CURE: Consistent Update with Redundancy Reduction in SDN

Ilora Maity, Student Member, IEEE, Ayan Mondal, Student Member, IEEE,
Sudip Misra, Senior Member, IEEE, and Chittaranjan Mandal, Senior Member, IEEE *

Abstract

In this paper, we address the issue of rule duplication during network updates in Software Defined Networking (SDN). In SDN, network update involves the controller in sending update packets to desired set of switches, where the update rules are installed. To ensure update consistency, old flow-rules are stored until the total update procedure is complete. Higher consumption of TCAMs during update increases the cost of network update and decreases the scalability of SDN. In this work, we propose an approach for consistent update with redundancy reduction, named CURE, that reduces TCAM usage during update. CURE prioritizes switches according to their usage pattern and schedules updates based on priority zones. The proposed approach guarantees that highly loaded switches are updated first. CURE also maintains packet-level consistency by implementing a multilevel queueing approach. In this framework, each switch in the current update region stores the incoming packets in individual device queues until the switch completes update. Therefore, after the initiation of an update, packets are processed according to new rules only. The results of performance evaluation depict that the average rule space utilization during update using CURE is 29.954% less than using the two-phase update proposed in the existing literature.

Index terms— SDN, Network Update, OpenFlow, TCAM, Multiclass Classification, Queueing Theory

*The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, India (Email: imaity@iitkgp.ac.in; ayanmondal@iitkgp.ac.in; smisra@sit.iitkgp.ernet.in; chitta@iitkgp.ac.in).

1 Introduction

In traditional networks, each network device or switch includes both a control plane and a data plane. The control plane determines the forwarding rules for the incoming packets. The data plane stores the forwarding table, which is a collection of the forwarding rules determined by the control plane. Therefore, traditional networks are complex and resistant to changes. Software Defined Networking (SDN) is a new networking paradigm, which separates the control plane from the data plane to offer network services dynamically [1]. In SDN, a single controller or a cluster of controllers [2] determines the forwarding rules and installs them in the switches. The switches store the forwarding tables and forward the incoming packets based on matched table entries.

Similar to traditional networks, updates in SDN occur frequently. Major reasons for network update are — (1) optimization of flow-table, (2) flow swapping after arrival of new flow, (3) traffic monitoring, (4) maintaining state of shut-down switches, (5) expiration of flow-rules due to timeouts, and (6) switch or link failure[3]. Update in traditional network involves changing the configuration of each switch separately [4]. On the other hand, SDN update is triggered by the controller, which generates forwarding rules for new network configuration and installs those rules to the required switches. Additionally, the controller performs garbage collection by deleting the old rules [5].

However, according to the state-of-the-art, OpenFlow [6] switches have Ternary Content Addressable Memory (TCAM) to store forwarding rules. TCAMs, which are high-speed memories, can match the rules in parallel in $O(1)$ time. However, TCAMs consume

For personal use only

large amount of power [7] and occupy large footprints [8]. These constraints restrict the storage capacity of TCAMs [9]. Moreover, multiple TCAM entries can be generated for a single forwarding rule [10]. Therefore, the update of a single forwarding rule causes an update of a single or multiple TCAM entries.

In this scenario, the maximum number of flow-rules stored in a switch during update is required to be managed optimally considering the restricted storage capacity of TCAMs. However, existing SDN update approaches store old rules along with new rules until all switches are updated. Hence, for the worst case scenario, 50% of the storage space needs to be empty before starting the network update. Therefore, the cost of storing redundant rules decreases the scalability of the overall network. Furthermore, the number of large-scale applications are increasing [11]. The continuous flow of high-volume data generates high number of update entries for each flow-table. Therefore, provisioning storage space for redundant TCAM entries can be a bottleneck.

In this work, we propose a SDN update policy without storing the redundant rules. Consequently, the maximum number of flow-rules present in the network during update is reduced. Updates in SDN switches are scheduled in an optimized manner, so that the high priority switches are updated first. We classify the switches based on the frequency of matched rules. Packet-level consistency is also ensured by employing a packet-queueing mechanism. In brief, the primary contributions of our work are listed below.

- Initially, we design a priority-based algorithm for scheduling updates to SDN switches.
- We propose a packet queueing mechanism for maintaining the consistency of incoming packets during update.
- Further, we design a packet processing algorithm that process the queued packets consistently.
- We compare our approach with two-phase update [12], timed two-phase update [5], and buffered update [13] to highlight the benefits of the proposed scheme.

The remainder of this paper proceeds as follows. Section 2 discusses the existing approaches for SDN update. In Section 3, we define the network model and describe CURE, the proposed scheme, in detail. Section 4 depicts the experimental results and comparative studies with other existing approaches. Finally, Section 5 concludes the work.

2 Related Work

Existing literature in this field are categorized in four parts including *ordered*, *incremental*, *timed*, and *buffered updates*.

In case of ordered update, the controller partitions the total update procedure into multiple stages [14], [15], [16]. It waits for the completion of each stage, prior to starting the next stage. The last stage is garbage collection, where older rules are deleted. Francois *et al.* [14] proposed an ordered update scheme that ensures packet-level consistency by preventing the formation of loops. However, this approach requires a modification of network protocols as well as of the forwarding devices. Bera *et al.* [15] proposed a prediction-based mobility-aware update mechanism for Software-Defined IoT which inserts new rule at the next access device (AD), and performs garbage collection at the current AD. Clad *et al.* [16] generated an optimized sequence for updating the weights of links. The ordered update policy encounters service latency as each phase is restricted by completion of previous phase.

