TIME4: Time for SDN

Tal Mizrahi, Yoram Moses
Technion — Israel Institute of Technology

Abstract

In recent years, there has been growing interest in dynamic and centralized traffic engineering, where decisions about forwarding paths are taken from a network-wide perspective, based on the dynamic state of the network. 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.

We introduce TIME4, an approach that uses accurate time to coordinate network updates. We characterize a set of update scenarios called flow swaps, where TIME4 is the optimal update approach, yielding less packet loss than existing update approaches. We introduce 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. Our prototype will soon be 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.

1 Introduction
1.1 Background
Defining the network topology and routing paths dynamically, based on a complete picture of the network, can significantly improve the network performance compared to distributed routing protocols that take local decisions. Software Defined Networking (SDN) and OpenFlow [1, 2] have been leading trends in this context, but several other ongoing efforts offer similar concepts. The Interface to the Routing System (I2RS) working group [3], and the Forwarding and Control Element Separation (ForCES) working group [4] are two examples of such ongoing efforts in the Internet Engineering Task Force (IETF).

‘With great power comes great responsibility’. Centralized network updates, whether they are related to network topology, security policy, or other configuration attributes, 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.

Several recent works have explored the realm of dynamic path reconfiguration, with frequent updates on the order of minutes [5–7], enabled by SDN. While SDN was originally considered in the context of campus networks [1] and data centers [8], it is now also being considered for Wide Area Networks (WANs) [5,6], carrier networks, and mobile backhaul networks [9].

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 [10]. Other types of carrier network applications, such as storage and financial trading require even lower loss rates [11], on the order of $10^{-5}$. For example, 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. 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.

Over the last decade network time synchronization has evolved significantly; the Precision Time Protocol (PTP), defined in the IEEE 1588 [12] standard, can synchronize clocks to a very high degree of accuracy, typically
on the order of 1 microseconds [13,14]. Since its publication in 2008, PTP has matured and has become a common and affordable feature in commodity switches. Notably, 9 out of the 13 SDN-capable silicons listed in the Open Networking Foundation (ONF) SDN Product Directory [15] have native IEEE 1588 support [16–24]. We argue that since SDN products already have built-in capabilities for accurate clock synchronization, it is only natural to harness this powerful technology to coordinate events in SDNs.

1.2 Timed Network Updates

It has recently been demonstrated [25] that TCAMs can be used to implement time-based updates in SDN switches with an accuracy on the order of 1 microsecond. Based on this infrastructure, we explore the use of accurate time as a tool for performing coordinated network updates in a way that minimizes packet loss. We propose the use of TiME4, which is an update approach that performs multiple changes at different switches at the same time.

![Figure 1: Flow Swapping—Flows need to convert from the “before” configuration to the “after”.

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

In this paper we show that simultaneous updates are the optimal approach in the example above, whereas other update approaches may yield considerable packet loss.

Accuracy is a key requirement in the timed approach; 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 this paper show that the scheduling error in software switches is on the order of 1 millisecond. The TCAM-based solution of [25] can time scheduled events in existing hardware switches with an accuracy on the order of 1 microsecond.

Accurate time is a powerful abstraction for SDN programmers, not only for flow swaps, but also for timed consistent updates, as discussed by [26].

1.3 Related Work

Time and synchronized clocks have been used in various distributed applications, from mobile backhaul networks [13] to distributed databases [27]. Time-of-day routing [28] routes traffic to different destinations based on the time-of-day. Path calendaring [29] 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.

Various network update approaches have been analyzed in the literature. A common approach is to use a sequence of configuration commands [7, 30–32], whereby the order of execution guarantees that no anomalies are caused in intermediate states of the procedure. Two-phase updates [33] use configuration version tags to guarantee consistency. None of these approaches applies to flow swaps, or considered explicit use of time.

---

1The 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 $S_3$ are not updated at the same time, but at two different time, $T_1$ and $T_3$, that account for the different latencies.
To the best of our knowledge, our work is the first to analyze the use of accurate time for coordinating configuration or policy updates of network devices. The concept of using accurate time to trigger policy and routing changes was briefly discussed in [34]. Preliminary versions of this paper briefly introduced the timed update concept [35] and the flow swapping scenario [36]. The use of time for consistent updates was discussed in [26]. To the best of our knowledge, the current work is the first to present accurate time as a generic tool for performing network updates, and the first to analyze flow swaps, where timed updates are the optimal update approach.

1.4 Contributions

The main contributions of this paper are as follows:

• We formally prove that accurate time is an essential tool in the dynamic nature of SDN. We show that timed updates are optimal for flow swapping.

• We introduce the lossless flow allocation (LFA) problem, and use game-theoretic tools to formally show that given the online nature of resource allocation in SDN, scenarios that require simultaneous changes at multiple switches are bound to occur, with clock-based near-simultaneous scheduling offering an advantageous solution.

• 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 [37], and into the OpenFlow 1.3.x extension package [38]. The source code of our prototype will soon be publicly available.

• We present experimental results that demonstrate the advantage of timed updates over existing approaches.

2 Time-based Flow Swapping

Our analysis of flow swaps consists of two main components: (a) showing the inevitability of flow swaps, and (b) comparing Time4 to other update approaches in flow swap scenarios. In this section we briefly discuss each of these two components, and then the detailed analysis is presented in the rest of the paper.

2.1 Inevitable Flow Swaps

In Section 1.2 we presented an example 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. Detailed theoretical analysis is presented in Section 3.

Our analysis is based on representing the flow-swap problem as an instance of a multi-commodity flow (MCF) problem, as illustrated in Fig. 2b. The topology of the graph in the figure models the traffic behavior to a given destination in common multi-rooted network topologies such as fat-tree and Clos (Fig. 2a). Using this model we discover a key result; flow-swapping is an essential operation in flow allocation.

The multi-commodity flow (MCF) problem [39] has been thoroughly discussed in the literature; given a directed graph, a source node, a destination node, and a set of flow demands (commodities) between the source and the destination, 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 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.

We first show 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.

Claim 1. For the topology of Fig. 2b, there exists a strategy for the source, $S_s$, that forces every controller strategy, $S_{con}$, to perform a swap.
Next, we show a more general result; in some topologies the source has a strategy that forces the controller to perform an $n$-swap, i.e., a simultaneous swap that involves $n$ individual updates.

**Claim 2.** For the topology of Fig. 2b, there exists a strategy for the source, $S_s$, that forces every controller strategy, $S_{con}$, to perform an $n$-swap.

Claims 1 and 2 are formally phrased in Theorems 1 and 4. These results show that a scenario in which multiple flows must be updated at the same time are inevitable, implying the importance of timed updates. Indeed, in the latter case, the controller is forced to invoke $n$ individual commands that should optimally be performed at the same time. The naive approach causes 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.

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.

### 2.2 Timed Flow Swaps vs. Other Approaches

We now return to the flow swapping scenario of Fig. 1, and examine the possible alternatives to update the network to the new path configuration.

The simple-minded way to perform a set of $n$ network updates is to send the set of $n$ update commands as close as possible to instantaneously, and hope for the best. Throughout the paper we refer to this approach as the naive approach. Another method that has been widely studied in various forms is to use a sequence of configuration commands, whereby the order of execution guarantees that no anomalies are caused in intermediate states of the procedure [7,30–32]. Another approach to consistent updates [33] is based on two-phase commit procedures, using configuration version tags to guarantee consistency. In SWAN [5], the authors suggest that reserving some unused scratch capacity of 10-30% on every link can allow congestion-free updates in most scenarios. The B4 [6] approach prevents packet loss during path updates by temporarily reducing the bandwidth of some or all of the flows.

