Article

Towards Causal Consistent Updates in Software-Defined Networks

Amine Guidara ^{1,2,*}, Saúl E. Pomares Hernández ^{1,3,4} , Lil María X. Rodríguez Henríquez ^{1,5} ,
Hatem Hadj Kacem ² and Ahmed Hadj Kacem ²

¹ Department of Computer Science, Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE), Tonantzintla, Puebla 72840, Mexico; spomares@inaoe.mx (S.E.P.H.); lmrodriguez@inaoe.mx (L.M.X.R.H.)

² ReDCAD Laboratory, University of Sfax, Sfax 3029, Tunisia; hatem.hadjkacem@redcad.org (H.H.K.); ahmed.hadjkacem@redcad.org (A.H.K.)

³ CNRS, LAAS, 7 Avenue du Colonel Roche, F-31400 Toulouse, France

⁴ Université de Toulouse, LAAS, F-31400 Toulouse, France

⁵ Consejo Nacional de Ciencia y Tecnología (CONACYT), Av. Insurgentes Sur 1582, Col. Crédito Constructor Del. Benito Juárez C.P., Mexico City 03940, Mexico

* Correspondence: guidara.amine@inaoe.mx; Tel.: +52-(221)-160-4451

Received: 1 February 2020; Accepted: 12 March 2020; Published: 19 March 2020

Abstract: A network paradigm called the Software-Defined Network (SDN) has recently been introduced. The idea of SDN is to separate the control logic from forwarding devices to enable a centralized control platform. However, SDN is still a distributed and asynchronous system: events can be triggered by any network entity, while messages and packets are prone to arbitrary and unpredictable transmission delays. Moreover, the absence of a global temporal reference results in a broad combinatorial range space of event order. During network updates, an out-of-order execution of events may result in a deviation from desirable consistent network update properties, leading, for example, to forwarding loops and forwarding black holes, among others. In this paper, we introduce a study of the Transient Forwarding Loop (TFL) phenomenon during SDN updates; for this, we define a formal model of the TFL based on causal dependencies that capture the conditions under which it may occur. Based on this model, we introduce an algorithm that ensures the causal dependencies of the system oriented toward TFL-free SDN updating. We formally prove that it is sufficient to ensure the causal dependencies in order to guarantee TFL-free network updates. Finally, we analytically evaluate our algorithm and discuss how it outperforms the state-of-the-art in terms of updating overhead.

Keywords: software-defined networks; OpenFlow; Transient Forwarding Loop (TFL); pattern; happened-before relation; causal order delivery; consistent network updates

1. Introduction

Currently, Software-Defined Networks (SDNs) present a revolution in the field of computer networks since they have reshaped several concepts of IP networks [1]. An important concept is that the network control logic is decoupled from forwarding devices. Indeed, the control logic is moved from forwarding devices, and it is encapsulated into an external entity called the controller (control plane). Through the latter, and by leveraging a logically centralized network view and standardized communication protocols (e.g., OpenFlow [2] and ForCES [3]), forwarding devices (data plane) become programmable. Despite the concept of a logically centralized controller, SDN remains a distributed system wherein the control and data planes cooperate and communicate via an asynchronous communication (asynchronous communication implies arbitrary and unpredictable

transmission delays) interface to establish networking. Furthermore, an out-of-order execution of events may occur since no global temporal reference is shared between network entities and message delays are arbitrary. This leads to the following problem: when updating a network while packet flows are taking routes to their destination, an out-of-order execution could give rise to non-deterministic behavior that temporarily deviates from network properties, which in turn may result in an inconsistent network update. Moreover and as a result, the network-wide view from the controller can transitorily be inconsistent with the current data plane state, which could affect the consistency of future network updates. To ensure consistent updates, depending on the network application, the network should align with some properties, such as no Transient Forwarding Loop (TFL) and no forwarding black hole, among others. The no TFL is one of the essential network properties desired by several network applications, including traffic engineering, virtual machine migration, and planned maintenance [4]. Informally, it ensures that a packet is never forwarded along a loop back during an arbitrary time interval to a forwarding device in the network where it was previously processed. To the best of our knowledge, (i) no study formally specifies under which conditions TFLs may occur in the context of SDNs. Furthermore, (ii) no solution to this problem aligns with the distributed and asynchronous nature of the SDNs. Indeed, the proposed solutions are centralized or synchronized. In fact, a centralized-based solution is associated with memory overhead: to perform updates, it makes use of the centralized controller with full knowledge of the network, i.e., the complete network forwarding graph. In addition, calculating consistent updates based on full knowledge of the network in large-scale networks becomes costly in terms of performance overhead. On the other hand, a synchronized-based solution comes with bandwidth overhead: it relies on synchronizing switch clocks to perform updating operations simultaneously. Apart from the control information overhead, such a solution presents a risk of inconsistency during the transition phases of updates since synchronization protocols do not perfectly synchronize switch clocks. We argue that achieving updates in a loop-free manner with a trade-off between ensuring consistent updates and performing efficient updates is still a challenging open problem.

In this paper, we introduce the first of a family of patterns: TFL is defined to capture asynchronously and avoid preventively it at run-time during SDN updates. The main contributions are the following:

1. OpenFlow-based SDN updates are modeled at the event level according to the distributed and asynchronous nature of SDNs.
2. A formal model of the TFL based on temporal and causal dependencies (the causal dependencies are based on the happened-before relation defined by Lamport in [5] (see Section 5 for more details) that capture the conditions under which it may occur is presented. We highlight that this study is a key contribution in this work since it defines the root cause behind the triggering of TFLs and allows us to define how to achieve a TFL-free property when updating SDNs.
3. A causal consistent update algorithm oriented to ensure the TFL-free property is presented. This algorithm is based on the work of Prakash et al. [6]
4. A proof of correctness that shows that the algorithm is TFL-free is provided.
5. The proposed algorithm is analytically evaluated, concluding that it enhances the update performance overhead.

The rest of this paper is structured as follows. Section 2 presents the preliminaries for the rest of this paper. Section 3 motivates the importance of the problem. A frame and a discussion of related work are presented in Section 4. Section 5 describes the network model. A study of the problem from a temporal perspective is described in Section 6. The proposed solution is presented in Section 7. An example scenario is described in Section 8. We evaluate and discuss the proposed approach in Section 9. Conclusions and future work are discussed in Section 10.

2. Preliminaries

2.1. Fundamental Abstractions of SDNs

A Software-Defined Network (SDN) refers to a new generation in the evolution of computer networks. SDN was launched in 2008. It was standardized by the Open Network Foundation [7] and implemented by a number of original equipment manufacturers (HP, CISCO, IBM, Juniper, NEC, and Ericsson).

SDN is an emerging paradigm that relies on decoupling the tightly coupled implementation of the network control logic and network devices. In fact, the control logic (control plane) is separated from network devices (data plane) and is implemented in a logically centralized controller [8]. Thus, SDN mainly consists of three planes: the application plane, the control plane, and the data plane. Figure 1 depicts a simplified view of the SDN architecture. The application plane is presented as a set of network applications that implement network control logic (e.g., firewall, traffic engineering, load-balancer, etc.) leveraging a northbound interface, e.g., REST API [9], which offers universal network abstraction data models and functionality to developers [1,8]. This set of network applications also either explicitly or directly notifies the network behavior to the control plane by means of a northbound interface. The former is presented by a logically centralized controller, which is also named the Network Operating System (NOS), and supports the translation of the network requirements and the desired behavior of the application plane to the data plane based on a southbound interface, e.g., OpenFlow [2] and ForCES [3]. Indeed, a southbound interface formalizes the way in which the control plane and the data plane communicate. Finally, the data plane is presented by the set of network devices, e.g., switches and routers, which remain a set of simple forwarding devices [1,8].

Figure 1. The general architecture of SDN.

2.2. Network Traffic Handling in an OpenFlow-Based SDN

This work of research is built on an OpenFlow-based SDN [2]. OpenFlow proposes a flow-based traffic forwarding decision approach. This forwarding approach enables flexibility on programming network traffic as all packets that belong to the same flow receive the same forwarding decisions [1]. To (re)configure switches, the controller interacts with switches via an OpenFlow channel interface that connects each switch with the controller and by exchanging OpenFlow messages based on the OpenFlow switch protocol. This protocol provides a reliable message delivery and processing but does not ensure ordered message processing [2] since the communication between the OpenFlow controller and the switches is asynchronous. In this paper, we are interested in a single type of message: a message instructing an OpenFlow switch in the network to update its flow table with new entries, namely the *FlowMod* controller-to-switch message (see the details in Section 5). In fact, each switch contains one or more flow tables that store a set of entries (forwarding rules). An entry consists of

matching fields and forwarding actions. Each entry matches a flow or a set of flows and performs certain actions (dropping, forwarding, modifying, etc.) on the matched flows.

3. Problem Description: TFL as an Inconsistent Network Update Problem

Despite the centralization of the logic control, SDN remains a distributed and an asynchronous system. Indeed, the data plane is managed by a logically-centralized controller that communicates with the data plane only by OpenFlow message passing via asynchronous communication channels. In the real world, during updates, applications running on the controller may compile several entries, requiring the controller to disseminate OpenFlow messages to install entries on switches. Therefore, the switches may deliver OpenFlow messages that are injected by the controller and in-fly packets, which interleave with update messages in any order, leading to inconsistent updates.

Flow swaps: a motivating scenario. In order to analyze the TFL phenomenon in SDNs, we use a common and inevitable network update scenario known as flow swaps. Figure 2 illustrates an example of a basic flow swap scenario. In this scenario, the topology is composed of an OpenFlow controller and two switches. Initially, Switch 1 (S_1) contains an entry that forwards a specific packet flow f_i to Switch 2 (S_2) (see Figure 2a). Due to a network policy change, f_i should be forwarded from S_2 to S_1 . To this end, the OpenFlow controller starts by instructing, based on two controller-to-switch OpenFlow messages of type *FlowMod*, (1) S_1 to delete the entry directing f_i from S_1 to S_2 (see Figure 2b) and, then, (2) S_2 to install a new entry to forward f_i from S_2 to S_1 . While the controller sends Operation (1) before Operation (2), the second operation finishes before the first one. In this case, an in-fly packet flow f_j may interleave with the installation of the rule in S_2 , forwarding network traffic to S_1 , and then f_j enters into a TFL between S_1 and S_2 .

Figure 2. Flow swap scenario: (a) initial forwarding policy; (b) after updating the initial forwarding policy.

Based on the previous example, an SDN update may result in a broad combinatorial range space of messages and packets ordering, and thus, the controller may not assume anything about the data plane state. Therefore, the network wide-view from the controller may be temporarily or permanently inconsistent with the current data plane state. Intuitively, this affects the consistency of network updates. Indeed, three of eleven bugs reported in [10] were caused by inconsistent views from the controller. Alternatively, an OpenFlow controller can explicitly request acknowledgments by sending a barrier request message (*BarrierReq*) after sending a message, that is it sets a synchronization point between messages. An OpenFlow switch of a destination should then respond with a barrier response message (*BarrierRes*) once the message and all other messages received before the message have been processed [2]. However, in which case and time should the controller explicitly request to install a *BarrierReq* message? What about the overhead of messages in large-scale networks? On the other hand, such a solution forces the system to stop processing any new requests until all messages sent before the *BarrierReq* message are completely processed. Indeed, this solution can harm performance by increasing the number of control messages, controllers, switch operations, and delays.