In the incremental update approach, the network is updated in multiple phases or rounds, where each round updates a portion of flow-rules or a subset of switches. Reitblatt *et al.* [12] proposed a two-phase update approach where the internal and ingress switches are updated in phase 1 and phase 2, respectively. Updated ingress switches attach new version tags to the incoming packets. The incoming packets are processed by either old or new rules (not both) based on the version tag. Older rules are deleted after all packets with old version tag are processed. This method increases the load on the ingress switches, as they have to modify the incoming packets. Moreover, memory overhead is incurred for storing old rules. In

For personal use only

another work, Canini *et al.* [17] discussed an incremental update approach, which is similar to database transactions, where either all switches are updated or none are. Therefore, the ordered and incremental update approaches require extra flow-table space for accommodating duplicate rules. Moreover, the controller is involved until all switches complete update.

To reduce this overhead, Mizrahi *et al.* [5] proposed an extension of OpenFlow protocol by scheduling the update phases at particular time instants for both ordered and incremental updates. This approach preserves packet-level consistency by avoiding conflicts in updates. This technique reduces the duration required to store older rules in SDN switches. However, synchronizing updates to all the switches encounters computational complexity and depends on the characteristics of particular forwarding devices.

Buffered update approach [13] identifies the incoming packets, whose routes are going to be affected by the upcoming update, and redirects the packets to the controller by installing an intermediate ruleset to all switches. These packets are buffered in the control plane until the switches are updated. After the completion of update, the packets are processed according to the new rules. The major limitation of this approach is that it overloads the controller and increases the service latency. Further, additional overhead is incurred due to the installation of the intermediate ruleset.

In this paper, we propose an update procedure where switches are updated according to their workload. In addition, we process packets consistently by maintaining a multilevel queueing approach. The proposed scheme is different from the existing ordered and incremental approaches, as we do not store old rules, once the new rules are installed. Our approach is also different from the buffered update approach, where the entire network is updated at a time and all the affected packets are buffered at the controller until the update completes. In CURE, switches are updated incrementally based on their usage pattern. Additionally, switch buffers are prioritized over the controller buffer to reduce the response time and controller load.

3 CURE: The Proposed Scheme

In this section, we describe the network model considered for our proposed scheme, CURE. We also discuss the approach for implementing a redundancy-free consistent update of SDN.

3.1 Network Model

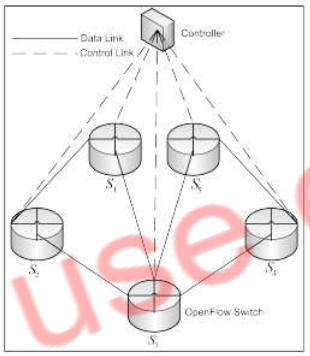


Figure 1: SDN architecture

We model the network as a graph $\mathcal{G} = (\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ is the set of nodes, and $\mathcal{L}$ is the set of links between the nodes. The set $\mathcal{N}$ is expressed mathematically as:

$$\mathcal{N} = \mathcal{C} \cup \mathcal{S}, \tag{1}$$

where $\mathcal{C}$ is the set of controllers, and $\mathcal{S}$ is the set of OpenFlow switches. Figure 1 shows the proposed network model. The upper bound of the number of flow-rules which can be stored in an OpenFlow switch $\mathcal{S}_i$ is denoted as U_i. Each switch $\mathcal{S}_j$ has an associated device queue denoted as Q^j. The set of links $\mathcal{L}$ is defined as:

$$\mathcal{L} = \mathcal{L}_{cc} \cup \mathcal{L}_{cs} \cup \mathcal{L}_{ss}, \tag{2}$$

where $\mathcal{L}_{cc}$ is the set of links between the controllers, $\mathcal{L}_{cs}$ is the set of control links between the controllers and the OpenFlow switches, and $\mathcal{L}_{ss}$ is the set of data links between the OpenFlow switches for packet forwarding.

For personal use only

For simplicity, we assume a centralized control plane containing a single controller C. Hence, $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_{|\mathcal{N}|-1}\}$, $\mathcal{L}_{cc} = \phi$ and the number of links in $\mathcal{L}_{cs}$ is $|\mathcal{S}| = |\mathcal{N}| - 1$. Each link $\mathcal{L}_i$ has an associate link capacity c_i.

Each switch stores the flow-rules in one or multiple flow-tables [6]. A flow-rule R_i^j in $\mathcal{S}_j$ is a ternary string denoted by a tuple $< Pr_i^j, M_i^j, A_i^j >$, where Pr_i^j denotes rule priority, M_i^j denotes the set of match fields, and A_i^j denotes the set of action values. Each flow-rule also contains a set of counters for storing the rule statistics, timeout value, cookie, and flags [6]. If an incoming packet matches with multiple rules, then the rule with the highest priority value is selected and the corresponding action is taken.

***Definition* 1** (State of a Switch)**.** *The state of $\mathcal{S}_j$ at time t is defined by:*

$$\Lambda_j(t) = \{R^j(t), \mathcal{L}_{cs}^j(t), \mathcal{L}_{ss}^j(t), \tau^j(t)\}, \tag{3}$$

where $R^j(t)$ is the set of flow-rules of $\mathcal{S}_j$ at time t, $\mathcal{L}_{cs}^j(t) \in \mathcal{L}_{cs}$ is the set of control links involving $\mathcal{S}_j$ at time t, $\mathcal{L}_{ss}^j(t) \in \mathcal{L}_{ss}$ is the set of data links involving $\mathcal{S}_j$ at time t, and τ^j is the last update time of $\mathcal{S}_j$ at time t.