Table 1 presents a comparison of these different update approaches.

As discussed in the table, the order and two-phase approaches are not applicable to flow swaps. The SWAN approach avoids packet loss, but at a potentially high cost of excess capacity. The B4 approach prevents packet
<table>
<thead>
<tr>
<th>Approach</th>
<th>Pros</th>
<th>Cons</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time4</strong></td>
<td>Near-zero loss, simplicity, scalability.</td>
<td>Packet loss is low but not zero.</td>
<td>As discussed in Section 1.2, Time4 offers a simple approach that yields very little packet loss, that is a function of the scheduling accuracy.</td>
</tr>
<tr>
<td>Naïve</td>
<td>Simplicity.</td>
<td>Packet loss during the update. Increases with the size of the network.</td>
<td>Using the naïve approach, flows $f_1$ and $f_2$ cannot be swapped simultaneously, as the controller cannot send two update messages at the same time. Moreover, the latency of rule installations has been shown to range from milliseconds to seconds [7, 40], significantly affecting the update coordination. Thus, temporary congestion will be caused, either at $S_2$ or at $S_4$, causing packets to be lost.</td>
</tr>
<tr>
<td><strong>SWAN [5]</strong></td>
<td>Zero packet loss.</td>
<td>Requires scratch capacity.</td>
<td>This approach prevents packet loss, but requires costly resources, using a scratch overhead of 100% on at least one of the links, $S_2 \rightarrow S_5$ or $S_4 \rightarrow S_5$.</td>
</tr>
<tr>
<td><strong>B4 [6]</strong></td>
<td>Zero packet loss.</td>
<td>Requires reducing the bandwidth at the source.</td>
<td>This approach prevents packet loss, but requires the sources to temporarily reduce their traffic rate. While this approach worked well in [6], it is not necessarily applicable to service provider networks, where the provider has no control over the bandwidth of traffic flows. The provider must take care not to compromise the Service Level Agreement (SLA) during path updates.</td>
</tr>
<tr>
<td>Ordered [7,30–32]</td>
<td>—</td>
<td>Not applicable to flow swaps.</td>
<td>The order approach cannot be applied to flow swap scenarios such as Example 1, as there does not exist a congestion-free update sequence. In [7] it was observed that in some update scenarios, referred to as deadlocks, there is no ordering that avoids congestion.</td>
</tr>
<tr>
<td><strong>Two-phase [33]</strong></td>
<td>—</td>
<td>Not applicable to flow swaps.</td>
<td>Two-phase commits guarantee that each flow is forwarded either according to the old path or according to the new path, but not according to a mixture of the two configurations. However, in the topology of Fig 1 the two-phase approach cannot avoid the temporary congestion during the update. The authors of [33] observed that the two-phase approach guarantees consistent updates for a class of updates referred to as trace properties, and does not guarantee consistency for other properties, such as congestion-freedom. Hence, the two-phase approach is not applicable to Example 1.</td>
</tr>
</tbody>
</table>

Table 1: Comparison of the different network update approaches with respect to the scenario of Fig. 1.

loss, but requires the network operator to be able to control the bandwidth of the flows, an assumption that is not viable in carrier networks. Hence, in the rest of the paper we focus on the two approaches that can address the flow swapping scenario without requiring excess capacity and without controller the flow bandwidth. Hence, in our evaluation (Section 5) we focus on Time4 and the naïve approach.

### 3 The Lossless Flow Allocation (LFA) Problem

#### 3.1 Model and Definitions

We now introduce a special case of the MCF problem called lossless flow allocation (LFA): it is not presented as an optimization problem, but rather as a game between two players: a source that generates traffic flows (commodities) and a controller that configures the network’s forwarding rules. 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. We focus on the unsplittable multi-commodity flow problem, i.e., all packets of a flow must
be forwarded through the same path. 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 in a way that forces the controller’s reconfiguration procedure to include 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.

The term flow in classic flow problems typically refers to the amount of traffic that is forwarded through each edge of the graph. Since our analysis focuses on SDN, we slightly divert from the common flow problem terminology, and 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 (or commodity) 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, \( \mathbb{V}_m \). Thus, \( \mathbb{V} = \mathbb{V}_m \cup \{s, d\} \). 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, \( R_{\text{con}} : \mathbb{F} \times \mathbb{V}_m \rightarrow \mathbb{V}_m \cup \{d\} \). Every node (switch) has a flow table, consisting of a set of entries; an element \( v \in \mathbb{F} \times \mathbb{V}_m \) is referred to as an entry for short. An update of \( R_{\text{con}} \) is defined to be a partial function \( u : \mathbb{F} \times \mathbb{V}_m \rightarrow \mathbb{V}_m \cup \{d\} \). 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 3.5 we discuss \( k \)-swaps for values of \( k > 2 \).

\(^2\)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 [41].

---

**Source Procedure**

1. \( \mathbb{F} \leftarrow \emptyset \)
2. repeat at every step
3. \( (a, F) \leftarrow S_a(\mathbb{F}, R_{\text{con}}) \)
4. if \( a = \text{Add} \) then
5. \( \mathbb{F} \leftarrow \mathbb{F} \cup F \)
6. else /\ a = \text{Remove} \ then
7. \( \mathbb{F} \leftarrow \mathbb{F} \setminus F \)

Figure 3: The LFA game: the source’s procedure.
Controller Procedure

1. repeat at every step
2. \{u_1, \ldots, u_\ell\} \leftarrow S_{con}(R_{con}, a, F)
3. for \( j \in [1, \ell] \)
4. Update \( R_{con} \) according to \( u_j \)

Figure 4: The LFA game: the controller’s procedure.

3.2 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 \( S \) is a function that defines for each flow set \( F \) and forwarding function \( R_{con} \) for \( F \), a pair \((a, F)\) representing the source’s next step, where \( a \in \{\text{Add, Remove}\} \) 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 \( S_{con}(R_{con}, a, F) = U \), where \( U = \{u_1, \ldots, u_\ell\} \) 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 \( F \). 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.

**Theorem 1.** Let \( G \) be an LFA graph. In the LFA game over \( G \), there exists a strategy for the source, \( S_s \), that forces every controller strategy, \( S_{con} \), 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 \( e_2 \). 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. Following 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:

- 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 \( F_5 \) 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 the same edge. 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.

- 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.

Thus, in both possible cases the controller is forced to swap a flow from \( o_1 \) with a flow from \( o_2 \), and thus the controller is forced to perform a 2-swap. We have proven the claim for \( m = 2 \).

For \( m \geq 2 \) the source generates \( m - 2 \) flows with a bandwidth of 1 each, and thus the controller forwards these flows through \( m - 2 \) edges connected to node \( d \) (without loss of generality \( e_3, \ldots, e_m \)) leaving two free edges, \( e_1 \) and \( e_2 \). From this point, the proof is identical to the case of \( m = 2 \). \( \square \)
3.3 The Impact of Flow Swaps

The proof of Theorem 1 presented an inevitable flow swap scenario. We try to define a metric for flow swaps, by considering the oversubscription that is caused if the flows are not swapped simultaneously, but updated using the naive approach.

