Timed Consistent Network Updates in Software-Defined Networks

Tal Mizrahi, Efi Saat, and Yoram Moses

***Abstract*— Network updates, such as policy and routing changes, occur frequently in software-defined networks (SDNs). Updates should be performed consistently, preventing temporary disruptions, and should require as little overhead as possible. Scalability is increasingly becoming an essential requirement in SDNs. In this paper, we propose to use time-triggered network updates to achieve consistent updates. Our proposed solution requires lower overhead than the existing update approaches, without compromising the consistency during the update. We demonstrate that accurate time enables far more scalable consistent updates in the SDN than previously available. In addition, it provides the SDN programmer with fine-grained control over the tradeoff between consistency and scalability.**

***Index Terms*— SDN, PTP, IEEE 1588, clock synchronization, management, time, consistent updates.**

I. Introduction

A. Background

TRADITIONAL network management systems are in charge of initializing the network, monitoring it, and allowing the operator to apply occasional changes when needed. Software Defined Networking (SDN), on the other hand, requires a central controller to routinely perform frequent policy and configuration updates in the network.

The centralized approach used in SDN introduces challenges in terms of *consistency* and *scalability*. The controller must take care to minimize network anomalies during update procedures, such as packet drops or misroutes caused by temporary **inconsistencies**. Updates must also be planned with **scalability** in mind; update procedures must scale with the size of the network, and cannot be too complex. In the face of rapid configuration changes, the update mechanism must allow a high update rate.

Two main methods for consistent network updates have been thoroughly studied in the last few years.

- *Ordered Updates:* This approach uses a sequence of phases of configuration commands, whereby the *order* of execution guarantees that no anomalies are caused in intermediate states of the procedure [2]–[5]; at each phase the controller waits until all the switches have completed their updates, and only then invokes the next phase in the sequence.
- *Two-Phase Updates:* In the *two-phase* approach [6], [7], configuration version tags are used to guarantee consistency; in the first phase the new configuration is installed in all the middle-stage switches of the network, and in the second phase the ingress switches are instructed to start using a version tag that represents the new configuration. During the update procedure every switch maintains two sets of entries: one for the old configuration version, and one for the new version. The version tag attached to the packet determines whether it is processed according to the old configuration or the new one. After the packets carrying the old version tag are drained from the network, garbage collection is performed on the switches, removing the duplicate entries and leaving only the new configuration.

In previous work [8] we argued that time is a powerful abstraction for coordinating network updates. We defined an extension [9] to the OpenFlow protocol [10] that allows time-triggered operations. This extension has been approved and integrated into OpenFlow 1.5 [11], and into the OpenFlow 1.3.x extension package [12].

B. Time for Consistent Updates

In this paper we study the use of *accurate time* to trigger *consistent* network updates. We define a time-based *order* approach, where each phase in the sequence is scheduled to a different execution time, and a time-based *two-phase* approach, where each of the two phases is invoked at a different time.

We show how the *order* and *two-phase* approaches benefit from time-triggered phases. Contrary to the conventional *order* and *two-phase* approaches, timed updates do not require the controller to wait until a phase is completed before invoking the next phase, significantly simplifying the controller's involvement in the update process, and reducing the update duration.

The time-based method significantly reduces the time duration required by the switches to maintain duplicate policy rules for the same flow. In order to accommodate the duplicate policy rules, switch flow tables should have a set of spare flow entries [6], [7] that can be used for network updates. Timed updates use each spare entry for a shorter duration than untimed updates, allowing higher scalability.

Manuscript received May 21, 2015; revised November 24, 2015; accepted January 11, 2016; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Chen. Date of publication March 2, 2016; date of current version December 15, 2016. This work was supported by the Israel Science Foundation under Grant 1520/11. Preliminary results were presented at the ACM SIGCOMM Symposium on SDN Research (SOSR), 2015 [1].

The authors are with the the Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology, Haifa 32000, Israel (e-mail: dew@tx.technion.ac.il; efisaat@tx.technion.ac.il; moses@ee.technion.ac.il).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2016.2529058

Accurate time synchronization has evolved over the last decade, as the Precision Time Protocol (PTP) [13] has become a common feature in commodity switches, allowing sub-microsecond accuracy in practical use cases (e.g., [14]). However, even if switches have perfectly synchronized clocks, it is not guaranteed that updates are *executed* at their scheduled times; a scheduling mechanism that relies on the switch's software may be affected by the switch's operating system and by other running tasks. We argue that a carefully designed switch can schedule updates with a high degree of accuracy. Moreover, we show that even if switches are not optimized for accurate scheduling, the timed approach outperforms conventional update approaches.

The use of time-triggered updates accentuates a **tradeoff** between update **scalability** and **consistency**. At one end of the scale, consistent updates come at the cost of a potentially long update duration, and expensive memory waste due to rule duplication.[1] At the other end, a network-wide update can be invoked simultaneously (e.g., using TIME4 [9]), allowing a short update time, preventing the need for rule duplication, but yielding a brief period of inconsistency. In this paper we show that timed updates can be tuned to any intermediate point along this scale.

C. Related Work

Various consistent network update approaches have been analyzed in the literature. Several solutions have been proposed [2]–[5], in which consistency is guaranteed by applying updates in a specific order. Another approach is to guarantee consistency using tags that are attached to the packet headers [6], [7], [15]. None of these solutions use accurate time and synchronized clocks as a means to coordinate the updates. In this paper we show that time can be used to improve these methods, allowing better performance during update procedures.

Scalability of network updates is another topic that has been discussed in several works. Using multiple SDN controllers to perform network updates, e.g., [16], [17], can improve scalability when the controller's performance is a bottleneck. Incremental methods [7], [18] can improve the efficient use of flow table space in switches by breaking each update into multiple independent rounds, thereby reducing the total overhead consumed in each separate round. The timed approach we present in this paper can be used in conjunction with each of these approaches, in order to improve scalability and efficiency even further.

The use of time in distributed applications has been widely analyzed, both in theory and in practice. Analysis of the usage of time and synchronized clocks, e.g., Lamport [19], [20] dates back to the late 1970s and early 1980s. Accurate time has been used in various different applications, such as distributed databases [21], industrial automation systems [22], automotive networks [23], and accurate instrumentation and measurements [24]. While the usage of accurate time in distributed systems has been widely discussed in the literature, we are not aware of similar analyses of the usage of accurate time as a means for performing consistent updates in computer networks.

Time-of-day routing [25] routes traffic to different destinations based on the time-of-day. Path calendaring [26] can be used to configure network paths based on scheduled or foreseen traffic changes. The two latter examples are typically performed at a low rate and do not place demanding requirements on accuracy.

In [27] the authors briefly mentioned that it would be interesting to explore using time synchronization to instruct routers or switches to change from one configuration to another at a specific time, but did not pursue the idea beyond this observation. Our previous work [8], [28] introduced the concept of using time to coordinate updates in SDN. The OpenFlow protocol [11], [12] currently supports time-based network updates. In [29] we presented a practical method to implement accurately scheduled network updates in hardware switches using timestamp-based TCAM rules. In this paper we analyze the use of time in *consistent* updates, and show that time can improve the scalability of consistent updates.

D. Contributions

The main contributions of this paper are as follows.

- We propose to use time-triggered network updates in a way that requires a lower overhead than existing update approaches without compromising the consistency during the update.
- We show that timed consistent updates require a shorter duration than existing consistent update methods. We also discuss hybrid approaches that combine the advantages of timed updates with those of other update methods.
- We define an inconsistency metric, allowing to quantify how consistent a network update is.
- We show that accurate time provides the SDN programmer with a knob for fine-tuning the tradeoff between consistency and scalability.
- We present experimental results that demonstrate the significant advantage of timed updates over other update methods. Our evaluation is based on experiments performed on a 50-node testbed, as well as simulation results.

II. TIME-BASED CONSISTENT UPDATES

We now describe the concept of time-triggered consistent updates. We assume that switches keep local clocks that are synchronized to a central reference clock by a synchronization protocol, such as the Precision Time Protocol (PTP) [13] or REVERSEPTP [30], [31], or by an accurate time source such as GPS. The controller sends network update messages to switches using an SDN protocol such as OpenFlow [11]. An update message may specify *when* the corresponding update is scheduled to be performed.

[1] As shown in [7], the **duration** of an update can be traded for the update rate. The flow table will typically include a limited number of excess entries that can be used for duplicated rules. By reducing the update duration, the excess entries are used for a shorter period of time, allowing a higher number of updates per second.

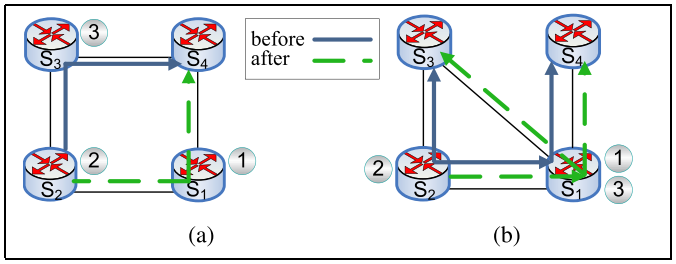

Fig. 1. Update procedure examples. (a) Ordered update of a path. (b) Two-phase update of a multicast distribution tree.

UNTIMED ORDERED UPDATE

1 Controller sends the 'after' configuration to S_1.
2 Controller sends the 'after' configuration to S_2.
3 Controller updates S_3 (garbage collection).

Fig. 2. Ordered update procedure for the scenario of Fig. 1a.

