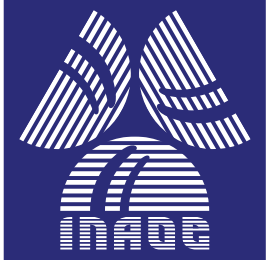

A Causal-Based Consistent Update Approach for Software-Defined Networks

By

Amine Guidara

Dissertation submitted in partial
fulfillment of the requirements for the degree of

PhD. in Computer Science

Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE)
Tonantzintla, Puebla, México
January, 2021

Advisors:

Dr. Saúl Eduardo Pomares Hernández
Coordination of Computer Science
INAOE, México

Dr. Lil María Xibai Rodríguez Henríquez
Coordination of Computer Science
CONACyT-INAOE, México

Dr. Ahmed Hadj Kacem
ReDCAD Laboratory
University of Sfax, Tunisia

Agradecimientos

A mis padres Mondher e Ibtissem quienes con su paciencia y esfuerzo me han permitido llegar a cumplir un sueño más. A mi hermana Yesmine, por su cariño y apoyo incondicional. A mi tía Salwa y mi tío Salim quienes me apoyaron mucho durante todo este proceso, por estar conmigo en todo momento. A mis abuelos, gracias por sus bendiciones y oraciones. A mi esposa Abir, por ser el apoyo incondicional, que con su amor y respaldo me ayuda alcanzar mis objetivos. A mi bebé Fedi, que con su nacimiento me dio los ánimos para seguir adelante. A mi hermano Josué, le agradezco por ser un alma bondadosa que ayuda sin esperar nada a cambio, por esas palabras de aliento, por ese tiempo tan preciado que me dedicaste para mejorar mi español. A mi hermano Sami, te agradezco por tu apoyo y tu amistad. A mis amigos, Alejandro, Laritza, Diana, Carlos, Hussein y Mariano con quienes he compartido los mejores momentos durante mi estancia. A mis paisanos en México Riadh y Mokhtar, gracias por estar conmigo en los momentos más defíciles.

A mi asesor el Dr. Saúl E. Pomares Hernández quien me ha transmitido sus conocimientos y me brindó la orientación para la realisazión de esta tesis. A mi co-asesora la Dra. Lil María X. Rodríguez Henríquez, por su dedicación que permitió el logro de este trabajo de investigación. A mi co-asesor el Dr. Ahmed Hadj Kacem, por su valiosa orientació. Al jurado de este trabajo, Dr. Felipe Orihuela-Espina, Dra. Hayde Peregrina-Barreto, Dr. René Armando Cumplido Parra y la Dra. Claudia Feregrino-

Uribe, por contribuir a que este proyecto de tesis pudiera llegar a un mejor término. También agradezco al Dr. Salvador Villarreal Reyes, por participar como miembro del jurado.

Al INAOE, por ser mi casa durante mas de tres años y permitirme obtener nuevos conocimientos. Al CEPE-Taxco, donde he aprendido el idioma español y he pasado los mejores momentos de mi vida. Al AMEXCID, por otorgarme los insumos económicos durante los tres primeros años del programa de doctorado.

A mi país Túnez, por enseñarme todo lo que realmente importa en esta vida. A México, por atendenrme y compartir conmigo su cultura.

Abstract

Software-Defined Network (SDN) is a network paradigm that has been recently introduced. Unlike traditional networks, e.g. IP networks, SDNs separate the network control logic from forwarding devices, and delegate network management tasks to a logically-centralized entity called the controller. However, SDN is still a distributed and asynchronous system. In fact, during forwarding policy updates, any network entity may trigger update events at any time, e.g. the sending of messages or data packets, while they are prone to arbitrary and unpredictable transmission delays. Moreover, the absence of an agreed and common temporal reference results in a broad combinatorial range space of event order. An out-of-order execution of events may lead to invariant violations, e.g. forwarding loops and forwarding blackholes, referred to as inconsistent updates. Some works tackle the issue of inconsistent updates by imposing global time references; however, they do not compromise consistency during updates as clocks of entities cannot be perfectly synchronized. Other solutions lie on performing updates on different rounds, i.e. steps, while each update step guarantees consistency. These solutions compromise consistency during updates; however, performing updates over different steps may congest the communication canals between the controller and the forwarding devices, leading to bandwidth overhead. In this dissertation, we propose a causal-based consistent update approach that ensures the connectivity update properties: transient forwarding loop-free and transient forwarding blackhole-free. This

is achieved by defining a formal model of the two connectivity invariant violations as a specification of the Happened-Before relation of Lamport. Based on this model, network update policies are introduced by establishing causal dependencies between relevant update events. These update policies are reflected by an update mechanism oriented towards transient connectivity inconsistency-free SDN updates. To prove the correctness of the update mechanism, it was demonstrated that it is sufficient to ensure the transient forwarding loop-free and the transient forwarding blackhole-free properties. In terms of findings, the formal modelisation of the two connectivity update properties defines the root cause of their triggering and capture the conditions under which they may occur. Accordingly, the proposed update mechanism is designed to prevent the triggering of these conditions, without requiring the use of global time references. Furthermore, as to the update scheduling, $O(1)$-step is required to perform an update. Unlike the other solutions, the mechanism promotes the availability of forwarding paths for data packets during updates. This is by permitting to packet flows to traverse both forwarding paths, i.e. the initial one related to the old forwarding policy and the final one related to the new forwarding policy. Finally, it can be concluded that connectivity consistency update is achieved by ensuring causal dependencies between events making end to the initial network policy and events setting the final one. As to the continuation of this work, the research community of network area may leverage from this investigation by proving the correctness of their update mechanisms across the formal definitions of the connectivity inconsistency phenomenons. Also, the proposed update mechanism can be adopted to be emulated in read world SDN update tasks.

Contents

List of Figures

List of Tables

List of Acronyms

SDN	Software-Defined Network
REST	REpresentational State Transfer
ForCES	FORwarding and Control Element Separation
OVSDB	Open vSwitch Database
POF	Protocol Oblivious Forwarding
ROFL	Revised Open-Flow Library
FIS	Flow Instruction Set
ONF	Open Networking Foundation
HBR	Happened-Before Relation
IDR	Immediate Dependency Relation
ENOS	Existing nodes
INOS	Inserted nodes
FNOS	Forgotten nodes

List of Notations

src	A source forwarding device
s_1,..., s_n	Intermediates forwarding device
dst	A destination forwarding device
cp	The controller process
rp_1, ..., rp_n	The routing processes related to the forwarding devices
m_1, ..., m_n	The messages exchanged between the different processes
pkt_1, ..., pkt_n	The data packets
mp_1, ..., mp_n	Messages or data packets
$fpath_1$, ..., $fpath_2$	Forwarding paths between sources and destinations forwarding devices
$fpath_1^{initial}$, ..., $fpath_n^{final}$	Initial forwarding paths between sources and destinations
$fpath_1^{final}$, ..., $fpath_n^{final}$	Final forwarding paths between sources and destinations
$match_1$, ..., $match_n$	Flag that identify packets, flow packets and forwarding paths
$SdM(m)$	The sending event of a message
$FwdP(pkt)$	The forwarding of a data packet
$RecMP(mp)$	The reception of a message or a data packet
$DlvMP(mp)$	The delivery of a message or a data packet
$T(m)$	A function that returns the transmission time of a message m
PT	An instance of the physical time
CI_i	A control information vector
$Delivery_i$	A matrix of messages/packets delivery related to a process p_i
$\rightarrow$	Happened-Before Relation
$\downarrow$	Immediate Dependency Relation

Chapter 1

Introduction

1.1 Motivation

Nowadays, Software-Defined Networks (SDNs) have became the spine of daily and critical used services, e.g. data management of banking platforms. Supporting such applications and services requires frequently SDN updates. Reasons behind performing updates became manifold, among them, security policy changes, traffic engineering, maintenance work, adding or removal of services, etc. [FSV16].

Applications sensitive to disruption, such as data delay or loss, cannot tolerate such errors, affecting network quality of service (QoS). To ensure a better QoS, it is important that networks obey correctness criteria during transition phases, i.e. throughout updating networks. For this, network operators have to guarantee that some desirable properties are preserved, ensuring consistent network updates. Preserving connectivity consistency on updating SDNs, which includes preserving forwarding loop-free and forwarding blackhole-free properties, is one of the network consistency categories. In fact, connectivity consistency concerns the delivery of data packets to their respective destinations, and therefore prevents data loss.

Different approaches have been proposed to deal with inconsistent connectivity update problems in SDNs. Recent results show that preserving connectivity consistency may be achieved based on different methods and techniques. However, an optimal trade-off between ensuring consistent connectivity updates and optimizing their performance remain a challenge. One network update performance criteria is the number of rounds required, i.e. the number of steps required to execute an update, while guar-

anteeing consistency. Ensuring consistent connectivity updates while minimizing the number of rounds is commonly considered in the state-of-the-art. For instance, results provided by [FLMS18] prove that, in the worst case, $\emptyset(n)$-rounds loop-free update schedules always exist where n is the number of network entities. Indeed, this reflects that performing consistent updates require incrementally installing intermediate network configurations. This qualifies updates as bandwidth-consuming tasks as more an update requires to install intermediate configuration more it requires communication between network entities.

On reviewing the state-of-the-art, and as yet, we have not found any work that has studied and formally specified the root cause(s) behind the triggering of inconsistent connectivity update problems during updates. Such studies are important as they allow to identify and describe the problem clearly, referring to define how a SDN passes from a connectivity consistent network state to a connectivity inconsistent network state. Understanding the latter mentioned allows distinguishing between the root cause of the problem and other causal factors, clarifying the view over the triggering of inconsistent connectivity behaviours and helping to design better update consistent methods in terms of performance.

1.2 Problem description

1.2.1 The SDN update problem

A SDN offers a new level of abstraction, reducing the complexity of network update tasks. This is through its single point of control named the control plane or the controller. Unlike the IP networks, updates in SDNs are no longer the responsibility of the forwarding devices, entitled the data plane in a SDN context. The SDN control plane is the entity responsible for supporting the different network management requirements,

including the update of forwarding policies. Therefore, no distributed computations are needed anymore on forwarding devices, e.g. switches, to perform updates. Instead, the controller has to compute a sequence of operations that changes the packet-processing rules installed on forwarding devices, and then communicate the rules to them via message exchanges. Given an initial and final forwarding policies, an update consists in moving from the initial policy to the final one by applying the computed sequence of operations on the underlying switches. For instance, in Figure 1.1, the initial forwarding policy forces the outgoing packets from the source switch *src* to reach the destination switch *dst* along the forwarding path (*src*, intermediate switches (...), s_i, s_{inter}, ..., s_j, ..., s_k, *dst*). The network update problem consists in replacing the initial forwarding rules with new ones to consider the new policy forwarding packets to *dst* along the path (*src*, ..., s_k, ..., s_j, ..., s_{inter}, ..., s_i, *dst*).

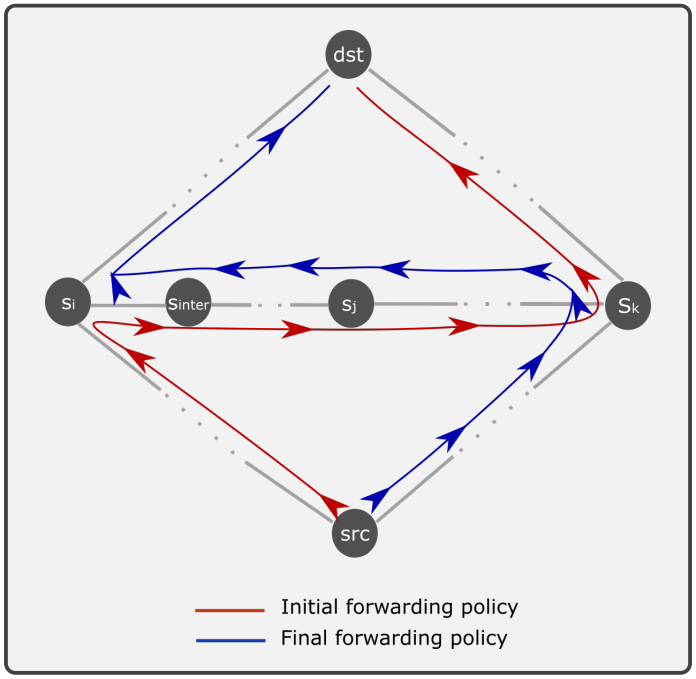

Figure 1.1: Packet flow forwarding policy changes: an example

1.2.2 The difficulty in solving inconsistent connectivity update problems

The SDN inconsistent connectivity update problem consists in avoiding two inconsistent phenomenons during updates, i.e. during the interval of time when moving from one forwarding policy to another. These two phenomenons are the forwarding loop and the forwarding blackhole. Informally, we can define them as follows:

- **Forwarding loop.** It occurs when a data packet is forwarded along a loop back during an arbitrary time interval to a forwarding device in the network where it was previously processed.

- **Forwarding blackhole.** It occurs during an update when a data packet arrives at a forwarding device and there is no a matching forwarding rule into the forwarding device table to forward them to its next hop.

Avoiding the triggering of these inconsistency connectivity phenomena during updates depends on :

1. The order in which operations appear in the computed sequence of update operation.

   For example, if the controller calculates that the intermediate switch s_{inter} should update its new forwarding rule before s_i, then this operational sequence triggers a forwarding loop between s_i and s_{inter}. Even worst, if the controller considers that all intermediate switches situated between s_j and s_i should update their new forwarding rules to forward all ingoing packets from s_j to s_i before deleting the initial ones that forward the ingoing packets from s_i to s_j, then this operational sequence gives rises to a forwarding loop between s_i and s_j passing by all the intermediate switches between them.