We define the oversubscription of an edge, \( e \), with respect to a forwarding function, \( R_{\text{con}} \), to be the difference between the total bandwidth of the flows forwarded through \( e \) according to \( R_{\text{con}} \), 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.

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

Example 2. We observe the scenario described in the proof of Theorem 1, and consider what would happen if the two flows had not been swapped simultaneously. The scenario had two cases; in the first case, a 0.35 flow was swapped with a 0.45 flow. If the controller first reroutes the 0.45 flow, and then reroutes the 0.35 flow, then during the intermediate transition period, the first edge is oversubscribed by 0.15, and hence the impact of this swap is 0.15. In the second case, if the 0.35 flow is rerouted and then the 0.45 flow, a temporary oversubscription of 0.05 is caused, and thus the flow swap impact in the second case is 0.05.

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. In the LFA game over \( G \), for every \( 0 < \alpha < 0.5 \), there exists a strategy for the source that forces every controller strategy, \( S_{\text{con}} \), to perform a swap with an impact of \( \alpha \).

Proof. We define \( \epsilon \) such that \( 5\epsilon = \frac{1}{2} - \alpha \). We use the source’s strategy from the proof of Theorem 1, with the exception that the bandwidths of flows \( F_1, \ldots, F_7 \) are: \( f_1 = f_2 = \frac{1}{2} - 2\epsilon \), \( f_3 = f_4 = \frac{1}{2} - \epsilon \), \( f_5 = 4\epsilon \), and \( f_6 = f_7 = 3\epsilon \).

As in the proof of Theorem 1, there are two possible cases. In the first case, the controller routes symmetrically through the two paths, utilizing \( 1 - 3\epsilon \) of the bandwidth of each path. The source adds \( F_5 \) in response. To accommodate \( F_5 \) the controller swaps \( F_1 \) and \( F_3 \). We determine the impact of this swap by considering the oversubscription of performing a naive update; the controller first reroutes \( F_1 \), and only then reroutes \( F_3 \). Hence, the temporary oversubscription is \( 1 - 3\epsilon + \frac{1}{2} - 2\epsilon - 1 = 1.5 - 5\epsilon - 1 \). Thus, the impact is \( 0.5 - 5\epsilon = \alpha \). In the second case, the controller forwards \( F_1 \) through the same path as \( F_2 \), and \( F_3 \) through the same path as \( F_4 \). The source responds by generating \( F_6 \) and \( F_7 \). Again, the controller is forced to swap between \( F_1 \) and \( F_3 \). We compute the impact by considering the naive update, where the controller reroutes \( F_3 \) first, causing an oversubscription of \( 1 - 4\epsilon + 0.5 - \epsilon - 1 = 0.5 - 5\epsilon = \alpha \).

Hence, in both cases the source inflicts a flow swap with a swap impact of \( \alpha \).

Intuitively, the latter theorem 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.

3.4 Network Utilization

Theorem 1 demonstrates that regardless of the controller’s policy, flow swaps cannot be prevented. However, the proof of Theorem 1 uses a scenario in which the edges leading to node \( d \) are almost fully utilized, suggesting that perhaps flow swaps are inevitable only when the traffic bandwidth is nearly equal to the max-flow of the graph.

If the max-flow capacity is \( m \cdot c \), then arguably by limiting the total bandwidth of the flows of \( F \) to \( \gamma \cdot m \), for \( \gamma < 1 \), it may be possible to avoid flow swaps. In the next theorem we show that if \( \gamma > \frac{7}{8} \), then flow swaps are inevitable.

Theorem 3. Let \( G \) be an LFA graph. In the LFA game over \( G \), if \( \gamma > \frac{7}{8} \), then there exists a strategy for the source, that forces every controller strategy, \( S_{\text{con}} \), to perform a swap.
Proof. We define \( \epsilon \) such that \( 0 < \epsilon < \frac{1}{3} \).

The proof is similar to the proof of Theorem 1, except the flow bandwidths; we define the bandwidth of flows \( F_1, \ldots, F_7 \) to be \( f_1 = f_2 = (\frac{1}{2} + \epsilon), f_3 = f_4 = (\frac{1}{2} - \epsilon), f_5 = (\frac{1}{2} + \epsilon), \) and \( f_6 = f_7 = 3 \cdot \epsilon \).

In the first case of the proof of Theorem 1, the total bandwidth of the flows is \( F_1 + F_2 + F_3 + F_4 + F_5 = (\frac{7}{4} + \epsilon) \). In the second case, the total bandwidth is \( F_1 + F_2 + F_3 + F_4 + F_5 + F_7 = (\frac{9}{4} + 6 \cdot \epsilon) \). In both cases, for any \( \gamma > \frac{7}{8} \), we can choose a sufficiently small \( \epsilon \) that guarantees that the total bandwidth does not exceed \( \gamma \).

The analysis of [5] showed that a scratch capacity of 10% was enough to address the reconfiguration scenarios that were considered in that work. Theorem 3 shows that utilization capacity of 87.5% suffices to force flow swaps. It follows that the 10% reserve that [5] suggest is not necessarily sufficient in general for consistent configuration.

3.5 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 \( \mathcal{O} \).

Theorem 4. Let \( G \) be an LFA graph. In the LFA game over \( G \), there exists a strategy for the source, \( S_s \), that forces every controller strategy, \( S_{con} \), to perform an \( n \)-swap.

Proof. For \( n = 1 \), the claim is trivial. For \( n = 2 \), the claim was proven in Theorem 1. Thus, we assume \( n \geq 3 \).

If \( m > 2 \), the source first generates \( m - 2 \) flows with a rate \( c \) each, and we assume without loss of generality that after the controller allocates these flows only \( e_1 \) and \( e_2 \) remain unused. Thus, we focus on the case where \( m = 2 \).

We describe a strategy, \( S_s \) as required; \( s \) generates three types of flows:

- **Type A:** two flows \( F_1, F_2 \), at a rate \( h \) each: \( F_1 = (1, h, o_1) \), and \( F_2 = (2, h, o_1) \).
- **Type B:** \( n \) flows, \( F_3, \ldots, F_{n+2} \), with a total rate \( g \), i.e., at a rate \( \frac{g}{n} \) each. The source sends each of the \( n \) flows through a different node of \( \mathcal{O} \).
- **Type C:** \( n - 1 \) flows, \( F_{n+3}, \ldots, F_{2n+1} \) with a total rate \( g \), i.e., \( \frac{g}{n-1} \) each. The source sends each of the \( n - 1 \) flows through a different node of \( o_2, \ldots, o_n \).