TIMED ORDERED UPDATE

0 Controller sends timed updates to all switches.
1 S_1 enables the 'after' configuration at time T_1.
2 S_2 enables the 'after' configuration at time $T_2 > T_1$.
3 S_3 performs garbage collection at time $T_3 > T_2$.

Fig. 3. Timed Ordered update procedure for the scenario of Fig. 1a.

A. Ordered Updates

Fig. 1a illustrates an ordered network update.[2] We would like to reconfigure the path of a traffic flow from the 'before' to the 'after' configuration. An ordered update proceeds as described in Fig. 2; the phases in the procedure correspond to the numbers in Fig. 1a.

In the ordered update procedure every phase is performed after the previous phase was completed, and this guarantees consistency. Note that packets may arrive out-of-order to S_4 due to the update. However, the update is per-packet consistent [6], i.e., each packet is forwarded either according to the 'before' configuration or according to the 'after' configuration, and no packets are dropped. A **time-based** order update procedure is described in Fig. 3.

Notably, the ordered approach requires the controller to be involved in the entire update procedure, making the update process sensitive to the load on the controller, and to the communication delays at the time of execution. In contrast, in the time-base protocol, the controller is only involved in phase 0, and if T_1 is chosen correctly, the update process is not influenced by these issues.

B. Two-Phase Updates

An example of a *two-phase* update is illustrated in Fig. 1b; the figure depicts a multicast distribution tree through a network of three switches. Multicast packets are distributed along the paths of the 'before' tree. We would like to reconfigure the distribution tree to the 'after' state.

[2]An ordered update proceeds as a sequence of k phases, which must be performed according to a specific order. The update in the current example is performed in three phases.

UNTIMED TWO-PHASE UPDATE

1 Controller sends the 'after' configuration to S_1.
2 Controller instructs S_2 to start using the 'after' configuration with the new version tag.
3 Controller updates S_1 (garbage collection).

Fig. 4. Two-phase update procedure for the scenario of Fig. 1b.

TIMED TWO-PHASE UPDATE

0 Controller sends timed updates to all switches.
1 S_1 enables the 'after' configuration at time T_1.
2 S_2 enables the 'after' configuration with the new version tag at time $T_2 > T_1$.
3 S_1 performs garbage collection at time $T_3 > T_2$.

Fig. 5. Timed two-phase update procedure for the scenario of Fig. 1b.

The *two-phase* procedure [6], [7] is described in Fig. 4. In the first phase, the new configuration is installed in S_1, instructing it to forward packets that have the new version tag according to the 'after' configuration. Thus, as defined in [6], the value of the version tag field is part of the flow match rule, and S_1 has two rules in the flow table, one corresponding to the 'before' action, and the other corresponding to the 'after' action. In the second phase, S_2 is instructed to forward packets according to the 'after' configuration using the new version tag. The 'before' configuration is removed in the third phase. As in the ordered approach, the *two-phase* procedure requires every phase to be invoked after the previous phase was completed.

In the timed *two-phase* approach, specified in Fig. 5, phases 1, 2, and 3 are scheduled in advance by the controller. The switches then execute phases 1, 2, and 3 at times T_1, T_2, and T_3, respectively.

C. k-Phase Consistent Updates

The *order* approach guarantees consistency if updates are performed according to a specific order. More generally, we can view an ordered update as a sequence of k phases, where in each phase j, a set of N_j switches is updated. For each phase j, the updates of phase j must be completed before any update of phase $j+1$ is invoked.

The *two-phase* approach is a special case, where $k = 2$; in the first phase all the switches in the middle of the network are updated with the new policy, and in the second phase the ingress switches are updated to start using the new version tag.

D. The Overhead of Network Updates

Both the *order* method and the *two-phase* method require duplicate configurations to be present during the

update procedure. In each of the protocols of Figures 2-5, both the 'before' and the 'after' configurations are stored in the switches' expensive flow tables from phase 1 to phase 3. The unnecessary entries are removed only after garbage collection is performed in phase 3.

In the timed protocols of Fig. 3 and 5 the switches receive the update messages in advance (phase 0), and can temporarily store the new configurations in an inexpensive memory. The switches install the new configuration in the expensive flow table memories only at the scheduled times, thereby limiting the period of duplication to the duration from phase 1 to phase 3.

The overhead cost of the duplication depends on the time elapsed between phase 1 and phase 3. Hence, throughout the paper we use the *update duration* as a metric for quantifying the overhead of a consistent update that includes a garbage collection phase.

III. Terminology and Notations

A. The Network Model

We reuse some of the terminology and notations of [6]. Our system consists of $N+1$ nodes: a controller c, and a set of N switches, $\mathbb{S} = \{S_1, \ldots, S_N\}$. A *packet* is a sequence of bits, denoted by $pk \in \mathbb{P}k$, where $\mathbb{P}k$ is the set of possible packets in the system. Every switch $S_i \in \mathbb{S}$ has a set $\mathbb{P}r_i$ of ports.

The sources and destinations of the packets are assumed to be external; packets are received from the 'outside world' through a subset of the switches' ports, referred to as *ingress ports*. An *ingress switch* is a switch that has at least one ingress port. Every packet pk is forwarded through a sequence of switches $(S_{i_1}, \ldots, S_{i_m})$, where the first switch S_{i_1} is an ingress switch. The last switch in the sequence, S_{i_m}, forwards the packet through one of its ports to the outside world.

When a packet pk is received by a switch S_i through port $p \in \mathbb{P}r_i$, the switch uses a forwarding function $\mathbb{F}_i : \mathbb{P}k \times \mathbb{P}r_i \longrightarrow \mathbb{A}$, where $\mathbb{A}$ is the set of possible actions a switch can perform, e.g., 'forward the packet through port q''. The packet content and the port through which the packet was received determine the action that is applied to the packet.

It is assumed that every switch maintains a local clock. As is standard in the literature (e.g., [32]), we distinguish between *real time*, an assumed Newtonian time frame that is not directly observable, and *local clock time*, which is the time measured on one of the switches' clocks. We denote values that refer to real time by lowercase letters, e.g. t, and values that refer to clock time by uppercase, e.g., T.

We define a *packet instance* to be a tuple (pk, S_i, p, t), where $pk \in \mathbb{P}k$ is a packet, $S_i \in \mathbb{S}$ is the ingress switch through which the packet is received, $p \in \mathbb{P}r_i$ is the ingress port at switch S_i, and t is the time at which the packet instance is received by S_i.

B. Network Updates

We define a *singleton update* u of switch S_i to be a partial function, $u : \mathbb{P}k \times \mathbb{P}r_i \rightharpoonup \mathbb{A}$. A switch applies a singleton update, u, by replacing its forwarding function, $\mathbb{F}_i$ with a new forwarding function, $\mathbb{F}'_i$, that behaves like u in the domain of u, and like $\mathbb{F}_i$ otherwise. We assume that every singleton update is triggered by a set of one or more messages sent by the controller to **one** of the switches.

We define an *update* to be a set of singleton updates $U = \{u_1, \ldots, u_m\}$. We define an *update procedure* to be a set $\mathbb{U} = \{(u_1, t_1, phase(u_1)), \ldots, (u_m, t_m, phase(u_m))\}$ of triples, such that for all $1 \leq j \leq m$, we have that u_j is a singleton update, $phase(u_j)$ is a positive integer specifying the *phase number* of u_j, and t_j is the time at which u_j is performed. Moreover, it is required that for every $1 \leq i, j \leq m$, if $phase(u_i) < phase(u_j)$ then $t_i < t_j$. This definition implies that an update procedure is a sequence of one or more phases, where each phase is performed after the previous phase is completed, but there is no guarantee about the order of the singleton updates within each phase.

A k-phase update procedure is an update procedure $\mathbb{U} = \{(u_1, t_1, phase(u_1)), \ldots, (u_m, t_m, phase(u_m))\}$ in which for all $1 \leq j \leq m$ we have $1 \leq phase(u_j) \leq k$, and for all $1 \leq i \leq k$ there exists an update u_j such that $(u_j, t_j, i) \in \mathbb{U}$.

We define a *timed* singleton update u^T to be a pair (u, T), where u is a singleton update, and T is a clock value that represents the scheduled time of u. We assume that every switch maintains a local clock, and that when a switch receives a message indicating a timed singleton update u^T it implements the update as close as possible to the instant when its local clock reaches the value T. Similar to the definition of an update procedure, we define a *timed update procedure* $\mathbb{U}^T$ to be a set $\mathbb{U}^T = \{(u_1^T, t_1, phase(u_1^T)), \ldots, (u_m^T, t_m, phase(u_m^T))\}$.

An update procedure, $\mathbb{U} = \{(u_1, t_1, phase(u_1)), \ldots, (u_m, t_m, phase(u_m))\}$, and a timed update procedure, $\mathbb{U}^T = \{(v_1^T, t_1, phase(v_1^T)), \ldots, (v_n^T, t_n, phase(v_n^T))\}$, are said to be *similar*, denoted by $\mathbb{U}^T \sim \mathbb{U}$ if $m = n$ and for every $1 \leq j \leq m$ we have $u_j = v_j$ and $phase(u_j) = phase(v_j)$.