4. Related Work

Current network requirements have changed, and even more after the introduction of SDNs, which support and facilitate network updates.

As SDNs have facilitated the updating of the networks, the problems related to them have been studied intensively. This scope has many axes that are relevant, including updating techniques and performance goals/objectives. That is why there is more than one classification of works that have addressed the problem, such as the proposal of [11]. The work consists of a profound review of the consistent updating approaches in SDNs. This overview generalizes the network updating problem and proposes a taxonomy of consistent network updating problems in which they are classified by the type of consistency network update (connectivity, policy, and capacity consistency) and by objectives (when link-based, to make a new links available as soon as possible; when round-based, to minimize the number of makespan; and when cross-flow, to minimize the number of interactions between network entities during updates). Indeed, this taxonomy provides an orthogonal view in terms of the attacked network update problems and the works' objectives. However, in this section, we focus on classifying works that attack the forwarding loop problem by these techniques. These techniques contribute to the prevention of the mentioned problem while measuring their impact on the network performance. For more details about this, please refer to Section 9.

In the literature, works that have been proposed to tackle the problem fall into one of the following approaches: either the ordered update [12–15], the n-phase commit update [4,15–18], the timed update [19,20], or the causal update [21] approach. On the other hand, another set of works focuses on studying optimization problems [22–27] of consistent updates in SDNs. These two kinds of works are briefly discussed in this section.

Ordered updating approaches [12–15] use a sequence of steps to execute reconfigurations (controller commands) on switches, where the order of execution ensures that no inconsistent behavior appears throughout the transition from one configuration to another. During the transition at each step, the controller should wait until all the underlying switches finish their updates (referred to as rule replacements) and then inform it (through acknowledgment messages) in order to be able to initiate the next step of the updating sequence. An update finishes after the achievement of all the updating steps.

As for the n-phase commit update approaches, we distinguish between the two-phase commit and the one-commit update approaches. Works like [4,16], which were based on the two-phase commit, provided a strong per-packet consistency guarantee: packets are forwarded to their destinations based on the initial or the final configuration and not both, covering a wide range of consistent updating properties that include connectivity and policy consistencies. These works are based on a packet tag-match mechanism. Initially, all in-fly packets are tagged by switches with the current forwarding configuration k . The controller starts by disseminating all updating commands of configuration $k + 1$ and then waits for acknowledgments from the underlying switches. Once all switches acknowledge the execution of commands (adding forwarding rules) of configuration $k + 1$, then the controller instructs them to tag all incoming packets with $k + 1$ to the match forwarding rules of the new configuration. During the update transition phase, all switches should maintain the forwarding rules of both configurations (referred by rule additions) until all packets tagged with k leave the network. When they leave, the controller instructs the switches to remove commands of version k .

On the other hand, ez-Segway [17] guarantees the following network properties: loop-freedom, black hole-freedom, and congestion-freedom. The scope of this work consists of having the controller delegate the task of performing network update operations to the switches implicated in the updates. In fact, the controller computes and sends the information needed to the switches once per update, which qualifies it as a one-phase commit approach. This is based on the basic-update technique. Once the first switch is involved in the new path, a message is sent from it towards their successors and until the destination switch to acknowledge that no packets will be forwarded anymore along the old path. Therefore, every switch that receives this new message removes the old forwarding entry

of the old path and afterwards forwards it to its successor on the old path. To speed up the flow update, a segmentation of basic-update is proposed. The idea here is to split the update of a flow path of the old and the new paths into subpaths and, at the same time, perform their updates. Taking the forwarding loop case, the controller classifies segments into two categories, called InLoop and NotInLoop segments, and calculates a dependency mapping that assigns each InLoop segment to a NotInLoop segment. On the other hand, switches should install the new path and remove the old path based on the dependency mappings established by the controller and exchange messages among them to establish a selection of update operations to be executed.

Recently, the authors of [15] proposed FLIP, an updating approach built on the dualism between the ordered update and the two-phase commit update approaches. FLIP combines the rule replacements and rule additions for matching packets with their forwarding rules in all switches, preserving the per-packet consistency property. The main contribution of this work involves reducing the rule additions during the transition update phase in the two-phase commit update approaches. This is based on two core procedures: constraint extraction and constraint swapping. The former consists of identifying the constraints (replacement and/or addition constraints) that ensure a safe update. Through the identified constraints, alternative constraints to the extracted ones are also inferred. After having extracted all the constraints, FLIP tries to calculate a sequence of update steps that satisfy all active constraints by applying a linear program where the objective is to minimize the number of update steps. If a solution can be found, FLIP outputs the operational sequences, otherwise it jumps to apply the constraint swapping procedure. This consists of considering some alternative constraints to replace some active constraints to calculate the operation steps. Based on these procedures, FLIP eventually reaches a combination between the two approaches for which a solution exists.

A timed consistent network update approach [19,20] has been proposed. This approach is based on establishing an accurate time to trigger network updates, that is synchronizing clocks of switches to simultaneously perform the executions of controller commands. An important advantage is that timed updates quickly become available for the in-fly packets as controller commands (adding and removing rules) are executed on the switches at the same time. Furthermore, it prevents switch congestion during the updates. In terms of consistency, a timed update achieves the per-packet consistency; however, it does not compromise the consistency during updates due to the risk that switch clocks may not effectively be synchronized.

Recently, the authors of [21] proposed a new update approach based on the suffix causal consistency property. This property ensures, during routing policy updates, that a packet traverses the most recent path specified for it and for which it encounters forwarding rules, ensuring consistency properties, like bounded looping, among others. The authors of this work proposed to adopt the Lamport timestamp to tag packets in order to reflect the rules that correspond to each switch. This ensures that an in-fly packet is forwarded based on rules with a timestamp at least equal to the timestamps of rules where it has been processed in upstream switches. To do so, a controller module operation and switch module operation are designed to manage and maintain the required timestamps. Regarding the proposed update algorithm, there are four important steps: the first is the backward closure step whose role is to include the new forwarding rules that precede those already included. This step permits propagating the installation of new rules backward along routing paths. The same technique is applied to include the new rules that follow those already included, ensuring the second step named forward closure. The third step is tagging timestamps, which sets the new added rule timestamps that have not been set in the preceding steps. The send-back rules is the fourth step, in which a new and temporary rule is added if a packet takes its old path and encounters a switch at which the rule that would have matched in the old configuration has been already deleted. In this case, this temporary rule allows forwarding the packet in the direction of the new path from where it can continue traveling to its destination.

Works like [22–27] proposed studying optimization problems on consistent updates in SDNs. The authors of [22] had as an objective to update as many switches as possible at a time, in what is known as node-based, in a loop-free manner. The authors of [23] showed that the node-based optimization problem is NP-hard. In the same context, the authors of [24] demonstrated that scheduling consistent updates for two destinations is NP-hard for a sublinear number of update rounds. In [25], the authors presented an approach that aims to minimize the number of messages needed between the controller and the switches to update networks, known as round-based, in a loop-free manner. In the same context, the authors in [26] distinguished between two loop-free network updates, the strong loop-freedom and the relaxed loop-freedom. The strong loop-freedom requires that the entries (i.e., forwarding rules) stored at the switches be loop-free at all points of time, whereas the relaxed loop-freedom requires entries stored by switches along a forwarding path to be loop-free. The problem was shown to be difficult in the strong loop-freedom case. In the worst case, $\Omega(n)$ rounds may be required, where n is the DSdefinition. On the other hand, $O(\log n)$ rounds always exist in the relaxed loop-freedom case. In [27], the authors studied local checkability properties like s-treachability (i.e., whether the network is acyclic or contains a cycle), introduced in [28]. This work studies the applicability of such theory in the context of SDNs by considering the case of updating spanning trees in a loop-free manner. Unlike previous works, the scope of this work is to determine a round-based update with a constant number of rounds, in such a way that every switch should be able to locally decide if it can change its forwarding behavior.

5. System Model

We consider a network represented through a set of nodes $N = \{n_1, n_2, \dots\}$ (defined later as a set of processes) and a generic (random) forwarding path, leading messages and packet flows, between a node n_i and another n_j (represented later as a set of forwarding paths), which is subject to being updated.

To formalize this system, we introduce a model of executions of message- and packet-passing between OpenFlow-based SDN entities during updates. First of all, in Section 5.1, we present the model related to a typical distributed system. Then, in the next subsection, we define relevant concepts for this paper, basically the Happened-Before Relation (HBR) and the Immediate Dependency Relation (IDR) definitions. After that, in Section 5.3, we introduce our SDN model by adapting the typical distributed system model to the SDN context and extending it to cover other sets specific to SDNs. Finally, in Section 5.4, we discuss the causal order delivery, which is a fundamental property for our proposed solution.

5.1. Distributed System Model

To introduce our SDN model, we start by presenting the model of a typical distributed system as a point of departure. A typical distributed system is composed of different separated entities that communicate with each other by exchanging messages. It is assumed that no global time reference exists and the message transmission delay is arbitrary and unpredictable. Furthermore, for simplicity in this work, it is assumed that no message is lost.

At a high level of abstraction, a distributed system can be described based on the following sets: P , M , and E , which correspond, respectively, to the set of processes, the set of messages, and the set of events.

- **Processes:** programs, instances of programs, or threads running simultaneously and communicating with other programs, instances of programs or threads. Each process belongs to the set of processes P . A process $p \in P$ can only communicate with other processes by message passing over an asynchronous communication network.
- **Messages:** Abstractions of any type of messages, which can contain either arbitrarily simple or complex data structures. Each message in the system belongs to the set of messages M . A message $m \in M$ is sent considering an asynchronous and reliable network.

- **Events:** An event e is an action or occurrence performed by a process $p \in P$. Each event e in the system belongs to the set E . There are two types of events under consideration: internal events and external events. An internal event is an action that occurs at a process in a local manner. An external event is an action that occurs in a process, but it is seen by other processes and affects the global system state. The external events are the send and delivery events. A send event identifies the emission event of a message $m \in M$ executed by a process, whereas a delivery event identifies the execution performed or the consumption of an ingoing message m by a recipient process.

5.2. Time and Causal Order

Time is an important theoretical construct to understand how the transactions are executed in distributed systems. Indeed, it is difficult to determine if an event takes place before another one in the absence of a global physical time. However, the logical time introduces an agreed time between all processes based on which one can establish the execution order relation between any two events in a distributed system. This order relation was defined by Lamport [5] by means of the Happened-Before Relation (HBR). The HBR is a strict partial order, and it establishes precedence dependencies between events. The HBR is also known as the relation of causal order or causality.