We define \( h \) and \( g \) such that:

\[
\frac{1}{3} < h < g < \frac{1}{2} \quad (1)
\]

\[
g > (n^2 - n)(1 - 2h) \quad (2)
\]

We claim that for every \( n \) there exist \( g \) and \( h \) that satisfy (1) and (2). We prove this claim by finding \( g \) and \( h \) that satisfy the two conditions. We choose an arbitrary \( g \) in the range \( (\frac{11}{24}, \frac{1}{2}) \). We find a valid \( h \) by solving \( g > (n^2 - n)(1 - 2h) \). The latter yields \( h > \frac{1}{2} - \frac{9}{2(n^2 - n)} \). Since \( n \geq 3 \), we have \( n^2 - n \geq 6 \), and thus \( \frac{9}{2(n^2 - n)} < \frac{9}{2 \times 6} = \frac{1}{4} \). Clearly, \( \frac{9}{2(n^2 - n)} > 0 \). It follows that every \( h \) that satisfies \( \frac{1}{2} - \frac{1}{4} < h < \frac{1}{2} - 0 \), also satisfies \( h > \frac{1}{3} \). Hence, every \( g \) and \( h \) in the range \( (\frac{11}{24}, \frac{1}{2}) \) that satisfy \( h < g \), also satisfy (1) and (2).

Intuitively, for \( h \) and \( g \) sufficiently close to \( \frac{1}{2} \) (but less than \( \frac{1}{2} \)) (1) and (2) satisfied.

We now prove that after generating the flows \( F_1, \ldots, F_{2n+1} \), the function \( R_{con} \) forwards all type B flows through the same path, and all type C flows through the same path. Assume by way of contradiction that there is a forwarding function \( R_{con} \) that forwards flows \( F_1, \ldots, F_{2n+1} \) without loss, but does not comply to the latter claim. We consider two distinct cases: either the two type A flows are forwarded through the same edge, or they are forwarded through two different edges.
If the two type A flows are forwarded through two different paths, then we first assume that \( F_1 \) and the \( n \) type B flows are forwarded through \( e_1 \) and that \( F_2 \) and the \( n - 1 \) type C flows are forwarded through \( e_2 \). Thus, at this point each of the two edges sustains traffic at a rate of \( g + h \). By the assumption, there exists an update that swaps \( i < n \) flows of type B with \( j < n - 1 \) flows of type C, such that after the swap none of the edges exceeds its capacity. Thus, the update adds the bandwidth \( |j \cdot \frac{g}{n-1} - i \cdot \frac{2}{n}| \) to one of the edges, and this additional bandwidth must fit into the available bandwidth before the update, \( 1 - g - h \). Hence, \( |j \cdot \frac{g}{n-1} - i \cdot \frac{2}{n}| < c - g - h \). Note that \( 1 - g - h < 1 - 2h < \frac{n-1}{n-1} - \frac{2}{n} \), following (1) and (2). Thus we get \( |j \cdot \frac{g}{n-1} - i \cdot \frac{2}{n}| < \frac{g}{n-1} - \frac{2}{n} \). It follows that \( |j \cdot n - i \cdot n + i| < 1 \). Since \( j, i, n \) are integers, we get that \( j \cdot n - i \cdot n + i = 0 \), and thus \( j = i \cdot \frac{n-1}{n} \). Now since \( i \leq n \) and \( j \leq n - 1 \) are both natural numbers, the only solution is \( j = n - 1 \) and \( i = n \), which means that the flows from type B are all forwarded through the same path, as well as the flows of type C, contradicting the assumption.

If the two type A flows are forwarded through the same edge, their total bandwidth is \( 2h \), and thus the remaining bandwidth through this edge is \( 1 - 2h \). From (2) we have \( \frac{g}{n-1} > \frac{n}{n-1} - \frac{2}{n} \). We note that (i) \( \frac{g}{n-1} > \frac{9}{n-1} - \frac{2}{n} \), and (ii) \( \frac{2}{n} > \frac{g}{n-1} - \frac{2}{n} \). It follows that \( \frac{g}{n-1} > 1 - 2h \), and also \( \frac{2}{n} > 1 - 2h \), and thus none of the type B or type C flows fit on the same path with \( F_1 \) and \( F_2 \). Thus, all the type B and type C flows are on the same path, contradicting the assumption.

We have shown that all flows of type B, denoted by \( \mathbb{R}^B \), must be forwarded through the same path, and that all flows of type C, denoted by \( \mathbb{R}^C \), are forwarded through the same path. Thus, after the source generates the \( 2 \cdot n + 1 \) flows, there are two possible scenarios:

- The two type A flows are forwarded through the same path, and the type B and type C flows are forwarded through the other path. In this case \( s \) generates two flows at a rate \( 1 - h - g \) each. To accommodate both flows the controller must swap the flows of \( \mathbb{R}^B \) with \( F_1 \) or the flows of \( \mathbb{R}^C \) with \( F_2 \). Both possible swaps involve \( n \) entries, and thus it is an \( n \)-swap.

- One path is used for \( F_1 \) and the flows of \( \mathbb{R}^C \), and the other path is used for \( F_2 \) and the flows of \( \mathbb{R}^B \). In this case the source generates a flow with a bandwidth of \( 1 - 2h \), again forcing the controller to swap the flows of \( \mathbb{R}^B \) with \( F_1 \) or the flows of \( \mathbb{R}^C \) with \( F_2 \).

In both cases the controller is forced to perform a swap that involves the \( n \) nodes, i.e., an \( n \)-swap.
4 Design and Implementation

4.1 Protocol Design

4.1.1 Overview

A Time4-enabled system is comprised of two main components:

- **OpenFlow time extension.** Time4 is built upon the OpenFlow protocol. We define an extension to the OpenFlow protocol that enables timed updates; the controller can attach an *execution time* to every OpenFlow command it sends to a switch, defining when the switch should perform the required command. It should be noted that the Time4 approach is not limited to OpenFlow; we have defined a similar time extension to the NETCONF protocol [42], but in this paper we focus on Time4 in the context of OpenFlow, as described in the next subsection.

- **Clock synchronization.** Time4 requires the switches and controller to maintain a local clock, allowing time-triggered events. A clock synchronization mechanism should be used, in order to align the different clocks. The OpenFlow time extension we defined does not mandate a specific clock synchronization method. Various mechanisms may be used, e.g., the Network Time Protocol (NTP), the Precision Time Protocol (PTP) [12], or GPS-based synchronization. The prototype we designed and implemented uses ReversePTP [43], as described below.

4.1.2 OpenFlow Time Extension

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 Section A.³

Our extension makes use of the OpenFlow [2] 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 rules 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 [2], 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, Ts. The switch then executes the desired command(s) at time Ts.

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 command, 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 Ts, the switch can verify that it can dedicate the required resources to execute the command as close as possible to Ts. If the switch’s resources are not available, for example due to another command that is scheduled to Ts, 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 the naïve approach is used.

The OpenFlow time extension also defines Bundle Feature Request messages, which allow the controller to query switches about whether they support the time extension, and to configure some of the switch parameters related to Scheduled Bundles.

³A preliminary version of this extension is discussed in [44].
4.1.3 Clock Synchronization: ReversePTP

In the last decade, PTP, based on the IEEE 1588 [12] standard, has become a common feature in commodity switches, typically providing a clock accuracy on the order of 1 microsecond.

In [43, 45] we introduced ReversePTP, a PTP variant for SDNs. ReversePTP is based on PTP, but is conceptually reversed. In PTP a single node periodically distributes its time to the other nodes in the network. In ReversePTP all nodes in the network (the switches) periodically distribute their time to a single node (the controller). The controller keeps track of the offsets, denoted by $\text{offset}_i$, between its clock and each of the switches’ clocks.

ReversePTP allows the complex clock algorithms to be implemented by the controller, whereas the ‘dumb’ switches only need to distribute their time to the controller. Following the SDN paradigm, the ReversePTP algorithmic logic can be programmed and dynamically tuned at the controller without affecting the switches.