We define *consistent forwarding* based on the per-packet consistency definition of [6]. Intuitively, given an untimed update U, a packet is consistently forwarded if it is processed by all switches either according to the new configuration, after the update U was applied, or according to the old one, but not according to a mixture of the two. Formally, let (pk, S_{i_1}, p_1, t) be a packet instance that is forwarded through a sequence of switches $S_{i_1}, S_{i_2}, \ldots, S_{i_m}$ through ports $p_1, p_2, \ldots, p_m$, respectively, and is assigned the actions $a_1, a_2, \ldots, a_m$. Let $\mathbb{F}_{i_j}$ be the forwarding function of S_{i_j} before the update is applied, and let $\mathbb{F}'_{i_j}$ be the forwarding function after the update. The packet instance (pk, S_{i_1}, p_1, t) is said to be *consistently forwarded* if either of the following is satisfied:

(i) $\mathbb{F}_{i_j}(pk, p_j) = a_j$ for all $1 \leq j \leq m$, or
(ii) $\mathbb{F}'_{i_j}(pk, p_j) = a_j$ for all $1 \leq j \leq m$.

A packet instance that is not consistently forwarded, is said to be *inconsistently forwarded*.

C. Delay-Related Notations

Table I presents key notations related to delay and performance. The attributes that play a key role in our analysis are D_c, D_n, and δ. These attributes are discussed further in Section IV.

TABLE I
DELAY-RELATED NOTATIONS

D_c	An upper bound on the *controller-to-switch delay*, including the network latency, and the internal switch delay until completing the update.
D_n	An upper bound on the end-to-end *network delay*.
Δ	An upper bound on the time interval between the transmission times of two consecutive update messages sent by the controller.
δ	An upper bound on the scheduling error; an update that is scheduled to be performed at T is performed in practice during the time interval $[T, T+\delta]$.
T_{su}	The timed update setup time; in order to invoke a timed update that is scheduled to time T, the controller sends the update messages no later than at $T - T_{su}$.

IV. UPPER AND LOWER BOUNDS

A. Delay Upper Bounds

Both the *order* [2]–[5] and the *two-phase* [6], [7] approaches implicitly assume the existence of two upper bounds, D_c and D_n (see Table I):

- D_c: Both approaches require previous phases in the update procedure to be completed before invoking the current phase. Therefore, after sending an update message, the controller must wait for a period of D_c until it is guaranteed that the corresponding update has been performed; only then can it invoke the next phase in the procedure. Alternatively, explicit acknowledgments can be used to indicate update completions, as further discussed in Section IV-B.
- D_n: Garbage collection can take place after the update procedure has completed, and all en-route packets have been drained from the network. Garbage collection can be invoked either after waiting for a period of D_n after completing the update, or by using *soft timeouts*.[3] Both of these approaches assume there is an upper bound, D_n, on the end-to-end network latency.

Is it practical to assume that the upper bounds D_c and D_n exist? Network latency is often modeled using long-tailed distributions such as exponential or Gamma [33], [34], implying that network latency is often **unbounded**.

We demonstrate the long-tailed behavior of network latency by analyzing measurements performed on production networks. We analyze 20 delay measurement datasets from [35], [36] taken at various sites over a one-year period, from November 2013 to November 2014. The measurements capture the round-trip time (RTT) using ICMP Echo requests. The measurements show (Fig. 6) that in some networks the 99.999^{th} percentile is almost two orders of magnitude higher than the average RTT. Table II summarizes the ratio between tail latency values and average values in the 20 traces we analyzed.

In typical networks we expect D_n to have long-tailed behavior. Similar long-tailed behavior has also been shown for D_c in [3] and [37].

[3]Soft timeouts are defined in the OpenFlow protocol [11] as a means for garbage collection; a flow entry that is configured with a soft timeout, D_n, is cleared if it has not been used for a duration D_n.

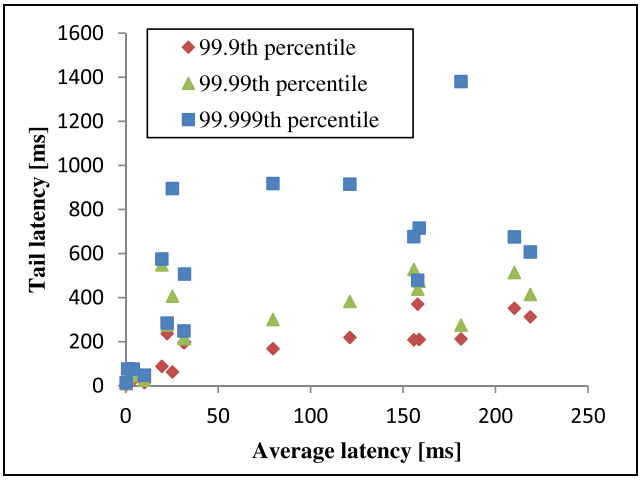

Fig. 6. Long-tail latency.

TABLE II
THE MEAN RATIO BETWEEN THE TAIL LATENCY AND THE AVERAGE LATENCY

99.9^{th} percentile	99.99^{th} percentile	99.999^{th} percentile
4.88	10.49	19.45

At a first glance, these results seem troubling: if network latency is indeed unbounded, neither the *order* nor the *two-phase* approaches can guarantee consistency, since the controller can never be sure that the previous phase was completed before invoking the next phase.

In practice, typical approaches will not require a true upper bound, but rather a latency value that is exceeded with a sufficiently low probability. Service Level Agreement (SLA) in carrier networks is a good example of this approach; per the MEF 10.3 specification [38], a Service Level Specification (SLS) defines not only the mean delay, but also the Frame Delay Range (FDR), and the percentile defining this range. Thus, service providers must guarantee that the rate of frames that exceed the delay range is limited to a known percentage.

Throughout the paper we use D_c and D_n, to denote the upper bounds of the delays. In practice, these may refer to a sufficiently high percentile delay. Our analysis in Section VI revisits the upper bound assumption.

B. Explicit Acknowledgment

If a switch can explicitly notify the controller when it completes an update operation, then the controller has a definitive indication of when the update is completed. Unfortunately, OpenFlow [11], [39] currently *does not* support such an acknowledgment mechanism. Hence, one can either use a different SDN protocol that supports explicit acknowledgment (as was assumed in [3]), or use an update procedure in which the controller waits for a fixed period (D_c) until the switch is guaranteed to complete the update.

In the absence of ACKs, update procedures are planned according to a *worst-case analysis* (Section V), both in the timed and in the untimed approaches; the controller waits

until it is guaranteed that the update was completed with a sufficiently high probability. In both the timed and untimed approaches, an update cannot be completed with 100% consistency in the absence of ACKs. Hence, in Section VI we introduce a metric that quantifies the level of consistency during an update.

In the presence of ACKs (as assumed in [3]), update procedures can sometimes be completed earlier than without using ACKs. Furthermore, ACKs enable updates to be performed dynamically [3], whereby at the end of each phase the controller dynamically plans the next phase.

Whether ACKs are available or not, garbage collection can only be performed after all packets that were en-route during the update have been drained from the network, requiring the controller to wait for a period of D_n units. Fortunately, the timed and untimed approaches can be combined. For example, in the presence of an acknowledgment mechanism, update procedures can be performed in a dynamic, untimed, ACK-based manner, with a timed garbage collection phase at the end. Such a flexible mix-and-match approach allows the SDN programmer to enjoy the best of both worlds. This hybrid approach is further discussed in Section V-E.

C. Delay Lower Bounds

Throughout the paper we assume that the lower bounds of the network delay and the controller-to-switch delay are zero. This assumption simplifies the presentation, although the model can be extended to include non-zero lower bounds on delays.

D. Scheduling Accuracy Bound

As defined in Table I, δ is an upper bound on the scheduling error, indicating how accurately updates are scheduled; an update that is scheduled to take place at time T is performed in practice during the interval $[T, T+\delta]$.[4] A switch's scheduling accuracy depends on two factors: (i) how accurately its clock is synchronized to the system's reference clock, and (ii) its ability to perform real-time operations.

Most high-performance switches are implemented as a combination of hardware and software components. A scheduling mechanism that relies on the switch's software may be affected by the switch's operating system and by other running tasks, consequently affecting the scheduling accuracy. Furthermore, previous work [3], [37] has shown high variability in rule installation latencies in Ternary Content Addressable Memories (TCAMs), resulting from the fact that a TCAM update might require the TCAM to be rearranged.

Nevertheless, existing switches and routers practice real-time behavior, with a predictable guaranteed response time to important external events. Traditional protection switching and fast reroute mechanisms require the network to react to a path failure in less than 50 milliseconds, implying that each individual switch or router reacts within a few milliseconds, or in some cases less than one millisecond (e.g. [40]). Operations, Administration, and Maintenance (OAM) protocols such as the IEEE 802.1ag [41] require faults to be detected within a strict timing constraint of ± 0.42 milliseconds.[5]

Measures can be taken to implement accurate scheduling of timed updates:

- Common real-time programming practices can be applied to ensure guaranteed performance for time-based update, by assigning a constant fraction of time to timed updates.
- When a switch is aware of an update that is scheduled to take place at time T_s, it can avoid performing heavy maintenance tasks near this time, such as TCAM entry rearrangement.
- Untimed update messages received slightly before time T_s can be queued and processed after the scheduled update is executed.
- If a switch receives a time-based command that is scheduled to take place at the same time as a previously received command, it can send an error message to the controller, indicating that the last received command cannot be executed.
- It has been shown that timed updates can be scheduled with a very high degree of accuracy, on the order of 1 microsecond, using TIMEFLIP [29]. This approach provides a high scheduling accuracy, potentially at the cost of some overhead in the switch's flow tables.

Observation 1: In typical settings $\delta < D_c$.

The intuition behind Observation 1 is that δ is only affected by the switch's performance, whereas D_c is affected by both the switch's performance and the network latency. We expect Observation 1 to hold even if switches are not designed for real-time performance. We argue that in switches that use some of the real-time techniques above, $\delta \ll D_c$, making the timed approach significantly more advantageous, as we shall see in the next section.