***Definition* 2** (Network Configuration)**.** *Network configuration at time t is defined by:*

$$\Gamma(t) = \bigcup_{j=1}^{|\mathcal{S}|} \Lambda_j(t) \tag{4}$$

***Definition* 3** (Network Update)**.** *Network update in SDN is migration from one network configuration Γ to another configuration $\Gamma^{'}$ such that,*

$$\Gamma(t_i) \neq \Gamma^{'}(t_j), \text{ where } t_i \neq t_j \tag{5}$$

Major objective for this work is to minimize the maximum TCAM usage during update without congesting the links and to maintain packet-level consistency. For a network update from $\Gamma(t_i)$ to $\Gamma^{'}(t_j)$, the optimization problem is formulated as follows:

$$\min \max \sum_{t=t_i}^{t_j} \sum_{j=1}^{|\mathcal{S}|} |R^j(t)| \tag{6}$$

Equation (6) minimizes the maximum number of rules in the whole network, subject to the following constraints:

$$|R^j(t)| \leqslant U_j, \forall \mathcal{S}_j \in \mathcal{S} \tag{7}$$

Equation (7) expresses the switch capacity constraint for storing flow-rules.

$$\begin{aligned} &M_r^j = M_s^j \text{ and } A_r^j = A_s^j, \forall \Lambda_j(t_i) = \{R^j(t_i), \mathcal{L}_{cs}^j(t_i), \mathcal{L}_{ss}^j(t_i), \\ &\tau^j(t_i)\}, \forall \Lambda_j(t_k) = \{R^j(t_k), \mathcal{L}_{cs}^j(t_k), \mathcal{L}_{ss}^j(t_k), \tau^j(t_k)\}, t_i < t_k, \\ &R_r^j \in R^j(t_i), R_s^j \in R^j(t_k), \text{Duration}(R_r^j) < \text{Duration}(R_s^j), \end{aligned} \tag{8}$$

where $\text{Duration}(R_i^j)$ is a counter [6], which denotes the elapsed time after installation of the flow-rule R_i^j. Equation (8) prohibits the storage of older and newer versions of a rule in a switch simultaneously.

3.2 Redundancy-free Consistent Update

In this section, we describe the proposed scheme, CURE, for SDN update. Based on workload, we first classify the to-be-updated switches into three priority regions, namely high, medium, and low. Thereafter, we design an algorithm for scheduling updates among the switches of different priority regions. Next, we propose a packet queueing mechanism to maintain packet-level consistency during update. Finally, we propose an algorithm for processing the queued packets.

3.2.1 Switch Classification

Each OpenFlow switch flow-table maintains a counter field, which records the details of the matching packets. Based on the counter value, we build a training data set. Therefore, we employ the existing One-Vs-All (OvA) multiclass classification algorithm [18] to classify the to-be-updated switches into three priority zones — low, medium, and high. This classification depends on the network topology, packet arrival rate, and existing flows in the network. If the traffic load in all the switches are approximately equal, CURE uses the number of active entries in each flow-table as a metric for the classification. The

For personal use only

number of active entries in each flow-table is also stored as a counter field [6].

3.2.2 Rule Update

Algorithm 1 schedules the update based on the priority zones. Before starting the update, C sends *UPDATE* signal at time T_0 to mark the set of switches which are to be updated. Therefore, the network configuration before update is $\Gamma(T_0)$. C waits for δ time interval before sending the first update packet. Heavily loaded switches are updated first at time $T_{high} > T_0$. Next, medium priority switches are updated at time $T_{medium} > T_{high}$. Finally, low priority switches are updated at time $T_{low} > T_{medium}$. During the update procedure at a switch, the set of new rules is installed first and the older rules are deleted thereafter. In other words, garbage collection at each switch is performed right after the completion of update at the switch. Therefore, this algorithm complies with the constraints stated in Equations (7) and (8). When every switch is updated, the network reaches a configuration $\Gamma(T_{complete})$ at time $T_{complete} > T_{low}$.

***Definition* 4** (Old Packet)**.** *After T_0, a packet is marked old, if it is processed by a switch, which is yet to be updated.*

***Definition* 5** (New Packet)**.** *After T_0, a packet is marked new, if it is processed by a updated switch.*

Let P_{old} and P_{new} denote the sets of *old* and *new* packets, respectively. When C selects a priority region for update, all $p \in P_{old}$ in that region are processed before starting the installation of new rules. This ensures that a packet, which is already processed by an old rule, is processed by old rules only. If an *old* packet reaches an updated switch, the packet is sent to C for further decision. Similarly, if a *new* packet reaches a to-be-updated switch, which is not in the current update region, the packet is sent to C for further decision.

