating the property of drop-freedom. But deleting a rule cannot be indefinitely delayed because TCAM space is scarce.

Since the controller monitors information in the switch on a per-rule basis and not on a per-flow basis, it does not have any means to know whether there are active flows associated with a (wild-carded) rule before it issues a delete command for that rule. 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 [13]. 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*. It is desirable to have a mechanism to delete the old rule set independent of the above considerations.

C. Usage of Software Rules Table

Since TCAM space is scarce, it is desirable to use a software cache to supplement the TCAM such that rules are installed both in the TCAM and in a rules table implemented in software, called the Software Rules Table (SRT), a model of which is described in [14]. Since switching in TCAM is faster, all high priority rules can be periodically moved to TCAM and the remaining rules can be in the SRT. Adding (removing) rules to (from) a TCAM incurs considerable overhead [15], while doing the same to an SRT is faster.

D. Enhanced 2 Phase Update with SRT (E2PU-SRT)

E2PU-SRT addresses failures F1 through F6 and the other issues identified in section II-B, while preserving the update property of per-packet consistency.

The switch on which a rule needs to be inserted or modified is called an affected switch.

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 received 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 to $v1$. *After receiving "Commit OK", a switch may move the $v1$ rules to TCAM at any time*.
4) Each switch that processed "Commit OK" sends "Ack Commit OK" to the controller. Each *ingress* switch sends the current time with this message, called T_o.
5) After receiving "Ack Commit OK" from all the affected *ingresses*, the controller notes the latest value of T_o received and saves it as T_l. Now it sends "Discard Old" to all the switches where rules need to be deleted. It sends T_l 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 $T_c > T_l + M$, 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.

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 [2].

After T_l, 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.

Handling Failures and Application Aborts:[2] If one (or more) of the switches is unable to process a "Commit" due to any of the failures F1 through F6 in section II-B, 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. In that case, the controller sends a "Commit" for adding $v1$ rules, using the tag preserved for this application. It appends the delete commands meant for the $v0$ rules and the delete commands for the completed $v1$ rules, if any, 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", the application must issue a new update, if required. If the application requests the controller to abort the update before the controller has received all the "Ack Commit OK" messages, the controller aborts the update. If the controller aborts the update, all $v1$ rules are deleted.

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 $T_l + M$, 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. This needs further exploration.

III. Per-Flow Consistent Updates

A. Challenges in PFC

When the controller instructs the ingress to switch to $v1$, some of the old $v0$ packets are very likely to be present in the network. It is possible that they reach the egress switch later than the new packets tagged with $v1$, as the $v0$ packets may follow a different path. Switches take a lot of care not to re-order packets [16], but updates could result in packets that are re-ordered, causing performance degradation [17] and if updates need to be performed frequently, the impact will be severe. This is one of the reasons for needing a PFC. Another reason is that a load balancer splits packets destined to a single server across multiple servers, by restamping the destination IP address of every received packet. It determines the destination address depending upon the rule installed in it, which usually allocates the destination address based on the source address of the packet. During an update to a load balancing switch [18], connections existing at the time of the update will be broken unless the update is per-flow consistent. Finally, applications will also need PFC; for example, a policy change that drops packets may disrupt a large file transfer.

For a PFC, the affected flows must complete without any disruption due to the update, regardless of the duration of the

[2]Unlike DBMS updates, a failure will not block the update

flow or the flow rate and these flows *must not* switch over to the new version of the rule. At the same time, new flows that originate after a certain stage in the update must follow the new version of rules. Ingress switches need to know when the affected flows will stop and they cannot switch over to $v1$ tagging like in PPC. Internal switches need to delete old rules, but the method used in PPC will not work as $v0$ packets may continue to come from existing flows while packets of new flows may be tagged as $v1$. So only when all old flows have ceased, can an internal switch delete $v0$ rules.

B. Existing Algorithms

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 [19] proposes a switch that can support a "clone" command and usage of this to support PFC is examined in [1]. 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. 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. 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. *This method has the following issues: 1) Microrules are always present and so the overheads of these microrules will affect the performance of a switch 2) It is unclear when the microrules get deleted 3) As in PPCs, when the old rules are deleted is unspecified and in the absence of acknowledgements from the switch, the update installation may be unrecoverable*.

