# Software Defined Networks: It's About Time

Tal Mizrahi, Yoram Moses*
Technion — Israel Institute of Technology
Email: {dew@tx, moses@ee}.technion.ac.il

***Abstract*—With the rise of Software Defined Networks (SDN), there is growing interest in dynamic and centralized traffic engineering, where decisions about forwarding paths are taken dynamically from a network-wide perspective. Frequent path reconfiguration can significantly improve the network performance, but should be handled with care, so as to minimize disruptions that may occur during network updates.**

**In this paper we introduce TIME4, an approach that uses accurate time to coordinate network updates. We characterize a set of update scenarios called *flow swaps*, for which TIME4 is the optimal update approach, yielding less packet loss than existing update approaches. We define the *lossless flow allocation problem*, and formally show that in environments with frequent path allocation, scenarios that require simultaneous changes at multiple network devices are inevitable.**

**We present the design, implementation, and evaluation of a TIME4-enabled OpenFlow prototype. The prototype is publicly available as open source. Our work includes an extension to the OpenFlow protocol that has been adopted by the Open Networking Foundation (ONF), and is now included in OpenFlow 1.5. Our experimental results demonstrate the significant advantages of TIME4 compared to other network update approaches.**

*Time is what keeps everything from happening at once*
*– Ray Cummings*

## I. INTRODUCTION

### A. It's About Time

The use of synchronized clocks was first introduced in the $19^{th}$ century by the Great Western Railway company in Great Britain. Clock synchronization has significantly evolved since then, and is now a mature technology that is being used by various different applications, from mobile backhaul networks [1] to distributed databases [2].

The Precision Time Protocol (PTP), defined in the IEEE 1588 standard [3], can synchronize clocks to a very high degree of accuracy, typically on the order of 1 microsecond [1], [4], [5]. PTP is a common and affordable feature in commodity switches. Notably, 9 out of the 13 SDN-capable switch silicons listed in the Open Networking Foundation (ONF) SDN Product Directory [6] have native IEEE 1588 support.

In this work we introduce TIME4, a **generic** tool for using time in SDN. One of the products of this work is a new feature that enables timed updates in OpenFlow, and has been incorporated in OpenFlow 1.5. Furthermore, we present a class of update scenarios in which the use of accurate time is provably optimal, while existing update methods are sub-optimal.

*Yoram Moses is the Israel Pollak academic chair at Technion.

### B. The Challenge of Dynamic Traffic Engineering in SDN

Defining network routes dynamically, based on a complete view of the network, can significantly improve the network performance compared to the use of distributed routing protocols. SDN and OpenFlow [7], [8] have been leading trends in this context, but several other ongoing efforts offer similar concepts (e.g., [9]).

Centralized network updates often involve multiple network devices. Hence, updates must be performed in a way that strives to minimize temporary anomalies such as traffic loops, congestion, or disruptions, which may occur during transient states where the network has been partially updated.

While SDN was originally considered in the context of campus networks [7] and data centers [10], it is now also being considered for Wide Area Networks (WANs) [11], [12], carrier networks, and mobile backhaul networks [13].

WAN and carrier-grade networks require a very low packet loss rate. Carrier-grade performance is often associated with the term *five nines*, representing an availability of 99.999%. Mobile backhaul networks require a Frame Loss Ratio (FLR) of no more than $10^{-4}$ for voice and video traffic, and no more than $10^{-3}$ for lower priority traffic [14]. Other types of carrier network applications, such as storage and financial trading require even lower loss rates [15], on the order of $10^{-5}$.

Several recent works have explored the realm of dynamic path reconfiguration, with frequent updates on the order of minutes [11], [12], [16], enabled by SDN. Interestingly, for voice and video traffic, a frame loss ratio of up to $10^{-4}$ implies that service must not be disrupted for more than 6 milliseconds per minute. Hence, if path updates occur on a per-minute basis, then transient disruptions must be limited to a short period of no more than a few milliseconds.

### C. Timed Network Updates

We explore the use of *accurate time* as a tool for performing coordinated network updates in a way that minimizes packet loss. We introduce TIME4, which is an update approach that performs multiple changes at different switches at the same time.

**Example 1.** *Fig. 1 illustrates a* flow swapping *scenario. In this scenario, the forwarding paths of two flows, $f_1$ and $f_2$, need to be reconfigured, as illustrated in the figure. It is assumed that all links in the network have an identical capacity of 1 unit, and that both $f_1$ and $f_2$ require a bandwidth of 1 unit. In the presence of accurate clocks, by scheduling $S_1$ and $S_3$ to update*

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

*their paths at the same time, there is no congestion during the update procedure, and the reconfiguration is smooth. As clocks will typically be reasonably well synchronized, albeit not perfectly synchronized, such a scheme will result in a very short period of congestion.*

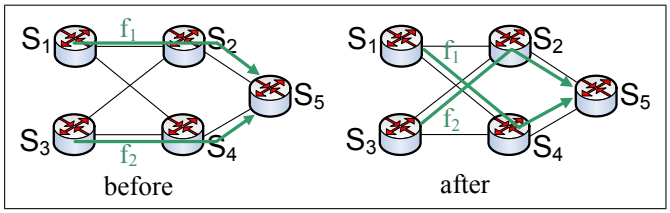



Fig. 1: Flow Swapping—Flows need to convert from the "before" configuration to the "after".

In this paper we show that in a dynamic environment, where flows are frequently added, removed or rerouted, flow swaps are inevitable. A notable example of the importance of flow swaps is a recently published work by Fox Networks [17], in which accurately timed flow swaps are essential in the context of video switching.

One of our key results is that simultaneous updates are the optimal approach in scenarios such as Example 1, whereas other update approaches may yield considerable packet loss, or incur higher resource overhead. Note that such packet loss can be reduced either by increasing the capacity of the communication links, or by increasing the buffer memories in the switches. We show that for a given amount of resources, TIME4 yields lower packet loss than other approaches.

**Accuracy** is a key requirement in TIME4; since updates cannot be applied at the exact same instant at all switches, they are performed within a short time interval called the *scheduling error*. The experiments we present in Section IV show that the scheduling error in **software** switches is on the order of 1 millisecond. The TCAM-based **hardware** solution of [18] can execute scheduled events in existing switches with an accuracy **on the order of 1 microsecond**.