***Definition* 6** (Update Duration)**.** *Update duration is the time interval between the dispatch of the first update message by C, and update completion of the last switch, including garbage collection.*

Algorithm 1 Rule Update Algorithm

INPUTS: $\mathcal{S}^{low}$, $\mathcal{S}^{medium}$, $\mathcal{S}^{high}$ ▷ Sets of low, medium, and high priority switches
OUTPUT: $\mathcal{S}''$ ▷ Set of updated switches
PROCEDURE:
function UPDATESWITCHES($\mathcal{S}^{Reg}$)
  for all $\mathcal{S}_j \in \mathcal{S}^{Reg}$ **do**
    Process P_{old} ▷ Process the old packets
    Insert R'^j ▷ Add the set of new rules
    Remove R^j ▷ Remove the set of old rules
    $\mathcal{S}'' \leftarrow \mathcal{S}'' \cup \{\mathcal{S}_j\}$
  end for
end function
$\mathcal{S}'' \leftarrow \emptyset$
for all $\mathcal{S}_j \in \mathcal{S}^{low} \cup \mathcal{S}^{medium} \cup \mathcal{S}^{high}$ **do**
  $SIGNAL(\mathcal{S}_j, UPDATE)$ ▷ C sends update signal
  $WAIT(\delta\ ms)$ ▷ C waits for δ ms
end for
UPDATESWITCHES($\mathcal{S}^{high}$)
UPDATESWITCHES($\mathcal{S}^{medium}$)
UPDATESWITCHES($\mathcal{S}^{low}$)
return $\mathcal{S}''$

***Definition* 7** (Inconsistent Packet)**.** *A packet $p \in P_{old}$ is termed inconsistent, if it reaches an updated switch. A packet $p \in P_{new}$ is termed inconsistent if it reaches a switch, which is not updated and is not in the current update region.*

3.2.3 Packet Queueing

Algorithm 2 Packet Queueing Algorithm

INPUTS:
$\mathcal{S}''$ ▷ Set of updated switches
$\mathcal{S}_j$ ▷ A switch in current update region
P^j ▷ Set of incoming packets at $\mathcal{S}_j$
OUTPUT: P_{count}
PROCEDURE:
function STOREPACKET(p, $\mathcal{S}_j$, j)
  if Q^j is not full **then**
    Store p in Q^j
  else if $\mathcal{S}_{neighbor} \neq NULL$ **then** ▷ Neighbor of $\mathcal{S}_j$
    Store p in $Q^{neighbor}$
    $P_{count} \leftarrow P_{count} + 1$
  else
    Buffer p at C
    $P_{count} \leftarrow P_{count} + 1$
  end if
end function
for all $p \in P^j$ **do**
  if $\mathcal{S}_j \in \mathcal{S}''$ **then**
    Process p ▷ Standard packet processing
  else
    STOREPACKET(p, $\mathcal{S}_j$, j) ▷ Insert p to Q_j
  end if
end for
return P_{count}

Algorithm 2 depicts a queueing mechanism for the

For personal use only

consistent processing of incoming packets during an ongoing update procedure. The packet queueing algorithm (PQA) is triggered for each to-be-updated switch $\mathcal{S}_j \in \mathcal{S}$ in the present update region after C starts update in that region. If $\mathcal{S}_j$ has received an *UPDATE* signal recently, PQA checks statistics at C to verify whether the switch is already updated. PQA stores the packet if the update process is incomplete in the corresponding switch.

Packets are stored in Q^j until it is full. Thereafter, the packets are redirected to the least priority switch $\mathcal{S}_{neighbor}$ which belongs to a lower priority region and has free buffer space within one-hop neighbors of $\mathcal{S}_j$. In this scenario, a switch-identifier flag is added to the packet header specifying the switch id where the packet arrived initially. The packets are buffered at C when no such neighbor exists. For each switch, we maintain a counter P_{count} that counts the number of packets stored outside of the switch's own buffer.

3.2.4 Packet Processing

Algorithm 3 Packet Processing Algorithm

INPUT: $\mathcal{S}_u$
OUTPUT: P'' ▷ Set of packets in *secondary buffer*
PROCEDURE:
1: **if** $|Q^u| == K$ **then**
2: $P'' \leftarrow \emptyset$
3: Process first K packets in Q^u
4: **for all** $p \in P^j$ stored at $\mathcal{S}_{neighbor}$ **do**
5: Copy p to *secondary buffer* of Q^u
6: $P'' \leftarrow P'' \cup \{p\}$
7: **end for**
8: **for all** $p \in P^j$ buffered at the C **do**
9: Copy p to *secondary buffer* of Q^u
10: $P'' \leftarrow P'' \cup \{p\}$
11: **end for**
12: Process packets in *secondary buffer*
13: Merge *secondary buffer* with Q^u
14: **end if**
15: Process packets in Q^u
16: **return** P''

After the completion of update, each switch $\mathcal{S}_u$ triggers C by informing that it is ready for processing packets. Algorithm 3 describes the procedure of processing the waiting-packets. If Q^u is full and the buffer size is K, the packet processing algorithm processes the first K packets waiting at Q^u. Then a portion of Q^u is reserved for storing the waiting packets with matching switch-identifier flag in the one-hop neighbor. We name this buffer space as *secondary buffer*. The size of *secondary buffer* is determined from the available counter value. Packets waiting in $Q^{neighbor}$ and/or C are shifted to the *secondary buffer*. After processing these packets, the *secondary buffer* space is merged with the switch's original buffer before processing the new ones.

3.3 Queueing Model

Assuming a Markovian server per switch, the queue of each switch $\mathcal{S}_j$ is modeled as a $M/M/1/K/\alpha$ queueing system [19, 20] where the incoming packets follow Poisson's distribution and those packets are processed by $\mathcal{S}_j$ with an exponentially distributed service time. Let, $\frac{1}{\mu_j}$ and $\frac{1}{\lambda_j}$ denote the mean service time and mean inter-arrival time at $\mathcal{S}_j$, respectively. We also consider that each switch has a finite queue length K. Figure 2 depicts the queueing model for SDN.