The reason we chose to use ReversePTP in our system, is that it simplifies the experiment setup. A clock synchronization protocol requires a long setup time, typically tens of minutes. ReversePTP allows the controller to keep track of the state of each clock, providing an indication of when the setup process has completed.

![ReversePTP Diagram](image)

Figure 6: ReversePTP in SDN: switches distribute their time to the controller. Switches’ clocks are not synchronized.

As shown in [43], ReversePTP can be effectively used to perform timed updates; in order to have switch $i$ perform a command at time $T_s$, the controller instructs $i$ to perform the command at time $T_{is}$, where $T_{is} = T_s + \text{offset}_i$, takes the offset between the controller and switch $i$ into account, causing $i$ to perform the action at time $T_s$ according to the controller’s clock.

4.2 Prototype Design and Implementation

We have designed and implemented a software-based prototype of Time4, as illustrated in Fig. 7. The components we implemented are marked in black. These components run on Linux, and will soon be publicly available as open source.

Our Time4-enabled OFSoftswitch prototype was adopted by the ONF as the official prototype of Scheduled Bundles.\footnote{The ONF process for adding new features to OpenFlow requires every new feature to be prototyped.}

**Switches.** Every switch $i$ runs an OpenFlow switch software module. Our prototype is based on the open source CPqD OFSoftswitch [46],\footnote{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.} incorporating the switch scheduling module that we implemented. When the switch receives a Scheduled Bundle from the controller, the switch scheduling module identifies that the bundle...
includes an execution time, \( T_s \), and schedules the respective OpenFlow command to the desired time of execution, \( T_s \). This module also replies to Bundle Feature Request messages received from the controller.

Each switch also 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 [47] module. Our software-based implementation uses the Linux clock as the reference for PTPd, and for the switch’s scheduling module.

**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).

## 5 Evaluation

### 5.1 Evaluation Method

**Environment.** We evaluated our prototype on a 71-node testbed in the DeterLab [48] 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. We note that while we considered using Mininet [49] in our evaluation, we decided against it, as Mininet is an emulation environment that runs on a single machine, making it impractical for emulating simultaneous or time-triggered events. We did, however, run our prototype over Mininet in some of our preliminary testing and verification.

**Performance attributes.** Three performance attributes play a key role in our evaluation, as shown in Table 2.
Figure 8: Performance attribute measurement.

Table 2: Performance Attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>∆</td>
<td>The average time elapsed between two consecutive messages sent by the controller.</td>
</tr>
<tr>
<td>I&lt;sub&gt;R&lt;/sub&gt;</td>
<td>Installation latency range: the difference between the maximal rule installation latency and the minimal installation latency.</td>
</tr>
<tr>
<td>δ</td>
<td>Scheduling error: the maximal difference between the actual update time and the scheduled update time.</td>
</tr>
</tbody>
</table>

Intuitively, ∆ and I<sub>R</sub> determine the performance of naïve updates. ∆ indicates the controller’s performance; an OpenFlow controller can handle as many as tens of thousands [50] to millions [51] of packets per second, depending on the type of controller and the machine’s processing power. Hence, ∆ can vary from 1 microsecond to several milliseconds. I<sub>R</sub> indicates the installation latency variation. The installation latency is the time elapsed from the instant the controller sends 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 [7, 40], and grows dramatically with the number of installations per second.

The attribute that affects the performance of timed updates is the switches’ scheduling error, δ. When an update is scheduled to be performed at time T<sub>0</sub>, it is performed in practice at some time t ∈ [T<sub>0</sub>, T<sub>0</sub> + δ].<sup>7</sup> The scheduling error, δ, 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 a clock that is perfectly synchronized to real time. 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 [12] is typically on the order of 1 microsecond [13, 14].

It is an important observation that in a typical system we expect that δ < I<sub>R</sub>, as I<sub>R</sub> is affected by the network latency variation between the controller and switches, as well as by the switch’s software latency, whereas δ is unaffected by the network latency.

**Software-based evaluation.** Our experiments measure the three performance attributes in a setting that uses software switches. While the values we measured do not necessarily reflect on the performance of systems that use hardware-based switches, the merit of our evaluation is that we vary these parameters and analyze how they affect the network update performance with the naïve approach and with TIME4.

<sup>7</sup>An alternative representation of the accuracy, δ, assumes a symmetric error, T<sub>0</sub> ± δ. The two approaches are equivalent.
5.2 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 3. Each attribute was measured 100 times on each machine type, and Fig. 8 illustrates our results. The figure graphically depicts the values \( \Delta \), \( I_R \), and \( \delta \) of machine Type I as an example.

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>( \Delta )</th>
<th>( I_R )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>I Intel Xeon E3 LP</td>
<td>9.64</td>
<td>1.3</td>
<td>1.23</td>
</tr>
<tr>
<td>2.4 GHz, 16 GB RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II Intel Xeon</td>
<td>9.6</td>
<td>1.47</td>
<td>1.18</td>
</tr>
<tr>
<td>2.1 GHz, 4 GB RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II Intel Dual Xeon</td>
<td>14.27</td>
<td>2.72</td>
<td>1.19</td>
</tr>
<tr>
<td>3 GHz, 2 GB RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Measured attributes in milliseconds.

The measured scheduling error, \( \delta \), was slightly more than 1 millisecond in all the machines we tested. Our experiments showed that the clock accuracy using ReversePTP over the DeterLab testbed is on the order of 100 microseconds. The measured value of \( \delta \) in Table 3 shows the execution accuracy, which is an order of magnitude higher. As expected, the installation latency range, \( I_R \), is slightly higher than \( \delta \), around 1 to 3 milliseconds. The measured value of \( \Delta \) was high, on the order of 10 milliseconds, as Dpctl is not optimized for performance.

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.

5.3 Microbenchmark: Video Swapping

To demonstrate how TiME4 is used in a real-life scenario, we reconstructed the video swapping topology of [52], as illustrated in Fig. 9a. Two video cameras, A and B, transmit an uncompressed video stream to targets A and B, respectively. At a given point in time, the two video streams are swapped, so that the stream from source A is transmitted to target B, and the stream from B is sent to target A. As described in [52], the swap must be performed at a specific time instant, in which the video sources transmit data that is not visible to the viewer, making the swap unnoticeable.

![Topology](image)

(a) Topology.

![Video swapping accuracy](chart)

(b) Video swapping accuracy.

Figure 9: Microbenchmark: video swapping.

The authors of [52] noted that the precisely timed swap cannot be performed by an OpenFlow switch, as currently OpenFlow does not provide abstractions for performing accurately timed changes. Instead, they used
source timing, where sources A and B were time-synchronized, and determined the swap time by using a swap indication in the packet header. The OpenFlow switch acted upon the swap indication to determine the correct path for each stream. We note that the main drawback of this source-timed approach is that the SMPTE 2022-6 video streaming standard [53], which was used in [52], does not currently define an indication about where in the video stream a packet comes from, and specifically does not include an indication about the correct swapping time. Hence, off-the-shelf streaming equipment does not provide this indication. In [52], the authors used a dedicated Linux server to integrate the non-standard swap indication.