V. WORST-CASE ANALYSIS

A. Worst-Case Update Duration

We define the *duration* of an update procedure to be the time elapsed from the instant at which the first switch updates its forwarding function to the instant at which the last switch completes its update.

We use Program Evaluation and Review Technique (PERT) charts [42] to illustrate the worst-case update duration analysis. Fig. 7 illustrates a PERT chart of an untimed ordered k-phase update, where three switches are updated in each phase. Switches S_1, S_2, and S_3 are updated in the first phase, S_4, S_5, and S_6 are updated in the second phase, and so on. In this procedure, the controller waits until phase j is guaranteed to have been completed before starting phase $j+1$.

Each node in the PERT chart represents an event, and each edge represents an activity. A node labeled $C_{j,i}$ represents the event 'the controller starts transmitting a phase j update message to switch S_i'. A node labeled $S_{j,i}$ represents 'switch S_i has completed its phase j update'. The weight of each edge

[4] An alternative representation of δ assumes a symmetric error, $T \pm \delta/2$. The two approaches are equivalent.

[5] Faults are detected using Continuity Check Messages (CCM), transmitted every 3.33 ms. A fault is detected when no CCMs are received for a period of 11.25 ± 0.42 ms.

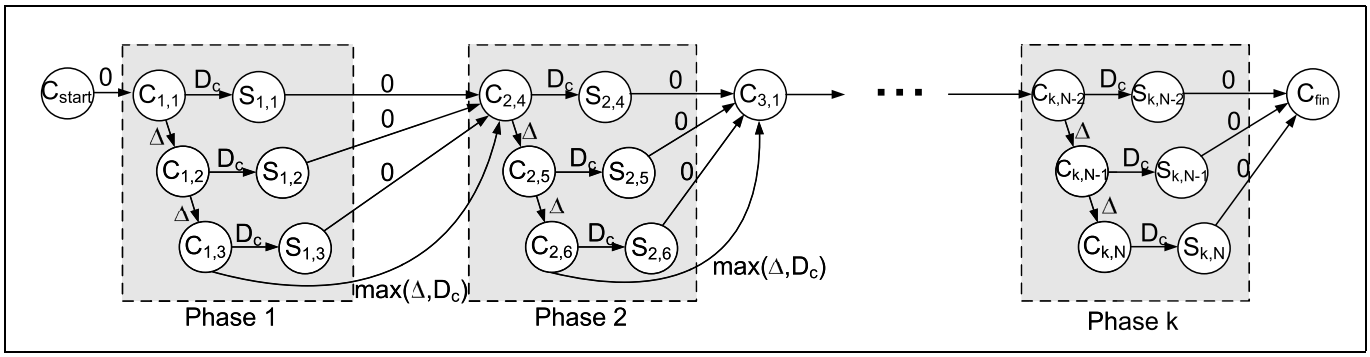

Fig. 7. A PERT chart of a k-phase update.

indicates the maximal delay to complete the transition from one event to another. C_{start} and C_{fin} represent the start and finish times of the update procedure, respectively. The worst-case duration between two events is given by the longest path between the two corresponding nodes in the graph.

Throughout the section we focus on *greedy* update procedures. An update procedure is said to be *greedy* if the controller invokes each update message at the earliest possible time that guarantees that for every phase j all the singleton updates of phase j are completed before those of phase $j+1$ are initiated.

B. Worst-Case Analysis of Untimed Updates

1) Untimed Updates: We start by discussing untimed k-phase update procedures, focusing on a single phase, j, in which N_j switches are updated. In Lemma 1 and in the upcoming lemmas in this section we focus on *greedy* updates.

Lemma 1: If $\mathbb{U}$ is a multi-phase update procedure, then the worst-case duration of phase j of $\mathbb{U}$ is:

$$(N_j - 1) \cdot \Delta + D_c \tag{1}$$

Proof: Assume that the controller transmits the first update message of phase j at time t. Since N_j switches take part in phase j, and Δ is the upper bound on the duration between two consecutive messages, the controller invokes the last update message of phase j no later than at $t + (N_j - 1) \cdot \Delta$. Since D_c is the upper bound on the controller-to-switch delay, the update is completed at most D_c time units later. Hence, the worst-case update duration is $(N_j - 1) \cdot \Delta + D_c$. □

The following lemma specifies the worst-case update duration of a k-phase update. The intuition is straightforward from Fig. 7.

Lemma 2: The worst-case update duration of a k-phase update procedure is:

$$\sum_{j=1}^{k} (N_j - 1) \cdot \Delta + (k-1) \cdot \max(\Delta, D_c) + D_c \tag{2}$$

Proof: Each phase j delays the controller for $(N_j - 1) \cdot \Delta$. Since the update is greedy, at the end of each of the first $k-1$ phases the controller waits $\max(\Delta, D_c)$ time units to guarantee that the phase has completed, and then immediately proceeds to the next phase. The update is completed, in the worst case, D_c time units after the controller sends the last update message of the k^{th} phase. The claim follows. □

Specifically, in *two-phase* updates $k = 2$, and thus:

Corollary 1: If $\mathbb{U}$ is a two-phase *update procedure, then its worst-case update duration is*:

$$(N_1 + N_2 - 2) \cdot \Delta + \max(\Delta, D_c) + D_c \tag{3}$$

2) Untimed Updates With Garbage Collection: In some cases, garbage collection is required for some of the phases in the update procedure. For example, in the *two-phase* approach, after phase 2 is completed and all en-route packets have been drained from the network, garbage collection is required for the N_1 switches of the first phase.

More generally, assume that at the end of every phase j the controller performs garbage collection for a set of N_{Gj} switches. Thus, after phase j is completed the controller waits D_n time units for the en-route packets to drain, and then invokes the garbage collection procedure for the N_{Gj} switches.

After invoking the last message of phase j, the controller waits for $\max(\Delta, D_c + D_n)$ time units. Thus, the worst-case duration from the transmission of the last message of phase j until the garbage collection of phase j is completed is given by Eq. 4.

$$\max(\Delta, D_c + D_n) + (N_{Gj} - 1) \cdot \Delta + D_c \tag{4}$$

Fig. 8 depicts a PERT chart of a *two-phase* update procedure that includes a garbage collection phase. No garbage collection is required at the end of phase 1, and thus $N_{G1} = 0$. At the end of the second phase, garbage collection is performed for the policy rules of phase 1, affecting $N_{G2} = 3$ switches: S_1, S_2, and S_3. This is in fact a special case of a 3-phase update procedure, where the third phase takes place only after all the en-route packets are guaranteed to have been drained from the network. The main difference between this example and the general k-phase graph of Fig. 7 is that in Fig. 8 the controller waits at least $max(\Delta, D_c + D_n)$ time units from the transmission of the last message of phase 2 until starting to invoke the garbage collection phase.

Note that in a multi-phase update there may be several garbage collection phases, each performed at a different stage of the update.

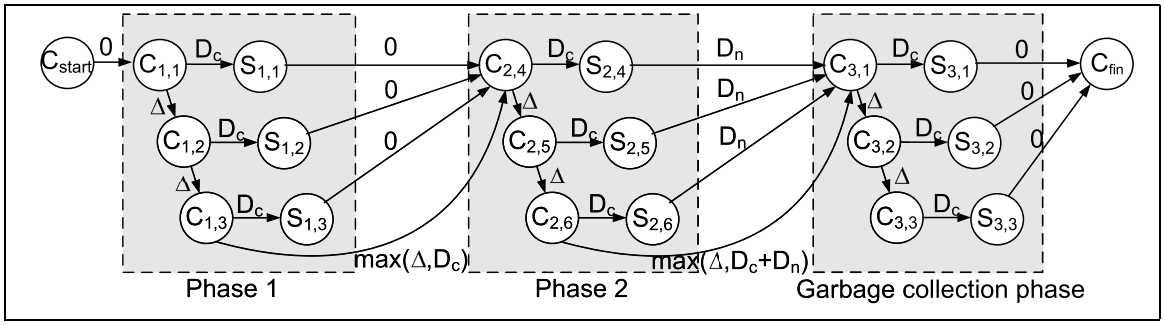

Fig. 8. A PERT chart of a two-phase update with garbage collection, performed after phase 2 is completed. Garbage collection removes the 'before' configuration (see Fig. 1) from the switches that took part in phase 1.

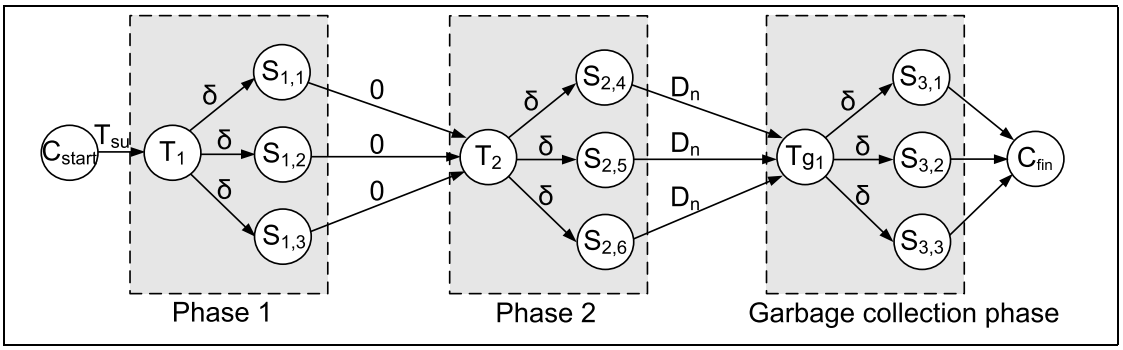

Fig. 9. A PERT chart of a timed two-phase update with garbage collection. The node labeled 'T_1' represents the event 'the current time is T_1', and thus all switches perform their scheduled updates for phase 1 no later than δ time units afterwards.