As to forwarding blackholes, for example, if the controller calculates that s_i has to update its forwarding rules by deleting the initial rule that forwards the ingoing packets before that src updates their forwarding rules by deleting the rule forwarding packets to s_i, then this operational sequence results that packets forwarded from src to s_i enter in forwarding blackhole as the rule responsible to forwarding the packet in question is already deleted.

2. The order of execution of the operational sequence by forwarding devices.

   Even if the controller is able to calculate the correct operation update sequences, e.g. updating s_{inter}'s rule after s_i's one in the first previous example. However, the order in which the underlying switches execute update operations is not necessarily the same order calculated by the controller. Indeed, the controller may start by sending a message instructing s_i to update, and then sending another messages to s_{inter} instructing it to update, ending by updating s_{inter}'rule before s_i one. This is can be explained due to the asynchronous communication between the controller and switches. The communicated update messages by the controller and in-fly data packets are prone to arbitrary and unpredictable transmission delays. Moreover, the absence of a global temporal reference in SDNs results in a broad combinatorial range space of order of which update operations may take place.

1.3 Research questions and Hypothesis

The described problem raises the following research question:

- How can we ensure consistent connectivity updates in SDNs to enforce the correctness of data plane when network policies change, maintaining an asynchronous communication between network entities?

This research question evokes further questions, which help us to lead the research. In a first instance, a question about how a SDN can be modeled to achieve connectivity consistency during updates. Thus, we formulate the following question:

- How can we model a consistent execution view of a SDN while facing an inconsistent network view from the control plane?

Furthermore, specific questions about how to achieve the connectivity consistency in SDNs are evoked. Thus, we formulate the following questions:

- How can we model SDN's events to identify patterns of events orderings that can give rise invariant violations in SDNs?
- How can we ensure an update order, needed to support consistent connectivity updates in SDNs when network policies change?

To lead this research, we propose the following hypothesis:

In SDNs, when network policies change, network connectivity consistency can be achieved by ensuring causal dependencies between events that eliminate the initial network policies and events that set the final ones.

1.4 Objectives

1.4.1 General objective

The general objective for this research is:

- To design a mechanism to ensure consistent connectivity correctness properties: forwarding loop free and forwarding blackhole-free, which serves as support to reach network consistency in SDNs during updates.

1.4.2 Specific objectives

- To model SDN updates at the event level according to the distributed and asynchronous nature of SDNs.
- To identify patterns of events order that can give rise inconsistent connectivity invariant violations in SDN.
- To design a mechanism, based on causal order principles, to support network consistency when network traffic evolves.

1.5 Proposed solution

In this dissertation we propose as main contribution, a formalization of the inconsistent connectivity update problem, forwarding loops and forwarding blackholes, during SDNs updates. Based on this formalization, the network update policies and an associated update mechanism are provided.

For our proposal, SDN updates are modeled at the event level according to their distributed and asynchronous nature.

The formal model of the forwarding loop and forwarding blackhole invariant violations are modeled based on causal dependencies[1], capturing the conditions under which connectivity inconsistency may occur. This formal model is a key contribution in this work since it defines event-based patterns, defining the root causes behind the triggering of each invariant violation leading to connectivity inconsistency during SDN updates.

Update policies establish causal dependencies between relevant update events in order to break the triggering of the defined connectivity inconsistency event-based pat-

[1]This is based on the happened-before relation defined by Lamport in [Lam78]

terns. Such policies allows us to build an update mechanism that does not require to synchronise clocks of forwarding devices and/or to use global references.

1.6 Associated publications

The findings of this research work were subject of the following two publications:

1. A study of the forwarding blackhole phenomenon during software-Defined network updates. [GHH+19]

2. Towards causal consistent updates in Software-Defined Networks. [GHH+20]

1.7 Thesis organisation

This document is organized as follows. In Chapter 2, the main concepts of SDNs and distributed systems are presented to support the rest of this dissertation. In Chapter 3, a frame and a discussion of related work are presented. In Chapter 4, the problem of inconsistent connectivity updates in SDNs is formalized. Based on this formalization, in Chapter 5, network update policies are introduced and an update mechanism is presented based on these policies. Also, a formal proof that the mechanism is free from connectivity violations during updates is provided. In Chapter 6, we evaluate and discuss the proposed approach, comparing it with the state-of-the-art. Finally, the conclusion and the future work of this research are summarized in Chapter 7.

Chapter 2

Background and Definitions

In this Chapter, we focus on presenting fundamental concepts related to the research areas: Software-Defined Networks (SDNs) and Distributed Systems, serving the development of this thesis.

2.1 Software-Defined Network paradigm

Software-Defined Network (SDN) refers to a new generation of computer networks. SDN was launched in 2008. It was standardized by the Open Network Foundation (ONF) [ONF12] and implemented by a number of original equipment manufacturers (e.g., HP, CISCO, IBM, Juniper, NEC and Ericsson).

This new paradigm of network architecture has reshaped several concepts of the classical IP network model. First, it breaks the strong coupling between the control plane and the data plane. By control plane, it refers to entities that decide how to handle network traffic, and by data plane to entities whose responsibility is to forward traffic according to the decisions made by the control plane. In IP networks, each entity has its own operating system. Network handling is a shared responsibility between operating systems of all entities. In SDNs, however, the control plane represents a logically-centralized entity representing the operating system of a whole network, named also a controller. It is important to highlight that a logically-centralized entity does not force that the controller needs to be implanted in a single entity [JMD14]. However, a controller can be implanted in physical/virtual distributed entities. Second, the data plane becomes a set of network entities where their unique responsibility is to forward

network traffic, then delegating the control logic management to the control plane.

Second, SDN, and as it name says, is a programmable network. The concept of a programmable network was born by means of the mentioned separation between the two planes. Indeed, tasks such as the definition of network policies, their implementation in switching hardware and the forwarding of traffic are now programmable. A well-defined application programming interfaces (API) are implemented between the network devices and the controller as well as between the controller and the application plane. The later is the highest level SDN's plane that permits network administrators to manage their networks [KRV+15]. SDN may be defined as follow.

Definition 1 *Software-Defined Network (SDN) is a network management approach that separates the control logic from forwarding devices to be logically-centralized into a network operating system, enabling programmatically configuring the network using software interfaces [JMD14, KRV+15].*

In the rest of this chapter, an overview of the general architecture of SDN is presented. Also, network traffic management in SDN is discussed.

2.1.1 Architecture and terminology

As mentioned above, SDN is an emerging paradigm that relies on decoupling the tightly coupled implantation of the control and data planes, where the control logic is separated from network devices and it is implanted in a logically centralized controller [JMD14]. Thus, SDN mainly consists of three planes: the *application plane*, the *control plane* and the *data plane*. Figure 2.1 depicts a simplified view of the SDN architecture. The application plane is presented as a set of network applications which implement network control logic (e.g. firewall, traffic engineering, load-balancer, etc.) leveraging a *northbound interface*, e.g. Representational state transfer (REST) API [ZLLC14],

which offers universal network abstraction data models and functionality to developers [JMD14, KRV+15]. This set of network applications also either explicitly or directly notifies the network behaviour to the control plane by means of a northbound interface. The controller supports the translation of the network requirements and the desired behaviour of the application plane to the data plane based on a *southbound interface*, e.g. OpenFlow [ONF15] (see more details about other southbound APIs in subsection 2.2.2. 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 [JMD14, KRV+15].

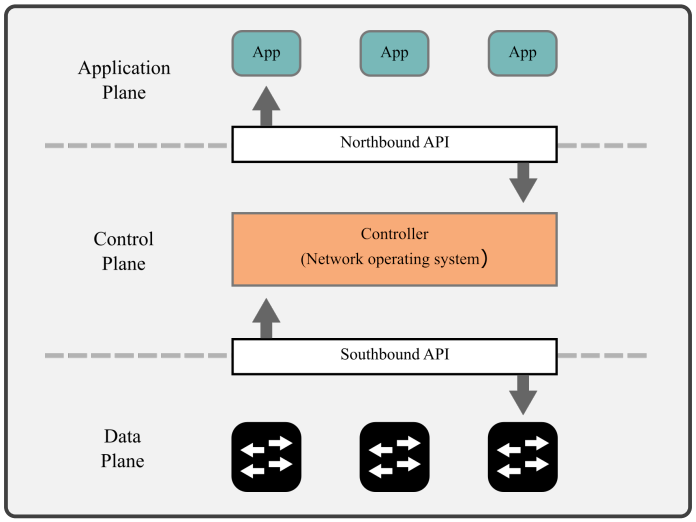

Figure 2.1: The general architecture of SDN.

2.2 Network update in Software-Defined Networks

In this Section, we start by describing how traffic is handled in SDNs. Then, we move to distinguish between two standards of routing models adopted by SDNs. Finally, we present the concept of consistent update in SDNs.

2.2.1 Network traffic handling in an OpenFlow-based SDN

In SDNs, network traffic is managed by the controller, qualified as the network brain [KRV+15], which is responsible to make decisions about the traffic. The controller acts as a translator between the application and data planes. It provides the requirements communicated by the application plane, via the northbound APIs, by communicating them as a set of instructions to the data plane, via the southbound APIs, allowing them to handle traffic.

Definition 2 *OpenFlow is a southbound API implementation that provides a message-based communication protocol defining the way a control plane and data plane communicate [ONF12].*

OpenFlow is the Open Networking Foundation (ONF) [ONF12] southbound standard API. This standard started with academic studies and then it gained traction in the industry. Many vendors of commercial forwarding devices (e.g., switches) include support of the OpenFlow API in their equipment. In this dissertation, the notation of all defined abstractions is built on an OpenFlow-based SDN.

To establish network traffic, the controller interacts with switches via an OpenFlow channel interface that connects each switch with the controller, permitting them to exchange OpenFlow messages based on the OpenFlow protocol. This protocol provides a reliable message delivery and processing but does not ensure ordered message processing [ONF15].

The set of OpenFlow messages is classified into different categories [ONF15]. The controller-to-switch is one of the OpenFlow message categories and is the relevant message category for this thesis. It includes the modify-state messages. These messages are sent by the controller to handle the state of the switches. FlowMod is a controller-to-switch message that instructs an OpenFlow switch in the network to add/update/delete

entries of its flow table. A flow table is a switch table that contains entries, named also forwarding rules. Based on its table of entries, a switch can perform actions on an ingoing data packet/packet flow. When receiving a data packet, a switch looks up into its forwarding table for an entry that matches the data packet/packet flow, and then performs, as defined in the entry, the action over the matched packet/packet flow (dropping, forwarding, modifying, etc.).

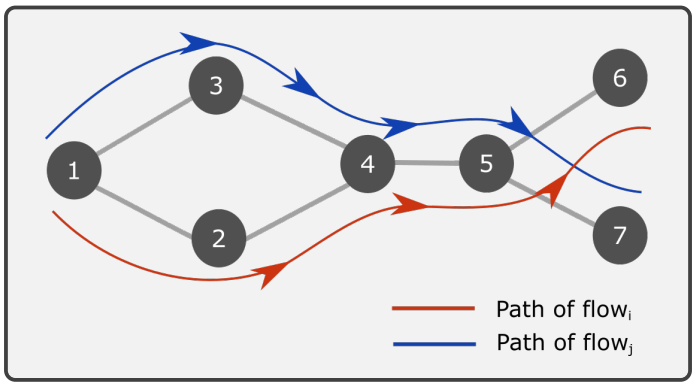

Figure 2.2: An example illustrating the flow-based traffic management in SDNs.

OpenFlow is considered as a SDN implementation that adopts a flow-based traffic routing model[1] [KRV+15]. A flow is defined as a set of packet field values (flags) acting as a match (filter) criterion and a set of actions (instructions). Also, it is defined as a sequence of packets between a source and a destination entities. Hence, each OpenFlow packet of a specific flow that carries the same flags is identified and equally treated at the forwarding devices. The example of Figure 2.2 shows a toy example that explains how an OpenFlow SDN handle packet flows. The example illustrates two packet flows: $flow_i$ where its forwarding path is colored with red, and $flow_j$ where its forwarding path is colored with blue. Imagine that packets of $flow_i$ carries "a.z" and packets of $flow_j$ carries "b.z" as a match field.

Despite that $switch_4$ and $switch_5$ process packets of $flow_i$ and $flow_j$, however, both

[1]In the next subsection, we distinguish between flow-based and destination-based routing models.

Match	Action
*.z	Fwd to $switch_5$

Table 2.1: Flow table of $switch_4$

Match	Action
a.z	Fwd to $switch_6$
b.z	Fwd to $switch_7$

Table 2.2: Flow table of $switch_5$

switches use different forwarding rules. On the one hand, as shown in Table 2.1, the flow table of $switch_4$ contains a single rule. This rule corresponds to all packets that piggyback a match with prefix "*", i.e. any prefix, and a suffix "z". This rule will match packets of flow $flow_i$ and $flow_j$ and it ends by forwarding them to $switch_5$. On the other hand, the Table 2.2 illustrates the flow table of $switch_5$ containing two forwarding rules: the first one is to forward all packet piggybacking a match= "a.z" (packet of $flow_i$) to $switch_6$. The second one is to forward all packet piggybacking a match= "b.z" (packets of $flow_j$) to $switch_7$. Thus, the two flows reach their destinations ($switch_6$ and $switch_7$).