Definition 1. The Happened-Before Relation (HBR) [5] “ $\rightarrow$ ” is the smallest relation on a set of events E satisfying the following conditions:

- If a and b are events that belong to the same process and a occurred before b , then $a \rightarrow b$.
- If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

In practice, the use of HBR to maintain the causality is expensive since the relation between each pair of events should be considered, including the transitively Happened-before relationships between events (defined in the third condition of the HBR definition). To obliterate the notion that causality is expensive to set up in distributed systems, the author of [29] proposed the Immediate Dependency Relation (IDR). The IDR is the minimal binary relation of the HBR that has the same transitive closure. We use the IDR to send the necessary and sufficient amount of control information sent per message to ensure causal order.

Definition 2. The Immediate Dependency Relation (IDR) [29] “ $\downarrow$ ” is the transitive reduction of the HBR, and it is defined as follows:

$$a \downarrow b \text{ if } a \rightarrow b \wedge \forall c \in E, \neg(a \rightarrow c \rightarrow b)$$

5.3. SDN Model

We developed the SDN model by adapting the sets P , M (also extended to the set MP), and E , presented in the distributed system model of Section 5.1, to the SDN context and by adding the sets $MATCH$, MP , $PFLOW$, and $FPATH$, which correspond, respectively, to the set of matches, the set of messages and data packets, the set of packet flows, and the set of forwarding paths, which are specific to the SDN model.

- **Processes:** The system under consideration is composed of a set of processes $P = \{p_1, p_2, \dots\} \mid P = P_{cp} \cup P_{rp}$ where $P_{cp} = \{cp\}$ represents the controller process cp and $P_{rp} = \{rp_1, rp_2, \dots\}$ represents the set of routing processes of OpenFlow switches.
- **Matches:** We consider a finite set of matches $MATCH = \{match_1, match_2, \dots\}$. A match $match \in MATCH$ is a field value that identifies a forwarding path in the network.

It is important to mention that the match is a key attribute in establishing the proposed updating algorithm (see Section 7.2). In fact, each update is performed based on a match. This is because an update makes reference to a forwarding path $fpath_\mu \in FPATH$, a packet flow $pflow_\tau \in PFLOW$, which is taking route to its destination according to $fpath_\mu$, and to any OpenFlow message $m \in M$ disseminated by the controller to update any routing process $rp \in fpath_\mu$ that shares the same match $match_\lambda \in MATCH$.

- Messages and data packets: The system includes a set of messages and data packets $MP = \{mp_1, mp_2, \dots\} \mid MP = M \cup PKT$ where $M = \{m_1, m_2, \dots\}$ represents the OpenFlow messages and $PKT = \{pkt_1, pkt_2, \dots\}$ represents the data packets.

In addition to the set of messages M , we consider the set of OpenFlow message types M_{type} [2] where $f_{message} : M \mapsto M_{type}$. Furthermore, we note that each message $m \in M$ corresponds to a $match \in MATCH$, denoted by $m \hat{=} match$. Furthermore, a $match$ may also correspond to a subset of OpenFlow messages. We denote a message $m \in M$ as $m = (cp, t_{cp}, OpenFlow_message, rp_j)$ where a controller cp sends an $OpenFlow_message$ to a $rp_j \in P_{rp}$ at t_{cp} (the logical clock of cp). Note that the tuple (cp, t_{cp}) represents the identifier of a message $m \in M$.

In this paper, we consider controller-to-switch messages, and in particular the *FlowMod* message. This message allows the controller to modify the state of OpenFlow switches either by adding, modifying, or deleting entries. The *FlowMod* message is composed of various structures (see the details in [2]). In this work, we consider *match* and *command* as the relevant *FlowMod* message structures. In fact, the former specifies to which packet flow an entry corresponds, whereas the latter specifies the action to be performed on the matched packet flow. We consider *FlowMod* as a relevant message, where *command* and *match* represent the “*OpenFlow_message*” structure in the specification of a message $m \in M$.

A packet pkt in the subset PKT mentioned above is denoted as $pkt = (rp_i, t_i, header, data, rp_j)$, where an rp_i forwards at the logical clock t_i a data packet, composed of a *header* and *data*, to an rp_j such that $rp_i, rp_j \in P_{rp}$ and $(rp_i \neq rp_j)$. Note that the tuple (rp_i, t_i) represents the identifier of a data packet $pkt_i \in PKT$. The header of each data packet piggybacks a $match \in MATCH$ that corresponds to all *match* of packets belonging to the same packet flow. The data consist of the payload (the current intended message).

- Packet flow: We consider a finite set of packet flows $PFLOW = \{pflow_1, pflow_2, \dots\}$. A $pflow \in PFLOW$ (also $pflow \subset PKT$) is a sequence of packets between a source rp_i and a destination rp_j , ($rp_i, rp_j \in P_{rp}$). Furthermore, we consider the bijection $f_{pflow} : PFLOW \mapsto MATCH$, that is each $pflow$ corresponds to a $match \in MATCH$ denoted by $pflow \hat{=} match$.
- Forwarding paths: A finite set of forwarding paths $FPATH = \{fpath_1, fpath_2, \dots\}$ is considered. An $fpath \in FPATH$ is a subset of routing processes $Rp = \{rp_k, rp_{k+1}, rp_{k+2}, \dots, rp_{k+n}\}$ between an rp_{src} source and an rp_{dst} destination, where $rp_i, rp_j \in P_{rp}$ and $Rp \subset P_{rp}$. Furthermore, we take into consideration the bijection $f_{fpath} : FPATH \mapsto MATCH$, that is each $fpath$ corresponds to a $match \in MATCH$ denoted by $fpath \hat{=} match$.
- Events: As mentioned in Section 5.1, there are two types of events: internal and external ones. We note that the internal events are not relevant to the rest of the paper. However, we define them for the completeness of the formal specification. The set of finite internal events $E_{internal}$ is the following:
 - $PrM(rp_i, t_i, m, cp)$ denotes that at t_i , an $rp_i \in P_{rp}$ processes a message $m \in M$ sent by the cp .
 - $PrP(rp_i, t_i, pkt, rp_j)$ denotes that at t_i , an $rp_i \in P_{rp}$ processes a data packet $pkt \in PKT$ forwarded by an $rp_j \in P_{rp}$ ($rp_i \neq rp_j$).

The external events considered are the send, the receive, and the delivery events. The set of external events is represented as a finite set $E_{external} = E_{send} \cup E_{receive} \cup E_{delivery}$. The set of send events E_{send} is the following:

- $SdM(m)$ denotes that the cp sends a message $m \in M$ to an $rp_j \in P_{rp}$.
- $FwdP(pkt)$: denotes that an $rp_i \in P_{rp}$ forwards a data packet $pkt \in PKT$ to an $rp_j \in P_{rp}$ ($rp_i \neq rp_j$).

$E_{receive}$ is composed of one event:

- $RecMP(mp)$: denotes that an $rp_j \in P_{rp}$ receives a message or a data packet $mp \in MP$. Such an event only notifies the reception of an mp by rp_j .

Furthermore, $E_{delivery}$ is composed of a unique event:

- $DlvMP(mp)$: denotes that an $rp_j \in P_{rp}$ delivered a message or a data packet $mp \in MP$. A delivery event identifies the execution performed or the consumption of an ingoing mp by rp_j .

The set of events associated with MP is the following:

$$E(MP) = \{SdM(m), FwdP(pkt)\} \cup \{RecMP(mp)\} \cup \{DlvMP(mp)\} \quad (1)$$

The whole set of events in the system is the finite set:

$$E = E_{internal} \cup E(MP) \quad (2)$$

The order of occurrence of events can be collected based on the causal dependencies between them. The representation $\hat{E}$ expresses the causality between events E in P , using the happened-before relation ($\rightarrow$) (see Definition 1) where:

$$\hat{E} = \{E, \rightarrow\}. \quad (3)$$

5.4. Causal Order Delivery

Causal order delivery is a fundamental property in the field of distributed systems, and it is required for various distributed applications (e.g., distributed simulation). In fact, it is useful for synchronizing distributed protocols by inducing the causal order of events. Indeed, this property means that if two send messages are causally related and are sent to the same process p_i , then their delivery should be held according to the causal order, that is by respecting the send order.

The author of [29] showed that in order to ensure a causal order delivery for a multicast communication, it suffices to ensure the causal delivery of immediately related send events. Therefore, to ensure message-data packet causal order delivery in SDNs, the set of events is determined to be $R = \{SdM(m), FwdP(pkt) | m, pkt \in MP\}$. For simplicity, we represent the two send events $SdM(m), FwdP(pkt)$ as $send(mp)$. Formally, the message causal delivery based on the IDR (see Definition 2) is defined for the SDN context as follows:

Definition 3. Causal order delivery in the SDN context:

$$\forall((send(mp), send(mp')) | mp, mp' \hat{=} match) \in R, send(mp) \downarrow send(mp') \text{ where } Dest(mp) = Dest(mp') \\ \Rightarrow delivery(mp) \rightarrow delivery(mp').$$

Therefore, causal order delivery establishes that if the diffusion of a message-data packet mp causally precedes the diffusion of message mp' , then the delivery of mp should precede the delivery of mp' at a common destination process.

6. Modeling the TFL Pattern from a Temporal Perspective

In this section, we study the use of physical time as a tool to ensure consistent network updates, and we analyze if physical time will be sufficient to deal with the TFL inconsistency problem.

We return to the flow swap update scenario of Figure 2. In this scenario, the topology is composed of an OpenFlow controller and two switches (S_1 and S_2). According to our network model (see Section 5), the set of processes for this scenario is $P = \{cp, rp_1, rp_2\}$, where cp is the OpenFlow controller and rp_1, rp_2 represent the switches S_1 and S_2 . During the update, the OpenFlow controller cp sends to rp_1 a FlowMod message $m_1 = (cp, 1, (match, delete), rp_1)$ where a $pflow \hat{=} match$, and then, it sends to rp_2 a FlowMod message $m_2 = (cp, 2, (match, add), rp_2)$ where also $pflow \hat{=} match$. The routing process rp_2 receives m_2 and installs the entry directing all $pkt_i \in pflow$ to rp_1 . Accordingly, rp_2 directs $pkt_1 \in pflow$ to rp_1 (see Figure 3). Subsequently, pkt_1 enters rp_1 before m_1 is delivered to rp_1 due to a delay of the reception. Finally, as pkt_1 matches the entry already installed in rp_1 , the former ends by redirecting pkt_1 to rp_2 , generating a TFL between rp_1 and rp_2 .

Figure 3. The communication diagram corresponding to Figure 2.

Upon analyzing the communication diagram corresponding to the execution diagram of Figure 3, we can observe that the TFL between rp_1 and rp_2 is created when the transmission time interval of m_1 is greater than the transmission time interval of m_2 plus the packet forwarding time of $pkt_1 \in pflow$ to rp_1 . We formally define the TFL pattern from a temporal perspective.