### D. Related Work

Various network update approaches have been analyzed in the literature. A common approach is to use a sequence of configuration commands [16], [19]–[21], whereby the **order** of execution guarantees that no anomalies are caused in intermediate states of the procedure. However, as observed by [16], in some update scenarios, known as **deadlocks**, there is no order that guarantees a consistent transition. ***Two-phase*** updates [22] use configuration version tags to guarantee consistency during updates. However, as per [22], *two-phase* updates cannot guarantee congestion freedom, and are therefore not effective in flow swap scenarios, such as Fig. 1. Hence, in flow swap scenarios the *order* approach and the *two-phase* approach produce the same result as the simple-minded approach, in which the controller sends the update commands as close as possible to instantaneously, and hopes for the best.

In this paper we present TIME4, an update approach that is most effective in flow swaps and other deadlock [16] scenarios, such as Fig. 1. We refer to update approaches that do not use time as **untimed** update approaches.

In SWAN [11], the authors suggest that reserving unused *scratch* capacity of 10-30% on every link can allow congestion-free updates in most scenarios. The B4 [12] approach prevents packet loss during path updates by temporarily reducing the bandwidth of some or all of the flows. Our approach does not require scratch capacity, and does not reduce the bandwidth of flows during network updates. Furthermore, in this paper we show that variants of SWAN and B4 that make use of TIME4 can perform better than the original versions.

Rearrangeably non-blocking topologies (e.g., [23]) allow new traffic flows to be added to the network by rearranging existing flows. The analysis of flow swaps presented in this paper emphasizes the requirement to perform *simultaneous* reroutes during the rearrangement procedure, an aspect which has not been previously studied.

Preliminary work-in-progress versions of the current paper introduced the concept of using time in SDN [24] and the flow swapping scenario [25]. The use of time for *consistent* updates was discussed in [26]. TimeFlip [18] presented a practical method of implementing timed updates. The current work is the first to present a generic protocol for performing timed updates in SDN, and the first to analyze *flow swaps*, a natural application in which timed updates are the optimal update approach.

### E. Contributions

The main contributions of this paper are as follows:

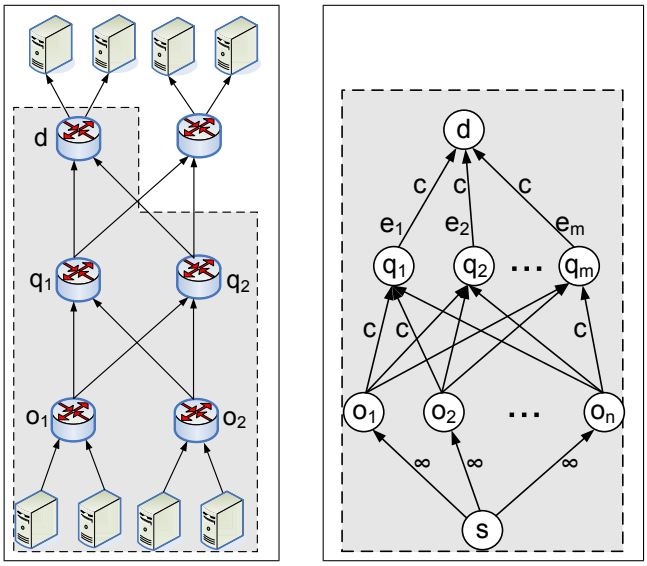
- We consider a class of network update scenarios called *flow swaps*, and show that simultaneous updates using synchronized clocks are provably the optimal approach of implementing them. In contrast, existing approaches for consistent updates (e.g., [16], [22]) are not applicable to flow swaps, and other update approaches such as SWAN [11] and B4 [12] can perform flow swaps, but at the expense of increased resource overhead.
- We present the design, implementation and evaluation of a prototype that performs timed updates in OpenFlow.
- Our work includes an extension to the OpenFlow protocol that has been approved by the ONF and integrated into OpenFlow 1.5 [27], and into the OpenFlow 1.3.x extension package [28]. The source code of our prototype is publicly available [29].
- We present experimental results that demonstrate the advantage of timed updates over existing approaches. Moreover, we show that existing update approaches (SWAN and B4) can be improved by using accurate time.

Due to space limits, some of the proofs and experimental results are presented in [30].

## II. THE LOSSLESS FLOW ALLOCATION (LFA) PROBLEM

### A. Inevitable Flow Swaps

Fig. 1 presents a scenario in which it is necessary to *swap* two flows, i.e., to update two switches at the same time. In this section we discuss the inevitability of flow swaps; we




(a) Clos network. (b) Unsplittable flow graph.

Fig. 2: Modeling a Clos topology as an unsplittable flow graph.

show that there does not exist a controller routing strategy that avoids the need for flow swaps.

Our analysis is based on representing the flow-swap problem as an instance of an unsplittable flow problem, as illustrated in Fig. 2b. The topology of the graph in Fig. 2b models the traffic behavior to a given destination in common multi-rooted network topologies such as fat-tree and Clos (Fig. 2a).

The unsplittable flow problem [31] has been thoroughly discussed in the literature; given a directed graph, a source node $s$, a destination node $d$, and a set of flow demands (commodities) between $s$ and $d$, the goal is to maximize the traffic rate from the source to the destination. In this paper we define a *game* between two players: a **source**[1] that generates traffic flows (commodities) and a **controller** that reconfigures the network forwarding rules in a way that allows the network to forward all traffic generated by the source without packet losses.

Our main argument, phrased in Theorem 1, is that the source has a strategy that **forces** the controller to perform a flow swap, i.e., to reconfigure the path of two or more flows at the same time. Thus, a scenario in which multiple flows must be updated at the **same time** is inevitable, implying the importance of timed updates.

Moreover, we show that the controller can be forced to invoke $n$ individual commands that should optimally be performed at the same time. Update approaches that do not use time, also known as **untimed** approaches, cause the updates to be performed over a long period of time, potentially resulting in slow and possibly erratic response times and significant packet loss. Timed coordination allows us to perform the $n$ updates within a short time interval that depends on the scheduling error.

[1]The source player does not represent a malicious attacker; it is an 'adversary', representing the worst-case scenario.

Although our analysis focuses on the topology of Fig.2b, it can be shown that the results are applicable to other topologies as well, where the source can force the controller to perform a swap over the edges of the min-cut of the graph.