2.2.2 Network traffic handling based on others SDN Southbound APIs

In the previous subsection, we described how traffic is handled based on an Openflow southbound API. However, OpenFlow is not the only specification of southbound API in the SDNs. There are other API proposals such as Forwarding and Control Element Separation (ForCES) [DSH+10], Protocol Oblivious Forwarding (POF) [Son13], Open vSwitch Database (OVSDB) [PD13], OpenState [BBCC14] and Revised Open-Flow Library (ROFL) [SAJ+14].

ForCES [DSH+10] proposes a network management approach without changing the current architecture of the traditional network. This means that logically centralized external controller is not needed. The separation of the control and data plane is implemented but in the same network element.

POF [Son13] provides an approach to enhance the current forwarding plane of OpenFlow-based SDNs. In OpenFlow, switches have to extract the field values of the ingoing data packets from their headers to be matched with the flow tables entries. This parsing represents a significant cost for devices. To tackle this, POF proposes a generic flow instruction set (FIS). With this approach, forwarding devices have not to be aware about the structure of the matches fields of data packets. However, packet parsing is the responsibility of the controller that results in generating a key values table lookup that should be installed in the forwarding devices.

The OVSDB [PD13] is a complementary southbound API protocol to OpenFlow designed for virtual Open vSwitchs. It offers other networking functions beyond the capabilities of OpenFlow, e.g, allowing the control elements to create multiple virtual switch instances and tying interfaces to the virtual switches.

OpenState [BBCC14] and ROFL [SAJ+14] do not propose a FIS to handle the matching of data packet with entries of forwarding devices. However, on the one hand, OpenState proposes to extend the OpenFlow matching abstraction by finite machines, allowing the implementation of several stateful tasks inside forwarding devices. On the other hand, ROFL implements a new layer as a facade to hide the complexity of the different OpenFlow packet header versions and then provides an API which simplifies the application deployment [KRV+15].

2.2.3 Routing models

Forwarding devices support rules that route packet flows based on one of these two alternative routing models: destination-based and flow-based routing models.

Destination-based routing. Following this routing model, devices forward packets based only on their destinations. This model permits confluent paths, i.e. two different packet flows with different sources and a same destination may intersect in an

intermediate device in the network, and then continue to their destination based on the same path. This means that one packet flow may be processed and controlled by more than one specific service policy when traversing its route from the source to destination.

Flow-based routing. On the other hand, with a flow-based routing model, routes are not confluent. In fact, rules forward packets based on the flow its belong to. A flow may be identified based on various criteria, e.g., source, destination, etc. The benefits of such routing model is that a single flow is processed and controlled based on one specific service policy. We highlight that a typical SDN network uses a flow-based routing [KRV+15]. In this thesis, we assume a flow-based routing model. In Chapter 4, we show how this routing model is a key point that permits us to identify inconsistency problems.

2.2.4 Consistent updates

Consistent updates mean that the network should meet strict requirements in terms of correctness and availability when updating from one network policy to another. The correctness and the availability of network are measured in terms of consistency properties. These properties are classified by category [FSV16]: connectivity-based, policy-based and capacity-based.

Connectivity-based. This guarantees the correct delivery of packets to their respective destinations. The properties to be ensured are forwarding loop-free and forwarding blackhole-free. Informally, on the one hand, forwarding loop-free 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 [GHH+20]. On the other hand, forwarding blackhole-free guarantees that a forwarding device tat receives a packet is able to forward it to its next hop. This thesis focuses on connectivity-based consistent updates.

Capacity-based. Networks inherently are limited in capacity. The consistency in this category is defined as not violating any link capacities. The consistency property to ensure is congestion-free. Migration from one forwarding policy to another should not bypass the capacity of any link that connects any two forwarding devices in a network. To locate exactly congestion-free consistency, other properties are taken into account, e.g., buffer sizes of forwarding devices.

Policy-based. Policy-based refers to applying a set of rules to forward specific packet flows from their sources to their destinations. In this category, requirements bypass connectivity-based and capacity-based and are focused on the paths, sub-paths, or even nodes from which a packet passes at a given time. The consistency property guarantees that each packet is processed by either the in-prior of update policy or the configuration in-place after updating. This is was formalized by means of the per-packet consistency property [RFR+12] as an abstraction that guarantees that every packet traversing the network is processed by only one specific network policy.

2.3 Distributed system and causal ordering

2.3.1 Distributed systems model

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 [GHH+20].

- Processes: programs or instances of programs running simultaneously and communicating with other programs or instances of programs. Each process belongs to the set of processes $P = \{p_i, p_j, ...\}$. and can only communicate with other processes by message passing over an asynchronous communication network.

- Messages: Abstractions which represent either arbitrarily simple or complex data structures. Each message in the system belongs to the set of messages $M = \{m_i, m_j, ...\}$.

- Events: An event e_n represents an instant execution performed by a process $p \in P$. Each event e_n in the system belongs to the set $E = \{e_i, e_j, ...\}$. We can distinguish two types of events: internal and external events. An internal event is an action that locally occurs at a process, e.g. the computation of the value of a local variable. 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, the receive and the delivery events. A send event identifies the emission event of a message $m \in M$ executed by a process. A receive event denotes the notification of the reception of a message $m \in M$ by a recipient process, whereas a delivery event identifies the execution performed or the consumption of m.

2.3.2 Partial and total order relations

In distributed systems, there is two adopted approaches to ordering events: the partial and the total order relations. As to the partial ordering, only the order of certain events in E is identified. In other words, one can not be sure of the exact order of all the events in the system. The partial order relation between every pair of event should satisfy the following properties: the reflexivity, the antisymmetry and transitivity.

- The reflexivity refers to that a relation R relates each element x of a set X to itself. Formally, this property can be expressed as follows:

$$\forall x \in X : xRx \tag{2.1}$$

- The antisymmetry refers to that a relation R relates only pairs of distinct elements of a set X. Formally, this property can be written as follows:

$$\forall a, b \in X : aRb \text{ with } a \neq b \tag{2.2}$$

- The transitivity refers to that if a relation R that relates an element a of X to another element b of X and also it relates the same element b to another element c of X, then R also relates a to c. This is formally denoted as follows:

$$\forall a, b, c \in X : \text{if } (aRb) \text{ and } (bRc) \text{ then } (aRc) \tag{2.3}$$

Partial order relation is qualified as strict when it satisfies the irreflexive property instead of the reflexive property.

- The irreflexivity refers to that a relation R does not relate any element of a set X to itself. Formally, this property can be expressed as follows:

$$\forall x \in X : x \not{R} x \tag{2.4}$$

As to the total order relation, it allows to identify when each of the events of E occurred and then identifies the total order of all the events in a system. This relation should satisfy the properties of a partial order as well as the connexity property.

- The connexity refers to that a relation R relates an element a of a set X to another element b of X or relates b to a and not both. Formally, this property can be written as follows:

$$\forall a, b \in X : aRb \text{ or } bRa \tag{2.5}$$

2.3.3 Logical Time and causal ordering

In distributed systems, time represents an important theoretical construct allowing to identify when events are executed . However, in a typical distributed system, it is difficult to determine whether an event takes place before another one due to the absence of a global physical time. In this context, the logical time, expressed by means of the Happened-Before Relation (HBR), establishes an agreed time between all processes that can establish the execution order relation between any two events in a distributed system. This order relation was defined by Lamport [Lam78]. 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 3 *The Happened-Before Relation (HBR) [Lam78] "$\rightarrow$"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 $a \rightarrow c$.*

Based on Definition 3, the author of [Lam78] defines that a pair of events is concurrently related as follows:

Definition 4 *Two events, a and b, are concurrent if $a \nrightarrow b$ and $b \nrightarrow a$, which is denoted by $a \parallel b$.*

The use of HBR to maintain the causality is expensive. This is due to the transitively Happened-before relationships between events (defined in the third condition

of the HBR definition) that should be considered between pairs of events. The author of [PH15] proposed the Immediate Dependency Relation (IDR), the minimal binary relation of the HBR that has the same transitive closure.

Definition 5 *The Immediate Dependency Relation (IDR) [PH15] "$\downarrow$"is the transitive reduction of the HBR, and it is defined as follows:*

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

2.3.4 Causal order delivery

Causal order delivery represents a fundamental property in the field of distributed systems. This property is useful for synchronizing distributed protocols by inducing the causal order of events. Informally, this means that if we have two send messages m and m' that are causally related and are sent to the same process p_i, then p_i should ensure that the delivery is held according to the causal order, i.e., respecting the sending order.

In [PH15], authors demonstrate that to ensure a causal order delivery for a multicast communication, it suffices to ensure the causal delivery of immediately related send events where the set of relevant events is determined to be $R = \{sent(m) : m \in M\}$. Formally, the message causal delivery based on the IDR-R, "$\downarrow_R$" (see the definition in [PH15]), was defined as follows:

Definition 6 ***Causal order delivery***

$$\forall((send(m), send(m')) \in R, send(m) \downarrow_R send(m') \Rightarrow$$
$$\forall p \in Dest(m) \bigcap Dest(m') : delivery(p, m) \rightarrow delivery(p, m')$$

2.4 Chapter summary

This chapter presents the fundamentals concepts and terminologies related to SDNs and distributed systems. On the one hand, fundamentals of SDNs was presented to clarify the purpose behind designing such computer network architecture, and to explain how traffic is handled based on such network architecture, specifically based on the OpenFlow communication protocol. On the other hand, time and ordering concepts in distributed systems was formalized to present the theoretical framework in which we will base to tackle inconsistency connectivity problems in SDNs, including Lampord' logical time and causal ordering concepts.

Chapter 3

Related work

3.1 Consistent Network update

Since the emergence of IP networks at the beginning of the years 80, consistent update problems have been faced by network administrators. Generally speaking, a consistent network update consists of defining a set of operations that changes the packet-processing rules installed on network devices to migrate from one forwarding policy to another, guaranteeing network consistency through preserving relevant network properties. In IP networks, the packet-processing updates are computed by routing protocols that run distributed algorithms. However, distributedly updating network topology and configurations may give rise to transient and/or permanent inconsistent behaviour, e.g. packets loops between a set of forwarding devices. Most works were focused on studying the Interior Gateway Protocol (IGP), a standard used to control packet-forwarding within a single network. Extensions to IGP are proposed to avoid forwarding disruptions, among them, the work proposed by Francois and Bonaventure in [FB07] guaranteeing the absence of forwarding loops and the work introduced by Dube R. et al in [ARA06] avoiding forwarding blackholes. An overview of these contributions is presented in [FSV16].

After the emergence of SDNs in 2008, consistent network update has got more interest in the network community. On the one hand, updates, with the SDNs, are no longer configurable tasks. However, they are programmable, leading to fast updates. On the other hand, current network requirements have been changed. For example, in the context of a traffic engineering application, an administrator may frequently decide

to reroute parts of the traffic along different forwarding paths to guarantee a better load balancing between links that connect forwarding devices. Such reason multiplies the purposes by which an administrators need to perform updates. Fastly and frequently performing network updates in SDNs makes them critical tasks.

In the rest of this chapter, the state-of-the-art of the consistent network update approaches are presented. To introduce the update techniques, a taxonomy of these proposed techniques is presented. Then, in a second part, an overview of works that have focused on studying consistent update techniques oriented to performance objectives is presented.

3.2 Consistent Network update approaches in SDNs

The scope of consistent network updates has been studied intensively in the last years, resulting in touching many study axes. This includes analytical studying the different inconsistent network phenomenons which can occurred [FMW16, FLMS18], updating techniques [RFR+12, LRFS14, MM16, LSM19], optimizing network-update performance [MW13, ALMS16, FW16, LMS15, FLMS18, FLSW18] and simulating update mechanisms [VC16, FMW16, ME16]. A survey studying the state-of-the-art in this field is presented in [FSV16]. The mentioned survey introduces a profound review of the consistent updating approaches in SDNs, generalizes the network updating problem and proposes a taxonomy of consistent network updating problems. The taxonomy provides a classification of the update approaches per category of consistency properties: connectivity-based, policy-based, and capacity-based properties (more details about these concepts are provided in Section 2.2.4). Furthermore, it presents a per objective classification : link-based, round-based and cross-flow objectives.

In this section, we introduce a technique-based taxonomy of the proposed solutions.

This classification presents an overview of the algorithmic techniques proposed in the literature to solve the specific class of update problems attacked in this thesis: forwarding loops and forwarding blackholes. This permits to clear up the advantages and drawbacks of each proposed updating approach, leading to a discussion of the inherent limitations and trade-off between the achievement of consistent updates and its impact on network performance. This discussion is presented later in Chapter 6.

3.2.1 A taxonomy of consistent updating techniques

Based on the reviewed literature, works that have been proposed to tackle the consistent network update problem fall into the following update techniques: the n-phase commit update [RFR+12, VC16, KRW13, NT17, FSYM14], the ordered update [LRFS14, LwZ+13, FMW16, VC16], the timed update [MM16, ME16], and the causal update [LSM19] techniques.