Lemma 3: If $\mathbb{U}$ *is a* two-phase *update procedure with a garbage collection phase, then its worst-case update duration is*:

$$(N_1 + N_2 + N_{G2} - 3) \cdot \Delta + \max(\Delta, D_c) + \max(\Delta, D_c + D_n) + D_c \quad (5)$$

Proof: In each of the three phases the controller waits at most Δ time units between two consecutive update messages, summing up to $(N_1 + N_2 + N_{G2} - 3) \cdot \Delta$. The controller waits for $\max(\Delta, D_c)$ time units at the end of phase 1, guaranteeing that all the updates of phase 1 have been completed before invoking phase 2. At the end of phase 2 the controller waits for $\max(\Delta, D_c + D_n)$ time units, guaranteeing that phase 2 is completed, and that all the en-routed packets have been drained before starting the garbage collection phase. Finally, D_c time units after the controller sends the last message of the garbage collection phase, the last update is guaranteed to be completed. □

C. Worst-Case Analysis of Timed Updates

1) Worst-Case-Based Scheduling: Based on a worst-case analysis, an SDN program can determine an update schedule, $\mathbb{T} = (T_1, \ldots, T_k, T_{g_1}, \ldots, T_{g_k})$. Every timed update u^T is performed no later than at $T + \delta$. Consequently, we can derive the worst-case scheduling constraints below.

Definition 1 (Worst-Case Scheduling): If $\mathbb{U}$ *is a timed* k*-phase update procedure, then a schedule* $\mathbb{T} = (T_1, \ldots, T_k, T_{g_1}, \ldots, T_{g_k})$ *is said to be a* worst-case schedule *if it satisfies the following two equations*:

$$T_j = T_{j-1} + \delta \quad for\ every\ phase\ 2 \le j \le k \quad (6)$$

$$T_{g_j} = T_j + \delta + D_n \quad (7)$$

for every phase j *that requires garbage collection.*

Note that a *greedy timed* update procedure uses *worst-case scheduling*.

Every schedule $\mathbb{T}$ that satisfies Eq. 6 and 7 guarantees consistency. For example, the timed *two-phase* update procedure of Fig. 9 satisfies the two scheduling constraints above.

2) Timed Updates: A timed update starts with the controller sending scheduled update messages to all the switches, requiring a setup time T_{su}. Every phase is guaranteed to take no longer than δ. An example of a timed *two-phase* update is illustrated in Fig. 9.

Lemma 4: The worst-case update duration of a k*-phase timed update procedure with a worst-case schedule is* $k \cdot \delta$.

Proof: The lemma follows directly from the worst-case scheduling constraints of Eq. 6 and 7. □

Based on the latter, we derive the following lemma.

Lemma 5: If $\mathbb{U}$ *is a* two-phase *timed update procedure with a garbage collection phase using a worst-case schedule, then its worst-case update duration is* $D_n + 3 \cdot \delta$.

Proof: By Lemma 4, the first two phases take $2 \cdot \delta$ time units. The garbage collection phase requires δ additional time units, and D_n time units to allow all en-route packets to drain from the network. Thus, the update duration is $D_n + 3 \cdot \delta$. □

D. Timed vs. Untimed Updates

We now study the conditions under which the timed approach outperforms the untimed approach.

Based on Lemmas 2 and 4, we observe that a timed k-phase update procedure has a shorter update duration than a similar untimed k-phase update procedure if:

$$k \cdot \delta \; < \; \sum_{j=1}^{k}(N_j - 1) \cdot \Delta + (k-1) \cdot \max(\Delta, D_c) + D_c \quad (8)$$

Lemma 6: Let $\mathbb{U}^T$ be a greedy timed k-phase update procedure, with a worst-case update duration D_1. Let $\mathbb{U}$ be a greedy untimed k-phase update procedure with a worst-case update duration D_2. If $\delta < D_c$ and $\mathbb{U}^T \sim \mathbb{U}$, then $D_1 < D_2$.

Proof: By Lemma 4, we have $D_1 = k \cdot \delta$. Lemma 2 yields $D_2 = \sum_{j=1}^{k}(N_j - 1) \cdot \Delta + (k-1) \cdot \max(\Delta, D_c) + D_c$. Thus, $D_1 = k \cdot \delta < k \cdot D_c < (k-1) \cdot \max(\Delta, D_c) + D_c < \sum_{j=1}^{k}(N_j - 1) \cdot \Delta + (k-1) \cdot \max(\Delta, D_c) + D_c = D_2$. It follows that $D_1 < D_2$. □

Now, based on Lemma 3 and Lemma 5, we observe that a timed *two-phase* update procedure with garbage collection has a shorter update duration than a similar untimed *two-phase* update procedure if:

$$D_n + 3 \cdot \delta < (N_1 + N_2 + N_{G2} - 3) \cdot \Delta + \max(\Delta, D_c) + \max(\Delta, D_c + D_n) + D_c \quad (9)$$

Lemma 7: Let $\mathbb{U}^T$ be a greedy timed two-phase *update procedure with a garbage collection phase, with a worst-case update duration D_1. Let $\mathbb{U}$ be a greedy untimed* two-phase *update procedure with a worst-case update duration D_2. If $\delta < D_c$ and $\mathbb{U}^T \sim \mathbb{U}$, then $D_1 < D_2$.*

Proof: By Lemma 5 we have $D_1 = D_n + 3 \cdot \delta$, and by Lemma 3 we have $D_2 = (N_1 + N_2 + N_{G2} - 3) \cdot \Delta + \max(\Delta, D_c) + \max(\Delta, D_c + D_n) + D_c$.

Thus, $D_1 = D_n + 3 \cdot \delta < D_n + 3 \cdot D_c < (N_1 + N_2 + N_{G2} - 3) \cdot \Delta + D_n + 3 \cdot D_c \leq (N_1 + N_2 + N_{G2} - 3) \cdot \Delta + \max(\Delta, D_c) + \max(\Delta, D_c + D_n) + D_c = D_2$. It follows that $D_1 < D_2$, as claimed. □

We have shown that if $\delta < D_c$ the timed approach yields a shorter update duration than the untimed approach, and is thus more scalable. Based on Observation 1, even if switches are not designed for real-time performance we have $\delta < D_c$. We conclude that **the timed approach is superior in typical settings**.

E. Using Acknowledgments

As discussed in Section IV-B, explicit acknowledgment allows the controller to have a clear indication of when an update is completed. In this section we discuss how untimed acknowledged *two-phase* updates can be combined with a timed garbage collection phase, combining the reliability of using ACKs with the short update duration of the timed approach.

Consider a *two-phase* update in which the first two phases are untimed, and where in each phase every switch sends an ACK to the controller upon completing the update. Note that the use of ACKs alleviates the need for a worst-case analysis of the first two phases. However, the garbage collection phase can be applied only when all packets that were en-route during the first phase are drained from the network, requiring a worst-case waiting time from the controller despite the acknowledgment mechanism. Hence, we focus our worst-case analysis on the garbage collection phase.

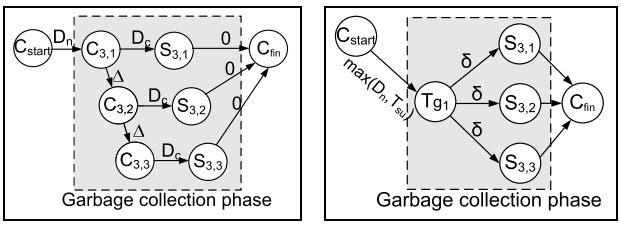

Fig. 10. PERT charts of the garbage collection phase of an ACK-based update. (a) Untimed garbage collection. (b) Timed garbage collection.

Fig. 10 presents the PERT charts of an untimed garbage collection phase (Fig. 10a) and a timed garbage collection phase (Fig. 10b). In these figures C_{start} refers to the time at which phase 2 has completed, i.e., the point at which the controller received the last ACK of phase 2.

Lemma 8: The worst-case duration of an untimed garbage collection phase in a two-phase update is:

$$D_g^{UT} = D_n + (N_{G2} - 1) \cdot \Delta + D_c \quad (10)$$

Proof: The controller waits for D_n time units to allow en-route packets to be flushed from the network. Then, the controller sends update messages to N_{G2} switches, and waits at most Δ time units between two consecutive update messages, summing up to $(N_{G2} - 1) \cdot \Delta$. Finally, the last update is completed at most D_c time units after the controller sends the last update message. Thus, the worst-case duration is $D_n + (N_{G2} - 1) \cdot \Delta + D_c$. □

Lemma 9: The worst-case duration of a timed garbage collection phase in a two-phase update is:

$$D_g^T = \max(D_n, T_{su}) + \delta \quad (11)$$

Proof: The controller waits for D_n time units to allow en-route packets to be flushed from the network. The controller allows T_{su} time units for the update messages to propagate to the switches. Since the two latter durations overlap, the worst-case is $\max(D_n, T_{su})$. Since the scheduling error is δ, the worst-case duration is $\max(D_n, T_{su}) + \delta$. □

We observe that for a sufficiently large value of D_n the timed approach produces a lower update duration. This results from the fact that for a high D_n we have $D_g^T = D_n + \delta$. By Observation 1, in typical settings $\delta < D_c$, and thus $D_g^T = D_n + \delta < D_n + D_c < D_g^{UT}$. We further demonstrate the advantage of the timed approach in Section VII-C.