### B. Model and Definitions

We now introduce the *lossless flow allocation (LFA)* problem; it is not presented as an optimization problem, but rather as a game between two players: a **source** and a **controller**. As the source adds or removes flows (commodities), the controller reconfigures the forwarding rules so as to guarantee that all flows are forwarded without packet loss. **The controller's goal** is to find a forwarding path for all the flows in the system without exceeding the capacity of any of the edges, i.e., to completely avoid loss of packets from the given flows. **The source's goal** is to progressively add flows, without exceeding the network's capacity, forcing the controller to perform a flow swap. We shall show that the source has a strategy that forces the controller to swap traffic flows simultaneously in order to avoid packet loss.

Our model makes three basic assumptions: (i) each flow has a **fixed bandwidth**, (ii) the controller strives to **avoid packet loss**, and (iii) flows are **unsplittable**. We discuss these assumptions further in Sec. V.

We use the term *flow* in its OpenFlow sense, i.e., a set of packets that share common properties, such as source and destination network addresses. A flow in our context, can be seen as a session between the source and destination that runs traffic at a fixed rate.

The network is represented by a directed weighted acyclic graph (Fig. 2b), $G = (\mathbb{V}, E, c)$, with a source $s$, a destination $d$, and a set of intermediate nodes, $\mathbb{V}_{in}$. Thus, $\mathbb{V} = \mathbb{V}_{in} \cup \{s, d\}$. The nodes directly connected to $s$ are denoted by $\mathbb{O} = \{o_1, o_2, \ldots, o_n\}$. Each of the outgoing edges from the source $s$ has an infinite capacity, whereas the rest of the edges have a capacity $c$. For the sake of simplicity, and without loss of generality, throughout this section we assume that $c = 1$. Such a graph $G$ is referred to as an *LFA graph*.

The source node progressively transmits traffic flows towards the destination node. Each flow represents a session between $s$ and $d$; every flow has a constant bandwidth, and cannot be split between two paths. A centralized controller configures the forwarding policy of the intermediate nodes, determining the path of each flow. Given a set of flows from $s$ to $d$, the controller's goal is to configure the forwarding policy of the nodes in a way that allows all flows to be forwarded to $d$ without exceeding the capacity of any of the edges.

The set of flows that are generated by $s$ is denoted by $\mathbb{F} ::= \{F_1, F_2, \ldots, F_k\}$. Each flow $F_i$ is defined as $F_i ::= (i, f_i, r_i)$, where $i$ is a unique flow index, $f_i$ is the bandwidth satisfying $0 < f_i \leq c$, and $r_i$ denotes the node that the controller forwards the flow to, i.e., $r_i \in \{o_1, o_2, \ldots, o_n\}$.

It is assumed that the controller monitors the network, and thus it is aware of the flow set $\mathbb{F}$. The controller maintains a forwarding function, $R_{con} : \mathbb{F} \times \mathbb{V}_{in} \longrightarrow \mathbb{V}_{in} \cup \{d\}$. Every node (switch) has a flow table, consisting of a set of *entries*;

an element $v \in \mathbb{F} \times \mathbb{V}_{in}$ is referred to as an *entry* for short. An update of $R_{con}$ is defined to be a partial function $u : \mathbb{F} \times \mathbb{V}_{in} \rightharpoonup \mathbb{V}_{in} \cup \{d\}$. We define a *reroute* as an update $u$ that has a single entry in its domain. We call an update that has more than one entry in its domain a *swap*, and it is assumed that all updates in a *swap* are performed at the same time. We define a $k$-swap for $k \geq 2$ as a swap that updates entries in at least $k$ different nodes. Note that a $k$-swap is possible only if $n \geq k$, where $n$ is the number of nodes in $\mathbb{O}$. We focus our analysis on 2-swaps, and throughout the section we assume that $n \geq 2$. In Section II-E we discuss $k$-swaps for values of $k > 2$.

### C. The LFA Game

The lossless flow allocation problem can be viewed as a game between two players, the source and the controller. The game proceeds by a sequence of steps; in each step the source either adds or removes a single flow (Fig. 3), and then waits for the controller to perform a sequence of updates (Fig. 4). The source's strategy $\mathbb{S}_s(\mathbb{F}, R_{con}) = (a, F)$, is a function that defines for each flow set $\mathbb{F}$ and forwarding function $R_{con}$ for $\mathbb{F}$, a pair $(a, F)$ representing the source's next step, where $a \in \{\textit{Add}, \textit{Remove}\}$ is the action to be taken by the source, and $F = (j, f_j, r_j)$ is a single flow to be added or removed. The controller's strategy is defined by $\mathbb{S}_{con}(R_{con}, a, F) = \mathbb{U}$, where $\mathbb{U} = \{u_1, \ldots, u_\ell\}$ is a sequence of updates, such that (i) at the end of each update no edge exceeds its capacity, and (ii) at the end of the last update, $u_\ell$, the forwarding function $R_{con}$ defines a forwarding path for all flows in $\mathbb{F}$. Notice that when a flow is to be removed, the controller's update is trivial; it simply removes all the relevant entries from the domain of $R_{con}$. Hence our analysis focuses on *adding* new flows.

The following theorem, which is the crux of this section, argues that the source has a strategy that forces the controller to perform a swap, and thus that flow swaps are inevitable from the controller's perspective.

**Theorem 1.** *Let $G$ be an LFA graph. In the LFA game over $G$, there exists a strategy, $\mathbb{S}_s$, for the source that forces every controller strategy, $\mathbb{S}_{con}$, to perform a 2-swap.*

*Proof.* Let $m$ be the number of incoming edges to the destination node $d$ in the LFA graph (see Fig 2b). For $m = 1$ the claim is trivial. Hence, we start by proving the claim for $m = 2$, i.e., there are two edges connected to node $d$, edges $e_1$ and $e_2$.

SOURCE PROCEDURE

```
1  F ← ∅
2  repeat at every step
3    (a, F) ← S_s(F, R_con)
4    if a = Add
5      F ← F ∪ F
6      Wait for the controller to complete updates
7    else // a = Remove
8      F ← F \ F
```

Fig. 3: The LFA game: the source's procedure.

CONTROLLER PROCEDURE

```
1  repeat at every step
2    {u_1, ..., u_ℓ} ← S_con(R_con, a, F)
3    for j ∈ [1, ℓ]
4      Update R_con according to u_j
```

Fig. 4: The LFA game: the controller's procedure.