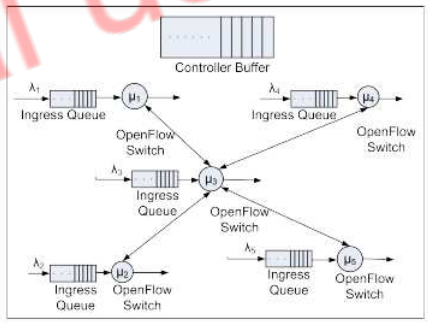


Figure 2: SDN Queueing Model

Figure 3 shows the state-transition-rate diagram of our proposed queueing model for a single switch. The average packet arrival rate and average service rate for the switch be λ and μ, respectively. Therefore, the traffic intensity is $\rho = \frac{\lambda}{\mu}$. The switch is in region $r \in \{high \cup medium \cup low\}$. Initially, C sends update signal to the switch. As depicted in Figure 3, we consider that the update procedure of an OpenFlow switch consists of three stages. In the first stage, the switch receives update signal and region r has not started update. The second stage begins when r starts update. The final stage begins when

For personal use only

the switch completes update. The switch continues processing until the second stage begins. During the second stage, the switch queues the received packets, unless it completes an update. Therefore, the service rate for this stage is $\mu = 0$. If the switch queue is full, the packets are buffered at the neighbor queue or at the controller buffer according to Algorithm 2. Hence, the increased traffic intensity of a neighbor switch $\mathcal{S}_a$ for buffering packets of the current switch is given by:

$$\rho_a^{over} = \left(\frac{\lambda + \lambda_a}{\mu_a}\right) \tag{9}$$

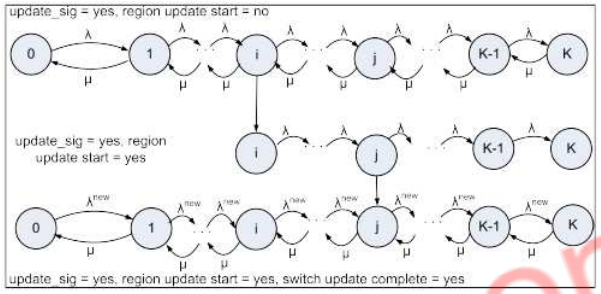


Figure 3: State-Transition-Rate Diagram of CURE for a Switch

During the final stage, the switch processes the packets from the neighbor buffer as well as its own buffer, as mentioned in Algorithm 3. Therefore, the new packet arrival rate is $\lambda^{new} = \lambda + \lambda^{neighbor}$, where $\lambda^{neighbor}$ is the rate at which the packets arrive at the current switch from the buffer of the neighbor switch. The traffic intensity in this scenario is $\rho^{new} = \frac{\lambda^{new}}{\mu}$. After the switch processes all the packets stored in neighbor queue, $\lambda^{neighbor} = 0$ and $\lambda^{new} = \lambda$.

The probabilities that the switch has a packets in the three stages are denoted by P_a^1, P_a^2, and P_a^3, respectively. However, as per our assumption, the processing of packets at a switch is a Poisson process. Therefore, according to queueing theory, the steady state probability that the switch has i packets in the first stage is given by:

$$P_i^1 = \rho^i P_0^1 \tag{10}$$

We consider the scenario that region r starts update when the switch has i packets queued and completes update when it has j packets queued. We know, $P_i^2 = P_i^1$. During the second stage, packets are added to the queue at the rate of λ and no processing is performed. Hence, we get:

$$P_i^2 = P_{i+1}^2 = \ldots = P_K^2 = P_i^1 \tag{11}$$

Similarly, from Equation (11), we get:

$$P_j^3 = P_j^2 = P_i^1 \tag{12}$$

The probability P_j^3 is also expressed as:

$$P_j^3 = (\rho^{new})^j P_0^3 \tag{13}$$

From Equations (10), (12), and (13) we have:

$$P_0^3 = \frac{\rho^i}{(\rho^{new})^j} P_0^1 \tag{14}$$

According to queueing theory for finite queue length, at steady state:

$$P_0^1 = \frac{1-\rho}{1-\rho^{K+1}},\ P_0^3 = \frac{1-\rho^{new}}{1-(\rho^{new})^{K+1}} \tag{15}$$

Hence, from Equations (14) and (15), the probability P_0^1 is defined as:

$$P_0^1 = \frac{(\rho^{new})^j(1-\rho^{new})}{\rho^i(1-(\rho^{new})^{K+1})} \tag{16}$$

Let L and L^{new} be the expected number of packets in the switch before starting update and after the completion of update, respectively. Mathematically,

$$L = \frac{\rho(1 + K\rho^{K+1} - (K+1)\rho^K)}{(1-\rho)(1-\rho^{K+1})} \tag{17}$$

$$L^{new} = \frac{\rho^{new}(1 + K(\rho^{new})^{K+1} - (K+1)(\rho^{new})^K)}{(1-\rho^{new})(1-(\rho^{new})^{K+1})} \tag{18}$$

For personal use only

Let W and W^{new} be the mean waiting time at the switch before starting update and after the completion of update, respectively. Therefore, the increase in mean waiting time at the OpenFlow switch due to update is given by:

$$W^{new} - W = \left(\frac{L^{new}}{\lambda^{new}} - \frac{L}{\lambda}\right) = \frac{1}{\mu}\left(-\frac{1+K\rho^{K+1}-(K+1)\rho^{K}}{(1-\rho)(1-\rho^{K+1})} + \frac{1+K(\rho^{new})^{K+1}-(K+1)(\rho^{new})^{K}}{(1-\rho^{new})(1-(\rho^{new})^{K+1})}\right) \quad (19)$$

The value $W^{new} - W$ provides an estimate of the latency incurred due to rule update. After the switch completes processing the packets stored in the neighbor queue, $W^{new} - W$ becomes zero, eventually.