Definition 4. We have a $match_\lambda \in MATCH$ and two OpenFlow messages m and m' , where $m = (cp, t_{cp}, (match_\lambda, delete), rp_i)$ and $m' = (cp, t'_{cp}, (match_\lambda, add), rp_j)$, such that (i) $time(m) < time(m')$, (ii) $(rp_i \neq rp_j)$ and (iii) $\exists$ a forwarding path $fpath_\mu = \{rp_i, \dots, rp_j\}$ from rp_i to rp_j , such that intermediates rp_r may exist where $fpath_\mu \in FPATH \mid fpath_\mu \hat{=} match_\lambda$. A TFL pattern from rp_i to rp_j exists iff there is a data packet flow $pflow_\tau = pkt_1, pkt_2, \dots, pkt_n$ ($n \geq 1$), where $pflow_\tau \in PFLOW \mid pflow_\tau \hat{=} match_\lambda$, such that:

$$T(m) > T(m') + \sum_{k=1}^n T(pkt_k) \quad (4)$$

where $T(mp)$ is a function that returns the transmission time of an OpenFlow message or a data packet $mp \in MP$ from a p_i to a p_j with $p_i, p_j \in P$, and $time(m)$ gives the local physical time at the moment a message $m \in M$ is sent.

Based on Definition 4, a possible solution to avoid TFLs in the data plane is to establish temporal references and perform timed execution of updating operations. However, and as was discussed in Section 4, such a solution is not effective since it is quite difficult to perfectly synchronize clocks across network entities. Indeed, any clock synchronization mechanism (e.g., Network Time Protocol (NTP) [30]) presents a clock synchronization accuracy. In this case, we will not be able to find out or to force the execution order of any pair of events, which may harm the update consistency.

7. The TFL-Free Property

As studied in Section 6, a TFL occurs during an update due to the no ordered execution of events. This is because networks are not able to reason about the global real-time ordering of events across entities. In this section, we explore the use of the causality (more detail in Section 5) as a theoretical construction, firstly, to define the TFL pattern (see Section 7.1), and secondly, to perform coordinated network updates, ensuring the TFL-free property (see Section 7.2).

7.1. Modeling the TFL Pattern from a Causal Perspective

In this subsection, we analyze the TFL phenomenon during OpenFlow-based SDN updates by using the scheme of the happened-before relation (See Definition 1).

Figure 4 illustrates the generic scenario in which an out-of-order execution of messages/packets leads to a TFL during updates. In this scenario, the topology is composed of an OpenFlow controller and n OpenFlow switches. According to our network model (see Section 5), the set of processes for this scenario is $P = \{cp, rp_{src}, rp_1, rp_2, \dots, rp_n, rp_{dst}\}$ where cp is the OpenFlow controller, rp_{src} and rp_{dst} represent, respectively, the *Source* and the *Destinations* switches, and $rp_1, rp_2, \dots, rp_n$ represent the *intermediateswitches* $S_1, S_2, \dots, S_n$. Initially, each intermediate routing process rp_i , except rp_n , contains an entry r directing a $pflow \in PFLOW$ to its rp_{i+1} (see the solid lines in Figure 4a). Due to a network policy change, $pflow$ should be forwarded from rp_n to rp_1 (see the dashed lines in Figure 4a). To this end, the cp starts by instructing each rp_i , except rp_n , to delete the entry directing $pflow$ to its rp_{i+1} and then instructs again each rp_i , except rp_1 , to install an entry directing $pflow_\tau$ to its rp_{n-1} . In an OpenFlow-based SDN, the cp has to send $2 \times (n - 1)$ controller-to-switch OpenFlow messages of type *FlowMod*. For brevity, and as it is sufficient to express the generality of the phenomenon, Figure 4b depicts only one message (Message (1) depicted in the dashed line) for deleting the rule in rp_1 and all the other messages (from (2) to (n)) for adding the new forwarding path from rp_n to rp_1 .

Figure 4. The generic Transient Forwarding Loop (TFL) scenario: (a) initial and final forwarding paths; (b) a sequence of messages to redirect a $pflow$ from S_n to S_1 ends with a TFL.

The communication diagram of Figure 5, which corresponds to the generic scenario of Figure 4b, is used to characterize the phenomenon. As shown, the cp starts by sending to rp_1 a *FlowMod* message $m_1 = (cp, 1, (match, delete), rp_1)$ of command type *delete* and *match* as a match, where $pflow \hat{=} match$, and then, it sends to rp_2 a *FlowMod* message $m_2 = (cp, 2, (match, add), rp_2)$ of command type *add* and also to $m_1, match$ as a match, as it is about forwarding the same packet flow $pflow$. The rest of the OpenFlow messages from m_i to m_j (represented in Figure 4b by Messages (3), ..., (n)) are sent to their corresponding rp to add the entries directing $pflow$ from rp_n to rp_1 . Upon the reception of the messages, and due to the asynchronous communication between the controller and all the underlying switches, $rp_n, rp_{n-1}, \dots$ and rp_2 receive their messages and install the entries while packets $pkt_i \in pflow$ hit rp_n , directing them to rp_1 (see Figure 5). Hence, pkt_n enters rp_1 before m_1 is delivered to rp_1 (see Figure 5). Consequently, pkt_n matches the entry r already installed in rp_1 directing all $pkt_i \in pflow$ to rp_2 . Finally, rp_1 ends by redirecting pkt_n to rp_2 (see the solid line between S_1 and S_2 in Figure 4b), which generates a TFL. Indeed, the fact that pkt_n is delivered to rp_1 and went back through rp_1 means that pkt_n has already entered into a TFL.

We define below an abstraction of the TFL pattern in SDN, as a specification of Lamport's happened-before relation to express the phenomenon from a causal perspective.

Figure 5. The communication diagram corresponding to Figure 4.

Definition 5. We have a $match_\lambda \in MATCH$ and two FlowMod messages $m, m' \in M$ where $m = (cp, t_{cp}, (match_\lambda, delete), rp_i)$ and $m' = (cp, t'_{cp}, (match_\lambda, add), rp_j)$, such that (i) $m \rightarrow m'$, (ii) $(rp_i \neq rp_j)$, and (iii) $\exists$ a forwarding path $fpath_\mu = \{rp_i, \dots, rp_j\}$ from rp_i to rp_j , such that intermediates rp_r may exist where $fpath_\mu \in FPATH \mid fpath_\mu \hat{=} match_\lambda$. A TFL from rp_i to rp_j exists iff there is a data packet flow $pflow_\tau = pkt_1, pkt_2, \dots, pkt_n$ ($n \geq 1$), where $pflow_\tau \in PFLOW \mid pflow_\tau \hat{=} match_\lambda$, such that:

1. pkt_1 is sent by rp_j after the delivery of m' ,
2. if pkt_k ($1 \leq k \leq n$) is delivered by rp_r ($rp_r \neq rp_i$), then pkt_{k+1} is the next data packet sent by rp_r , and
3. pkt_n is delivered by rp_i before the delivery of m .

It can be interpreted that a TFL occurs due to the violation of the causal order delivery of the OpenFlow update message(s) of type delete and matched packets/packet flows (see the three conditions in Definition 5). Based on the abstraction defined in Definition 5, we present how to capture and avoid the occurrence of the defined pattern during updates, ensuring the TFL-free property.

7.2. Algorithm for TFL-Free SDN Updating

An update algorithm based on the algorithm of [6] is presented in this subsection. Based on the study provided in Section 7.1, we show how the presented algorithm allows capturing asynchronously and avoiding preventively the occurrence of TFLs during updates, referred to as TFL-free updates.

7.2.1. Algorithm Overview

Throughout an update, an OpenFlow-based SDN expresses message/packet-passing between processes P , specifically, between the controller process P_{cp} and the set of routing processes P_{rp} , or between P_{rp} themselves. Therefore, we distributed the algorithm over the P_{cp} and the set of P_{rp} . It can be summarized as follows:

- Input: The algorithm takes the list of all OpenFlow update messages and ingoing data packets as input. As described in Section 5, updates are *match*-based performed, i.e., update messages and in-fly data packets are grouped and processed by match. This allows updates of different matches to be executed concurrently without the risk of harming the TFL-free property.
- Condition: Let $match_\lambda \in MATCH$ be a match. All matched OpenFlow messages of type delete should be disseminated from the P_{cp} to the P_{rp} before all OpenFlow messages of type add.
- Execution model: All OpenFlow update messages are asynchronously disseminated from P_{cp} to P_{rp} . No upper bound on messages/data packets transmission delay is required. P_{cp} never waits for an acknowledgment message from P_{rp} once a message is delivered, and clocks of all entities are not in synchronization with each other, that is the execution model is fully asynchronous.

- **Data structures:** At a process-level, each process $p_i \in P$ maintains a vector of control information CI_i to store direct dependency information between the set of messages/packets MP with respect to the corresponding *match*. Furthermore, each $rp_i \in P_{rp}$ maintains a matrix $Delivery_i$ to track dependency information (see the data structures' subsection below for more details) when an rp receives an $mp \in MP$. We will also make reference to the state of the delivery matrix structure when a process p_i sends an mp , denoted as *ForDelivery_i*.
- **Functionally:** The algorithm is implemented at the time of sending an OpenFlow message $m \in M$ from the P_{cp} , and/or at the time of forwarding a data packet $pkt \in PKT$ from an $rp \in P_{rp}$, and/or at the time of receiving an $mp \in MP$ by a routing process $rp \in P_{rp}$. At the sending of an OpenFlow message from the P_{cp} , and besides the message m , the algorithm encapsulates into the message a vector CI_{cp} containing control information on the send message events that directly depend on it (see Algorithm 1). Similarly, an outgoing packet pkt in the data plane piggybacks a vector CI_{rp} that carries control information of the OpenFlow message/packet send events that directly depend on it (see Algorithm 2). At the reception of an mp , and based on the control information encapsulated into CI_{mp} , the receiver rp_j decides if it can deliver mp or if it should wait for the reception of another/other mp' (s) to then be able to deliver mp (see Algorithm 3).
- **Ensured properties:** Based on the algorithm functionality, neither an OpenFlow message nor a packet, which share the same match, will be delivered out of causal order, ensuring the per-match causal order delivery property (see Definition 3). In Section 7.3, we prove that the presented algorithm ensures the TFL-free property.

Algorithm 1 : Controller-to-switch message sending.

Input: The set of OpenFlow update messages

Condition: $\forall m \in M : (m = (cp, t_{cp}, (match_{\lambda}, delete), rp_d^1) \rightarrow m' = (cp, t'_{cp}, (match_{\lambda}, add), rp_d))$

1: $t_{cp} := 0, CI_{cp}[j] = \{\} \ \forall j : 1 \dots N$

2: **for all** OpenFlow messages $\hat{=}$ $match_{\lambda}$ **do**

3: $t_{cp} := t_{cp} + 1$

4: $m = (cp, t_{cp}, OpenFlow_message, rp_i)$

5: $SdM(m, CI_{cp})$

6: $CI_{cp}[i] := (cp, t_{cp})$

Algorithm 2 : Switch-to-switch packet forwarding.