VI. Time as a Consistency Knob

A. An Inconsistency Metric

As discussed in Section IV, the upper bounds D_c and D_n do not necessarily exist, or may be very high. Thus, in practice consistent network updates only guarantee consistent

forwarding with a high probability, raising the need for a way to measure and *quantify* to what extent an update is consistent.

Definition 2 (Test Flow): A set of packet instances $\mathbb{PI}$ is said to be a test flow *if for every two packet instances $(pk_1, S_1, p_1, t_1) \in \mathbb{PI}$ and $(pk_2, S_2, p_2, t_2) \in \mathbb{PI}$, all the following conditions are satisfied:*

- $S_1 = S_2$.
- $p_1 = p_2$.
- $pk_1 = pk_2$.[6]
- *Packet instances are received at a constant packet arrival rate R, i.e., if both $t_2 > t_1$ and there is no packet instance $(pk_3, S_3, p_3, t_3) \in \mathbb{PI}$ such that $t_2 > t_3 > t_1$, then $t_2 = t_1 + 1/R$.*

We assume a method that, for a given test flow f and an update u, allows to measure the number of packets $n(f, u)$ that are forwarded inconsistently.[7]

Definition 3 (Inconsistency Metric): Let f be a test flow with a packet arrival rate $R(f)$. Let U be an update, and let $n(f, U)$ be the number of packet instances of f that are forwarded inconsistently due to update U. The inconsistency $I(f, U)$ of a flow f with respect to U is defined to be:

$$I(f, U) = \frac{n(f, U)}{R(f)} \tag{12}$$

The inconsistency $I(f, U)$ is measured in time units. Intuitively, $I(f, U)$ quantifies the amount of time that flow f is disrupted by the update.

B. Fine Tuning Consistency

Timed updates provide a powerful mechanism that allows SDN programmers to tune the degree of consistency. By **setting** the update times $T_1, T_2, \ldots, T_k, T_{g_1}, \ldots, T_{g_k}$, the controller can play with the consistency-scalability tradeoff; the update overhead can be reduced at the expense of some inconsistency, or vice versa.[8]

Example 1: We consider a two-phase *update with a garbage collection phase. We assume that $\delta = 0$ and that all packet instances are subject to a constant network delay, D_n. By assigning $T = T_1 = T_2 = T_{g_1}$, the controller schedules a simultaneous update. This approach is referred to as* TIME4 *in [9]. All switches are scheduled to perform the update at the same time T. Packets entering the network during the period $[T - D_n, T]$ are forwarded inconsistently. The inconsistency metric in this example is $I = D_n$. The advantage of this approach is that it completely relieves the switches from the overhead of maintaining duplicate entries between the phases of the update procedure.*

Example 2: Again, we consider a two-phase *update (Fig. 11), with $\delta = 0$ and a constant network delay, D_n.*

[6]For simplicity, we define that all packets of a test flow are identical. It is in fact sufficient to require that all packets of the flow are indistinguishable by the switch forwarding functions, for example, that all packets of a flow have the same source and destination addresses.

[7]This measurement can be performed, for example, by per-flow match counters in the switches.

[8]In some scenarios, such as security policy updates, even a small level of inconsistency cannot be tolerated. In other cases, such as path updates, a brief period of inconsistency comes at the cost of some packets being dropped, which can be a small price to pay for reducing the update duration.

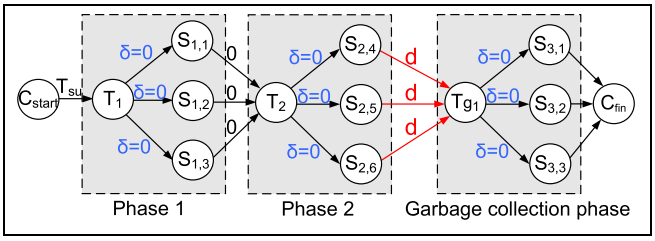

Fig. 11. Example 2: PERT chart of a timed two-phase update. The delay d (red in the figure) is a knob for consistency.

TABLE III
THE MEASURED 99.9^{th} PERCENTILE OF EACH OF THE DELAY ATTRIBUTES IN MILLISECONDS

D_n	D_c	δ	Δ
0.262	4.865	1.297	5.24

We assign $T_2 = T_1 + \delta$ according to Eq. 6, and T_{g_1} is assigned to be $T_2 + \delta + d$, where $d < D_n$. The update is illustrated in the PERT chart of Fig. 11. Hence, packets entering the network during the period $[T_2 - D_n + d, T_2]$ are forwarded inconsistently. The inconsistency metric is equal to $I = max(D_n - d, 0)$. In a precise sense, the delay d is a knob for tuning the update inconsistency.

VII. EVALUATION

Our evaluation was performed on a 50-node testbed in the DeterLab [43], [44] environment. The nodes (servers) in the DeterLab testbed are interconnected by a user-configurable topology.

Each testbed node in our experiments ran a software-based OpenFlow switch that supports time-based updates, also known as *Scheduled Bundles* [11]. A separate machine was used as a controller, which was connected to the switches using an out-of-band control network.

The OpenFlow switches and controller we used are a version of OFSoftSwitch and Dpctl [45], respectively, that supports Scheduled Bundles [9]. We used REVERSEPTP [30], [31] to guarantee synchronized timing.

A. Experiment 1: Timed vs. Untimed Updates

We emulated a typical leaf-spine topology (e.g., [46]) of N switches, with $\frac{2N}{3}$ leaf switches, and $\frac{N}{3}$ spine switches (see Fig. 13). The experiments were run using various values of N, between 6 and 48 switches.

We measured the delay upper bounds, D_n, D_c, δ, and Δ. Table III presents the 99.9^{th} percentile delay values of each of these parameters. These are the parameters that were used in the controller's greedy updates.

We observed a low network delay D_n, as it was measured over two hops of a local area network. In Experiment 2 we analyze networks with a high network delay. Note that the values of δ and D_c were measured over software-based switches. Since hardware switches may yield different values, some of our experiments were performed with various synthesized values of δ and D_c, as discussed below. The measured value

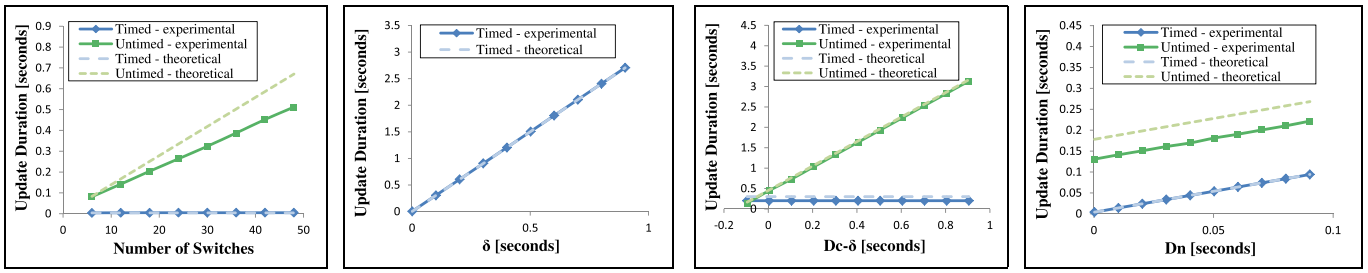

Fig. 12. Timed updates vs. untimed updates. Each figure shows the experimental values, and the theoretical worst-case values, based on Lemmas 3 and 5. (a) The update duration as a function of the number of switches. (b) The update duration as a function of the scheduling error, for $N = 12$. (c) The update duration as a function of $D_c - \delta$, for $N = 12$, $\delta = 100$ ms, various values of D_c. (d) The update duration as a function of the end-to-end network delay D_n, for $N = 12$.

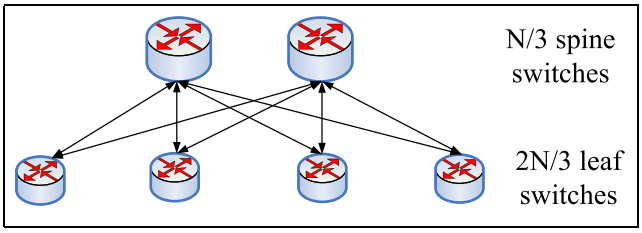

Fig. 13. Leaf-spine topology.

of Δ was high, on the order of 5 milliseconds, as Dpctl is not optimized for performance.

The experiments consisted of 3-phase updates of a policy rule: (i) a phase 1 update, involving all the switches, (ii) a phase 2 update, involving only the leaf (ingress) switches, and (iii) a garbage collection phase, involving all the switches.

Results: Fig. 12a compares the update duration of the timed and untimed approaches as a function of N. Untimed updates yield a significantly higher update duration, since they are affected by $(N_1+N_2+N_{G2}-3)\cdot\Delta$, per Lemma 3.[9] Hence, **the advantage of the timed approach increases with the number of switches** in the network, illustrating its scalability.

The impact of the scheduling error on the update duration in the timed approach is illustrated in Fig. 12b. As expected, the update duration grows linearly with δ, however, the update duration of the untimed approach is expected to be higher, as typically $\delta < D_c$.