We show that the source has a strategy that, regardless of the controller's strategy, forces the controller to use a swap. In the first four steps of the game, the source generates four flows, $F_1 = (1, 0.35, o_1)$, $F_2 = (2, 0.35, o_1)$, $F_3 = (3, 0.45, o_2)$, and $F_4 = (4, 0.45, o_2)$, respectively. According to the Source Procedure of Fig. 3, after each flow is added, the source waits for the controller to update $R_{con}$ before adding the next flow. After the flows are added, there are two possible cases:

(a) The controller routes symmetrically through $e_1$ and $e_2$, i.e. a flow of $0.35$ and a flow of $0.45$ through each of the edges. In this case the source's strategy at this point is to generate a new flow $F_5 = (5, 0.3, o_1)$ with a bandwidth of $0.3$. The only way the controller can accommodate $F_5$ is by routing $F_1$ and $F_2$ through the same edge, allowing the new $0.3$ flow to be forwarded through that edge. Since there is no sequence of *reroute* updates that allows the controller to reach the desired $R_{con}$, the only way to reach a state where $F_1$ and $F_2$ are routed through the same edge is to swap a $0.35$ flow with a $0.45$ flow. Thus, by issuing $F_5$ the controller forces a flow swap as claimed.

(b) The controller routes $F_1$ and $F_2$ through one edge, and $F_3$ and $F_4$ through the other edge. In this case the source's strategy is to generate two flows, $F_6$ and $F_7$, with a bandwidth of $0.2$ each. The controller must route $F_6$ through the edge with $F_1$ and $F_2$. Now each path sustains a bandwidth of $0.9$ units. Thus, when $F_7$ is added by the source, the controller is forced to perform a swap between one of the $0.35$ flows and one of the $0.45$ flows.

In both cases the controller is forced to perform a 2-swap, swapping a flow from $o_1$ with a flow from $o_2$. This proves the claim for $m = 2$.

The case of $m > 2$ is obtained by reduction to $m = 2$: the source first generates $m-2$ flows with a bandwidth of 1 each, causing the controller to saturate $m - 2$ edges connected to node $d$ (without loss of generality $e_3, \ldots, e_m$). At this point there are only two available edges, $e_1$ and $e_2$. From this point, the proof is identical to the case of $m = 2$. □

### D. The Impact of Flow Swaps

We define a **metric** for flow swaps, by considering the oversubscription that is caused if the flows are **not** swapped simultaneously, but updated using an untimed approach.

We define the *oversubscription* of an edge, $e$, with respect to a forwarding function, $R_{con}$, to be the difference between the total bandwidth of the flows forwarded through $e$ according to $R_{con}$, and the capacity of $e$. If the total bandwidth of the flows through $e$ is less than the capacity of $e$, the oversubscription is defined to be zero.

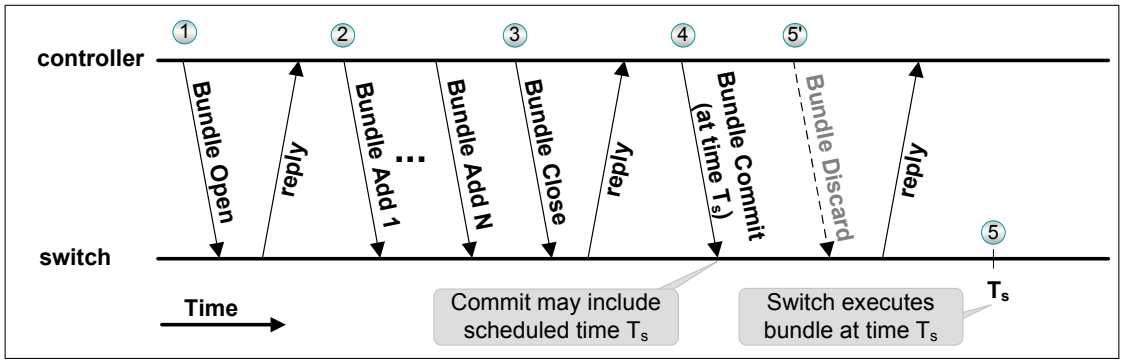



Fig. 5: A *Scheduled Bundle*: the *Bundle Commit* message may include $T_s$, the scheduled time of execution. The controller can use a *Bundle Discard* message to cancel the *Scheduled Bundle* before $T_s$.

**Definition 1** (Flow swap impact)**.** *Let $\mathbb{F}$ be a flow set, and $R_{con}$ be the corresponding forwarding function. Let $u : \mathbb{F} \times \mathbb{V} \rightharpoonup \mathbb{V} \cup \{d\}$ be a 2-swap, such that $u = u_1 \cup u_2$, where $u_i = (v_i, v_i)$, for $v_i \in \mathbb{F} \times \mathbb{V}$, $v_i \in \mathbb{V} \cup \{d\}$, and $i \in \{1, 2\}$. The* impact *of $u$ is defined to be the minimum of: (i) The oversubscription caused by applying $u_1$ to $R_{con}$, or (ii) the oversubscription caused by applying $u_2$ to $R_{con}$.*

The following theorem shows that in the LFA game, the source can force the controller to perform a flow swap with a swap impact of roughly $0.5$.

**Theorem 2.** *Let $G$ be an LFA graph, and let $0 < \alpha < 0.5$. In the LFA game over $G$, there exists a strategy, $\mathbb{S}_s$, for the source that forces every controller strategy, $\mathbb{S}_{con}$, to perform a swap with an impact of $\alpha$.*

The proof is presented in [30].

Intuitively, Theorem 2 shows that not only are flow swaps inevitable, but they have a high impact on the network, as they can cause links to be congested by roughly $50\%$ beyond their capacity.

### *E. n-Swaps*

As defined above, a $k$-swap is a swap that involves $k$ or more nodes. In previous subsections we discussed 2-swaps. The following theorem generalizes Theorem 1 to $n$-swaps, where $n$ is the number of nodes in $\mathbb{O}$.

**Theorem 3.** *Let $G$ be an LFA graph. In the LFA game over $G$, there exists a strategy, $\mathbb{S}_s$, for the source that forces every controller strategy, $\mathbb{S}_{con}$, to perform an $n$-swap.*

The proof is presented in [30].

## III. Design and Implementation

### *A. Protocol Design*

We present an extension that allows OpenFlow controllers to signal the time of execution of a command to the switches. This extension is described in full in [30].[2]

[2]A preliminary version of this extension was presented in [32].