Input: Ingoing data packets

1: $t_{rp_i} := 0, CI_{rp_i}[j] = \{\} \ \forall j : 1 \dots N$

2: $t_{rp_i} := t_{rp_i} + 1$

3: $pkt = (rp_i, t_{rp_i}, header, data, rp_j)$

4: $FwdP(pkt, CI_{rp_i})$

5: $CI_{rp_i}[j] := (rp_i, t_{rp_i})$

Algorithm 3 : Switch message/packet reception.**Input:** OpenFlow messages sent from the controller and in-fly data packets

-
- 1: $Delivery_{rp_i}[i, k] = 0 \forall i, k : 1 \dots N, CI_{rp_i}[i] = \{\} \forall i : 1 \dots N$
 - 2: $RecMP(mp = (p_i, t_i, content, p_j, CI_{mp}))$ (the content may be an OpenFlow message or a packet)
 - 3: $wait (\forall k (k, x) \in CI_{mp}[j] \mid Delivery_j[k, j] \geq x)$
 - 4: $DlvMP(mp)$
 - 5: $Delivery_j[i, j] := t_{mp}$
 - 6: $\forall k \mid (k, y) \in CI_{mp}[i]: Delivery_j[k, i] := \max(Delivery_j[k, i], y)$
 - 7: $CI_j[j] := (CI_j[j] \cup_{\max} \{(i, t_{mp})\}) -_{\max} CI_{mp}[j]$
 - 8: $\forall p_k \in P \mid k \neq i, j: CI_j[k] := CI_j[k] \cup_{\max} CI_{mp}[k]$
 - 9: $\forall p_k \in P \mid k \neq j:$
 - 10: $\forall (l, x) \in CI_j[k]$
 - 11: if $Delivery_j[l, k] \geq x$ then
 - 12: delete (l, x) from $CI_j[k]$
-

7.2.2. Data Structures

In the algorithm, each process $p_i \in P$ maintains a vector of control information CI_i of length N to store direct dependency information (N is the number of processes). Each element of CI_i is a set of tuples of the form $(process_{id}, logical_clock)$. For instance, let CI_i be the vector of a process p_i such that $(k, t) \in CI_i[j]$ ($i \neq j$). This implies that any message sent by a process p_i should be delivered to p_j after the message mp of sequence number t sent by p_k has been delivered to p_j . Furthermore, each process rp_i maintains an $N \times N$ integer matrix $Delivery_i$ to track dependency information. Each matrix $Delivery_i$ records the messages of the last sequence number delivered to other processes. For instance, $Delivery_i[j, k] = t$ denoted that p_i knows that messages sent by process p_j to p_k , whose sequence numbers are less than or equal to t , have been delivered to p_k .

7.2.3. Algorithm Details

Controller-to-switch message sending: This algorithm (see Algorithm 1) takes all update messages calculated by the controller that should be communicated to the routing processes as input. As a condition, all messages of type delete should be disseminated by the controller before all messages of type add. Before sending the messages, they should be grouped by *match* (see Line 2). In each message send event, the logical clock of cp (denoted by t_{cp}) is incremented (see Line 3) to associate a timestamp with m (see Line 4). Upon sending m , CI_{cp} is encapsulated into the send event (see Line 5). $CI_{cp}[k]$ contains information about the direct predecessors of m with respect to messages sent to p_k . After sending m , $CI_{cp}[i]$ is updated by adding (cp, t_{cp}) as a potential direct predecessor of future messages sent to rp_i after m (see Line 6). In general, $CI_{cp}[i]$ contains delivery constraints of messages sent to rp_i by cp .

Switch-to-switch packet forwarding: This algorithm (see Algorithm 2) takes ingoing data packets as input. Before forwarding a packet pkt to an rp_j , the logical clock of rp_i (denoted by t_{rp_i}) is incremented (see Line 2) to associate a timestamp with pkt (see Line 3). Upon forwarding a pkt to its next hop, CI_{rp_i} is encapsulated into the forwarding event (see Line 4). $CI_{rp_i}[k]$ contains information about the direct predecessors of pkt with respect to OpenFlow messages/packets sent to p_k . After forwarding pkt , $CI_{rp_i}[j]$ is updated by adding (rp_i, t_{rp_i}) as a potential direct predecessor of future packets sent to rp_j after pkt (see Line 5). The instructions of this algorithm are similar to the previous one. The difference is at the level of the communicating processes and of the type of data for sending.

Switch message/packet reception: From the point of view of the receiver rp_j (see Algorithm 3), an mp piggybacks delivery constraints encapsulated into CI_{mp} , that is messages that must be delivered to p_j before mp . Note that there is a distinction between the reception of a message mp (see Line 2) and its delivery (see Line 4). The delivery of mp to a process p_j implies that the message in question was received and all previous delivery constraints on p_j were satisfied (see Line 3). Thus, once the delivery constraints are satisfied, mp is delivered to p_j (see Line 4), and what follows is updating the control information of the receiver process p_j . Then, p_j updates its $Delivery_j$ matrix, indicating that the message sent from p_i , whose sequence number is equal to t_{mp} , was already delivered to p_j (see Line 5). The $Delivery_j$ matrix is also updated with respect to the messages mps delivered to process p_i (see Line 6). After the delivery of mp , p_j updates a new delivery constraint for future messages sent from p_j . Therefore, $CI_j[j]$ is updated by adding (i, t_{mp}) , using the $\cup_{max}$ operator as a new potential direct dependency on subsequent messages sent from p_j and by deleting older direct dependencies (transitive dependencies), using the $-_{max}$ operator (see Line 7), which were already satisfied. The operator $\cup_{max}$ (see Algorithm 4) ensures that if there are multiple constraints corresponding to a sender process, the most recent constraint is selected [6]. The $-_{max}$ operator (see Algorithm 5) deletes the delivery constraints already known to be satisfied (T2) from the current set of message delivery constraints (T1) [6]. Furthermore, and to maintain the causality property, a message sent by a process p_k to p_j , whose send event is causally dependent on messages sent by p_i to p_j , should be delivered to p_j only after the n^{th} message sent by p_i to p_j . Thus, for all processes p_k , except the ones for the sender and receiver, $CI_i[k]$ is updated by adding the delivery constraints piggybacked by CI_{mp} (see Line 8). As we have mentioned, the $Delivery$ matrix is used for garbage collection to reduce the communication overhead when ensuring the causal ordering of messages. Therefore, based on the $Delivery_j$ matrix of process p_j , CI_j is updated in such a way that it contains only the recent delivery constraint needed for the delivery of future messages (see Lines 9 to 12).

Algorithm 4 : The operator $\cup_{max}$: $(T1 \cup_{max} T2)$.

```

1: Boolean change, set of tuples T1, set of tuples T2, set of tuples T
2: change := true
3: T := T1  $\cup$  T2    (T1 and T2 contain the delivery constraints)
4: while (change) {
5:   change := false
6:   if  $(i, x) \in T$  and  $(i, y) \in T$  and  $(x < y)$ 
7:     { T := T −  $\{(i, x)\}$ 
8:     change := true } }
9: return(T)

```

Algorithm 5 : The operator $-_{max}$: $(T1 -_{max} T2)$.

```

1: Boolean change, set of tuples T1, set of tuples T2, set of tuples T
2: change := true
3: T := T1
4: while (change) {
5:   change := false
6:   if  $(i, x) \in T$  and  $(i, y) \in T2$  and  $(x \leq y)$ 
7:     { T := T −  $\{(i, x)\}$ 
8:     change := true } }
9: return(T)

```

7.3. Proof of Correctness

In the previous subsection, we presented an adaptation of a causal ordering algorithm as a solution for the TFL problem. We now demonstrate that the proposed algorithm is TFL-free.

When analyzing Definition 5, we can interpret that a TFL occurs in the data plane due to the violation of the causal order delivery of the relevant OpenFlow message *FlowMod*, and consequently, a packet or a flow of packets starts to loop between the underlying switches. Thus, and in order to prove that the algorithm is TFL-free, it is sufficient to demonstrate that there is no data packet flow that holds with the conditions specified in the TFL pattern of Definition 5. To accomplish the proof, we focus on the packet *pkt* that triggers the TFL pattern. The following theorem proves this observation.

Theorem 1. *The algorithm guarantees that $\nexists$ a data packet $pkt_k \in pflow_\tau \mid pflow_\tau \triangleq match_\lambda$, such that (i) $SdM(m') \rightarrow FwdP(pkt_k)$ and (ii) $DlvMP(pkt_k, t_i) \rightarrow DlvMP(m, t'_i)$ where $m, m' \triangleq match_\lambda$ and $SdM(m) \rightarrow SdM(m')$.*

Let us assume that the Algorithm 1 and the Algorithm 2 store knowledge of the latest message/packet $mp \in MP$ sent from the same process $p_i \in P$ to another process, through a local matrix named *ForDelivered_i* that has the same structure as the delivery matrix (for more details, see the structure description subsection). Therefore, for this proof, we consider this additional instruction in both mentioned algorithms: (6)*ForDelivered_i*[*i*, *j*] = *t*. This hypothetical instruction allows building a matrix that considers the message(s)/packet(s) that belong(s) to the causal history of a message/packet *mp* and that has (have) to be delivered before *mp* to a process p_j . Thus, if *ForDelivery_i*[*i*, *j*] is less than or equal to *Delivery_j*[*i*, *j*], this means that all message(s)/packet(s) that causally depend(s) on *mp* has (have) already been delivered to p_j . On the other hand, if *ForDelivery_i*[*i*, *j*] is greater than *Delivery_j*[*i*, *j*], this means that some of the message(s)/packet(s) that causally depend(s) on *mp* has (have) not been delivered to p_j .

Proof. This is proven by contradiction. Let us assume that there is a packet $pkt_k \in pflow_\tau$, such that:

1. $SdM(m) \rightarrow FwdP(pkt_k)$
According to Definition 5, this is by transitivity since $SdM(m) \rightarrow SdM(m')$ and $SdM(m') \rightarrow FwdP(pkt_k)$, such that *m* and pkt_k have rp_i as a common destination.
2. $DlvMP(pkt_k, t_i) \rightarrow DlvMP(m, t'_i)$

The existence of a pkt_k under the mentioned Conditions 1 and 2 implies that *ForDelivery_{cp}*[*cp*, rp_i] > *Delivery_{rp_i}*[*cp*, rp_i] when rp_i receives pkt_k . However, and based on the proof of [6], this cannot occur as the algorithm allows the delivery of pkt_k to rp_i only when *ForDelivery_{cp}*[*cp*, rp_i] ≤ *Delivery_{rp_i}*[*cp*, rp_i], which contradicts the initial assumption. □

8. Scenario Description

We present the example of Figure 6 to illustrate how our algorithm works. This example shows two independent updating processes within the same topology. Note that updating processes in OpenFlow-based SDNs are independent as forwarding decisions are flow-based: each packet flow is determined by match fields, as well as by update messages disseminated by the controller to update the forwarding tables of switches. The two updating processes shown in Figure 6 are distinguished by means of two different colors, corresponding to a *match₁* in purple and a *match₂* in blue. In both examples, data packets that interleave with the update messages sent from the controller (*cp*) (the *cp* is not depicted in Figure 6) and installed into the switches *source*, $S_1, S_2, \dots, S_n$ ($rp_{src}, rp_1, rp_2, \dots, rp_n$) may enter into TFLs between all intermediate switches $rp_1, rp_2, \dots, rp_n$. For brevity, only the update forwarding process corresponding to *match₁* is explored. However, and for completeness, Table 1 and Tables A1 and A2 in Appendix A include the information related to both matches. Update

messages are illustrated in Table 1. Table A1 shows piggybacked control information of update messages, and Table A2 depicts control information for future update messages.