Fig. 12c shows the update duration of the two approaches as a function of $D_c-\delta$, as we ran the experiment with synthesized values of δ and D_c. We fixed δ at 100 milliseconds, and tested various values of D_c. As expected (by Section V-D), the results show that for $D_c - \delta > 0$ the timed approach yields a lower update duration. Furthermore, only when the scheduling error, δ, is significantly higher than D_c does the untimed approach yield a shorter update duration. As discussed in Section IV-D, we typically expect $D_c - \delta$ to be positive, as δ is unaffected by high network delays, and thus we expect the timed approach to prevail. Interestingly, the results show that **even when the scheduling is not accurate**, e.g., if δ is 100 milliseconds worse than D_c, **the timed approach has a lower update duration**.

Fig. 12d illustrates the effect of the end-to-end network latency on the update duration. Both the timed and untimed approaches are linearly proportional to the network latency, following Lemmas 3 and 5. However, the timed approach allows a lower update duration, as it is not affected by N and Δ.

B. Experiment 2: Fine Tuning Consistency

The goal of this experiment was to study how time can be used to tune the level of inconsistency during updates. In order to experiment with real-life wide area network delay values, D_n, we performed the experiment using publicly available topologies.

Network Topology: Our experiments ran over three publicly available service provider network topologies [47], as illustrated in Fig. 14. We defined each node in the figure to be an OpenFlow switch. OpenFlow messages were sent to the switches by a controller over an out-of-band network (not shown in the figures).

Network Delays: The public information provided in [47] does not include the explicit delay of each path, but includes the coordinates of each node. Hence we derived the network delays from the beeline distance between each pair of nodes, assuming 5 microseconds per kilometer, as recommended in [48]. The DeterLab testbed allows a configurable delay value to be assigned to each link. We ran our experiments in two modes:

(i) Constant delay — each link had a constant delay that was configured to the value we computed as described above.

(ii) Exponential delay — each link had an exponentially distributed delay. The mean delay of each link in experiment (ii) was equal to the link delay of this link in experiment (i), allowing an 'apples to apples' comparison.

Test Flows: In each topology we ran five test flows, and measured the inconsistency during a timed network update. Each test flow was injected by an external source (see 14) to one of the ingress switches, forwarded through the network, and transmitted from an egress switch to an external destination. Test flows were injected at a fixed rate of 40 Mbps using Iperf [49].

[9]The slope of the untimed curve in Fig. 12a is Δ, by Lemma 3. The theoretical curve was computed based on the 99.9^{th} percentile value, whereas the mean value in our experiment was about 20% lower, explaining the different slopes of the theoretical and experimental curves.

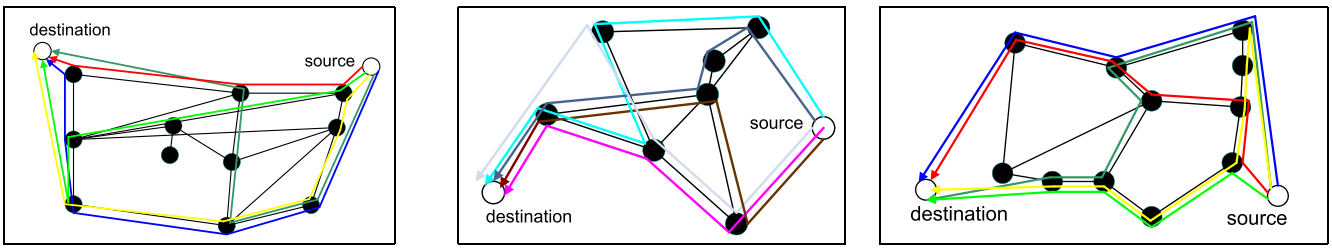

Fig. 14. Publicly available topologies [47] used in our experiments. Each path of the test flows in our experiment is depicted by a different color. Black nodes are OpenFlow switches. White nodes represent the external source and destination of the test flows in the experiment. (a) Sprint topology. (b) NetRail topology. (c) Compuserve topology.

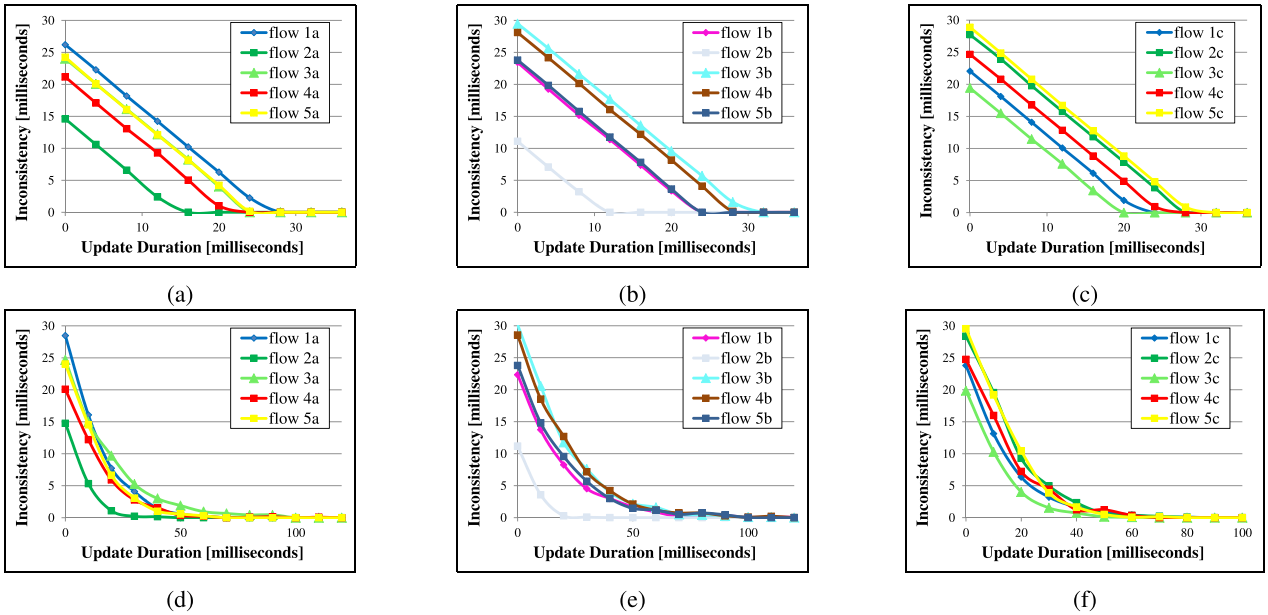

Fig. 15. Inconsistency as a function of the update duration. Modifying the update duration controls the degree of inconsistency. Two graphs are shown for each of the three topologies: exponential delay, constant delay. (a) Sprint - constant network delay. (b) NetRail - constant network delay. (c) Compuserve - constant network delay. (d) Sprint - exponential network delay. (e) NetRail - exponential network delay. (f) Compuserve - exponential network delay.

Network Updates: We performed *two-phase* updates of a Multiprotocol Label Switching (MPLS) label; a flow is forwarded over an MPLS Label-Switched Path (LSP) with label A, and then reconfigured to use label B. A garbage collection phase was used to remove the entries of label A. Conveniently, the MPLS label was also used as the version tag in the *two-phase* updates.

Inconsistency Measurement: For every test flow f, and update U, we measure the number of inconsistent packets during the update $n(f, U)$. Inconsistent packets in our context are either packets with a 'new' label arriving to a switch without the 'new' rule, or packets with an 'old' label arriving to a switch without the 'old' rule. We used the switches' OpenFlow counters to count the number of inconsistent packets, $n(f, U)$. We compute the inconsistency of each update using Eq. 12.

Results: We measured the inconsistency I during each update as a function of the update duration, $T_{g_1} - T_1$ (see Fig. 9). We repeated the experiment for each of the topologies and each of the test flows of Fig. 14.

The results are illustrated in Fig. 15. The figure depicts the tradeoff between the update duration, and the inconsistency during the update. A long update duration bares a cost on the switches' expensive memory resources, whereas a high degree of inconsistency implies a large number of dropped or misrouted packets.

Using a timed update, it is possible to tune the difference $T_{g_1} - T_1$, directly affecting the degree of inconsistency. An SDN programmer can tune $T_{g_1} - T_1$ to the desired sweet spot based on the system constraints; if switch memory resources are scarce, one may reduce the update duration and allow some inconsistency.

As illustrated in Fig. 15d, 15e, and 15f, this fine tuning is especially useful when the network latency has a long-tailed distribution. A truly consistent update, where $I = 0$, requires a very long and costly update duration. As shown in the figures, by slightly compromising I, the switch memory overhead during the update can be cut in half.

C. Simulation: Using ACKs

In this simulation we analyzed a *two-phase* hybrid approach that uses acknowledgments for the first two phases, and a timed garbage collection phase.

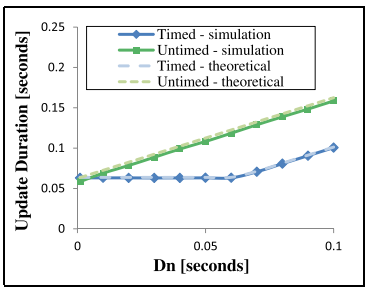

Fig. 16. Update duration of the garbage collection phase.

As discussed in Section IV-B, explicit acknowledgment is currently not supported by OpenFlow, and thus we could not experiment with ACKs in the OpenFlow-based testbed that was used in Experiments 1 and 2. Therefore, we used a simulation to evaluate the usage of ACKs. We assumed the topology of Experiment 1 with $N = 12$, and used the measured values of D_c, δ, and Δ, as presented in Table III. The simulation was implemented in Visual Basic.