Our extension makes use of the OpenFlow [8] **Bundle** feature; a Bundle is a sequence of OpenFlow messages from the controller that is applied as a single operation. Our time extension defines ***Scheduled Bundles***, allowing all commands of a Bundle to come into effect at a pre-determined time. This is a generic means to extend all OpenFlow commands with the scheduling feature.

Using Bundle messages for implementing TIME4 has two significant advantages: (i) It is a generic method to add the time extension to all OpenFlow commands without changing the format of all OpenFlow messages; only the format of Bundle messages is modified relative to the Bundle message format in [8], optionally incorporating an execution time. (ii) The Scheduled Bundle allows a relatively straightforward way to *cancel* scheduled commands, as described below.

Fig. 5 illustrates the *Scheduled Bundle* message procedure. In step 1, the controller sends a *Bundle Open* message to the switch, followed by one or more Add messages (step 2). Every *Add* message encapsulates an OpenFlow message, e.g., a *FLOW_MOD* message. A *Bundle Close* is sent in step 3, followed by the *Bundle Commit* (step 4), which optionally includes the scheduled time of execution, $T_s$. The switch then executes the desired command(s) at time $T_s$.

The *Bundle Discard* message (step 5′) allows the controller to enforce an all-or-none scheduled update; after the *Bundle Commit* is sent, if one of the switches sends an *error* message, indicating that it is unable to schedule the current Bundle, the controller can send a Discard message to all switches, canceling the scheduled operation. Hence, when a switch receives a scheduled commit, to be executed at time $T_s$, the switch can verify that it can dedicate the required resources to execute the command as close as possible to $T_s$. If the switch's resources are not available, for example due to another command that is scheduled to $T_s$, then the switch replies with an error message, aborting the scheduled commit. Significantly, this mechanism allows switches to execute the command with a guaranteed scheduling accuracy, avoiding the high variation that occurs when untimed updates are used.

The OpenFlow time extension also defines *Bundle Feature Request* messages, which allow the controller to query

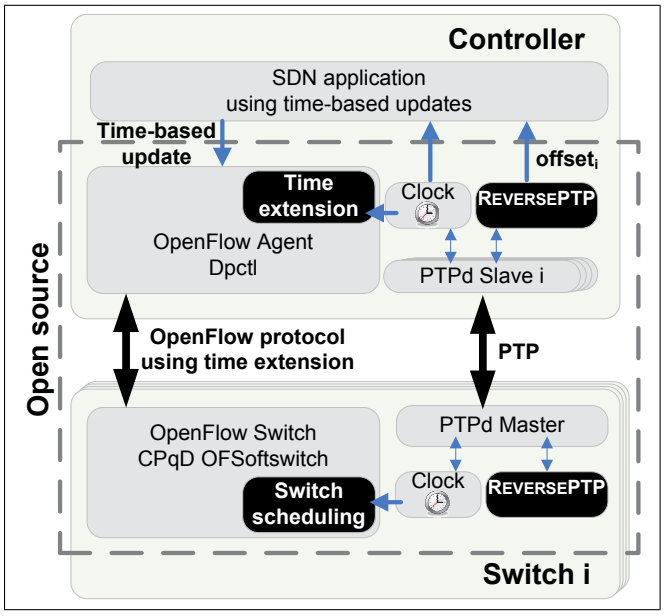



Fig. 6: TIME4 prototype design: the black blocks are the components implemented in the context of this work.

switches about whether they support Scheduled Bundles, and to configure some of the switch parameters related to *Scheduled Bundles*.

### B. Clock synchronization

TIME4 requires the switches and controller to maintain a local clock, enabling time-triggered events. Hence, the local clocks should be synchronized. The OpenFlow time extension we defined does not mandate a specific synchronization method. Various mechanisms may be used, e.g., the Network Time Protocol (NTP), the Precision Time Protocol (PTP) [3], or GPS-based synchronization. The prototype we designed and implemented uses REVERSEPTP [33], a variant of PTP that is customized for SDN.

### C. Prototype Design and Implementation

We have designed and implemented a software-based prototype of TIME4, as illustrated in Fig. 6. The components we implemented are marked in black. These components run on Linux, and are publicly available as open source [29].

Our TIME4-enabled OFSoftswitch prototype was adopted by the ONF as the official prototype of Scheduled Bundles.[3]

**Switches.** Every switch $i$ runs an OpenFlow switch software module. Our prototype is based on the open source CPqD OFSoftswitch [34],[4] incorporating the *switch scheduling* module (see Fig. 6) that we implemented. When the switch receives a *Scheduled Bundle* from the controller, the *switch scheduling* module schedules the respective OpenFlow command to the desired time of execution. The switch scheduling module also handles *Bundle Feature Request* messages received from the controller.

Each switch runs a REVERSEPTP master, which distributes the switch's time to the controller. Our REVERSEPTP prototype is a lightweight set of Bash scripts that is used as an abstraction layer over the well-known open source PTPd [35] module. Our software-based implementation uses the Linux clock as the reference for PTPd, and for the switch's scheduling module. To the best of our knowledge, ours is the first open source implementation of REVERSEPTP.

**Controller.** The controller runs an OpenFlow agent, which communicates with the switches using the OpenFlow protocol. Our prototype uses the CPqD Dpctl (Datapath Controller), which is a simple command line tool for sending OpenFlow messages to switches. We have extended Dpctl by adding the time extension; the Dpctl command-line interface allows the user to define the execution time of a *Bundle Commit*. Dpctl also allows a user to send a *Bundle Feature Request* to switches.

The controller runs REVERSEPTP with $n$ instances of PTPd in slave mode, where $n$ is the number of switches in the network. One or more SDN applications can run on the controller and perform timed updates. The application can extract the offset, $\text{offset}_i$, of every switch $i$ from REVERSEPTP, and use it to compute the scheduled execution time of switch $i$ in every timed update. The Linux clock is used as a reference for PTPd, and for the SDN application(s).

## IV. EVALUATION

### A. Evaluation Method

**Environment.** We evaluated our prototype on a 71-node testbed in the DeterLab [36] environment. Each machine (PC) in the testbed either played the role of an OpenFlow switch, running our TIME4-enabled prototype, or the role of a host, sending and receiving traffic. A separate machine was used as a controller, which was connected to the switches using an out-of-band network.