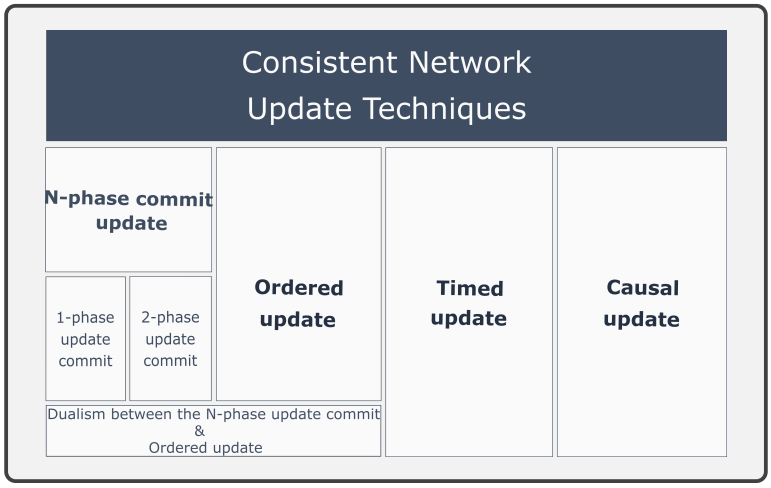

Figure 3.1: Taxonomy of consistent update techniques

Figure 3.1 overviews the taxonomy of the technical solutions proposed for the consistency update problem. In the rest of this section, we brevity discuss each proposed

update technique, discussing some advantages and drawbacks of each one. The Figure 1.1 of Chapter 1 is reproduced in this Chapter via Figure 3.2 for exemplification purposes.

N-phase commit update. we distinguish between two N-phase commit update techniques: 2-phase commit update and 1-commit update.

On the one hand, [RFR+12, KRW13] propose solutions based on the 2-phase commit approach. Such works are designed based on the packet tag-match mechanism. Initially, when data packet are "in the way" to their destinations, forwarding devices tag all in-fly packets with the current forwarding configuration k. In the example of Figure 3.2, initially, all packets that take the initial forwarding policy path are tagged with the current configuration version k where they match flow table rules related to the configuration k of switches that belongs to the initial forwarding path. On starting the updated policy, the controller (not depicted in Figure 3.2) starts by disseminating all updating messages of configuration $k + 1$ to install the new corresponding rules on forwarding devices, and then waits for their acknowledgements. When all switches acknowledge the installation of the new rules (referred to as rule addition), the controller instructs them to tag all incoming packets with $k+1$ to the match rules of the new policy to forward packet based on the final forwarding path (see Figure 3.2). Once packets tagged with k leave the network, the controller instructs the switches to remove rules of version k [GHH+20].

This update technique provides a strong per-packet consistency guarantee: packets are forwarded to their destinations based on the initial or the final configuration and not both. However, during the update transition phase, all switches have to maintain the forwarding rules of both forwarding policies until all packets tagged with k leave the network. This is in fact may give rise to switch memory overhead.

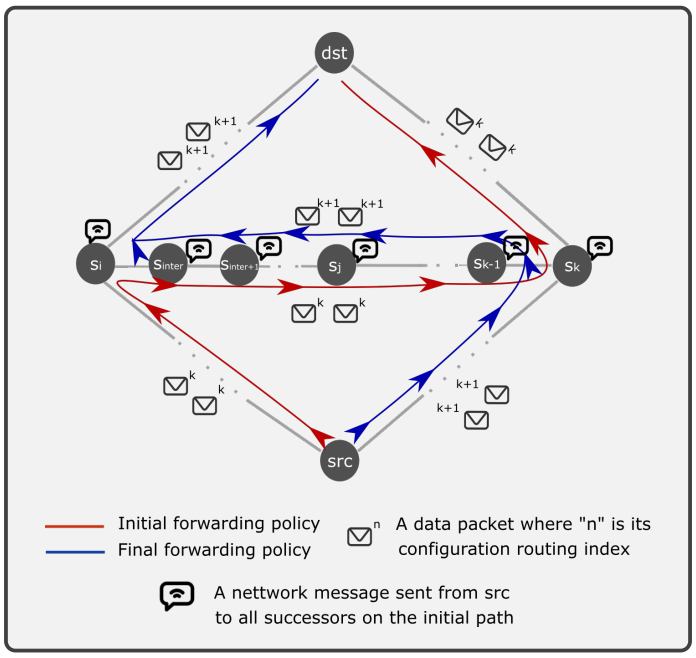

Figure 3.2: Forwarding policy changes example

On the other hand, Canini et al proposed ez-Segway in [NT17], a 1-phase commit update consistent mechanism. With ez-Segway, the controller computes and sends the information needed to the switches once per update, qualifying this update approach as a 1-phase commit. This is based on the basic-update technique: when a switch incorporates the new path related to the new forwarding policy, a message is sent from this switch to their successors and until the destination switch in order to inform them that no packets will be forwarded anymore along the old path. Thus, on receiving this message, each switch shall remove the old rule which forwards packets on the old path, ensuring the blackhole-freedom property. This means that when installing the new rule on the *src* switch to forward ingoing packets towards s_k instead of s_i, then *src* sends messages to all the successors on the initial forwarding path (see Figure 3.2) informing them that no packet belonging to this particular packet flow will pass in the future. Thus, this allows switches to safely remove the old rules without risking the violation of

the blackhole-freedom property. To accelerate the flow update, a segmentation of basic-update is proposed. It consists on splitting the update of a flow path into sub-paths and simultaneously perform their updates. For example, to update to the final forwarding policy of Figure 3.2, the update mechanism may split the update as follows: simultaneously update the segment from switch src to s_{k-1}, the segment from s_k to s_{inter} and the segment from s_i to dst. To tackle the forwarding loop phenomenon, the controller classifies segments into two categories, called InLoop and NotInLoop. Then, it calculates a dependency mapping that assigns each InLoop segment to a NotInLoop segment. Taking the segment from switch s_k to s_{inter} (see Figure 3.2). This segment is an Inloop segment as packet can enter into a forwarding loop when the final forwarding path (depicted in blue) will be installed before the removing of the initial one (depicted in red). Thus, these switches have to install new rules and remove old entries based on the information provided by the appropriate dependency mapping calculated by the controller to establish a NotInLoop segment [GHH+20]. An interesting contribution of this work is the ability to compute and send the information needed to update switches once per update. However, consistently update networks with ez-Segway requires message exchange between switches themselves which represents a bandwidth overhead into a typical SDN.

Ordered update. Works like [LRFS14, LwZ+13, FMW16, VC16] design mechanisms that calculates a sequence of steps to replace rules installed in flow tables of switches, where the order of execution ensures that no inconsistent behavior appears throughout the transition from one forwarding policy to another. For example, updating src, s_i and all their intermediate switches (see Figure 3.2) in one step guaranteeing an update free from forwarding loops. However, this can generate a forwarding blackhole in src as ingoing packets will not match a rule that forwards them to their next hope (i.e. switch/node).

Among the drawbacks of this update technique is that, during the transition at each step, the controller should wait until all the underlying switches finish their updates (referred to as rule replacement). Then, they should inform the controller, through acknowledgement messages, in order to be able to initiate the next step of the updating sequence. Thus, an update finishes after the achievement of all the updating steps.

Recently, the authors of [VC16] proposed FLIP, an updating approach built on the dualism between the ordered update and the 2-phase commit update approaches. FLIP uses rule replacement and rule addition techniques to match the in-fly packets with their forwarding rules in all devices, guaranteeing the per-packet consistency property. Technically, this is based on two core procedures: constraint extraction and constraint swapping. The constraint extraction identifies the constraints on replacing and/or adding entries, ensuring a safe update. Based on the identified constraints, alternatives one are also inferred. Once all the constraints are extracted, FLIP calculates a sequence of update steps that aligns with all the constraints. This is by applying a linear program where the objective is to minimize the number of update steps. If such solution can be found, FLIP apply the outputted sequence to perform the update, otherwise it proceeds with the constraint swapping procedure. The later consists in taking the alternative constraints to replace active constraints to calculate the operation steps [GHH+20]. Thus, the contribution of this work consists in reducing the number of rule additions during update phase comparing the 2-phase commit update approaches.

Timed update. The works [MM16, ME16] proposed an update technique based on synchronizing time references. 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. Backing to the example of Figure 3.2. This means that all switches will update in the same period of time, and then the final forwarding policy path will be available once the initial one is removed. An important

contribution 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. Each switch will perform the update at the interval time $[PT-\varepsilon, PT+\varepsilon]$ where ε denotes the clock synchronization accuracy. Two interval times related to two events (e.g. reception of a controller-to-switch message) may overlap and then, this may result to an inconsistent behaviour in the case when the execution order is not trivial to ensure consistency.

Causal update. Recently, the authors of [LSM19] proposed a new consistent update approach based on the suffix causal consistency property. This property implies that an in-fly packet is forwarded to its next hope only based on a forwarding rule that is equally or more up-to-date then rules of upstream switches where it was processed. This allows packets to traverse forwarding path based on a rule set which is related to a specific network policy, referred to as *Epoch* in [LSM19]. For example, if the controller launches the $Epoch_k$ to update from the initial forwarding policy to the final one (see Figure 3.2) and a packet pkt_j reaches the switch s_{inter} through the final path, then pkt_j will be forwarded to its next hope based on the same forwarding policy, i.e. pkt_j will be forwarded to $s_{inter+1}$. This prevents s_{inter} to forward pkt_k back to s_i based on the initial forwarding policy, avoiding pkt_j to enter a forwarding loop between s_i and s_{inter}.

This approach adopts Lamport timestamp to tag packets reflecting the counter numbers related to the forwarding rules that have processed them in the last hope (i.e. switch). When an in-fly packet reach an intermediate switch in the forwarding path, it is processed with a forwarding rule with a timestamp at least equal to the timestamps of rules where the same packet has been processed in upstream switches.

The proposed update algorithm is constituted from four important steps. The first is backward closure. This step is responsible to include the new forwarding rules that precede those already included, propagating the installation of new rules backward along routing paths. For example, this ensures that the installation of the new rule related to s_i (see Figure 3.2) is included in the update once the new rule related to s_{inter} is already installed. This ensures that all packet that pass by s_i will be stamped by a sequence number that will permit them to match the new rule already installed on s_inter, forwarding them based on the final forwarding policy. The second step is forward closure. Similarly to the first step, it is applied to include the new rules that follow those already included. The third step is responsible to tag the set of the rule to be installed on switches by the corresponding timestamps. The last step is send-back rules. This step handles in-fly packets that initially are forwarded from a source switch based on the initial path and then they hit a switch at which the matched rule that would continue to forward them via the initial path is already deleted. To manage this case, the controller install new temporary rules on the corresponding switches to backtrack these packets to their origin. These packets are buffered until the new rules that can forward these packets from the new path are installed [GHH+20]. For example, if a packet pkt_i is forwarded from switch src (see Figure 3.2) based on rules installed of the old configuration, i.e. forwarding packets based on the initial forwarding policy, reaches s_i where it already update its forwarding table by removing the rule that forwards pkt_i to s_{inter}. Then, the controller calculates and installs new temporary rules on s_i and the all upstream switches, i.e. between src and s_i, to backtrack pkt_i to src. Once pkt_i is backtracked to src, it will be buffered until src installs the new rules forwarding pkt_i based on the final forwarding policy. The contribution of this work consists in relaxing the limitation to forwarding in-fly packets during update based either the initial forwarding policy or the final one, ensuring consistent updates. As

described in the last update algorithm step, packets, and if the case requires, may reach their destinations passing by both paths during an update from one forwarding policy to another.

A summary of the discussed consistent update approaches is presented in Table 3.1. In the next sub-section, we present works which focused on optimizing the network update task, guaranteeing consistency updating.

3.2.2 Performance oriented-objective updating

Network updates are costly tasks. Among the reasons behind, updates require many exchanges of messages between the controllers and switches which can overload network resources (e.g. memories and buffers of switches, communication canal bandwidth, etc.) to ensure such tasks. Recently, works like [MW13, ALMS16, FW16, LMS15, FLMS18, FLSW18] proposed studying techniques oriented towards optimizing updates against one/some overhead criteria(s), guaranteeing update consistency. [FSV16] distinguished between three category of performance goals that have been proposed: the link(node)-based, the round-based and the cross-flow objectives.

Link/node-based objective. This class of network update objective aims to make new links/nodes available as soon as possible. In other words, the aim is to maximize the number of switch rules that can be used directly on updating the network, without violating consistency properties. This is of utmost importance for network applications that do not tolerate update time overhead. Authors of [MW13] studied the possibility of updating as many switch as possible at a time in a loop-free manner. Then, in another related work, [ALMS16] proved that node-based optimization problem is NP-hard. In the same context, [FW16] demonstrated that scheduling consistent updates is NP-hard for a sub-linear number of update rounds.

Round-based objective. The second class of network update objective aims to

minimize the total number of rounds, i.e. the number of steps needed to schedule an update, where each step is responsible to install a set of new forwarding rules in the underlying switches that is safe to update simultaneously. In this context, the authors of [LMS15] presented an approach that minimize the number of steps to update networks in a loop-free manner. In [FLMS18], authors defined two loop-free level: the strong loop-freedom and the relaxed loop-free. On the one hand, the strong loop-freedom requires that forwarding rules stored at the switches be loop-free at all points of time. On the other hand, the relaxed loop-freedom only requires that the 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 and it was not solved. However, for the relaxed loop-freedom, in the worst case, $\Omega(n)$ rounds may be required, where n is the number of switch to be updated. On the other hand, $O(log\ n)$ rounds always exist.

Cross-flow objective. A third class of network update objective where it takes in consideration the problem of the presence of multiple flows.

The authors of [DLS16] studied how to update multiple policies simultaneously in a loop-free manner. In this approach, the authors focus on minimizing the number of touches: the number of interactions of the controller with the switches in terms of exchanged messages. The idea is to reduce the number of touches if controllers encapsulate the updates of multiple flows to a given switch into a single message. Then in another work, authors of [ADSW16] proved that the problem is NP-hard for updating to independent policies.