4 Performance Evaluation

In this section, we evaluate the performance of CURE in terms of the following metrics: (a) update duration, (b) average rule space utilization, (c) average packet waiting time, and (d) inconsistent packet count. To evaluate the performance, we implemented a discrete event simulator in MATLAB and performed two experiments. In the first experiment, we measured the update duration and the average rule space utilization, while varying the number of switches in a leaf-spine topology with $\frac{2N}{3}$ leaf (ingress) switches and $\frac{N}{3}$ spine switches (e.g., [5]). In the second experiment, we simulated three network topologies available in the Internet Topology Zoo [21], namely Sprint, NetRail, and Compuserve. We run five test flows in each of these topologies to compute the performance with respect to the average packet waiting time and inconsistent packet count.

4.1 Simulation Parameters

Table I depicts the simulation parameters. We implement the leaf-spine topology by varying the total number of switches from 6 to 48. The maximum number of flow entries in a switch is fixed to 8000 [22]. We consider that the upper bounds on controller-to-switch delay, end-to-end network delay, and the time interval between generation of two consecutive update messages are 4.865 ms, 0.262 ms, and 5.240 ms, respectively [5]. The average packet arrival rate, average packet service rate, and average queue size per switch are $0.005 - 0.025$ million packets per second (mpps), 0.030 mpps [23], and 0.073 million packets, respectively. We consider that the flow-table lookup time for each packet is 33.333 μsec [23].

4.2 Result and Discussion

4.2.1 Update Duration

The update duration is the time interval between the dispatch of the first update message by the controller, and the update completion of the last switch. Garbage collection, i.e., the removal of old rules is included in the update duration, as defined in Definition 6.

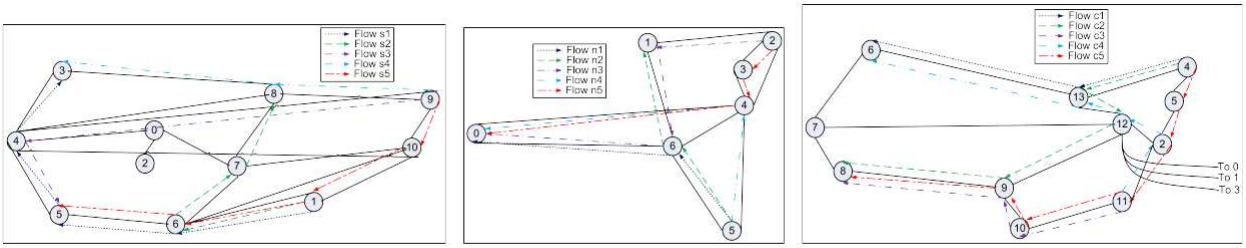
Figure 5 depicts the update duration for two-phase update [12], timed two-phase update [5], Buffered Update [13], and CURE in a leaf-spine topology. The two-phase update approach (both untimed and timed) updates the spine switches in phase 1, the leaf switches in phase 2 and performs garbage collection after completion of phase 2. From Figure 5, we can see that the update duration for timed two-phase update is 27.919% less than that of two-phase update. The update duration for CURE is 37.563% less than that of two-phase update. The update duration is almost similar for timed two-phase update and CURE. Duration for buffered update is high due to the overhead for the installation of intermediate rules. From Figure 5, we yield that the update duration for CURE is short as it does not have a separate garbage collection phase.

4.2.2 Average Rule Space Utilization

We calculate the average rule space utilization as the percentage of rule space used during different stages of update by N switches in the leaf-spine topology.

Figure 6 shows the rule space utilization percentage for two-phase update [12], timed two-phase update [5], Buffered Update [13], and CURE. CURE and buffered update utilize similar amount of rule space, as they both do not store redundant rules. Whereas, rule space utilization is almost similar for

For personal use only



(a) Sprint Topology (b) NetRail Topology (c) Compuserve Topology

Figure 4: Test Flows in Sprint, NetRail, and Compuserve Topology

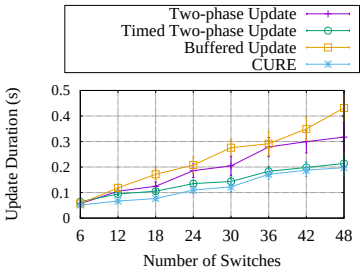


Figure 5: Update Duration

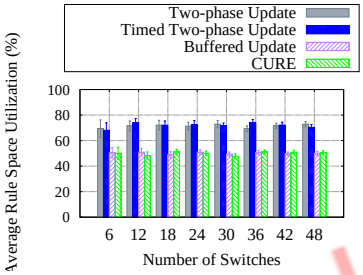


Figure 6: Average Rule Space Utilization

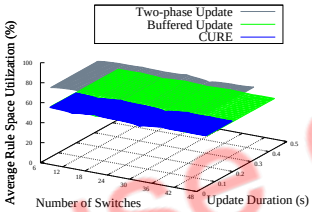


Figure 7: Update Duration and Average Rule Space Utilization

TABLE I: Simulation parameters

Parameter	Value
Number of switches in the leaf-spine topology	6 – 48
Rule space size in a switch	8000 flow entries [22]
Upper bound on controller-to-switch delay	4.865 ms [5]
Upper bound on end-to-end network delay	0.262 ms [5]
Upper bound on time interval between dispatch of two consecutive update messages	5.240 ms [5]
Average packet arrival rate per switch	0.005 – 0.025 mpps
Average packet service rate per switch	0.030 mpps [24]
Average queue size per switch	0.073 million packets
Flow-table lookup time	33.333 μsec [24]

the two-phase update and the timed two-phase update, as they both require to store both old and new rules until the start of the garbage collection phase. Average rule space requirement for CURE is 29.954% and 30.348% less than that of the two-phase update and timed two-phase update, respectively. As shown in Figure 6, we synthesize that the average rule space utilization is short in CURE, as storage of both versions of rules, simultaneously, is not required.