In this experiment we studied how Time4 can tackle the video swapping scenario, avoiding the above drawback. Each node in the topology of Fig. 9a was emulated by a DeterLab machine. We used two 10 Mbps flows, generated by Iperf [54], to simulate the video streams. Each swap was initiated by the controller 100 milliseconds in advance (as in [52]): the controller sent a Scheduled Bundle, incorporating two updates, one for each of the flows. We repeated the experiment 100 times, and measured the scheduling error.

The measurement was performed by analyzing capture files taken at the sources and at the switch’s egress ports. A swap that was scheduled to be performed at time $T$, was considered accurate if every packet that was transmitted by each of the source before time $T$ was forwarded according to the old configuration, and every packet that was transmitted after $T$ was forwarded according to the new configuration. The scheduling error of each swap (measured in milliseconds) was computed as the number of misrouted packets, divided by the bandwidth of the traffic flow. The sign of the scheduling error indicates whether the swap was performed before the scheduled time (negative error) or after it (positive error).

Fig. 9b illustrates the empirical Probability Density Function (PDF) of the scheduling error of the swap, i.e., the difference between the actual swapping time and the scheduled swapping time. As shown in the figure, the swap is performed within $\pm 0.6$ milliseconds of the scheduled swap time. We note that this is the achievable accuracy in a software-based OpenFlow switch, and that a much higher degree of accuracy, on the order of microseconds, can be achieved if two conditions are met: (i) A hardware switch is used, supporting timed updates with a microsecond accuracy, as shown in [25], and (ii) The cameras are connected to the switch over a single hop, allowing low latency variation, on the order of microseconds.

![Figure 10: Experimental evaluation: every host and switch was emulated by a PC 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.](image)

5.4 Flow Swap Evaluation

1) Experiment Setting

We evaluated our prototype on a 71-node testbed under the DeterLab [48] environment. We used the testbed to emulate an OpenFlow network with 32 hosts and 32 leaf switches, as depicted in Fig. 10, with $n = 32$.

Theorem 4 showed that in the topology of Fig. 10, the source player can generate a sequence of flows that forces the controller to perform a flow swap. Furthermore, by Theorem 2, the source can force the controller to perform a flow swap with an impact as high as roughly 0.5. Hence, our experiments focused on flow swaps with...
(a) The number of packets lost in a flow swap vs. the number of switches involved in the update.

(b) The update duration vs. the number of switches involved in the update.

Figure 11: Flow swap performance as a function of the number of switches.

As discussed in the introduction, two main approaches can be used when a flow swap is required: either the naïve approach, or our Time4-based approach. In each experiment we compared the number of packets lost during a Time4-based swap with the number of packets lost when using the naïve approach to reconfigure the flows.\footnote{The naïve approach bares 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. 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

Number of switches. We evaluated the effect of \( n \), the number of switches at the bottom layer of the topology, on the packet loss. We repeated the experiment for \( n = 2, 4, 8, 16, 32 \). As illustrated in Fig. 11a, the number of packets lost during a naïve 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\footnote{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 \((n - 1)\Delta\).} is longer (Fig. 11b), and hence more packets are lost during the update procedure.

Controller performance. In this experiment we explored how the controller’s performance, represented by \( \Delta \), affects the packet loss rate in a naïve update. As \( \Delta \) increases, the update procedure requires a longer period of time (Fig. 12b), and hence more packets are lost (Fig. 12a) during the process. We note that although previous work has shown that \( \Delta \) can be on the order of microseconds in some cases \cite{51}, Dpctl is not optimized for performance, and hence \( \Delta \) in our experiments was on the order of milliseconds. As shown in Fig. 12a, we synthetically increased \( \Delta \), and observed its effect on the packet loss during flow swaps.

Installation latency variation. Our next experiment (Fig. 13a) examined how the installation latency variation, denoted by \( I_R \), affects the packet loss during a naïve 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. 13a, the switch’s installation latency range, \( I_R \), dramatically affects the packet loss rate during a naïve update. Notably, when \( I_R \) is on the order of 1 second, as in the extreme scenarios of \cite{7, 40}, Time4 has a significant advantage over the naïve approach.
Figure 12: Flow swap performance as a function of the controller’s performance.

(a) The number of packets lost in a flow swap vs. ∆.

(b) The update duration vs. ∆.

Figure 13: Naïve flow swap: performance as a function of the installation latency variation.

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

(b) The update duration vs. the installation latency range, $I_R$.

**Scheduling error.** Figure 14 depicts the packet loss and the update duration as a function of the scheduling accuracy of Time4.

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

(b) The update duration vs. the scheduling error, $δ$.

Figure 14: Time4-based flow swap: performance as a function of scheduling error.

**Summary.** The experiments presented in this subsection demonstrate that if $δ$ is sufficiently low compared to $I_R$ and $(n - 1) \cdot ∆$, then Time4 outperforms the naïve approach. We note that Time4 can be implemented in a way that guarantees a very low value of $δ$, as discussed in Section 6, making Time4 the superior approach. Even if switches are not implemented with extremely low scheduling error, we expect Time4 to outperform the naïve approach, as typically $δ < I_R$ (see Section 5.1).
6 Discussion

Scheduling accuracy. The value 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 [13,14] using PTP [12]. However, a switch’s ability to accurately perform a scheduled action depends on its implementation. We discuss three cases:

- **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 [25] has shown that the scheduling error of timed events in hardware switches is on the order of 1 microsecond.

- **Software-based scheduling in hardware switches:** A scheduling mechanism that relies on the switch’s software may be affected by the switch’s operating system and by other running tasks. Measures can be taken to implement an accurate software-based scheduling in Time4: 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 at this time, such as TCAM entry rearrangement. Update messages received slightly before time $T_s$ can be queued and processed after the scheduled update is executed. Moreover, if a switch receives a timed 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.

Short term vs. long term scheduling. The OpenFlow time extension we presented in Section 4 is intended for short term scheduling; a controller should schedule an action to a near-future time, on the order of seconds in the future. The challenge in long term scheduling is that during the long period between the time at which the Scheduled Bundle was sent and the time at which it is meant to be executed various external events may occur: the controller may fail or reboot, or a second controllers\textsuperscript{10} may try perform a conflicting update. Near future scheduling guarantees that external events such as the examples above have a low probability of occurring; since near-future scheduling is on the order of seconds, this short potentially hazardous period is no worse than in conventional updates, where an OpenFlow command may be executed a few seconds after it was sent by the controller.

7 Conclusion

Time and clocks are valuable tools for coordinating actions at remote sites. Reconfiguration in an SDN network requires carefully coordinated actions. We have shown that in some cases, dynamic traffic steering by SDN controllers requires flow swaps, which are best performed as close to instantaneous as possible. Since clocks can be efficiently synchronized in an SDN network, we showed how 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, however, also be used for a variety of additional timed update scenarios that can help improve network performance during path and policy updates. We see great potential and room for future investigation of consistent time-based updates in SDN.

8 Acknowledgments

We gratefully acknowledge Oron Anschel and Nadav Shiloach for their help with the OFSoftswitch prototype implementation. 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.

\textsuperscript{10}In an SDN with a distributed control plane, where more than one controller is used.
References


Appendix A  A Time Extension to the OpenFlow Protocol

A.1 Introduction

This section defines a time extension to the OpenFlow protocol. This extension allows the controller to send OpenFlow commands that include an execution time, indicating to the switch when the respective command should be performed.