Approaches	**References**	Ensured consistent properties	Approach overview
2-phase commit update	[RFR+12, KRW13, NT17]	Per-packet consistency Connectivity consistency	Packets are forwarded either based on the initial forwarding policy or based on the final one
1-phase commit update	[NT17]	Connectivity consistency	All switches are aware about any performed change on the forwarding path they belong to. Any forwarding rule can be safty removed.
Ordered update	[LRFS14, LwZ+13, FMW16, VC16]	Connectivity consistency	Update are scheduled under different calculated steps where each update step ensures connectivity consistency.
Timed update	[MM16, ME16]	Per-packet consistency Connectivity consistency	Updates are simultaneously executed in the undelying devices
Causal update	[LSM19]	Connectivity consistency	Packets are forwarded based on an entry that is equally or more up-to-date then entries of upstream switches where it was processed.

Table 3.1: Overview of consistent update approaches

3.3 Chapter summary

This chapter presents and describes the state-of-the-art of works oriented to solve the inconsistency update problem, including solutions that were proposed to cover connectivity, congestion and policy update inconsistency problems. The different techniques used to attack the problem was presented via a taxonomy to provide a classification of the different approaches ensuring consistency updates in SDNs.

Chapter 4

The formalisation of the inconsistency connectivity update problem in Software-Defined Networks

In this chapter, we analytically study the problem of inconsistent connectivity update in SDNs. For this purpose, the focus is held on the two update problems: forwarding loops and forwarding blackholes, which disrupt packet delivery during an update from one forwarding policy to another. Informally, on the one hand, a forwarding loop takes place when a data packet is forwarded back from a forwarding device to another from which the data packet was processed. On the other hand, a forwarding blackhole happens when a forwarding device is not able to forward an ingoing packet to its next hop due to the absence of a forwarding rule matching the data packet.

Before formalized the problem of inconsistent connectivity update, an SDN update model is introduced. Then, the problem of forwarding loop and forwarding blackhole is analytically analysed and formulated.

4.1 Network model

The SDN model presented in this section is an extension of the typical system model presented in the Section 2.3 of Chapter 2. The sets P, M, and E, which correspond respectively, to the set of processes, the set of messages and the set of events, are adapted to the SDN context. However, 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, are added and are specific to the SDN network model [GHH+20].

- Processes: An SDN network is composed of a set of processes $P = \{p_1, p_2, \ldots\}$ | $P = P_{cp} \cup P_{rp}$ where $P_{cp} = \{cp\}$ represents the controller process cp and $P_{rp} = \{rp_1, rp_2, \ldots\}$ represents the set of routing processes of OpenFlow switches.

- Matches: A finite set of matches $MATCH = \{match_1, match_2, \ldots\}$ is considered.

  A match is a key attribute for establishing update tasks in SDNs. In fact, each update is performed based on a specific match value. This is due to that an update makes reference to a forwarding path $fpath_l \in FPATH$, a packet flow $pflow_m \in PFLOW$, taking route to its destination according to $fpath_l$, and to any OpenFlow message $m \in M$ disseminated by the controller to update any routing process $rp \in fpath_l$ that shares the same match $match_n \in MATCH$.

- Messages and data packets: An SDN network includes a set of messages and data packets $MP = \{mp_1, mp_2, \ldots\}$ | $MP = M \cup PKT$ where $M = \{m_1, m_2, \ldots\}$ represents the OpenFlow messages and $PKT = \{pkt_1, pkt_2, \ldots\}$ represents the data packets.

  Besides the set of messages M, the set of OpenFlow message types M_{type} where $f_{message} : M \mapsto M_{type}$ is considered [ONF15]. Each message $m \in M$ corresponds to a $match_n \in MATCH$, denoted by $m \triangleq match_n$. Furthermore, a $match_n$ may also correspond to a subset of OpenFlow messages. A message $m \in M$ is denoted 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 $m \in M$.

  As mentioned in Section 2.1 of Chapter 2, controller-to-switch messages are the considered OpenFlow messages in this thesis due to their relevance to the prob-

lem at hand, 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 and properties (see more details in [ONF15]). However, we only consider *match* and *command* as the relevant *FlowMod* message structures. In fact, the structure *match* specifies the packet/packet flow that correspond to an entry, whereas *command* defines the action to be performed on the forwarding rule. In this thesis, we consider to type of commands: add and delete. We distinguish between two subsets: the subset of messages of command type add $M_{add} \subset M$ (to add a new forwarding rule), and the subset of messages of command type delete $M_{delete} \subset M$ (to delete an existing forwarding rule). *FlowMod* is considered as a relevant message, and *command* and *match* represent the "*OpenFlow_message*" structures in the specification of a message $m \in M$.

A packet $pkt \in PKT$ is a tuple $pkt = (rp_i, t_i, header, data, rp_j)$, 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 contains a $match_n \in MATCH$ that corresponds to all *match* of packets belonging to the same packet flow. The data consist of the payload (the message content).

- Packet flow: A finite set of packet flows $PFLOW = \{pflow_1, pflow_2, \ldots\}$ is considered. A $pflow_m \in PFLOW$ (also $pflow_m \subset PKT$) 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 injection $f_{pflow_m} : PFLOW \mapsto MATCH$, that is each $pflow_m$ corresponds to a $match_n \in MATCH$ denoted by $pflow_m \mathrel{\widehat{=}} match_n$ [GHH+20].

- Forwarding paths: A finite set of forwarding paths $FPATH = \{fpath_1, fpath_2, \ldots\}$ is considered. An $fpath_l \in FPATH$ is a subset of routing processes $Rp = \{rp_k, rp_{k+1}, \ldots, rp_{k+n}\}$ 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 injection $f_{fpath_l} : FPATH \mapsto MATCH$, that is each $fpath_l$ corresponds to a $match_n \in MATCH$ denoted by $fpath_l \mathrel{\widehat{=}} match_n$ [GHH+20]. We denote a forwarding path $fpath_l \in FPATH$ before and after an update by $fpath_l^{initial}$ and $fpath_l^{final}$, respectively.

- Events: As shown in the Section 2.3 of Chapter 2, there are two types of events: internal and external ones. In this thesis, the internal events are not relevant. However, we define them for the completeness of the formal specification. The set of finite internal events $E_{internal}$ is the following [GHH+20]:

  - $PrM(rp_i, t_i, m, cp)$ 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 [GHH+20]:

  - $SdM(m)$ denotes that the cp sends a message $m \in M$ to an $rp_j \in P_{rp}$[1].
  - $FwdP(pkt)$: 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:

[1] In some part of this thesis, we specify the receiver of the message: $SdM(m, rp_j)$

– $RecMP(mp)$: 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:

$$E(MP) = \{SdM(m), FwdP(pkt)\} \cup \{RecMP(mp)\} \cup \{DlvMP(mp)\} \tag{4.1}$$

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

$$E = E_{internal} \cup E(MP) \tag{4.2}$$

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, using the happened-before relation ($\rightarrow$) (see Definition 3) where:

$$\hat{E} = \{E, \rightarrow\}. \tag{4.3}$$

4.2 Inconsistent connectivity update in Software-Defined Networks: Problem formulation

In the rest of this chapter, we separately formalize the problems of forwarding loop and forwarding blackhole during updates. Firstly, the forwarding loop phenomenon during updates is formally defined based on i) the physical time and ii) the causal order as

tools to express the conditions under which the phenomenon can occur during updates. Secondly, we present an novel study perspective which allows to formally delimits and define the three pattern defining conditions under which forwarding blackholes occurs during updates.

4.2.1 Transient forwarding loop

4.2.1.1 Transient forwarding loop from temporal perspective

Returning to the example of Figure 1.1 of Chapter 1 and taking the scenario of updating s_i and s_{inter} switches. An in-depth view of this scenario is shown in Figure 4.1.

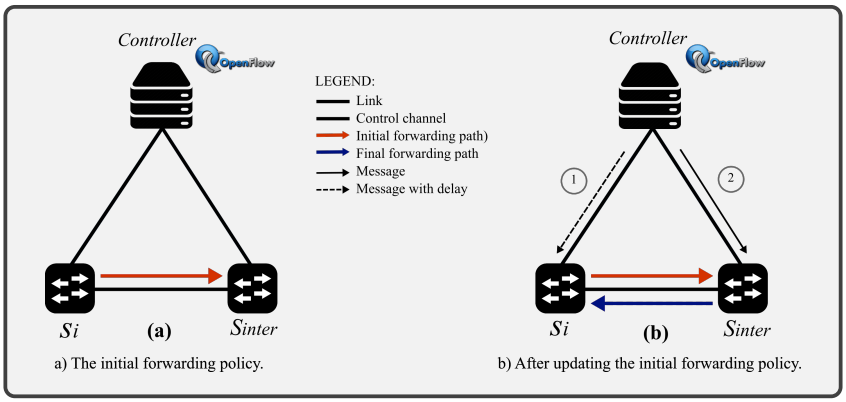

Figure 4.1: Triggering of a transient forwarding loop due to an out-of-order execution of update events [GHH+20].

In this scenario, the topology is composed of an OpenFlow controller and two switches. According to the network model presented in the previous section, the set of processes for this scenario is $P = \{cp, rp_i, rp_{inter}\}$, where cp is the OpenFlow controller and rp_i, rp_{inter} represent the switches S_i and S_{inter}. Also, the initial forwarding path (depicted in red) forwards all matched packet flow from rp_i to rp_{inter} (see Figure 4.1a). During the update, the OpenFlow controller cp sends to rp_i a *Flow-*

Mod message $m_1 = (cp, 1, (match_n, delete), rp_i)$ where a $pflow_m \triangleq match_n$, and then, it sends to rp_{inter} a *FlowMod* message $m_2 = (cp, 2, (match_n, add), rp_{inter})$ where also $pflow_m \triangleq match_n$ (see Figure 4.1b). The routing process rp_{inter} receives m_2 and installs the new rule directing all $pkt_i \in pflow$ to rp_i (see the final forwarding path depicted in blue) . Accordingly, rp_{inter} directs $pkt_1 \in pflow$ to rp_i. Subsequently, pkt_1 enters rp_i before m_1 is delivered to rp_i due to a delay of the reception. Finally, as pkt_1 matches the entry already installed in rp_i, the former ends by redirecting pkt_1 to rp_{inter}, generating a transient forwarding loop between rp_i and rp_{inter}.

On analysing the communication diagram corresponding to the execution diagram of Figure 4.2 [GHH+20], it can be observed that the transient forwarding loop between rp_i and rp_{inter} is created due to that 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_i. We note that we have observed the same cause of the triggering of transient forwarding loops in others example with more complicated network topologies. We formally define the Transient forwarding loop pattern from a temporal perspective [GHH+20].

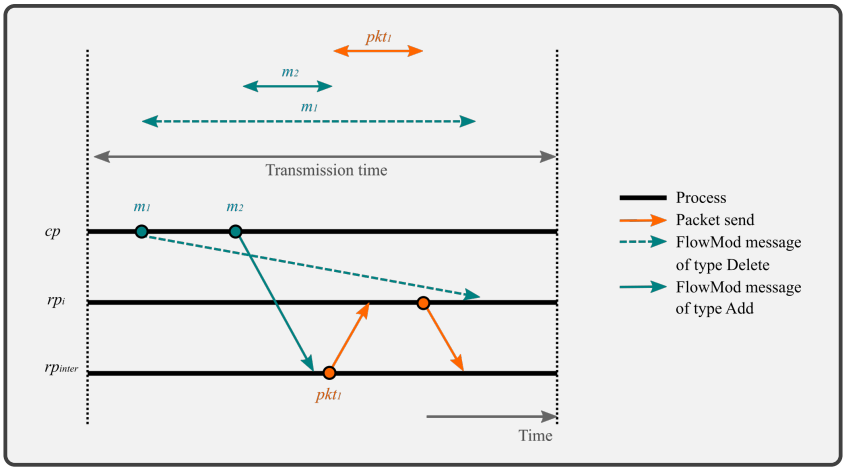

Figure 4.2: The communication diagram related to Figure 4.1 [GHH+20].

Definition 7 *Let be a* $match_n \in MATCH$ *and two OpenFlow messages* m *and* m'*, where* $m = (cp, t_{cp}, (match_n, delete), rp_i)$ *and* $m' = (cp, t'_{cp}, (match_n, add), rp_j)$*, such that (i)* $time(m) < time(m')$*, (ii)* $(rp_i \neq rp_j)$ *and (iii)* $\exists$ *a forwarding path* $fpath_l = \{rp_i, \ldots, rp_j\}$ *from* rp_i *to* rp_j*, such that intermediates* rp_r *may exist where* $fpath_l \in FPATH \mid fpath_l \mathrel{\widehat{=}} match_n$*. A Transient Forwarding Loop pattern from* rp_i *to* rp_j *exists iff there is a data packet flow* $pflow_m = pkt_1, pkt_2, ..., pkt_n$ $(n \geq 1)$*, where* $pflow_m \in PFLOW \mid pflow_m \mathrel{\widehat{=}} match_n$*, such that:*

$$T(m) > T(m') + \sum_{k=1}^{n} T(pkt_k) \tag{4.4}$$

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.

4.2.1.2 Transient forwarding loop from causal perspective

The same phenomenon is analysed during SDN updates by using the scheme of the happened-before relation (See Definition 3 of Chapter 2). Figure 4.3 shows the generic scenario in which an out-of-order execution of messages/packets leads to a transient forwarding loops during updates.