Figure 7 portrays the relation between the number of switches, average rule space utilization, and update duration for two-phase update, buffered update, and CURE. We see that CURE outperforms the others, considering both performance metrics — average rule space utilization and update duration.

4.2.3 Average Packet Waiting Time

For each of the three topologies — Sprint, NetRail, and Compuserve, we simulate five test flows, and calculate the average waiting time for the incoming packets that are either waiting in the switch queues or are in process. Figure 4 depicts the topologies and the test flows. We estimate the delay of each link based on the distance between the corresponding nodes. Similar to Ref. [5], we assume 5 microsecond delay per kilometer.

Figure 8 depicts the average packet waiting time for different packet arrival rate for each of the test flows in each of the topologies. The average packet

For personal use only

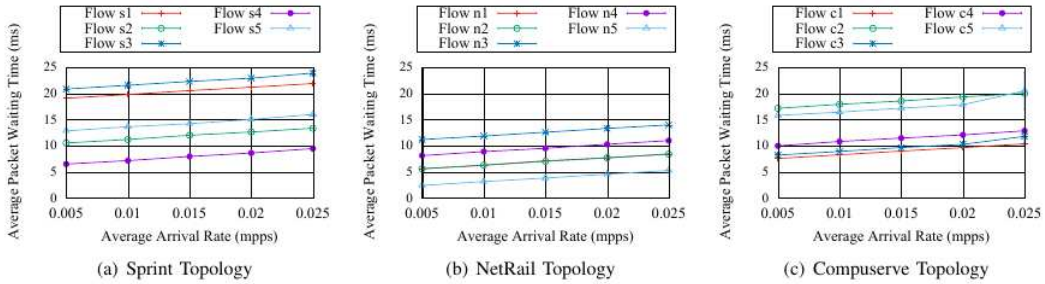


Figure 8: Average Packet Waiting Time

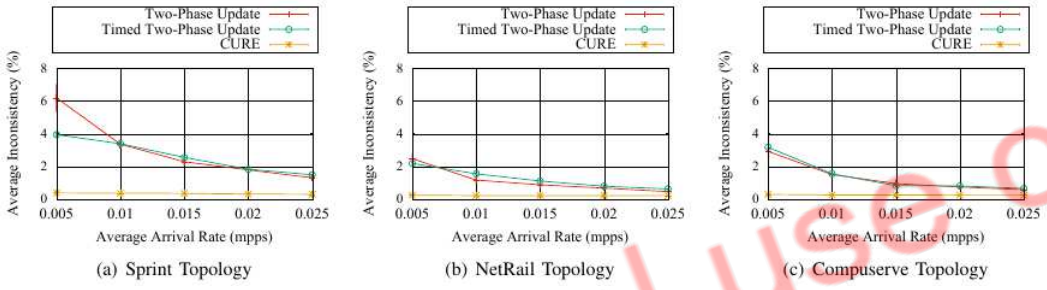


Figure 9: Average Packet Inconsistency

queue size is 0.073 million packets. Average packet waiting time increases with increasing packet arrival rate.

4.2.4 Inconsistent Packet Count

We measure inconsistency as a percentage of inconsistent packets in the system. Inconsistent packets are identified based on Definition 7.

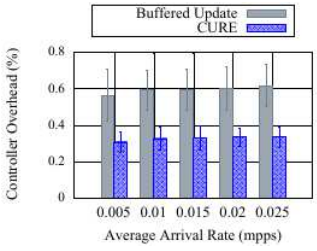


Figure 10: Controller Overhead in Sprint Topology

Figure 9 compares inconsistency count in CURE with two-phase update and timed two-phase update [5] for different average packet arrival rates. We simulate test flows $s1$, $n1$, and $c1$ in topologies Sprint, NetRail, and Compuserve, respectively. Average queue size per switch is 0.073 million packets. In the two-phase update approaches (both untimed and timed), inconsistency count decreases with increasing packet arrival rate. In two-phase update, the average inconsistency counts for Sprint, NetRail, and Compuserve are 2.976%, 1.118%, and 1.327%, respectively. In timed two-phase update, the average inconsistency counts for Sprint, NetRail, and Compuserve are 2.629%, 1.237%, and 1.389%, respectively. However, average inconsistency count for CURE is similar for different packet arrival rates. The average inconsistency count for Sprint, NetRail, and Compuserve is 0.322%, 0.205%, and 0.240%, respectively. Therefore, we yield that in CURE an initial percentage of incoming packets become inconsistent due to the ongoing network update and inconsistency count reduces as time elapses after completion of the update.

For personal use only

4.2.5 Controller Overhead

Controller overhead is calculated as the percentage of packets sent to controller during an ongoing update. In Sprint topology, CURE incurs 0.31% controller overhead for packet arrival rate 0.005 mpps. Figure 10 depicts that the controller overhead in buffered update is 82.209% higher than that in CURE. This is because CURE redirects packets to the controller only in the absence of a neighbor switch having lower priority and free buffer space. Whereas, buffered update keeps redirecting all the affected packets to the controller until the update completes.