[18] provides 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 R2, instead of replica R1. During the update, the controller installs transition rules that direct *all* 0* packets to the controller. The controller examines the next packet of all 0* 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 R2 and the old connections to R1. The controller examines 0* packets for 60 seconds to see if there is any missed connection ; if not, it installs the rule that directs 0* to R2, at a lower priority than the microrules. In the second solution, the controller first installs the new rule that directs all traffic to R2, 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 has the danger of some of the rules never timing out due to new connections being continuously established that match one of the temporary rules*.

C. Enhanced Per-Flow Consistent Update with SRT (EPCU-SRT)

In comparison with the first existing solution mentioned above [1], the overhead of microrules is greatly reduced as we restrict microrule generation between the switch receiving a "Commit" and "Commit OK". We also specify a method to delete the microrules. Comparing with the first solution in [18], our solution does not involve the controller and with the second solution, 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 follows, assuming one flow table:

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 enables microrule creation. For every new header of a packet which does not have a microrule, a new microrule for that header is created.
   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 (Openflow v1.5.0 [2] supports this). 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, then it is removed when the timeout associated with it is triggered.* This is discussed further in III-D.
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). 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_o. 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_o received and saves it as T_l. Now it sends "Discard Old" to all the switches where rules need to be deleted. It sends T_l 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 $T_c > T_l + M$, 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.

If a load balancer requires an update from $v0$ to $v1$, we assume that the load balancer is an internal switch that requires an update and we proceed with the update as outlined above. If multiple, pipelined flow tables exist, the algorithm can be modified to handle them. It is not described here due to space constraints.

D. 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 ([20] and [21]), 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.

E. Long-lived Flows

Every update need not be a PFC update. Data centers may have flows that are so long that waiting for them to complete during an update may render the rest of the update useless. For long-lived flows, applications must not instruct the controller to use a PFC update when such flows are affected by the update. Alternately, applications may exclude such flows from the update. However, it may be required to do a PFC update for such a flow - for example, an application wishes to install a policy change that affects a long-lived flow (or a set of long-lived flows), but the flow must not be disrupted and the policy change must be effective as soon as the flow is completed. In such cases, the application must first install a separate rule for the long-lived flow, which is of higher priority than the general rule (if a separate rule does not already exist). Now it must do the desired PFC update, with only this flow. [6] and [19] provide mechanisms to identify long-lived flows. In the internet, applications like BitTorrent have permanent TCP connections that use keep-alive timers during their idle time [22] - PFC updates are not meaningful for such flows.

TABLE I. SYMBOLS USED IN THE ANALYSIS

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
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
$t_{u\mu}$	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
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

IV. 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", need 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. In the case of a PFC, 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 V mentions further comparisons.

V. ANALYSIS OF THE ALGORITHM

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 4) **Parameter 4**: Time complexity - the total update time and 5) **Parameter 5**: Duration for which microrules are present at the ingress (relevant only for PFC). Parameters 3 and 4 are along the lines of complexity measures in distributed systems [7] .

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

The symbols used in the analysis is as per Table I. It is assumed that the time taken for the sum of the propagation times and switch delays between the controller and the switches is

TABLE II. ANALYSIS OF PPC AND PFC

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

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}+δ+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) 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.

Table II captures all the parameters for a PPC and a PFC. For PFC updates, 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 E2PU-SRT' and EPCU-SRT'.

Observations: 1) For networks without an SRT, parameters 1,2 and 4 worsen because the values of t_u and t_d increase significantly, though the formulae remain the same. However, the number of messages used reduces to $2k_o+4k_c+2k_n+2k_i$. 2) If v_0 rules for all the switches are in the SRT, the parameter values reduce as t_d 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) 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 or applications must use only short connections. 6)For both PPC and PFC (with SRT), the new rules get installed fairly quickly and the old rules are removed as soon as is feasible.

VI. RELATED WORK

Detection of the last packet: [23] proposes a method that reduces rule space in switches. zUpdate [24] and SWAN [5] ensure that updates do not cause congestion. [25] provides a dependency table that denotes at which switch a rule must be modified to guarantee a consistency property, before it uses a new rule. [26] proposes a method of creating a dependency graph between rules such that their toplogical ordering is a safe update. Dionysus [15] improves the update time. [27] provides a mechanism to enforce customizable consistency properties. None of these describes how and when to delete old rules. [28] 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 the method is different from a two-phase update. No paper addresses recoverability.