In this scenario, the topology is composed of an OpenFlow controller cp and n OpenFlow switches. The set of processes is $P = \{cp, rp_{src}, rp_1, rp_2, \ldots, rp_n, rp_{dst}\}$ where cp is the OpenFlow controller, rp_{src} and rp_{dst} represent, respectively, the *Source* and the *Destination* switches, and rp_1, rp_2, $\ldots, rp_n$ represent the intermediate switches $S_1, S_2, \ldots, S_n$. Initially, each intermediate routing process rp_i, except rp_n, contains an forwarding rule r directing a $pflow_m \in PFLOW$ to its rp_{i+1} (see the solid lines in Figure 4.3a). The network policy update consists to redirect $pflow$ from rp_n to rp_1

(see the dashed lines in Figure 4.3a). To establish the update, the cp instructs each rp_i, except rp_n, to update their forwarding table by deleting the rules directing $pflow$ to its rp_{i+1} and then instructs again each rp_i, except rp_1, to install a rule directing $pflow_m$ to its rp_{n-1}. For brevity, Figure 4.3b illustrates only one message (Message (1) depicted in the dashed line) to delete the rule from rp_1 and all the other messages (from (2) to (n)) for adding the new forwarding path from rp_n to rp_1.

To characterize the generic scenario, we based on its corresponding communication diagram 4.4. As illustrated, the cp starts by sending message $m_1 = (cp, 1, (match_n, delete), rp_1)$ to rp_1, deleting the initial rule of rp_1. Then, it sends the message $m_2 = (cp, 2, (match_n, add), rp_2)$, inserting a new rule into rp_n to consider forwarding the matched packet based on the new forwarding policy. The rest of the OpenFlow messages from m_i to m_j (represented in Figure 4.3b by Messages (3), ..., (n)) are sent to their corresponding rp to add the entries directing $pflow$ from rp_{n-1} to rp_1.

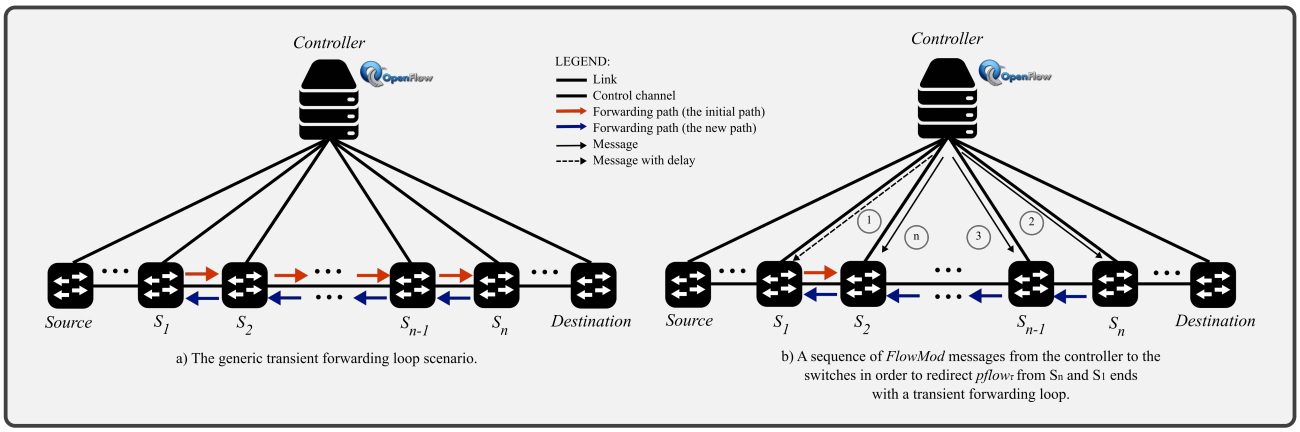

Figure 4.3: Transient forwarding loop: generic scenario [GHH+20].

Due to the asynchronous communication between the controller and all the underlying switches, it can be noted that rp_n, rp_{n-1}, ... and rp_2 receive their messages and therefore install their new rules before rp_1 besides that m_1 was sent to rp_1 before them. The installation of the new rules coincides with the arrival of a packet flow $pkt_i \in pflow$

that hits rp_n and corresponds to the new installed rule. Thus rp_n directs it to rp_{n-1}, then rp_{n-2} until reaching rp_1. pkt_n enters rp_1 before m_1 is delivered to rp_1 (see Figure 4.4). Consequently, pkt_n matches the forwarding 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 4.3b), which generates a transient forwarding rule.

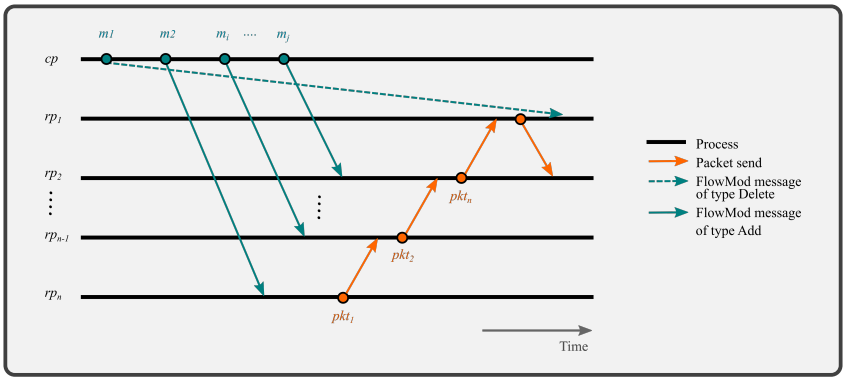

Figure 4.4: The communication diagram corresponding to Figure 4.3 [GHH+20].

We define below an abstraction of the transient forwarding loop in SDN, as a specification of Lamport's happened-before relation to express the phenomenon from a causal perspective [GHH+20].

Definition 8 *Let be a* $match_n \in MATCH$ *and two FlowMod messages* $m, m' \in M$ *where* $m = (cp, t_{cp}, (match_n, delete), rp_i)$ *and* $m' = (cp, t'_{cp}, (match_n, add), rp_j)$, *such that (i)* $m \rightarrow m'$, *(ii)* $(rp_i \neq rp_j)$, *and (iii)* $\exists$ *a forwarding path* $fpath_l = \{rp_i, \ldots, rp_j\}$ *from* rp_i *to* rp_j, *such that intermediates* rp_r *may exist where* $fpath_l \in FPATH \mid fpath_l \triangleq match_n$. *A Transient Forwarding Loop from* rp_i *to* rp_j *exists iff there is a data packet flow* $pflow_m = pkt_1, pkt_2, ..., pkt_n$ $(n \geq 1)$, *where* $pflow_m \in PFLOW \mid pflow_m \triangleq match_n$, *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.*

The Definition 8 specifies the transient forwarding loop *pattern*, representing an abstraction and a generalization of the phenomenon in question. The abstraction lies in highlighting the relevant preconditions and events that get in on the act and in ignoring the irrelevant events. On the other hand, the generalization consists of specifying the three conditions catching the phenomenon in its entirety, no matter how complex the network architecture is or how much forwarding devices are involved in the update.

4.2.2 Forwarding blackhole

In SDNs, a packet flow is identified by its match value, serving as a filter criterion to distinguish it from the other packet flows in the network. Indeed, each packet flow can be controller independently from any other packet flow. Hence, all packets of a specific flow receive identical service policies in the forwarding devices basing on their match value. If we want to find out whether an in-fly packet flow can enter in a forwarding blackhole switch, one analysis perspective is to verify the forwarding tables of the forwarding devices, which receive service policies related to the same packet flow, *with respect to switches belonging to the initial and the final forwarding paths* [GHH+19]. With this in mind, we introduce our analysis perspective of the forwarding blackhole phenomenon which is based on a per-node categorisation during updates.

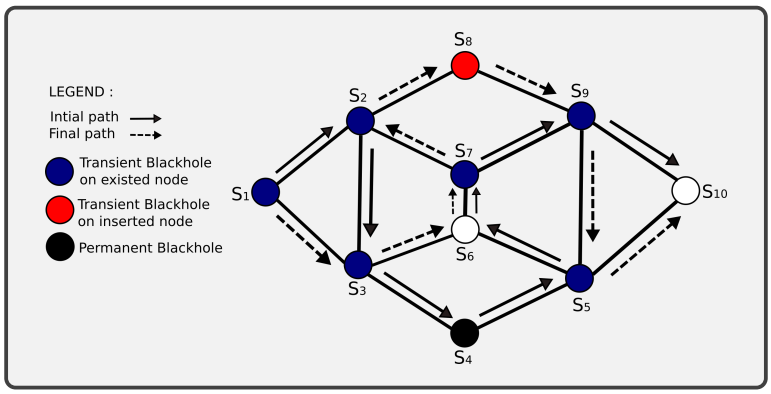

Figure 4.5: A per-node categorisation of forwarding blackhole occurrences during SDN updates [GHH+19].

In Figure 4.5, a routing updating scenario is depicted. The controller (not depicted for brevity) has to update the forwarding path of a specific flow from a source switch S_1 to a destination switch S_{10}. Two routing polices are presented. These forwarding policies are illustrated with two forwarding paths, the initial forwarding path (solid lines) and the final forwarding path (dashed lines). Based on the mentioned analysis perspective, we first start by categorizing switches (nodes) while updating the forwarding policy regarding the flow forwarding path which they belongs. Thus, and according to the forwarding path node compositions, there are only three cases of interest:

- Nodes which belong to both the initial and final forwarding path (the blue nodes),
- Nodes which belong only to the final forwarding path (the red node), and
- Nodes which belong only to the initial forwarding path (the black node).

We highlight that the white nodes are not considered, as these nodes do not receive any update and belong to the initial and the final forwarding path. This classification of nodes orients us to study the phenomenon from an abstraction allowing to delimit

and identify, for each node category, the conditions under which the phenomenon may occur.

In the rest of this section, we provide a case-by-case study of the forwarding blackhole phenomenon.

4.2.2.1 Forwarding Blackhole

In this subsection, we formalize the first forwarding blackhole pattern: *transient* forwarding blackhole. With this nomenclature, we are referring to switches which can be blackholes for some in-route packet flows *only* during the establishment of updates. Once the update is finished, the switch is no longer a blackhole. However, this can perturb the connectivity as packets may delay in reaching their destination, or even worst they can be lost. We distinguish two cases in which a transient forwarding blackhole may take place.

Transient forwarding blackhole on Existing Node. Transient forwarding blackhole should be considered on Existing Nodes (ENos) during updates. Nodes which belong to both the initial and the final forwarding paths. The blue nodes in Fig.4.5 depict these nodes. A transient forwarding blackhole occurs on a ENo when the controller instructs it by means of two messages; the first message m to add a new rule forwarding the matched packets to another node into the forwarding path, and a second message to delete the rule forwarding packets to the a node in the forwarding path where the order of execution of the message is not respected. Taking as example of the node S_2 of Figure 4.5. The communication diagram of message-exchange between the controller (represented as the controller process cp) and S_2 (represented as the routing process rp_i), and the interleaving of messages with in-fly packets $pkt_1, pkt_2, \ldots, pkt_k$ during the update is shown in Figure 4.6.

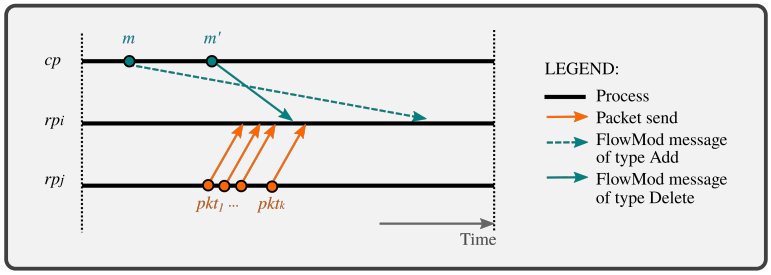

Figure 4.6: Communication diagram related to the triggering of a transient forwarding blackhole on a $rp_i \in fpath_l^{initial} \cap fpath_l^{final}$ [GHH+19].

In this case, $m' = (cp, 2, (match, delete), rp_i)$ is delivered to rp_i before $m = (cp, 2, (match, add), rp_i)$ despite the fact that the later was sent before. This is indeed due to the asynchronous communication between both. Therefore, rp_i deletes the old rule that forward the matched packets to their next hop before it adds the new rule. As a result, rp_i is converted to *transient* blackhole node for all incoming packets corresponding to the matched flow. We can observe that a sequence of packets emitted from S_1 (represented as rp_j) enters rp_i while it is receiving update messages. In this case, only the packet pkt_k enters into a transient forwarding blackhole as it reaches rp_i after the delivery of m' and before the delivery of m [GHH+19]. However, this is not the case for the others packets $pkt_1, pkt_2, \ldots, pkt_{k-1}$ as they reached rp_i before the delivery of m'. Thus, rp_i remains a transient forwarding blackhole for pkt_k until the delivery of m takes place. We formally define the transient forwarding blackhole pattern occurring on ENos during an update as follows:

Definition 9 *Let us consider a* $match_n \in MATCH$ *and two FlowMod messages* $m, m' \in M$ *where* $m = (cp, t_{cp}, (match_n, add), rp_b)$ *and* $m' = (cp, t'_{cp}, (match_n, delete), rp_b)$ *such that i)* $send(m) \rightarrow send(m')$*, ii)* $\exists$ *a forwarding path* $fpath_l^{initial} \in FPATH \mid fpath_l^{initial} \widehat{=} match_n$ *from a source* rp_i *to a destination* rp_j*, iii)* $rp_b \in fpath_l^{initial}$ *and iv)* $rp_b \neq rp_j$*.* **A transient forwarding blackhole on an Existing node** *occurs on* rp_b *if there is a data packet flow* $pflow_m =$

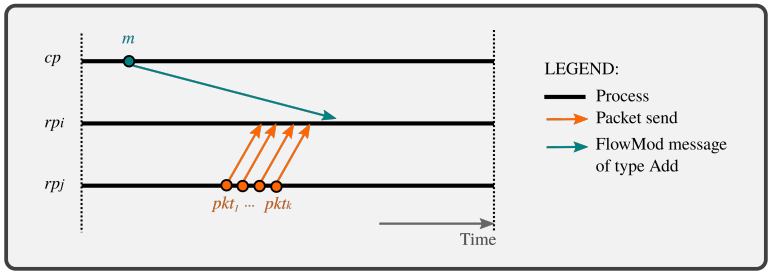

Figure 4.7: Communication diagram related to the triggering of a transient forwarding blackhole on a $rp_i \in fpath_l^{final}$ [GHH+19].