**Performance attributes.** Three performance attributes play a key role in our evaluation, as shown in Table I.

| | |
|---|---|
| $\Delta$ | The average time elapsed between two consecutive messages sent by the controller. |
| $I_R$ | Installation latency range: the difference between the maximal rule installation latency and the minimal installation latency. |
| $\delta$ | Scheduling error: the maximal difference between the actual update time and the scheduled update time. |

TABLE I: Performance Attributes.

Intuitively, $\Delta$ and $I_R$ determine the performance of untimed updates. $\Delta$ indicates the **controller**'s performance; an OpenFlow controller can handle as many as tens of thousands [37] to millions [38] of packets per second, depending on the type of controller and the machine's processing power. Hence, $\Delta$ can vary from 1 microsecond to several milliseconds. $I_R$ indicates the installation latency variation. The installation latency is the time elapsed from the instant the controller sends

[3]The ONF process for adding new features to OpenFlow requires every new feature to be prototyped.

[4]OFSoftswitch is one of the two software switches used by the Open Networking Foundation (ONF) for prototyping new OpenFlow features. We chose this switch since it was the first open source OpenFlow switch to include the Bundle feature.

a rule modification message until the rule has been installed. The installation latency of an OpenFlow rule modification (FLOW_MOD) has been shown to range from 1 millisecond to seconds [16], [39], and grows dramatically with the number of installations per second.

The attribute that affects the performance of **timed** updates is the switches' scheduling error, $\delta$. When an update is scheduled to be performed at time $T_0$, it is performed in practice at some time $t \in [T_0, T_0 + \delta]$.[5] The scheduling error, $\delta$, is affected by two factors: the device's *clock accuracy*, which is the maximal offset between the clock value and the value of an accurate time reference, and the *execution accuracy*, which is a measure of how accurately the device can perform a timed update, given run-time parameters such as the concurrently executing tasks and the load on the device. The achievable clock accuracy strongly depends on the network size and topology, and on the clock synchronization method. For example, the clock accuracy using the Precision Time Protocol [3] is typically on the order of 1 microsecond (e.g., [4]).

### B. Performance Attribute Measurement

Our experiments measured the three attributes, $\Delta$, $I_R$, and $\delta$, illustrating how accurately updates can be applied in software-based OpenFlow implementations. It should be noted that these three values depend on the processing power of the testbed machine; we measured the parameters for three types of DeterLab machines, Type I, II, and III, listed in Table II.

In software-based switches, the CPU handles both the data-plane traffic and the communication with the controller, and thus $I_R$ and $\delta$ can be affected by the rate of data-plane traffic through the switch. Hence, in our experiments we fixed the rate of traffic through each switch to 10 Mbps, allowing an 'apples-to-apples' comparison between experiments.

[5]An alternative representation of the accuracy, $\delta$, assumes a symmetric error, $T_0 \pm \delta$. The two approaches are equivalent.

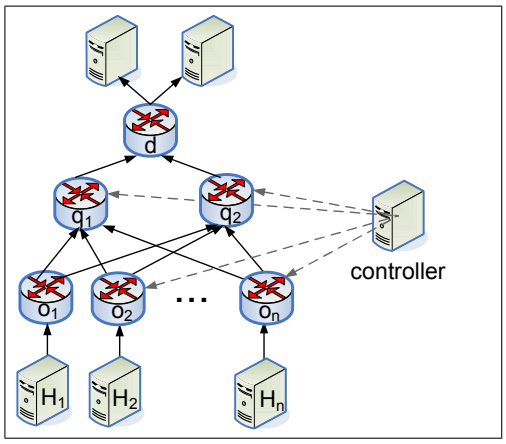



Fig. 7: Experimental evaluation: every host and switch was emulated by a Linux machine in the DeterLab testbed. All links have a capacity of 10 Mbps. The controller is connected to the switches by an out-of-band network.

| Machine Type | | $\Delta$ | $I_R$ | $\delta$ |
|---|---|---|---|---|
| I | Intel Xeon E3 LP 2.4 GHz, 16 GB RAM | 9.64 | 1.3 | 1.23 |
| II | Intel Xeon 2.1 GHz, 4 GB RAM | 9.6 | 1.47 | 1.18 |
| II | Intel Dual Xeon 3 GHz, 2 GB RAM | 14.27 | 2.72 | 1.19 |

TABLE II: Measured attributes in milliseconds.

### C. Flow Swap Evaluation

#### 1) Experiment Setting

We evaluated our prototype on a 71-node testbed under. We used the testbed to emulate an OpenFlow network with 32 hosts and 32 leaf switches, as depicted in Fig. 7, with $n = 32$.

**Metric.** A flow swap that is not performed in a coordinated way may bare a high cost: either packet loss, deep buffering, or a combination of the two. We use packet loss as a **metric** for the cost of flow swaps, assuming that deep buffering is not used.

We used Iperf to generate flows from the sources to the destination, and to measure the number of packets lost between the source and the destination.

**The flow swap scenario.** All experiments were flow swaps with a swap impact of $0.5$.[6] We used two static flows, which were not reconfigured in the experiment: $H_1$ generates a 5 Mbps flow that is forwarded through $q_1$, and $H_2$ generates a 5 Mbps flow that is generated through $q_2$. We generated $n$ additional flows (where $n$ is the number of switches at the bottom layer of the graph): (i) A 5 Mbps flow from $H_1$ to the destination. (ii) $n-1$ flows, each having a bandwidth of $\frac{5}{n-1}$ Mbps. Every flow swap in our experiment required the flow of (i) to be swapped with the $n-1$ flows of (ii). Note that this swap has an impact of $0.5$.

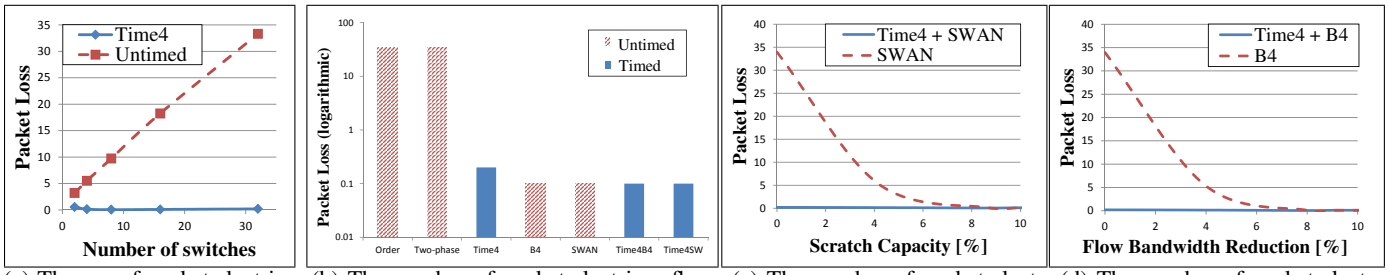
#### 2) Experimental Results

**TIME4 vs. other update approaches.** In this experiment we compared the packet loss of TIME4 to other update approaches described in Sec. I-D. As discussed in Sec. I-D, applying the *order* approach or the *two-phase* approach to flow swaps produces similar results. This observation is illustrated in Fig. 8b. In the rest of this section we refer to these two approaches collectively as the *untimed* approaches.