TFLs that occur when updating forwarding path related to $match_1$: We will follow a packet pkt_k from the origin to its destination taking into consideration the routing tables of switches corresponding to the initial policy of $match_1$ (colored with purple). Assuming that rp_{src} and rp_n update their flow tables, then rp_{src} forwards an ingoing matched data packet pkt_k to rp_n (see the dashed purple arrow between rp_{src} and rp_n). After delivering pkt_k , rp_n forwards it to rp_{n-1} , and the latter delivers pkt_k before updating its flow table. As a result, pkt_k ends in a TFL between rp_{n-1} and rp_n (see the dashed purple arrow between rp_n and rp_{n-1} and the solid purple arrow between rp_{n-1} and rp_n). If the last assumption is not the case, i.e., rp_{n-1} delivers pkt_k after having updated its flow table, then rp_{n-1} should forward pkt_k to rp_{n-2} . The same risk that pkt_k enters into a TFL between rp_{n-2} and rp_{n-1} exists if rp_{n-2} delivers pkt_k before updating its flow tables. pkt_k may indeed enter into TFLs between all the subsequent rp pairs, i.e., until pkt_k reaches rp_2 , and then, the risk may arise between rp_1 and rp_2 .

Figure 6. Two forwarding path update scenarios from the same source to the same destination.

How the update algorithm proceeds to capture and avoid TFLs: As already mentioned, in each hop, there is a risk of falling into a TFL during the update of the forwarding path. We will review how the initial condition and the algorithm prevent this from happening. Table 2 illustrates the behavior of the data packet pkt_k , which interleaves with the disseminated update messages corresponding to $match_1$. Note that in each hop, pkt_k gets a different identifier. In the first row, for example, pkt_k must travel from the Source switch to the switch S_n . Later, in the following hop, pkt_k is forwarded from S_n to S_{n-1} (second row). This continues until pkt_k reaches the Destination switch.

As mentioned above, before performing the update, the algorithm (see Section 7.2) begins with the initial condition by sorting update messages by their match fields and arranging the dissemination of messages as follows: all delete messages are disseminated before all add messages. We describe now the algorithmic part. Each disseminated message (see Table 1) piggybacks direct dependency information related to message(s), which directly depend(s) on it and correspond(s) only to their match fields. Table A1 shows the control information (CI_m) encapsulated into each message when sending it from the controller, and Table A2 illustrates the control information needed to be piggybacked for future messages. In Table A1, the vectors CI_m of tuples encapsulated into update messages are presented and referred to the messages that causally depend on the message, i.e., messages that should be delivered to a switch before the message in question, while Table A2 presents the vector of the controller CI_{cp} and the tuples stored just after sending each message and that should be encapsulated with the future messages. On the other hand, the control information piggybacked by pkt_k is built after delivering it in each hop.

Upon receiving messages/packets (see Table 3), switches also receive the control information CI_{mp} related to messages and packets. After each delivery, new potential direct dependencies on subsequent messages/packets are added to a receipt switch CI_{rp} . In this scenario, all add update messages sent to the intermediate routing processes $rp_1, rp_2, \dots, rp_n$ are delivered without any restriction as no message

or packet sent before them with a common destination exists. However, for example, the delivery of pkt_k with identifier pkt_2 is left on standby (wait) by rp_{n-1} as pkt_2 piggybacked the tuple $(cp, n-1)$, which means that the message disseminated from the controller cp with sequence number $n-1$ should be delivered before pkt_2 to rp_{n-1} (see Table 3). Once the message sent by cp with sequence number $n-1$ is delivered, pkt_2 can be delivered to rp_{n-1} . The algorithm treats all pkt_k in the same way in the subsequent hops, i.e., $rp_{n-2}, \dots, rp_1$, if they reach them before the delivery of the delete messages. Therefore, any forwarding of pkt_k generated by an installation of an add update message will not be delivered by the next hop switch before the delivery of the corresponding delete message (if it exists), preventing any pkt_k from going back to a switch from which it was forwarded, avoiding having them enter TFLs during the update.

Table 1. Update messages calculated for updating P_{rp} that correspond to $match_1$ and $match_2$.

Message	Message Content
Delete update messages that correspond to $match_1$	
m_1	$(cp, t_{cp} = 1, (match_1, delete), rp_{src})$
m_2	$(cp, t_{cp} = 2, (match_1, delete), rp_1)$
m_3	$(cp, t_{cp} = 3, (match_1, delete), rp_2)$
...	...
m_{n-2}	$(cp, t_{cp} = n-2, (match_1, delete), rp_{n-2})$
m_{n-1}	$(cp, t_{cp} = n-1, (match_1, delete), rp_{n-1})$
m_n	$(cp, t_{cp} = n, (match_1, delete), rp_n)$
Delete update messages that correspond to $match_2$	
m_{n+1}	$(cp, t_{cp} = n+1, (match_2, delete), rp_{src})$
m_{n+2}	$(cp, t_{cp} = n+2, (match_2, delete), rp_n)$
m_{n+3}	$(cp, t_{cp} = n+3, (match_2, delete), rp_{n-1})$
...	...
m_{n+m}	$(cp, t_{cp} = n+m, (match_2, delete), rp_2)$
m_{n+m+1}	$(cp, t_{cp} = n+m+1, (match_2, delete), rp_1)$
Add update messages that correspond to $match_1$	
m_{n+m+2}	$(cp, t_{cp} = n+m+2, (match_1, add), rp_{src})$
m_{n+m+3}	$(cp, t_{cp} = n+m+3, (match_1, add), rp_n)$
m_{n+m+4}	$(cp, t_{cp} = n+m+4, (match_1, add), rp_{n-1})$
...	...
m_{n+m+k}	$(cp, t_{cp} = n+m+k, (match_1, add), rp_2)$
$m_{n+m+k+1}$	$(cp, t_{cp} = n+m+k+1, (match_1, add), rp_1)$
Add update messages that correspond to $match_2$	
$m_{n+m+k+2}$	$(cp, t_{cp} = n+m+k+2, (match_2, add), rp_{src})$
$m_{n+m+k+3}$	$(cp, t_{cp} = n+m+k+3, (match_2, add), rp_1)$
$m_{n+m+k+4}$	$(cp, t_{cp} = n+m+k+4, (match_2, add), rp_2)$
...	...
$m_{n+m+k+j}$	$(cp, t_{cp} = n+m+k+j, (match_2, add), rp_{n-2})$
$m_{n+m+k+j+1}$	$(cp, t_{cp} = n+m+k+j+1, (match_2, add), rp_{n-1})$
$m_{n+m+k+j+2}$	$(cp, t_{cp} = n+m+k+j+2, (match_2, add), rp_n)$

Table 2. Packets interleaving with update messages that correspond to $match_1$.

Packet pkt_k	Packet Content
pkt_1	$(rp_{src}, trp_{src} = 1, (match_1, data), rp_n)$
pkt_2	$(rp_n, trp_n = 1, (match_1, data), rp_{n-1})$
pkt_3	$(rp_{n-1}, trp_{n-1} = 1, (match_1, data), rp_{n-2})$
...	...
pkt_n	$(rp_2, trp_2 = 1, (match_1, data), rp_1)$
pkt_{n+1}	$(rp_1, trp_1 = 1, (match_1, data), rp_{dst})$

Table 3. Delivery of update messages and data packets $mp \in MP$ corresponding to $match_1$.

Message/Package	CI_{mp}	Delivery Condition	CI_{rp}
Reception of the delete update message by rp_{src} m_1	CI_{m_1}	Delivered (✓)	$CI_{rp_{src}} = \cup_{max} CI_{m_1} + (cp, 1)$
Reception of the add update message by rp_{src} m_{m+n+2}	$CI_{m_{n+m+2}}$	✓	$CI_{rp_{src}} = \cup_{max} CI_{m_{n+m+2}} + (cp, n + m + 2)$
Reception of the add update message by rp_n m_{m+n+3}	$CI_{m_{n+m+3}}$	✓	$CI_{rp_n} = \cup_{max} CI_{m_{n+m+3}} + (cp, n + m + 3)$
Reception of the add update message by rp_{n-1} m_{m+n+4}	$CI_{m_{n+m+4}}$	✓	$CI_{rp_{n-1}} = \cup_{max} CI_{m_{n+m+4}} + (cp, n + m + 4)$
Reception of the add update message by rp_2 m_{m+n+k}	$CI_{m_{n+m+k}}$	✓	$CI_{rp_2} = \cup_{max} CI_{m_{n+m+k}} + (cp, n + m + k)$
Reception of the add update message by rp_1 $m_{m+n+k+1}$	$CI_{m_{n+m+k+1}}$	✓	$CI_{rp_1} = \cup_{max} CI_{m_{n+m+k+1}} + (cp, n + m + k + 1)$
Reception of pkt_1 by rp_n pkt_1	$CI_{rp_{src}}$	✓	$CI_{rp_n} = \cup_{max} CI_{rp_{src}} + (rp_{src}, 1)$
Reception of pkt_2 by rp_{n-1} pkt_2	$CI_{rp_n} = [\dots, (cp, n - 1), \dots]$	Wait (⌚)	—
Reception of the delete update message by rp_{n-1} m_{n-1}	$CI_{m_{n-1}}$	✓	$CI_{rp_{n-1}} = \cup_{max} CI_{m_{n-1}} + (cp, n - 1)$
Reception of the pkt_2 by rp_{n-1} after delivering m_{n-1} pkt_2	$CI_{rp_n} = [\dots, (cp, n - 1), \dots]$	✓	$CI_{rp_{n-1}} = \cup_{max} CI_{rp_n} + (rp_n, 1)$
Reception of pkt_3 by rp_{n-2} pkt_3	$CI_{rp_{n-1}} = [\dots, (cp, n - 2), \dots]$	⌚	—
Reception of the delete update message by rp_{n-2} m_{n-2}	$CI_{m_{n-2}}$	✓	$CI_{rp_{n-2}} = \cup_{max} CI_{m_{n-2}} + (cp, n - 2)$
Reception of the pkt_3 by rp_{n-2} after delivering m_{n-2} pkt_3	$CI_{rp_{n-1}} = [\dots, (cp, n - 2), \dots]$	✓	$CI_{rp_{n-2}} = \cup_{max} CI_{rp_{n-1}} + (rp_{n-1}, 1)$
The same for the other intermediate routing processes ...	...	...	...
Reception of pkt_n by rp_1 pkt_n	$CI_{rp_2} = [\dots, (cp, 2), \dots]$	⌚	—
Reception of the delete update message by rp_1 m_2	CI_{m_2}	✓	$CI_{rp_2} = \cup_{max} CI_{m_2} + (cp, 2)$
Reception of pkt_n by rp_1 after delivering m_2 pkt_n	$CI_{rp_2} = [\dots, (cp, 2), \dots]$	✓	$CI_{rp_1} = \cup_{max} CI_{rp_2} + (rp_2, 1)$
Reception of $pkt_n + 1$ by rp_{dst} $pkt_n + 1$	CI_{rp_1}	✓	$CI_{rp_{dst}} = \cup_{max} CI_{rp_1} + (rp_1, 1)$

9. Discussion

The problem of TFL during updates was intensively attacked by using different techniques. The proposed solutions can be classified into four approaches: ordered update, n-phase commit update, timed update, and the causal update (see Section 4 for more details). Although previous works have reached loop-free updates, they have not achieved effective and efficient solutions. We refer to the trade-off between ensuring TFL-free updates and performing updates with tolerable computational cost. To achieve a better trade-off, an important consideration is how to reason about the modeling of TFL-free updates. In this context, while all previous works set up synchronous execution models to perform updates, our solution was based on an asynchronous model. This allowed getting rid of the strong assumptions and certain constraints that come with a synchronous execution model. Indeed, solutions based on strong assumptions tend to be less applicable.