As specified in [2], a bundle is a sequence of (one or more) OpenFlow modification requests from the controller that is applied as a single OpenFlow operation. The controller uses a commit message to apply the set of requests in the bundle. Consequently, the switch applies all messages in the bundle as a single operation or returns an error.

This extension defines scheduled bundles; a bundle commit request may include an execution time, specifying when the bundle should be committed. A switch that receives a scheduled bundle, commits the bundle as close as possible to the execution time that was specified in the commit message.

This document also defines the bundle features message, allowing the controller to retrieve information about the switch’s bundle support, and specifically about its scheduled bundle support.

A.2 How It Works

A.2.1 Overview

This extension allows a bundle operation to be invoked at a scheduled time that is determined by the controller.

The time-based bundle procedure is illustrated in Figure 15:
1. The controller starts the bundle procedure by sending an **OFPBCT_OPEN_REQUEST**, and receives a reply from the switch.

2. The controller then sends a set of \( N \) **OFPT_BUNDLE_ADD_MESSAGE** messages, for some \( N \geq 1 \).

3. The controller MAY then send an **OFPBCT_CLOSE_REQUEST**. The close request is optional, and thus the controller may skip this step.

4. The controller sends an **OFPBCT_COMMIT_REQUEST**. The **OFPBCT_COMMIT_REQUEST** includes two time-related fields: the time flag and optionally the time property. When the time flag is set, it indicates that this is a *scheduled commit*. A scheduled commit request includes the time property field, which contains the scheduled time at which the switch is expected to apply the bundle.

5. After receiving the commit message, the switch applies the bundle at the scheduled time, \( T_s \), and sends a **OFPBCT_COMMIT_REPLY** to the controller.

![Figure 15: Scheduled Bundle Procedure](image)

**Discarding scheduled bundles.** The controller may cancel a scheduled commit by sending an **OFPT_BUNDLE_CONTROL** message with type **OFPBCT_DISCARD_REQUEST**. An example is shown in Figure 16; if the switch is not able to schedule the operation after receiving the commit message, it responds to the controller with an error message (see A.5.1). This indication may be used for implementing a coordinated update where either all the switches successfully schedule the operation, or the bundle is discarded; when a controller receives a scheduling error message from one of the switches it can send a discard message (step 5’ in in Figure 16) to other switches that need to commit a bundle at the same time, and abort the bundle.

**A.2.2 Timekeeping and Synchronization**

Every switch that supports scheduled bundles must maintain a clock. It is assumed that clocks are synchronized by a method that is outside the scope of this document, e.g., the Network Time Protocol (NTP) or the Precision Time Protocol (PTP).

23
Two factors affect how accurately a switch can commit a scheduled bundle; one factor is the accuracy of the clock synchronization method used to synchronize the switches’ clocks, and the second factor is the switch’s ability to execute real-time operations, which greatly depends on how it is implemented.

This document does not define any requirements pertaining to the degree of accuracy of performing scheduled operations. However, every switch that supports the time extension is able to report its estimated scheduling accuracy to the controller. The controller can retrieve this information from the switch using the bundle features message, defined in Section A.4.

Since a switch does not perform configuration changes instantaneously, the processing time of required operations should not be overlooked; in the context of the extension described in this paper the scheduled time and execution time always refer to the start time of the relevant operation.

A.2.3 Scheduling Tolerance

When a switch receives a scheduled commit message, it MUST verify that the scheduled time, $T_s$, is not too far in the past or in the future. As illustrated in Figure 17, the switch verifies that $T_s$ is within the scheduling tolerance range.

The lower bound on $T_s$ verifies the freshness of the packet so as to avoid acting upon old and possibly irrelevant messages. Similarly, the upper bound on $T_s$ guarantees that the switch does not take a long-term commitment to execute an action that may become obsolete by the time it is scheduled to be invoked.

The scheduling tolerance is determined by two parameters, $\text{sched\_max\_future}$ and $\text{sched\_max\_past}$. The default value of these two parameters is 1 second. The controller MAY set these fields to a different value using the bundle features request, as described in Section A.4.

If the scheduled time, $T_s$, is within the scheduling tolerance range, the scheduled commit is performed; if $T_s$ occurs in the past and within the scheduling tolerance, the switch applies the bundle as soon as possible. If $T_s$ is a future time, the switch applies the bundle at $T_s$. If $T_s$ is not within the scheduling tolerance range, the switch responds to the controller with an error message.
A.3 Time-based Bundle Messages

This section updates Section 7.3.9 of [2]. The reader is assumed to be familiar with Sections 6.8 and 7.3.9 of [2].

The time extension allows bundle commit messages to include a time property, defining when the bundle should be executed.

The time extension defines two time-related fields in OFPBCT_COMMIT_REQUEST messages:

- The time flag, denoted OFPBF_TIME.
- The time property.

All OFPT_BUNDLE_CONTROL messages include the OFPBF_TIME flag. In control messages with type OFPBCT_COMMIT_REQUEST the time flag MAY be set, indicating that the time property field is present. The time property incorporates the time at which the switch is scheduled to apply the bundle.

Control messages with a type that is not OFPBCT_COMMIT_REQUEST MUST have the OFPBF_TIME flag disabled, and this flag is ignored by the switch in these messages.

A.3.1 The Time Flag

This document updates ofp_bundle_flags by adding the OFPBF_TIME flag, as follows:

```c
/* Bundle configuration flags. */
enum ofp_bundle_flags {
    OFPBF_TIME = 1 << 2, /* Execute in a specific time. */
};
```

A.3.2 The Bundle Time Property

This document defines a new bundle property, the time property.

```c
/* Bundle property */
struct ofp_bundle_prop_time {
    uint16_t type; /* OFPBPT_TIME */
    uint16_t length; /* Length in bytes = 24 */
    uint8_t pad[4];
};
```
struct ofp_time scheduled_time; /* The scheduled time at which the switch should apply the bundle. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_prop_time) == 24);

The `type` field in the time property is set to the value `OFPBPT_TIME`, defined as follows:

```c
/* Bundle property types. */
enum ofp_bundle_prop_type {
    OFPBPT_TIME = 1, /* Time property. */
};
```

### A.3.3 Time Format

The time format defined in this extension is based on the one defined in [12]. It consists of two sub-fields; a `seconds` field, representing the integer portion of time in seconds\(^{11}\), and a `nanoseconds` field, representing the fractional portion of time in nanoseconds, i.e., \(0 \leq \text{nanoseconds} \leq (10^9 - 1)\).

```c
/* Time format */
struct ofp_time {
    uint64_t seconds;
    uint32_t nanoseconds;
    uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_time) == 16);
```

As defined in [12], time is measured according to the International Atomic Time (TAI) timescale. The epoch is defined as 1 January 1970 00:00:00 TAI.

### A.4 Bundle Features Request

The bundle features request defined in this document allows a controller to query a switch about its bundle capabilities, including its scheduled bundle capabilities.