5 Conclusion

This work emphasizes reduction of TCAM usage during SDN update with an aim to increase scalability required for handling large-scale data. This work modifies the update scheme of OpenFlow-enabled SDN and proposes a multilevel queue-based policy for ensuring packet-level consistency. We compared our scheme with the other approaches of SDN update to evaluate its performance. Results clearly depict that CURE significantly reduces the update duration and the average rule space requirement during update approximately by 38% and 30%, respectively.

The future work will include extension of the proposed scheme in distributed SDN control plane, where multiple controllers perform network update concurrently. We will consider flow-level consistency along with packet-level consistency.

References

[1] C. Metter, M. Seufert, F. Wamser, T. Zinner, and P. Tran-Gia, "Analytical Model for SDN Signaling Traffic and Flow Table Occupancy and its Application for Various Types of Traffic," *IEEE Trans. Netw. Service Manag.*, vol. PP, no. 99, pp. 1–1, 2017.

[2] M. T. I. ul Huque, W. Si, G. Jourjon, and V. Gramoli, "Large-Scale Dynamic Controller Placement," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 1, pp. 63–76, Mar. 2017.

[3] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C. N. Chuah, Y. Ganjali, and C. Diot, "Characterization of Failures in an Operational IP Backbone Network," *IEEE/ACM Trans. Netw.*, vol. 16, no. 4, pp. 749–762, Aug. 2008.

[4] N. Ansari, G. Cheng, and N. Wang, "Routing-oriented update schEme (ROSE) for link state updating," *IEEE Trans. Commun.*, vol. 56, no. 6, pp. 948–956, Jun. 2008.

[5] T. Mizrahi, E. Saat, and Y. Moses, "Timed Consistent Network Updates in Software-Defined Networks," *IEEE/ACM Trans. Netw.*, vol. 24, no. 6, pp. 3412–3425, Dec. 2016.

[6] "OpenFlow Switch Specification (Version 1.5.1): Open Networking Foundation," Mar. 2015.

[7] S. Baeg, "Low-Power Ternary Content-Addressable Memory Design Using a Segmented Match Line," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 55, no. 6, pp. 1485–1494, Jul. 2008.

[8] T. Mishra and S. Sahni, "PETCAM—A Power Efficient TCAM Architecture for Forwarding Tables," *IEEE Trans. Comput.*, vol. 61, no. 1, pp. 3–17, Jan. 2012.

[9] P. T. Congdon, P. Mohapatra, M. Farrens, and V. Akella, "Simultaneously Reducing Latency and Power Consumption in OpenFlow Switches," *IEEE/ACM Trans. Netw.*, vol. 22, no. 3, pp. 1007–1020, Jun. 2014.

[10] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "Optimal In/Out TCAM Encodings of Ranges," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 555–568, Feb. 2016.

[11] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for Internet of Things: A Survey," *IEEE Internet Things J.*, vol. 3, no. 1, pp. 70–95, Feb. 2016.

For Personal use only

[12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," in *Proc. of ACM SIGCOMM*, New York, NY, USA, 2012, pp. 323–334.

[13] R. McGeer, "A Safe, Efficient Update Protocol for Openflow Networks," in *Proc. of HOT SDN*, New York, NY, USA, 2012, pp. 61–66.

[14] P. Francois and O. Bonaventure, "Avoiding Transient Loops During the Convergence of Link-state Routing Protocols," *IEEE/ACM Trans. Netw.*, vol. 15, no. 6, pp. 1280–1292, 2007.

[15] S. Bera, S. Misra, and M. S. Obaidat, "Mobility-Aware Flow-Table Implementation in Software-Defined IoT," in *Proc. of IEEE GLOBECOM*, Dec. 2016, pp. 1–6.

[16] F. Clad, S. Vissicchio, P. Mrindol, P. Francois, and J. J. Pansiot, "Computing Minimal Update Sequences for Graceful Router-Wide Reconfigurations," *IEEE/ACM Trans. Netw.*, vol. 23, no. 5, pp. 1373–1386, Oct. 2015.

[17] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "Software Transactional Networking: Concurrent and Consistent Policy Composition," in *Proc. of HOT SDN*. New York, NY, USA: ACM, 2013, pp. 1–6.

[18] R. Anand, K. Mehrotra, C. K. Mohan, and S. Ranka, "Efficient classification for multiclass problems using modular neural networks," *IEEE Trans. Neural Netw.*, vol. 6, no. 1, pp. 117–124, Jan. 1995.

[19] A. F. Tayel, S. I. Rabia, and Y. Abouelseoud, "An Optimized Hybrid Approach for Spectrum Handoff in Cognitive Radio Networks With Non-Identical Channels," *IEEE Trans. Commun.*, vol. 64, no. 11, pp. 4487–4496, Nov. 2016.

[20] D. Li, W. Saad, I. Guvenc, A. Mehbodniya, and F. Adachi, "Decentralized Energy Allocation for Wireless Networks With Renewable Energy Powered Base Stations," *IEEE Trans. Commun.*, vol. 63, no. 6, pp. 2126–2142, Jun. 2015.

[21] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet Topology Zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011.

[22] D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," in *Proc. of the IEEE*, vol. 103, no. 1, Jan. 2015, pp. 14–76.

[23] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for OpenFlow Switch Evaluation," in *Proc. of PAM*. Springer, 2012, pp. 85–95.

For personal use only