In regards to the ordered update approaches [12–14], such as the loop-free routing update algorithms proposed by [14], it has to be noticed that algorithms are centralized into the controller side to calculate updates. During each routing policy update, they perform updates in a free-loop manner by calculating the possible edges that can be updated without generating loops. To do so, the algorithms take forwarding graph properties (set of nodes, set of edges to be added, set of edges to be removed, etc.) as input. In the worst case, the algorithms require $|E|$ number of new edge to be added $|E|$ loop-free update steps to perform one routing policy update [14], which is very expensive in terms of update time and bandwidth. The fact that the controller should stop disseminating update messages at each k -step of an update until receiving acknowledgment messages from all involved switches means that the update requires extra time to be completed. Furthermore, acknowledgment messages result in bandwidth overhead. In contrast, our algorithm proposed to disseminate update messages in one-shot. As shown in Section 7.2, the only required treatment is to sort update messages by their match fields and arranging the dissemination of messages following a predefined order (see Algorithm 1). In addition, switches are not required to send acknowledgment messages to the controller as the proposed algorithm works with one-step update, and unlike [14], it does not need acknowledgment messages that confirm the reception of the disseminated messages in order to start another update step.

Regarding the solutions based on the two-phase commit update [4,16], they require that each switch maintains the entries of both policies (the initial and the final ones) during the transition synchronous phase. This may end by congesting the memories of switches. Even worse, this approach may not be feasible when the number of entries bypasses the memory limit size of the switches. Our proposed solution avoids such a problem since the update algorithm is based on the replacement of entries. Firstly, it consists of computing an order in which the switches should consume the delete entry update message prior to the add entry update message, instructed by the controller. Secondly, by ensuring the computed order of message execution through causal order delivery of messages/packets, no switch will have entries of both policies at the same time.

As far as the one-phase commit update approach ez-Segway [17] is concerned, updating tasks are delegated to the switches, qualifying it as a decentralized approach. However, consistent updates are ensured through centralized calculus performed by the controller. This is based on a forwarding graph that contains the global network state at each moment. Therefore, even when the updating process is decentralized among the switches, the process to make decisions is centralized in the controller side and carried out in a synchronous manner. Furthermore, no switch can participate in a new update until the previous updating process has finished. As already mentioned, our proposal avoids the use of a global graph since it is based on the causal order delivery of messages/packets.

The timed update approach [19,20] ensures consistent updates. However, the consistency during the intervals of time of updates is not guaranteed. This is due to the adaptation of a synchronous execution model to tackle an asynchronous problem, i.e., the inconsistent update problem. In fact, updates on involved switches are synchronized to be processed simultaneously, assuming an upper bound on message delivery. Actually, each switch receives messages/packet flows at an interval time

$[T - \varepsilon, T + \varepsilon]$ where ε represents the clock synchronization error. In this case, two periods of events may overlap where the execution order is not trivial in order to ensure consistency. Indeed, as analyzed in Section 6, the reception of in-fly packets that interleave with the reception of update messages can be a question of inconsistency during the interval time when the delivery of messages may take place, leading to TFLs. Hence, one cannot find out or force the execution order of events by using mechanisms for synchronizing physical time.

The consistency update approach based on Suffix Causal Consistency (SCC) [21] introduced an update algorithm based on the Lamport timestamp, tackling the forwarding loop problem. We should highlight that this approach and ours are quite different. In fact, SCC works based on timestamps tagging packets to reflect the rules that correspond to each switch. On the other hand, our work is based on establishing a causal order between update messages and in-fly packets. The idea behind SCC is to ensure that an in-fly packet is routed based on its recently installed forwarding path, ensuring bounded looping. As mentioned in Section 4, the fourth step of the proposed algorithm requires the controller to calculate and install the extra temporarily forwarding rule, redirecting an in-fly packet to the recent forwarding path to reach its destination. The question here is how many extra rules would be required per policy update? Indeed, this generates extra overhead related to controller and switch memory, as well bandwidth as this requires message exchange between the controller and switches. On the other hand, our work proposes an OpenFlow-based SDN model where the relevant update events and an abstraction of the TFL phenomenon are defined. Based on this model, we demonstrated that ensuring causal order delivery of the defined relevant update events, based on our SDN model, was sufficient to ensure the TFL-free property. Contrary to the SCC, ensuring causal order delivery does not need any extra rule installation. Furthermore, packets should not only traverse the most recent path to avoid the occurrence of such phenomenon. As shown in the scenario of Section 8, a packet may start flying from the old path and then be forwarded to the destination based on the new path without the need for an extra rule to direct it to the new path. Thus, only the original calculated rules are required to perform updates.

Instead, our proposal established a message/packet causal order delivery to detect and avoid any TFL (refer to the correctness proof in Section 7.3). Obviously, this resulted in memory and bandwidth overhead. However, the causal order message/packet algorithm was designed based on the IDR (see Definition 2). In fact, only direct dependency information between messages/packets with respect to the destination process(es) was needed to be piggybacked with them. In the worst case, each component of a $CI_{cp}[j]$ (vector of the control information of the controller process) could have at most one tuple for each process. This was because there was no concurrent message sent from the controller to a switch, i.e., each update message was sent to a specific switch and not to others. Furthermore, each message only piggybacked control information corresponding to their match fields. Let us go back to the scenario example of Section 8, in Table A1, of control information CI_{cp} presented in Appendix A. Messages of $match_2$ do not piggyback control information of messages of $match_1$. On the other hand, a component of a $CI_{rp_i}[j]$ (vector of the control information of a routing process) can have at most N tuples for each process, i.e., for each vector component. This occurs when a routing process should, for example, flood a data packet to all outgoing links. Therefore, $O(N^2)$ control information is required to be piggybacked with each message/packet. Concerning memory overhead, a routing process rp_i needs to store a vector $CI_{rp_i}[j]$ and a matrix $Delivery_i$ of $N \times N$, which then requires $O(N^2)$ integers to be stored.

In this work, we introduced a new approach to tackle the TFL SDN update problem based on distributed system principles. By means of this approach, we outperformed the-state-of-the-art by proposing a solution that was more suitable for the characteristics of the SDNs. Indeed, the solution was totally distributed, where the execution model was asynchronous, no global references were required, and it did not assume a message upper bound delivery. The aspects mentioned above favored the following points: (1) the system configuration with the proposed update mechanism aligned with the pure features of a distributed and asynchronous system; (2) the proposed update

mechanism minimized the interaction of the controller with the switches during an update (update messages were sent in one shot per updating flow); (3) the controller was not the only network entity that calculated the update operations as the switches contributed in the calculation (based on the exchanged control information and the messages/packets delivery constraints ensured by the update mechanism).

10. Conclusions

A model of the TFL pattern based on causal dependencies in SDNs was presented. We characterized the TFL phenomenon in an OpenFlow-based SDN, by identifying the relevant events and conditions under which it could occur during updates. In Definition 6, we specified that a TFL occurred due to a violation of the causal order delivery of a data packet during the process to update flow tables. This updating process was performed through *FlowMod* controller-to-switch messages of type *add* and *delete*. Based on this abstraction, we proposed a causal consistent update algorithm oriented to ensure the TFL-free property. To demonstrate this, a proof of correctness of the algorithm was provided. Based on these results, we propose as a future work to extend our solution to manage temporal constraints by including the principle of Δ -causal proposed by Pomares et al. in [31]. Furthermore, we would like to extend the study to other network invariant violation patterns. This study will allow us to answer whether ensuring causal dependencies will be sufficient to cover consistent network updates in SDNs.

Author Contributions: Formal analysis, A.G., S.E.P.H., L.M.X.R.H., H.H.K. and A.H.K.; Investigation, A.G., S.E.P.H., L.M.X.R.H., H.H.K. and A.H.K.; Methodology, A.G., S.E.P.H., L.M.X.R.H., H.H.K. and A.H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: This work is supported by the Mexican Agency for International Development Cooperation (AMEXCID).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Tables of Control Information Related to the Scenario Description of Section 8

Table A1. Piggybacked control information corresponding to update messages of $match_1$ and $match_2$.

Message	CI_m
Piggybacked control information of delete messages that correspond to $match_1$	
m_1	$[null]$
m_2	$[(cp, 1)]$
m_3	$[(cp, 1), (cp, 2)]$
...	...
m_{n-2}	$[(cp, 1), (cp, 2), (cp, 3), \dots]$
m_{n-1}	$[(cp, 1), (cp, 2), (cp, 3), \dots, (cp, n-2)]$
m_n	$[(cp, 1), (cp, 2), (cp, 3), \dots, (cp, n-2), (cp, n-1)]$
Piggybacked control information of delete messages that correspond to $match_2$	
m_{n+1}	$[null]$
m_{n+2}	$[(cp, n+1)]$
m_{n+3}	$[(cp, n+1), (cp, n+2)]$
m_{n+4}	$[(cp, n+1), (cp, n+2), (cp, n+3)]$
...	...
m_{n+m}	$[(cp, n+1), (cp, n+2), (cp, n+3), (cp, n+4), \dots]$
m_{n+m+1}	$[(cp, n+1), (cp, n+2), (cp, n+3), (cp, n+4), \dots, (cp, n+m)]$

Table A1. Cont.