This section extends Section 7.3.5 of [2]. The reader is assumed to be familiar with Section 7.3.5 of [2]. The bundle features request is a new multipart message type, the `OFPMP_BUNDLE_FEATURES` message. This document updates `ofp_multipart_type` by adding the `OFPMP_BUNDLE_FEATURES` type, as follows:

```c
enum ofp_multipart_type {
    /* Bundle features. */
    /* The request body is ofp_bundle_features_request. */
    /* The reply body is struct ofp_bundle_features. */
    OFPMP_BUNDLE_FEATURES = 17,
};
```

### A.4.1 Bundle Features Request Message Format

The body of the bundle features request message is defined by `struct ofp_bundle_features_request`, as follows:

```c
/* Body of OFPMP_BUNDLE_FEATURES request. */
struct ofp_bundle_features_request {
    uint32_t feature_request_flags; /* Bitmap of "ofp_bundle_feature_flags". */
    uint8_t pad[4];

    /* Bundle features property list - 0 or more. */
    struct ofp_bundle_features_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_bundle_features) == 8);
```

The body consists of a flags field, followed by zero or more property TLV fields. The flags field, `feature_request_flags`, is defined as follows:

---

\(^{11}\)The seconds field in IEEE 1588 is 48 bits long. The seconds field used in this extension is a 64-bit field, but it has the same semantics as the seconds field in the IEEE 1588 time format.
/* Flags used in a OFPMP_BUNDLE_FEATURES request. */
enum ofp_bundle_feature_flags {
    OFPB_TIMESTAMP = 1 << 0,  /**< When enabled, the current request includes a timestamp, using */
                      /**< the time property */
    OFPB_TIME_SET_SCHED = 1 << 1,  /**< When enabled, the current request includes the sched_max_future */
                      /**< and sched_max_past parameters, using the time property */
}

If at least one of the flags OFPB_TIMESTAMP or OFPB_TIME_SET_SCHED is set, the bundle features request includes a time property.

The bundle features properties are specified in Section A.4.3.

A.4.2 Bundle Features Reply Message Format

If the features request is processed successfully by the switch, it sends a reply to the controller. The body of the bundle features reply message is struct ofp_bundle_features, as follows:

/* Body of reply to OFPMP_BUNDLE_FEATURES request. */
struct ofp_bundle_features {
    uint16_t capabilities; /**< Bitmap of "ofp_bundle_flags". */
                      /**< ofp_time pads[6];               */

    /**< Bundle features property list - 0 or more. */
    struct ofp_bundle_features_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_bundle_features) == 8);

A.4.3 Bundle Features Properties

The optional property fields are defined as TLVs with a common header format, as follows:

/* Common header for all bundle feature Properties */
struct ofp_bundle_features_prop_header {
    uint16_t type; /**< One of OFPTMPBF_. */
                    /**< length; /**< Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_features_prop_header) == 4);

The currently defined types are as follows:

/* Bundle features property types. */
enum ofp_bundle_features_prop_type {
    OFPTMPBF_TIME_CAPABILITY = 0x1, /**< Time feature property. */
    OFPTMPBF_EXPERIMENTER = 0xFFFF, /**< Experimenter property. */
};

The Bundle Features Time Property.

A bundle feature request in which at least one of the flags OFPB_TIMESTAMP or OFPB_TIME_SET_SCHED is set, incorporates the time property. A bundle feature reply that has the OFPB_TIME flag set incorporates the time property.

The time property is defined as follows:

struct ofp_bundle_features_prop_time {
    uint16_t type; /**< OFPTMPBF_TIME_CAPABILITY. */
                    /**< length; /**< Length in bytes of this property. */
    uint8_t pads[4];

    struct ofp_time sched_accuracy; /**< The scheduling accuracy, i.e., how accurately the switch can */
                                     /**< perform a scheduled commit. This field is used only in bundle */
                                     /**< features replies, and is ignored in bundle features requests. */
    struct ofp_time sched_max_future; /**< The maximal difference between the */
                                        /**< scheduling time and the current time. */
    struct ofp_time sched_max_past; /**< If the scheduling time occurs in the past, defines the maximal

The time property in a bundle features request includes:

- **sched_accuracy**: this field is relevant only to bundle features replies, and the switch must ignore this field in a bundle features request.

- **sched_max_future** and **sched_max_past**: a switch that receives a bundle features request with `OFPBF_TIME_SET_SCHED` set, should attempt to change its scheduling tolerance values according to the `sched_max_future` and `sched_max_past` values from the time property. If the switch does not successfully update its scheduling tolerance values, it replies with an error message.

- **timestamp**, indicating the controller’s time during the transmission of this message. A switch that receives a bundle features request with `OFPBF_TIMESTAMP` set, may use the received timestamp to roughly estimate the offset between its clock and the controller’s clock.

The time property in a bundle features reply includes:

- **sched_accuracy**, indicating the estimated scheduling accuracy of the switch. For example, if the value of `sched_accuracy` is 1000000 nanoseconds (1 ms), it means that when the switch receives a bundle commit scheduled to time $T_s$, the commit will in practice be invoked at $T_s \pm 1$ ms. The factors that affect the scheduling accuracy are discussed in Section A.2.2.

- **sched_max_future** and **sched_max_past**, containing the scheduling tolerance values of the switch. If the corresponding bundle features request has the `OFPBF_SET_TIME_TOLERANCE` flag enabled, these two fields are identical to the ones sent by the controller in the request.

- **timestamp**, indicating the switch’s time during the transmission of this feature reply. Every bundle feature reply that includes the time property also includes a timestamp. The timestamp may be used by the controller to get a rough estimate of whether the switch’s clock is synchronized to the controller’s.

### A.5 Errors

As defined in Section 7.5.4 of [2] the switch can send an error message to the controller, which includes a type and a code. This document extends Section 7.5.4 with additional codes, as specified below.

#### A.5.1 Bundle Error

When the switch has an error related to the bundle operation, it sends an error message with type `OFPET_BUNDLE_FAILED`. This document defines the following new codes:

- **OFPBFC_SCHED_NOT_SUPPORTED** - this code is used when the switch does not support scheduled bundle execution and receives a commit message with the `OFPBF_TIME` flag set.

- **OFPBFC_SCHED_FUTURE** - used when the switch receives a scheduled commit message and the scheduling time exceeds the `sched_max_future` (see Section A.2.3).

- **OFPBFC_SCHED_PAST** - used when the switch receives a scheduled commit message and the scheduling time exceeds the `sched_max_past` (see Section A.2.3).

The `ofp_bundle_failed_code` is updated as follows:

```c
enum ofp_bundle_failed_code {
    OFPBFC_SCHED_NOT_SUPPORTED = 16, /* Scheduled commit was received and scheduling is not supported. */
    OFPBFC_SCHED_FUTURE = 17, /* Scheduled commit time exceeds upper bound. */
    OFPBFC_SCHED_PAST = 18, /* Scheduled commit time exceeds lower bound. */
};
```
A.5.2 Bundle Features Error

When the switch has an error related to the `OFPMP_BUNDLE_FEATURES` request, it replies with an error message of type `OFPET_BAD_REQUEST`. The code `OFPBRC_MULTIPART_BAD_SCHED` indicates that the request had the `OFPBF_SET_TIME_TOLERANCE` flag enabled, and the switch failed to update the scheduling tolerance values.

The `ofp_bad_request_code` is updated as follows:

```c
enum ofp_bad_request_code {
    OFPBRC_MULTIPART_BAD_SCHED = 16, /* Switch received a OFPMP_BUNDLE_FEATURES request and failed
                                 * to update the scheduling tolerance. */
};
```