In our experiments we also implemented a SWAN-based [11] update, and a B4-based [12] update. In SWAN, we used a 10% scratch on each of the links, and in B4 updates we temporarily reduced the bandwidth of each flow by 10% to avoid packet loss. As depicted in Fig. 8b, SWAN and B4 yield a slightly lower packet loss rate than TIME4; the average number of packets lost in each TIME4 flow swap is $0.2$, while with SWAN and B4 only $0.1$ packets are lost on average.

To study the effect of using **time** in SWAN and in B4, we also performed hybrid updates, illustrated in Fig. 8c and 8d, and in the two right-most bars of Fig. 8b. We combined SWAN and TIME4, by performing a timed update on a network with scratch capacity, and compared the packet loss

[6]By Theorem 2, the source can force the controller to perform a flow swap with an impact as high as roughly $0.5$.




(a) The no. of packets lost in a flow swap vs. no. of switches involved in the update.

(b) The number of packets lost in a flow swap in different update approaches (with $n = 32$).

(c) The number of packets lost in a flow swap using SWAN and TIME4 + SWAN (with $n = 32$).

(d) The number of packets lost in a flow swap using B4 and TIME4 + B4 (with $n = 32$).

Fig. 8: Flow swap performance: in large networks (a) TIME4 allows significantly less packet loss than untimed approaches. The packet loss of TIME4 is slightly higher than SWAN and B4 (b), while the latter two methods incur higher overhead. Combining TIME4 with SWAN or B4 provides the best of both worlds; low packet loss (b) and low overhead (c and d).

to the conventional SWAN-based update. We repeated the experiment for various values of scratch capacity, from 0% to 10%. As illustrated in Fig. 8c, the TIME4+SWAN approach can achieve the same level of packet loss as SWAN with **less scratch capacity**. We performed a similar experiment with a timed B4 update, varying the bandwidth reduction rate between 0% and 10%, and observed similar results.

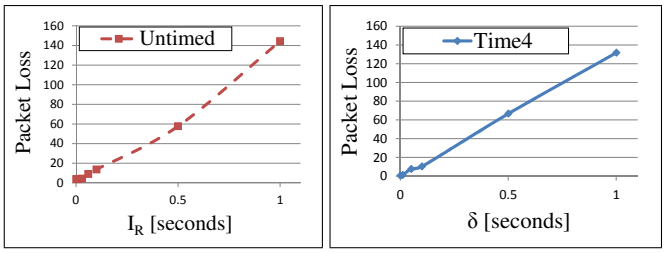
**Number of switches.** We evaluated the effect of $n$, the number of switches involved in the flow swap, on the packet loss. We performed an $n$-swap with $n = 2, 4, 8, 16, 32$. As illustrated in Fig. 8a, the number of packets lost during an untimed update grows linearly with the number of switches $n$, while the number of packets lost in a TIME4 update is less than one on average, and is not affected by the number of switches. As $n$ increases, the update duration[7] is longer, and hence more packets are lost during the update procedure.

**Installation latency variation.** Our next experiment (Fig. 9a) examined how the installation latency variation, denoted by $I_R$, affects the packet loss during an untimed update. We analyzed different values of $I_R$: in each update we synthetically determined a uniformly distributed installation latency, $I \sim U[0, I_R]$. As shown in Fig. 9a, the switch's installation latency range, $I_R$, dramatically affects the packet loss rate during an untimed update. Notably, when $I_R$ is on the order of 1 second, as in the extreme scenarios of [16], [39], TIME4 has a significant advantage over the untimed approach.

**Scheduling error.** Figure 9b depicts the packet loss as a function of the scheduling error of TIME4. By Fig. 8a, 9a and 9b, we observe that if $\delta$ is sufficiently low compared to $I_R$ and $(n-1)\Delta$, then TIME4 outperforms the untimed approaches. Note that even if switches are not implemented with extremely low scheduling error $\delta$, we expect TIME4 to outperform the untimed approach, as typically $\delta < I_R$, as further discussed in Section V.

**Summary.** The experiments presented in this section demonstrate that TIME4 performs significantly better than untimed approaches, especially when the update involves multiple switches, or when there is a non-deterministic installation latency. Interestingly, TIME4 can be used in conjunction with existing approaches, such as SWAN and B4, allowing the same level of packet loss **with less overhead** than the untimed variants.

[7]The **update duration** is the time elapsed from the instant the first switch is updated until the instant the last switch is updated. In our setting the update duration is roughly $(n-1)\Delta$.




(a) The number of packets lost in a flow swap vs. the installation latency range, $I_R$.

(b) The number of packets lost in a flow swap vs. the scheduling error, $\delta$.

Fig. 9: Performance as a function of $I_R$ and $\delta$.

## V. DISCUSSION

**Scheduling accuracy.** The advantage of timed updates greatly depends on the **scheduling accuracy**, i.e., on the switches' ability to accurately perform an update at its scheduled time. Clocks can typically be synchronized on the order of 1 microsecond (e.g., [4]) using PTP [3]. However, a switch's ability to accurately perform a scheduled action depends on its implementation.

- *Software switches:* Our experimental evaluation showed that the scheduling error in the software switches we tested was on the order of 1 millisecond.
- *Hardware-based scheduling:* The work of [18] has shown a method that allows the scheduling error of timed events in hardware switches to be as low as 1 microsecond.

It is an important observation that in a typical system we expect the scheduling error to be lower than the installation latency variation, i.e., $\delta < I_R$. Untimed updates have a non-deterministic installation latency. On the other hand, timed updates are predictable, and can be scheduled in a way that avoids conflicts between multiple updates, allowing $\delta$ to be typically lower than $I_R$.