We simulated the garbage collection phase of a *two-phase* update, as depicted in Fig. 10, and compared the update duration in the timed and untimed cases for various values of D_n. We defined T_{su} to be $(N_{G2} - 1) \cdot \Delta + D_c$, allowing enough time for the timed update messages to propagate from the controller to the switches. Fig. 16 depicts the duration of the garbage collection phase in the timed and in the untimed approaches. The theoretical values in the figure are based on Eq. 10 and 11. As illustrated in the graph, if D_n is higher than a few milliseconds, then the timed approach yields a significantly lower update duration, illustrating the advantage of a timed garbage collection phase.

VIII. Discussion

A. Failures

Switch failures during an update procedure may compromise the consistency during an update. For example, a switch may silently fail to perform an update, thereby causing inconsistency. Both the timed and untimed update approaches may be affected by failure scenarios. The OpenFlow Scheduled Bundle [11] mechanism provides an elegant mechanism for mitigating failures in timed updates; if the controller detects a switch failure *before* an update is scheduled to take place, it can send a cancellation message to all the switches that take part in the scheduled update, thus guaranteeing an *all-or-none* behavior.

B. Security Considerations

Security threats in SDNs and possible mitigations have been discussed in previous work (e.g., [50]–[52]). In the context of the current paper, other security threats may arise from the use of timed updates; A man-in-the-middle attacker can selectively delay control plane messages between the controller and switches, thereby preventing timed updates from being performed at their scheduled time. This attack can be mitigated by using an explicit acknowledgment mechanism, as discussed in Section IV-B. Since timed updates rely on a time synchronization protocol, an attack on the time protocol can compromise the timed update mechanism. Such attacks can be mitigated by securing the time protocol [53].

IX. Conclusion

Accurate time synchronization has become a common feature in commodity switches and routers. We have shown that it can be used to implement consistent updates in a way that reduces the update duration and the expensive overhead of maintaining duplicate configurations. Moreover, we have shown that accurate time can be used to tune the fine tradeoff between consistency and scalability during network updates. Our experimental evaluation demonstrates that timed updates allow scalability that would not be possible with conventional update methods.

Acknowledgments

The authors gratefully acknowledge the DeterLab project [43] for the opportunity to perform their experiments on the DeterLab testbed.

References

[1] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res. (SOSR)*, 2015, Art. ID 21.
[2] 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, Dec. 2007.
[3] X. Jin *et al.*, "Dynamic scheduling of network updates," in *Proc. ACM SIGCOMM*, 2014, pp. 539–550.
[4] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide IGP migrations," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 314–325, Aug. 2011.
[5] H. H. Liu *et al*. "zUpdate: Updating data center networks with zero loss," in *Proc. ACM SIGCOMM*, 2013, pp. 411–422.
[6] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, 2012, pp. 323–334.
[7] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 49–54.
[8] T. Mizrahi and Y. Moses, "Time-based updates in software defined networks," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 163–164.
[9] T. Mizrahi and Y. Moses, "Software defined networks: It's about time," in *Proc. IEEE INFOCOM*, Apr. 2016.
[10] *OpenFlow Switch Specification, Version 1.4.0*, Open Networking Foundation, Palo Alto, CA, USA, 2013.
[11] *OpenFlow Switch Specification, Version 1.5.0*, Open Networking Foundation, Palo Alto, CA, USA, 2015.
[12] *OpenFlow Extensions 1.3.x Package 2*, Open Networking Foundation, Palo Alto, CA, USA, 2015.
[13] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Standard 1588-2008, 2008.
[14] H. Li, "IEEE 1588 time synchronization deployment for mobile backhaul in China mobile," presented at the IEEE ISPCS, 2014.
[15] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags," in *Proc. NSDI*, 2014, pp. 543–546.
[16] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust SDN control plane for transactional network updates," in *Proc. IEEE INFOCOM*, Apr./May 2015, pp. 190–198.
[17] Z. Guo *et al.*, "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Comput. Netw.*, vol. 68, pp. 95–109, Aug. 2014.

[18] X. Wen *et al.*, "Compiling minimum incremental update for modular SDN languages," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 193–198.
[19] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[20] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 2, pp. 254–280, Apr. 1984. [Online]. Available: http://doi.acm.org/10.1145/2993.2994
[21] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *Proc. OSDI*, vol. 1. 2012, pp. 1–14.
[22] K. Harris, "An application of IEEE 1588 to industrial automation," in *Proc. IEEE ISPCS*, Sep. 2008, pp. 71–76.
[23] (2012). *IEEE, Time-Sensitive Networking Task Group*. [Online]. Available: http://www.ieee802.org/1/pages/tsn.html
[24] P. Moreira, J. Serrano, T. Wlostowski, P. Loschmidt, and G. Gaderer, "White rabbit: Sub-nanosecond timing distribution over Ethernet," in *Proc. IEEE ISPCS*, Oct. 2009, pp. 1–5.
[25] G. R. Ash, "Use of a trunk status map for real-time DNHR," in *Proc. Int. Teletraffic Congr. (ITC)*, 1985, pp. 1–2.
[26] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula, "Calendaring for wide area networks," in *Proc. ACM SIGCOMM*, 2014, pp. 515–526.
[27] A. Greenberg *et al.*, "A clean slate 4D approach to network control and management," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 5, pp. 41–54, 2005.
[28] T. Mizrahi and Y. Moses, "On the necessity of time-based updates in SDN," in *Proc. Open Netw. Summit (ONS)*, 2014.
[29] T. Mizrahi, O. Rottenstreich, and Y. Moses, "TimeFlip: Scheduling network updates with timestamp-based TCAM ranges," in *Proc. IEEE INFOCOM*, Apr./May 2015, pp. 2551–2559.
[30] T. Mizrahi and Y. Moses, "Using ReversePTP to distribute time in software defined networks," in *Proc. IEEE ISPCS*, Sep. 2014, pp. 112–117.
[31] T. Mizrahi and Y. Moses, "ReversePTP: A software defined networking approach to clock synchronization," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 203–204.
[32] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. ACM*, vol. 32, no. 1, pp. 52–78, Jan. 1985.
[33] A. Mukherjee, "On the dynamics and significance of low frequency components of Internet load," Dept. Comput. Inf. Sci., Univ. Pennsylvania, Philadelphia, PA, USA, Tech. Rep. MS-CIS-92-83, 1992, p. 300.
[34] O. Gurewitz, I. Cidon, and M. Sidi, "One-way delay estimation using network-wide measurements," *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 2710–2724, Jun. 2006.
[35] (2014). *PingER*. [Online]. Available: http://pinger.fnal.gov/
[36] (2014). *AMP Measurements*. [Online]. Available: http://erg.wand.net.nz
[37] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation," in *Passive and Active Measurement*. Springer, 2012, pp. 85–95.
[38] *Ethernet Services Attributes—Phase 3*, document MEF 10.3, Metro Ethernet Forum, 2013.
[39] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.
[40] Network Test Inc. (2010). *Virtual Chassis Performance: Juniper Networks EX Series Ethernet Switches*. [Online]. Available: http://www.networktest.com/
[41] *IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management*, IEEE Standard 802.1ag, 2007.
[42] D. G. Malcolm, J. H. Roseboom, C. E. Clark, and W. Fazar, "Application of a technique for research and development program evaluation," *Oper. Res.*, vol. 7, no. 5, pp. 646–669, 1959.
[43] (2015). *The DeterLab Project*. [Online]. Available: http://deter-project.org/about_deterlab
[44] J. Mirkovic and T. Benzel, "Teaching cybersecurity with DeterLab," *IEEE Security Privacy*, vol. 10, no. 1, pp. 73–76, Jan./Feb. 2012.
[45] (2014). *CPqD OFSoftswitch*. [Online]. Available: https://github.com/CPqD/ofsoftswitch13
[46] Cisco. (2010). *Cisco's Massively Scalable Data Center*. [Online]. Available: http://www.cisco.com/c/dam/en/us/td/docs/solutions/Enterprise/Data_Center/MSDC/1-0/MSDC_AAG_1.pdf
[47] (2015). *Topology Zoo*. [Online]. Available: http://topology-zoo.org/
[48] *One-Way Transmission Time*, document ITU-T G.144, ITU-T, 2003.
[49] (2014). *Iperf—The TCP/UDP Bandwidth Measurement Tool*. [Online]. Available: https://iperf.fr/
[50] D. Kreutz, F. M. V. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 55–60.
[51] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *Proc. IEEE SDN Future Netw. Services (SDN4FNS)*, Nov. 2013, pp. 1–7.
[52] M. Ambrosin, M. Conti, F. D. Gaspari, and R. Poovendran, "LineSwitch: Efficiently managing switch flow in software-defined networking while effectively tackling DoS attacks," in *Proc. ACM Symp. Inf., Comput. Commun. Secur. (ASIA CCS)*, 2015, pp. 639–644.
[53] T. Mizrahi, *Security Requirements of Time Protocols in Packet Switched Networks*, document RFC 7384, IETF, 2014.

Tal Mizrahi is currently pursuing the Ph.D. degree with Technion–Israel Institute of Technology. He is also a Switch Architect at Marvell, with over 15 years of experience in networking. He is an active participant in the Internet Engineering Task Force and the IEEE 1588 Working Group. His research interests include network protocols, switch and router architecture, time synchronization, and distributed systems.

Efi Saat is currently pursuing the M.S. degree with the the Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology. His interests include router design, chip design, and network security.

Yoram Moses is currently the Israel Pollak Academic Chair and a Professor of Electrical Engineering with Technion–Israel Institute of Technology. His research focuses on distributed and multiagent systems, with a focus on fault-tolerance and on the applications of knowledge and time in such systems. He has co-authored the book entitled *Reasoning About Knowledge*, and was a recipient of the Gödel Prize in 1997 and the Dijkstra Prize in 2009.