Per-flow consistency: [1] proposes associating a timeout with the old rule, but the rule may never timeout if it is coarse. The servers can send a list of active sockets or indicate the end of a flow to the controller but this involves changes to servers. Devoflow [19] proposes a switch that can support a "clone" command and usage of this to support per-packet consistency is examined in [1].[18] provides two solutions for updating (only) a load balancer in a per-connection consistent manner.

Handling Faults: There is existing work to provide an abstraction to develop programs that implement fault tolerance in the network [8], algorithms to implement local fast failover[9], [10], re-design of the controller to reduce fate-sharing between apps and the controller [11] etc. but none on failure during an update or recoverability.

VII. CONCLUSIONS

The paper identified areas where the basic update algorithm for SDNs is under-specified and described enhancements for update algorithms for PPC and PFC, 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 [29], 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. The latter method is not incorporated in the algorithms for ease of exposition.

Updates in a distributed system: How can concurrent updates from a controller be executed on a network, ensuring recoverability? What are the conditions under which updates can create conflicts and how can they be resolved? Also, handling system and communication failures need to be further specified.

Optimizations: The update algorithms above can be further optimized both in terms of speed of update and occupancy of TCAM. There can be improvements in the algorithms to move rules into and out of TCAMs into the SRT. Per-flow consistent updates can be further optimized to use lesser number of microrules.

Nature of updates: Policy updates and routing updates have been treated the same way in the above discussion. Since policy updates are large, how must they be handled? Virtualisation, middleboxes and live VM migration will have impacts on maintaining consistency properties during updates and need to be further explored.

REFERENCES

[1] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM 2012*. ACM, 2012, pp. 323–334.

[2] Open Networking Foundation, "Openflow switch specification version 1.5.0," 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.pdf

[3] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies, 2011*. ACM, 2011, p. 8.

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

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

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

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, 2013*. ACM, 2013, pp. 109–114.

[9] N. Beheshti and Y. Zhang, "Fast failover for control traffic in software-defined networks," in *Global Communications Conference (GLOBECOM) 2012*. IEEE, 2012, pp. 2665–2670.

[10] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms," in *Proceedings of the third workshop on Hot topics in software defined networking, 2014*. ACM, 2014, pp. 121–126.

[11] B. Chandrasekaran and T. Benson, "Tolerating SDN application failures with LegoSDN," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks, 2014*. ACM, 2014, p. 22.

[12] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[13] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *INFOCOM 2013, Proceedings IEEE*. IEEE, 2013, pp. 2130–2138.

[14] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Proceedings of the third workshop on Hot Topics in Software Defined Networking 2014*. ACM, 2014, pp. 175–180.

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

[16] O. Rottenstreich, P. Li, I. Horev, I. Keslassy, and S. Kalyanaraman, "The switch reordering contagion: Preventing a few late packets from ruining the whole party," *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1262–1276, 2014.

[17] M. Laor and L. Gendel, "The effect of packet reordering in a backbone link on application throughput," *IEEE Network*, vol. 16, no. 5, pp. 28–36, 2002.

[18] R. Wang, D. Butnariu, and J. Rexford, "Openflow-based server load balancing gone wild," in *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE'11*. USENIX Association, 2011.

[19] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *SIGCOMM 2011*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[20] J. Postel, "Transmission Control Protocol," RFC 793 (Standard), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[21] R. Braden, *RFC 1122 Requirements for Internet Hosts - Communication Layers*, Internet Engineering Task Force, October 1989. [Online]. Available: http://tools.ietf.org/html/rfc1122

[22] G. Urvoy-Keller, "On the stationarity of TCP bulk data transfers," in *Passive and Active Network Measurement*. Springer, 2005, pp. 27–40.

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

[24] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating data center networks with zero loss," *SIGCOMM 2013*, vol. 43, no. 4, pp. 411–422, 2013.

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

[26] Y. Yuan, F. Ivančić, C. Lumezanu, S. Zhang, and A. Gupta, "Generating consistent updates for software-defined network configurations," in *Proceedings of the third workshop on Hot topics in Software Defined Networking, 2014*. ACM, 2014, pp. 221–222.

[27] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *NSDI, 2015*. USENIX Association, May 2015, pp. 73–85.

[28] R. McGeer, "A safe, efficient update protocol for openflow networks," in *Proceedings of the first workshop on Hot topics in Software Defined Networks, 2012*. ACM, 2012, pp. 61–66.

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