$pkt_1, pkt_2, \ldots, pkt_n \ (n \geq 1)$ *where* $pflow_m \in PFLOW \mid pflow_m \mathrel{\widehat{=}} match_n$, *and* $pkt_k \in pflow_m \mid pkt_k = (rp_l, t_{rpl}, (header(match_n), data), rp_b)$ *such that*

1. m' *is delivered before* $pkt_k \ (1 \leq k < n)$ *by* rp_b
2. m *is delivered after* $pkt_k \ (1 \leq k < n)$ *by* rp_b
3. $rp_b \in fpath_l^{initial} \cap fpath_l^{final} \mid fpath_l^{final} \mathrel{\widehat{=}} match_n$

Transient forwarding blackhole on Inserted Node. The second case in which transient forwarding blackholes occur is on Inserted Nodes (INos). Nodes which only belong to the final forwarding path. The red node depicted in Figure 4.5 represents this category of nodes. In this case, the controller considers adding a new node to the final forwarding path. To this end, it sends a message instructing a switch, that belongs to the forwarding path, to add a new rule forwarding the matched packets to the new INo. We take the case of S_8 of Figure 4.5 where the controller cp sends a message m instructing S_8 to install a rule forwarding all the matched packets to S_9. The example is illustrated based on the communication diagram in Figure 4.7. In this scenario, S_2 (represented as rp_j) has already updated its forwarding tables, then it forwards a matched sequence of packets $pkt_1, \ldots, pkt_k$ to S_8 (represented as rp_i). This sequence of packets enters S_8 before it has received the message m. Therefore, S_8 is

not able to forward the packets to its next hope, qualified as a transient forwarding blackhole for packet of the matched flow. Once S_8 receives m, then it is no longer a transient forwarding blackhole node. We formally define the transient forwarding blackhole pattern on INos as follows.

Definition 10 *Let us consider a* $match_n \in MATCH$ *and a FlowMod message* $m \in M$ *where* $m = (cp, t_{cp}, (match_n, add), rp_b)$ *such that i)* $\exists$ *a forwarding path* $fpath_l^{initial} \in FPATH \mid fpath_l^{initial} \widehat{=} match_n$ *from a source* rp_i *to a destination* rp_j*, and ii)* $rp_b \notin fpath_l^{initial}$. **A transient forwarding blackhole on an Inserted node** *occurs on* rp_b *if there is a data packet flow* $pflow_m = pkt_1, pkt_2, \ldots, pkt_n$ $(n \geq 1)$ *where* $pflow_m \in PFLOW \mid pflow_m \widehat{=} match_n$*, and* $pkt_k \in pflow_m \mid pkt_k = (rp_l, t_{rpl}, (header(match_n), data), rp_b)$ *with* $send(m) \parallel send(pkt_k)$ $(1 \leq k \leq n)$*, such that*

1. pkt_k $(1 \leq k < n)$ *is delivered before* m *by* rp_b*, and*

2. $rp_b \in fpath_l^{final} \mid fpath_l^{final} \widehat{=} match_n$

4.2.2.2 Permanent forwarding blackhole

The third forwarding blackhole pattern is qualified as *permanent*. This pattern of forwarding blackhole occur on Forgotten Nodes (FNos). Nodes which belong only to the initial forwarding path. Unlike the *transient* forwarding blackhole, and as its name said, if a packet enters a permanent forwarding blackhole switch, this packet will lost and will never reach its destination. The black node depicted in Figure 4.5 represents this category of nodes. To eliminate a node from belonging to a forwarding path, the controller has to send a message instructing the transmitter switch to delete the rule that forward the matched packet to this node.

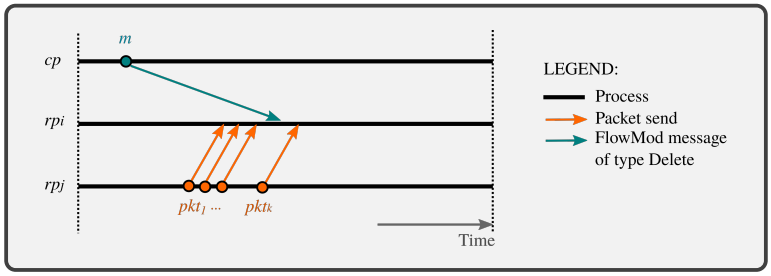

Figure 4.8: Communication diagram related to the triggering of a permanent forwarding blackhole on a $rp_i \in fpath_l^{initial}$ [GHH+19].

Taking the case of the forwarding path between the nodes S_4 and S_5. To delete it, the controller has to send a message m instructing S_4 to delete the rule forwarding the matched packets to S_5. Figure 4.8 shows the corresponding communication diagram. In this scenario, S_3 (represented as rp_j) forwards a sequence of packets $pkt_1, pkt_2, \ldots, pkt_k$ to S_4 (represented as rp_i) while the controller (represented as cp) sends m to S_4. Then, pkt_k enters S_4 after it deletes the rule forwarding all the matched sequence of packets to the next hope. In this case, S_4 will never be able to forward pkt_k to its next hope, and then S_4 is a permanent FB for pkt_k [GHH+19]. We formally define the permanent FB pattern that may occur during SDNs updates.

Definition 11 *Let us consider a* $match_n \in MATCH$ *and a FlowMod message* $m \in M$ *where* $m = (cp, t_{cp}, (match_n, delete), rp_b)$ *such that i)* $\exists$ *a forwarding path* $fpath_l^{initial} \in FPATH \mid fpath_l^{initial} \widehat{=} match_n$ *from a source* rp_i *to a destination* rp_j*, and ii)* $rp_b \in fpath_l^{initial}$. **A permanent forwarding blackhole on a Forgotten node** *occurs on* rp_b *if there is a data packet flow* $pflow_m = pkt_1, pkt_2, \ldots, pkt_n$ $(n \geq 1)$ *where* $pflow_m \in PFLOW \mid pflow_m \widehat{=} match_n$*, and* $pkt_k \in pflow_m \mid pkt_k = (rp_l, t_{rpl}, (header(match_n), data), rp_b)$ *with* $send(m) \parallel send(pkt_k)$ $(1 \leq k \leq n)$*, such that*

1. m *is received before* pkt_k $(1 \leq k < n)$ *by* rp_b*, and*

2. $rp_b \notin fpath_l^{final} \mid fpath_l^{final} \widehat{=} match_n$

As to the forwarding blackhole phenomenon, a per-node categorisation is proposed to study the problem. For each node category, preconditions, relevant events and conditions under which a blackhole occurs during SDN updates were identified and formally specified by the Definitions 9, 10 and 11. The first two definitions capture the occurrences of *transient* forwarding blackholes while the last definition catches the occurrences of *permanent* forwarding blackholes. This is regardless of the network architecture and the forwarding devices involved in the update. We should highlight that the introduced perspective study based on the per-node categorisation is not oriented towards designing a forwarding blackhole-free solution; however, this categorisation was only used for an analytical purpose of the problem at hand.

In this chapter, an SDN update model was introduced. Based on this model, the two connectivity invariant violations: forwarding loop and forwarding blackhole, were analytically studied. Our results consist of defining the patterns of network events under which each phenomenon may occur during SDN updates, allowing to understand the root causes behind the occurrence of these phenomenons. This result will be the key to introduce, in the next chapter, a new consistent connectivity update approach.

Chapter 5

Causal-based consistent connectivity update approach

In the previous Chapter, a mathematical model of the inconsistent connectivity update problem is introduced. In this model, two main phenomena giving rise to inconsistent connectivity updates are defined: the forwarding loop and the forwarding blackhole. Based on these results, new causal update policies are introduced in this chapter, guaranteeing consistent connectivity updates. Also, an algorithm formalizing the proposed update policies is presented. Finally, proofs that the algorithm is transient forwarding loop-free and transient forwarding blackhole-free are presented.

5.1 Transient forwarding loop-free update policy

An abstraction of the transient forwarding loop problem during update is introduced in section 4.2.1 of Chapter 4. By abstraction, we mean that any transient forwarding loop occurrence during updates can be identified. A first transient forwarding loop abstraction is specified by Definition 7. This Definition identifies the pattern under which the phenomena occurs in function of transmission time of update OpenFlow messages and in-fly packets (see equation 4.4). Solving the problem at hand based on the last mentioned requires to break the inequality of the Formula 4.4, i.e. ensuring that the transmission time of a delete message m is less or equal than the transmission time of an add message m' and the sum of the transmission time of all matched packets pkt_k where $k = \{1, 2, ..., n\}$. Thus, one update policy is to synchronize times at which relevant update events (see Definition 7) occur in order to simultaneously perform them. However,

such a solution is not effective since it is quite difficult to perfectly synchronize clocks across network entities. In fact, any clock synchronization mechanism (e.g. Network Time Protocol [Mil91]) presents a clock synchronization accuracy ε. As a result, each switch will perform the update at the interval time $[T-\varepsilon, T+\varepsilon]$. In this case, we will not be able to find out or to force the execution order of any pair of update events, harming the update consistency [GHH+20].

On the other hand, Definition 8 introduces an abstraction of the transient forwarding loop phenomena during updates as a specification of Lamport's happened-before relation (see Definition 3 of Chapter 2). The three conditions specified in Definition 8 covers and identifies any transient forwarding loop during updates from a causal perspective. On analysing these conditions, these conditions specify an event-based chain leading to transient forwarding loops between any two or more routing processes $P_{rp} = \{rp_i, rp_j, ...\}$ (i.e. forwarding devices). In fact, the first condition represents the initialisation of this event-based chain where a packet flow $pflow_r = \{pkt_1, pkt_2, ..., pkt_n\}$ starts taking the route to a first $rp_i \in P_{rp}$ due to the installation of a new rule matching $pflow_r$ on the delivery of a message $m' \in M$. Then, pkt_k will be delivered to a rp_r (first case: described by the second condition) or to rp_i (second case: described by the third condition). The first case is when $rp_r \neq rp_i$ where rp_r represents an intermediate rp in the forwarding path and then it forwards pkt_{k+1} to its next hop. This represents the generalisation of the transient forwarding loop phenomena, i.e. the case where $pflow_r$ holds into a transient forwarding loop between more than two forwarding devices. The second case is when $rp_r = rp_i$. In this case, rp_i is the triggering rp where the triggered event is when rp_i delivers pkt_k before having delivered a message $m \in M$ that deletes the old rule, forwarding pkt_k back to the rp from which pkt_k reaches rp_i. This is referred to as a causal violation since the delivery of pkt_k occurs before the delivery of the message m where the send of pkt_k (see the

first and the second conditions) causally depends from m (transitively as m' causally depends from m).

Policy. The proposal is to ensure that all messages of command type delete $M_{delete} = \{m_{del_1}, m_{del_2}, ..., m_{del_n}\} \mid M_{delete} \subset M$ are delivered before any incoming packet flow $pflow_m = \{pkt_1, pkt_2, ...pkt_n\} \mid pflow_m \in PFLOW$ where $pflow_m \wedge M_{delete} \hat{=} match_n$. To do so, the idea is to **i)** establish a causal order between M_{delete} and $pflow$, denoted by $Dependency_1 = \forall(m_{del} \in M_{delete}) \wedge \forall(pkt \in pflow) : m_{del} \rightarrow pkt$. The forwarding of any packet $pkt \in pflow$ causally depends on the sending of a message m_{add} of type command add where $m_{add} \in M_{add} \mid M_{add} \subset M$. Thus, this is an explicit causal dependency denoted by $Dependency_2 = \forall(m_{add} \in M_{add}) \wedge \forall(pkt \in pflow) : m_{add} \rightarrow pkt$. Another causal dependency $Dependency_3$ to be established is between the set of messages M_{delete} and the set of messages m_{add}, denoted by $Dependency_3 = \forall(m_{del} \in M_{delete}) \wedge \forall(m_{add} \in M_{add}) : m_{del} \rightarrow m_{add}$. Having $Dependency_2$ and establishing $Dependency_3$, the causal dependency $dependency_1$ is transitively established as any message $m_{del} \in M_{delete}$ causally precedes any message $m_{add} \in M_{add}$, and any message $m_{add} \in M_{add}$ causally precedes any packet $pkt \in pflow$. Therefore, any message $m_{del} \in M_{delete}$ causally precedes any $pkt \in pflow$. $Dependency_3$ is considered as a first precondition for the proposed update mechanism. A second part in this update policy is **ii)** to ensure the causal order delivery of the established order of i) (see details in section 5.3.2).

Property 1: Considering the first Precondition and ensuring a causal order delivery, transient forwarding loop-free is guaranteed when updating from one update forwarding policy to another.

We continue with studying update policies, ensuring forwarding blackhole-free.