**Model assumptions.** Our model assumes a *lossless* network with *unsplittable*, *fixed-bandwidth* flows. A notable example

of a setting in which these assumptions are often valid is a WAN or a carrier network. In carrier networks the maximal **bandwidth** of a service is defined by its bandwidth profile [15]. Thus, the controller cannot dynamically change the bandwidth of the flows, as they are determined by the SLA. The Frame **Loss** Ratio (FLR) is one of the key performance attributes [15] that a service provider must comply to, and cannot be compromised. **Splitting** a flow between two or more paths may result in packets being received out-of-order. Packet reordering is a key performance parameter in carrier-grade performance and availability measurement, as it affects various applications such as real-time media streaming [40]. Thus, all packets of a flow are forwarded through the same path.

**Network latency.** In Fig. 1, the switches $S_1$ and $S_3$ are updated at the same time, as it is implicitly assumed that all the links have the same latency. In the general case each link has a different latency, and thus $S_1$ and $S_3$ should not be updated at the same time, but at two different times, $T_1$ and $T_3$, that account for the different latencies.

## VI. Conclusion

Time and clocks are valuable tools for coordinating updates in a network. We have shown that dynamic traffic steering by SDN controllers requires flow swaps, which are best performed as close to instantaneously as possible. Time-based operation can help to achieve carrier-grade packet loss rate in environments that require rapid path reconfiguration. Our OpenFlow time extension can be used for implementing flow swaps and TIME4. It can also be used for a variety of additional timed update scenarios that can help improve network performance during path and policy updates.

## VII. Acknowledgments

We gratefully acknowledge Oron Anschel and Nadav Shiloach, who implemented the TIME4-enabled OFSoftswitch prototype. We thank Jean Tourrilhes and the members of the Extensibility working group of the ONF for many helpful comments that contributed to the OpenFlow time extension. We also thank Nate Foster, Laurent Vanbever, Joshua Reich and Isaac Keslassy for helpful discussions. We gratefully acknowledge the DeterLab project [36] for the opportunity to perform our experiments on the DeterLab testbed. This work was supported in part by the ISF grant 1520/11.

## References

[1] ITU-T G.8271/Y.1366, "Time and phase synchronization aspects of packet networks," *ITU-T*, 2012.
[2] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, 2013.
[3] IEEE TC 9, "1588 IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems Version 2," *IEEE*, 2008.
[4] H. Li, "IEEE 1588 time synchronization deployment for mobile backhaul in China Mobile," keynote presentation, IEEE ISPCS, 2014.
[5] IEEE Std C37.238, "IEEE Standard Profile for Use of IEEE 1588 Precision Time Protocol in Power System Applications," *IEEE*, 2011.
[6] "ONF SDN Product Directory," https://www.opennetworking.org/sdn-resources/onf-products-listing, January, 2015.
[7] N. McKeown, et al., "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, 2008.
[8] Open Networking Foundation, "OpenFlow Switch Specification," *Version 1.4.0*, 2013.
[9] "Interface to the Routing System (I2RS) working group," https://datatracker.ietf.org/wg/i2rs/charter/, IETF, 2016.
[10] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *NSDI*, 2010.
[11] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM*, 2013.
[12] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM*, 2013.
[13] Open Networking Foundation, "OpenFlow-enabled mobile and wireless networks," *ONF Solution Brief*, 2013.
[14] Metro Ethernet Forum, "Mobile backhaul - phase 2 implementation agreement," *MEF 22.1*, 2012.
[15] Metro Ethernet Forum, "Carrier ethernet class of service - phase 2 implementation agreement," *MEF 23.1*, 2012.
[16] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang, "Dionysus: Dynamic scheduling of network updates," in *ACM SIGCOMM*, 2014.
[17] T. G. Edwards and W. Belkin, "Using SDN to facilitate precisely timed actions on real-time data streams," in *ACM SIGCOMM Workshop on Hot topics in Software Defined Networks (HotSDN)*, 2014.
[18] T. Mizrahi, O. Rottenstreich, and Y. Moses, "TimeFlip: Scheduling network updates with timestamp-based TCAM ranges," in *IEEE INFOCOM*, 2015.
[19] P. François and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Transactions on Networking (TON)*, 2007.
[20] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide IGP migrations," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 314–325, 2011.
[21] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: updating data center networks with zero loss," in *ACM SIGCOMM*, 2013.
[22] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, 2012.
[23] N. Pippenger, "On rearrangeable and non-blocking switching networks," *Journal of Computer and System Sciences*, 1978.
[24] T. Mizrahi and Y. Moses, "Time-based updates in software defined networks," in *ACM SIGCOMM Workshop on Hot topics in Software Defined Networks (HotSDN)*, 2013.
[25] T. Mizrahi and Y. Moses, "On the necessity of time-based updates in SDN," in *Open Networking Summit (ONS)*, 2014.
[26] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2015.
[27] Open Networking Foundation, "OpenFlow switch specification," *Version 1.5.0*, 2015.
[28] Open Networking Foundation, "OpenFlow extensions 1.3.x package 2," 2015.
[29] "TIME4 source code," https://github.com/TimedSDN, 2015.
[30] T. Mizrahi and Y. Moses, "TIME4: Time for SDN," technical report, arXiv preprint, 2016.
[31] J. M. Kleinberg, "Single-source unsplittable flow," in *Symposium on Foundations of Computer Science (FOCS)*, 1996.
[32] T. Mizrahi and Y. Moses, "Time-based Updates in OpenFlow: A Proposed Extension to the OpenFlow Protocol," technical report, Technion, http://tx.technion.ac.il/%7Edew/OFTimeTR.pdf, 2013.
[33] T. Mizrahi and Y. Moses, "Using REVERSEPTP to distribute time in software defined networks," in *IEEE ISPCS*, 2014.
[34] "CPqD OFSoftswitch," https://github.com/CPqD/ofsoftswitch13, 2014.
[35] "Precision Time Protocol daemon," http://ptpd.sourceforge.net/, 2013.
[36] "The DeterLab project," http://deter-project.org/, 2015.
[37] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the datacenter," in *HotNets*, 2009.
[38] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Hot-ICE*, 2012.
[39] 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*, 2012.
[40] ITU-T Y.1563, "Ethernet frame transfer and availability performance," *ITU-T*, 2009.