Message	CI_m
Piggybacked control information of add messages that correspond to $match_1$	
m_{n+m+2}	$[CI_{cp} \text{ of } m_n + (cp, n)]$
m_{n+m+3}	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2)]$
m_{n+m+4}	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2), (cp, n + m + 3)]$
m_{n+m+5}	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2), (cp, n + m + 3)]$
...	...
m_{n+m+k}	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2), (cp, n + m + 3), (cp, n + m + 5), \dots]$
$m_{n+m+k+1}$	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2), (cp, n + m + 3), (cp, n + m + 5), \dots, (cp, n + m + k)]$
Piggybacked control information of add messages that correspond to $match_2$	
$m_{n+m+k+2}$	$[CI_{cp} \text{ of } m_{n+m+1} + (cp, n + m + 1)]$
$m_{n+m+k+3}$	$[CI_{cp} \text{ of } m_{n+m+1} + (cp, n + m + k + 2)]$
$m_{n+m+k+4}$	$[CI_{cp} \text{ of } m_{n+m+1} + (cp, n + m + k + 2), (cp, n + m + k + 3)]$
...	...
$m_{n+m+k+j}$	$[CI_{cp} \text{ of } m_{n+m+1} + (cp, n + m + k + 2), (cp, n + m + k + 3), (cp, n + m + k + 4), \dots]$
$m_{n+m+k+j+1}$	$[CI_{cp} \text{ of } m_{n+m+k+j}]$
$m_{n+m+k+j+2}$	$[CI_{cp} \text{ of } m_{n+m+k+j} + (cp, n + m + k + j + 1)]$

Table A2. Control information of update messages generated on the cp for future message corresponding to $match_1$ and $match_2$.

Message	CI_{cp}
Control information of delete messages that correspond to $match_1$	
m_1	$[(cp, 1)]$
m_2	$[(cp, 1), (cp, 2)]$
m_3	$[(cp, 1), (cp, 2), (cp, 3)]$
...	...
m_{n-2}	$[(cp, 1), (cp, 2), (cp, 3), \dots, (cp, n - 2)]$
m_{n-1}	$[(cp, 1), (cp, 2), (cp, 3), \dots, (cp, n - 2), (cp, n - 1)]$
m_n	$[(cp, 1), (cp, 2), (cp, 3), \dots, (cp, n - 2), (cp, n - 1), (cp, n)]$
Control information of delete messages that correspond to $match_2$	
m_{n+1}	$[(cp, n + 1)]$
m_{n+2}	$[(cp, n + 1), (cp, n + 2)]$
m_{n+3}	$[(cp, n + 1), (cp, n + 2), (cp, n + 3)]$
...	...
m_{n+m}	$[(cp, n + 1), (cp, n + 2), (cp, n + 3), (cp, n + 4), \dots, (cp, n + m)]$
m_{n+m+1}	$[(cp, n + 1), (cp, n + 2), (cp, n + 3), (cp, n + 4), \dots, (cp, n + m), (cp, n + m + 1)]$
Control information of add messages that correspond to $match_1$	
m_{n+m+2}	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2)]$
m_{n+m+3}	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2), (cp, n + m + 3)]$
m_{n+m+4}	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2), (cp, n + m + 3), (cp, n + m + 4)]$
...	...
m_{n+m+k}	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2), (cp, n + m + 3), (cp, n + m + 5), \dots, (cp, n + m + k)]$
$m_{n+m+k+1}$	$[CI_{cp} \text{ of } m_n + (cp, n + m + 2), (cp, n + m + 3), (cp, n + m + 5), \dots, (cp, n + m + k), (cp, n + m + k + 1)]$
Control information of add messages that correspond to $match_2$	
$m_{n+m+k+2}$	$[CI_{cp} \text{ of } m_{n+m+1} + (cp, n + m + k + 2)]$
$m_{n+m+k+3}$	$[CI_{cp} \text{ of } m_{n+m+1} + (cp, n + m + k + 2), (cp, n + m + k + 3)]$
$m_{n+m+k+4}$	$[CI_{cp} \text{ of } m_{n+m+1} + (cp, n + m + k + 2), (cp, n + m + k + 3), (cp, n + m + k + 4)]$
...	...
$m_{n+m+k+j}$	$[CI_{cp} \text{ of } m_{n+m+1} + (cp, n + m + k + 2), (cp, n + m + k + 3), (cp, n + m + k + 4), \dots, (cp, n + m + k + j)]$
$m_{n+m+k+j+1}$	$[CI_{cp} \text{ of } m_{n+m+k+j} + (cp, n + m + k + j + 1)]$
$m_{n+m+k+j+2}$	$[CI_{cp} \text{ of } m_{n+m+k+j} + (cp, n + m + k + j + 1), (cp, n + m + k + j + 2)]$

References

1. Kreutz, D.; Ramos, F.M.V.; Verissimo, P.; Rothenberg, C.E.; Azodolmolky, S.; Uhlig, S. Software-defined networking: A comprehensive survey. *Proc. IEEE* **2015**, *103*, 14–76. [\[CrossRef\]](#)
2. Openflow Switch Specification Version 1.5.1. Open Networking Foundation Tech. Rep. 2015. Available online: <https://www.opennetworking.org/sdn-resources/technical-library> (accessed on 20 February 2019).
3. Doria, A.; Salim, J.H.; Haas, R.; Khosravi, H.; Wang, W.; Dong, L.; Gopal, R.; Halpern, J. Forwarding and control element separation (forces) protocol specification. *Internet Eng. Task Force (IETF)* **2010**, *5810*, 1–124.
4. Reitblatt, M.; Foster, N.; Rexford, J.; Schlesinger, C.; Walker, D. Abstractions for Network Update. *ACM SIGCOMM Comput. Commun. Rev.* **2012**, *42*, 323–334. [\[CrossRef\]](#)
5. Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* **1978**, *21*, 558–565. [\[CrossRef\]](#)
6. Prakash, R.; Raynal, M.; Singhal, M. An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *J. Parallel Distrib. Comput.* **1997**, *41*, 190–204. [\[CrossRef\]](#)
7. Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. 2012. Available online: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wpsdn-newnorm.pdf> (accessed on 20 February 2019).
8. Jarraya, Y.; Madi, T.; Debbabi, M. A Survey and a Layered Taxonomy of Software-Defined Networking. *IEEE Commun. Surv. Tutor.* **2014**, *16*, 1955–1980. [\[CrossRef\]](#)
9. Zhou, W.; Li, L.; Luo, M.; Chou, W. REST API Design Patterns for SDN Northbound API. In Proceedings of the 2014 28th International Conference on Advanced Information Networking and Applications Workshops, Victoria, BC, Canada, 13–16 May 2014; pp. 358–365.
10. Canini, M.; Venzano, D.; Peresini, P.; Kostic, D.; Rexford, J. A NICE way to test OpenFlow applications. *NSDI* **2012**, *12*, 127–140.
11. Förster, K.-T.; Schmid, S.; Vissicchio, S. Survey of Consistent Software-Defined Network Updates. *IEEE Commun. Surv. Tutor.* **2018**, *21*, 1435–1461. [\[CrossRef\]](#)
12. Ludwig, A.; Rost, M.; Foucard, D.; Schmid, S. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies. In Proceedings of the 13th ACM Workshop on Hot Topics in Networks, Los Angeles, CA, USA, 27–28 October 2014; pp. 15:1–15:7.
13. Liu, H.H.; Wu, X.; Zhang, M.; Yuan, L.; Wattenhofer, R.; Maltz, D. zUpdate: Updating Data Center Networks with Zero Loss. *SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 411–422. [\[CrossRef\]](#)
14. Förster, K.-T.; Mahajan, R.; Wattenhofer, W. Consistent updates in software defined networks: On dependencies, loop freedom, and black holes. In Proceedings of the 2016 IFIP Networking Conference (IFIP Networking) and Workshops, Vienna, Austria, 17–19 May 2016.
15. Vissicchio, S.; Cittadini, L. FLIP the (Flow) Table: Fast Lightweight Policy-preserving SDN Updates. In Proceedings of the IEEE INFOCOM 2016—The 35th Annual IEEE International Conference on Computer Communications, San Francisco, CA, USA, 10–14 April 2016; pp. 1–9.
16. Katta, N.P.; Rexford, J.; Walker, D. Incremental Consistent Updates. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, China, 16 August 2013; pp. 49–54.
17. Nguyen, T.D.; Chiesa, M.; Canini, M. Decentralized Consistent Updates in SDN. In Proceedings of the Symposium on SDN Research, Santa Clara, CA, USA, 3–4 April 2017; pp. 21–33.
18. Fayazbakhsh, S.K.; Chiang, L.; Sekar, V.; Yu, M.; Mogul, J.C. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, Seattle, WA, USA, 2–4 April 2014.
19. Mizrahi, T.; Moses, Y. Software defined networks: It's about time. In Proceedings of the 35th Annual IEEE International Conference on Computer Communications, San Francisco, CA, USA, 10–14 April 2016; pp. 1–9.
20. Mizrahi, E.S.T.; Moses, Y. Timed consistent network updates in software-defined networks. *IEEE/ACM Trans. Netw.* **2016**, *24*, 3412–3425. [\[CrossRef\]](#)
21. Liu, S.; Benson, T.A.; Reiter, M.K. Efficient and Safe Network Updates with Suffix Causal Consistency. In Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, 25–28 March 2019; pp. 1–15.
22. Mahajan, R.; Wattenhofer, R. On Consistent Updates in Software Defined Networks. In Proceedings of the 12th ACM Workshop on Hot Topics in Networks, New York, NY, USA, 21–22 November 2013; pp. 1–7.

23. Amiri, S.A.; Ludwig, A.; Marcinkowski, J.; Schmid, S. Transiently Consistent SDN Updates: Being Greedy is Hard. In *International Colloquium on Structural Information and Communication Complexity*; Springer: Cham, Switzerland, 2016; pp. 391–406.
24. Förster, K.-T.; Wattenhofer, W. The power of two in consistent network updates: Hard loop freedom, easy flow migration. In *Proceedings of the 2016 25th International Conference on Computer Communication and Networks (ICCCN)*, Waikoloa, HI, USA, 1–4 August 2016.
25. Ludwig, A.; Marcinkowski, J.; Schmid, S. Scheduling Loop-free Network Updates: It's Good to Relax!. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 21–23 July 2015; pp. 13–22.
26. Förster, K.-T.; Ludwig, A.; Marcinkowski, J.; Schmid, S. Loop-Free Route Updates for Software-Defined Networks. *IEEE/ACM Trans. Netw.* **2018**, *26*, 328–341. [[CrossRef](#)]
27. Förster, K.-T.; Luedi, T.; Seidel, J.; Wattenhofer, R. Local checkability, no strings attached: (a)cyclicity, reachability, loop free updates in sdns. *Theor. Comput. Sci.* **2018**, *709*, 48–63. [[CrossRef](#)]
28. Naor, M.; Stockmeyer, L.J. What can be computed locally? In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, San Diego, CA, USA, 16–18 May 1993; pp. 184–193.
29. Pomares Hernández, S.E. The Minimal Dependency Relation for Causal Event Ordering in Distributed Computing. *Appl. Math. Inf. Sci.* **2015**, *9*, 57–61. [[CrossRef](#)]
30. Mills, D.L. Internet Time Synchronization: The Network Time Protocol. *IEEE/ACM Trans. Netw.* **1991**, *39*, 1482–1493. [[CrossRef](#)]
31. Pomares Hernandez, S.E.; Lopez Dominguez, E.; Rodriguez Gomez, G.; Fanchon, J. An Efficient Δ -Causal Algorithm for Real-Time Distributed Systems. *J. Appl. Sci.* **2009**, *9*, 1711–1718. [[CrossRef](#)]

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).