5.2 Forwarding blackhole-free update policies

In this subsection, network update policies to preventively guarantee forwarding blackhole-free updates are introduced. In a first part, we focus on studying update policies to avoid forwarding blackholes qualified as *transient* ones. In a second part, a network update policy to prevent the triggering of permanent forwarding blackholes during update is introduced. Similarly to the previous section, the study of the update policies is based on the formalization of the forwarding blackhole phenomena provided in Section 4.2.2 of Chapter 4. This is based on the Definitions 9 and 10 formalizing the patterns under which transient forwarding blackholes occur, and Definition 11 specifying how a forwarding device blackholed packet flows permanently.

We should recall that the formalization of the forwarding blackhole phenomena is based on the categorization of forwarding devices. A forwarding device is classified in function of the forwarding path that belongs to when updating from one forwarding policy to another. This results in delimiting the conditions under which a blackhole occurs based on the following classification: forwarding devices that belongs to both initial and final forwarding paths (qualified as an existing node), to only the final forwarding path (qualified as an inserted node), or only to the initial forwarding path (qualified as a forgotten node).

5.2.1 Transient forwarding blackhole-free update policy

A. Transient forwarding blackhole-free update policy on Existing Nodes. Based on Definition 9, a transient forwarding blackhole on an Existing Node (ENo) occurs when an ENo, denoted by rp_b, receives from the controller, firstly, a message $m_{del} \in M_{del} \mid M_{del} \subset M$ instructing rp_b to remove a rule forwarding a packet flow $pflow_m = \{pkt_1, pkt_2, ...\} \mid pflow_m \in PFLOW$ to its next hop, and secondly, a message

$m_{add} \in M_{add} \mid M_{add} \subset M$ instructing rp_b to add a new rule forwarding $pflow$ to its new next hop.

Policy. As an update policy for the ENos, a message $m_{add} \in M_{add}$ adding a new forwarding rule and matching a packet $pkt \in pflow$ should be sent to rp_b before a message m_{del} M_{delete} deleting a forwarding rule and matching pkt where $m_{add} \wedge m_{del} \wedge pkt \triangleq match_n$. This is considered as a second Precondition for the proposed update mechanism introduced posteriorly in Section 5.3.2. However, this indeed will not be sufficient to ensure that rp_b delivers m_{add} before m_{del} as the communication between the controller and rp_b is asynchronous, and thus, messages can be delivered in any order [GHH+19].

Property 2: Considering the second Precondition and ensuring a FIFO ordering, transient forwarding blackhole-free on an ENo is guaranteed.

B. Transient FBs-free Update policy on Inserted Nodes Following Definition 10, a transient forwarding blackhole on an Inserted Node (INo), denoted by rp_b, occurs, on the one hand, when its direct predecessor rp_{b-1} installs a rule forwarding a packet flow $pflow = \{pkt_1, pkt_2, ..., pkt_n\}$ to rp_b. On the other hand, when a packet $pkt \in pflow$ reaches rp_b where it has not yet delivered a message $m_{add} \in M_{add} \mid M_{add} \subset M$, installing a new rule forwarding pkt to its next hope. Then, rp_b is a transient forwarding blackhole for pkt [GHH+19].

Summary. A first part of this update policy consists on **i)** establishing causal dependencies between all messages $M_{add} = \{m_{add_1}, m_{add_2}, \ldots, m_{add_n}\}$ sent by the controller to all INos and the forwarding of any packet flow $pflow_m = \{pkt_1, pkt_2, ..., pkt_n\} \mid pflow_m \in PFLOW$ to INos coming from their respective predecessors where $pflow_m \wedge M_{add} \triangleq match_n$. A second part of the update policy is **ii)** to ensure that the delivery of M_{add} and $pflow$ respect the causal dependencies established in **i)**.

Policy details. To ensure transient forwarding blackhole-free on INos, the delivery

of all messages M_{add} adding new rules to the forwarding tables of INos should precede the delivery of $pflow$ where $pflow_m \wedge M_{add} \mathrel{\widehat{=}} match_n$. However, controlling the delivery order of a $m_{add} \in M_{add}$ and $pkt \in pflow$ is not possible as both are concurrent events (see Definition 4 in Chapter 2). Following this observation, **i)** we propose to create *causal dependencies* between the sent events of M_{add} to INOs and the forward of $pflow$ matching the set of messages M_{add}, that is, to establish a causal relationship between these events ensuring that a packet $pkt \in pflow$ is delivered by an INo, denoted by rp_{ino}, if and only if the corresponding $m_{add} \in M_{add}$, forwarding pkt to its next hope, is delivered to rp_{ino}. We denote this causal dependency $Dependency_1 = \forall(m_{add} \in M_{add}) \wedge \forall(pkt \in pflow) : SdM(m_{add}, rp_{ino}) \rightarrow FwdP(pkt, rp_{ino})$. To create $Dependency_1$, we, firstly, based on explicit causal dependencies between M_{add} sent to any direct predecessor of INos, denoted by rp_{pino} and the forward of any packet $pkt \in pflow$ to an INo, denoted by $Dependency_2 = \forall(m_{add} \in M_{add}) \wedge \forall(pkt \in pflow) : SdM(m_{add}, rp_{pino}) \rightarrow FwdP(pkt, rp_{ino})$. This is considered as explicit causal dependencies as the triggering of the forwarding of a packet pkt depends directly on the send event of the corresponding message m_{add}, adding a new rule to the predecessor node of an INo and enabling it to forward pkt to an INo. To establish $Dependency_1$, it remains to create causal dependencies between any $m_{add} \in M_{add}$, adding new forwarding rules to any INos, denoted by rp_{ino}, and any $m'_{add} \in M_{add}$, adding new forwarding rules to any direct predecessors of INos, denoted by rp_{pino}. This dependency is expressed as follows: $Dependency_3 = \forall(m_{add} \in M_{add}) : SdM(m_{add}, rp_{ino}) \rightarrow SdM(m'_{add}, rp_{pino})$. Hence, having $Dependency_2$ and establishing $Dependency_3$, $Dependency_1$ is transitively set up [GHH+19]. $Dependency_3$ is considered a precondition for the proposed update mechanism (see Algorithm 1). The second part in this update policy is **ii)** to ensure the causal order delivery of the established order of i) (see details in section 5.3.2).

Property 3: Considering the Precondition 3 and ensuring a causal order delivery,

transient FBs-free on Inserted Nodes is guaranteed.

5.2.2 Permanent forwarding blackhole-free update policy on forgotten nodes

A permanent forwarding blackhole occurs, based on Definition 11, when a Forgotten Node (FNo), denoted by rp_b, delivers an update message $m_{del} \in M_{delete} \mid M_{delete} \subset M$ instructing it to delete a forwarding rule before the delivery of an incoming packet flow $pflow = \{pkt_1, pkt_2, ..., pkt_n\}$ that matches the removed rule. Thus, $pflow$ is permanently blackholed by rp_b.

Policy. To break any occurrence of the permanent forwarding blackhole pattern, an update policy should ensure that the delivery of the subset of all messages M_{delete} to all FNos occurs after the delivery of the last packet $pkt_{last} \in pflow$ to each FNo. However, controlling the order of the delivery of these events is not possible as the send events of the subset M_{delete} to FNos and the forwarding events of $pkt_{last} \in pflow_r$ are concurrent events. Unlike the previous policy for blackholes on INos, it is not possible to establish causal dependencies between these events during the update as none of them are related to other relevant events which can be causally dependent on them [GHH+19].

We propose a tag/match update policy for permanent forwarding blackholes [GHH+19]. **Tag.** The tag operation consists in having the controller send a message instructing the *Decisive Node* (DNo) (see Definition 12), denoted by rp_b, to tag its final packet pkt_n in the buffer with *"final packet"* and to clear out its buffer. This operation should occur before the controller sends to rp_b the delete update message instructing to remove the rule forwarding the matched packet flow via the initial forwarding path. In fact, this can be guaranteed by applying a FIFO order. This ensures that the last packet forwarded via the initial forwarding policy is the tagged packet pkt_{last}. In this

way, we can control the last packet routed following the initial forwarding policy.

Definition 12 *A Decisive Node S_i (represented as $rp_i \in P_{rp}$) is the first Existing Node that switches the routing of the subset of packet flows $pf = pflow_1, pflow_2, \ldots, pflow_n \mid pf \widehat{=} match_n$ from the initial forwarding path $fpath_l^{initial} \in FPATH$ to the final forwarding path $fpath_l^{final} \in FPATH$ where $rp_i \in [fpath_l^{initial} \cap fpath_l^{final}] \widehat{=} match_n$.*

Match. The second part of the update policy is the operation. It consists to deliver all delete update messages M_{delete} by all FNos conditioned by the matching of the final tagged packet pkt_{last}. Indeed, in addition to the match structure, a message $m \in M_{delete}$ must correspond to the tagged packet to be subsequently delivered by rp_b. Hence, the delivery of the last packet pkt_{last} forwarded based on the initial forwarding policy will occur before the delivery of m by rp_b.

In the remainder of this Chapter, an algorithm formalizing the proposed update polices supporting consistent connectivity updates is presented. Then, we formally prove that the presented algorithm is transient forwarding loop-free and transient forwarding blackhole-free in the causal dependency sense.

5.3 Update mechanism free from transient connectivity inconsistencies

The update mechanism presented in this section is inspired from the algorithm presented in [PRS97]. Authors of [PRS97] presented a generic algorithm ensuring the causal order delivery of messages in a Multicast communication environment. We extend this algorithm to guarantee transient connectivity inconsistencies-free updates based on the policies introduced in the previous section [GHH+20].

5.3.1 Algorithm overview

The update mechanism is distributed over the set of processes P, i.e., the controller process cp and the set of routing processes $P_{rp} = \{rp_1, rp_2, ..., rp_n\}$. Throughout an update, a subset of messages and data packets is exchanged. The update mechanism can be summarized as follows [GHH+19]:

- Input: It consists of the list of update OpenFlow messages and ingoing data packets. As described in Section 4.1, updates are *match*-based performed, i.e., update messages and in-fly data packets are grouped and processed by match. This allows to separately process messages and data packets of different match values without worrying about the ordering in which they are executed.

- Preconditions: Let $match_n \in MATCH$ be a match that corresponds to a subset $MP' \subset MP$ where $(M_{add} \cup M_{delete}) \subseteq MP'$. Based on the defined update policies introduced for transient forwarding loops, transient forwarding blackholes on Existed Nodes (ENos) and transient forwarding blackholes on Inserted Nodes (INos), the list of the update preconditions, respectively, is:

- $\forall(m_{del} \in M_{delete}) \wedge \forall(m_{add} \in M_{add}) : SdM(m_{del}, rp_i) \rightarrow SdM(m_{add}, rp_j) \mid rp_i \neq rp_j$
- $\forall(m_{add} \in M_{add}) \wedge \forall(m_{del} \in M_{delete}) : SdM(m_{add}, rp_i) \rightarrow SdM(m_{del}, rp_j) \mid rp_i = rp_j$
- $\forall(m_{add} \in M_{add}) : SdM(m_{add}, rp_i) \rightarrow SdM(m'_{add}, rp_j) \mid rp_i = INo \wedge rp_j = ENo$

- Execution model: The set of messages and data packets MP is asynchronously exchanged between the set of process P: Clocks of all processes P are not in synchronization with each other. No upper bound on messages/data packets transmission delay is required. Also, no acknowledgement message from P_{rp} is required on the delivery of a message $m \in MP$. Therefore, the execution model is fully asynchronous.

- Data structures: Each process $p \in P$ contains a vector of control information CI_i to store direct dependency information between each subset $MP' \in MP$ with respect to a corresponding $match_n \in MATCH$.

  Also, each $rp \in P_{rp}$ maintains a matrix $Delivery_i$ to trace dependency information when an rp receives an $mp \in MP'$.

- Functionally: The mechanism is focused on i) establishing the sent order of a set of OpenFlow messages M to their appropriates P_{rp}, basing on the established update preconditions (see Algorithm 1), ii) encapsulating the set of OpenFlow messages M and the set of data packets PKT with the necessary control information related to the established order of events (see Algorithms 1 and 2), and iii) ensuring the delivery of MP in function of the piggybacked control information (see Algorithm 3). On sending an OpenFlow message m from the the cp,

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 each $pkt \in PKT$ piggybacks a vector CI_{rp} that carries control information of the OpenFlow message and packet send events that directly depend on it (see Algorithm 2). At the reception of an $mp \in MP$, a $rp \in P_{rp}$ checks, based on the control information encapsulated into CI_{mp}, if the delivery of mp respect or not the causal order. If yes, then rp delivers mp, else, rp will wait for the reception of another/other mp'(s) to then be able to deliver mp (see Algorithm 3).

- Ensured properties: The following properites are ensured:
  - Transient forwarding loop-free
  - Transient forwarding blackhole-free

5.3.2 Algorithm description

Data Structures. In the update mechanism each process $p \in P$ 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 example, suppose that CI_i is the vector of control information related to a process p_i such that $(k,t) \in CI_i[j]$ $(i \neq j)$. This means that any message sent by a process p_i should be delivered to p_j after the message or data packet $mp \in MP$ of sequence number t sent by p_k has been delivered to p_j. Furthermore, each process rp_i contains an $N \times N$ integer matrix $Delivery_i$ to track dependency information. This matrix stores the last sequence number of messages delivered to other processes. For instance, if $Delivery_i\ [j,k] = t$, this means that p_i is aware that messages, with sequence numbers that are less than or equal to t, sent by

process p_j to p_k have been delivered to p_k [GHH+20].

Algorithm 1: Controller-to-switch message sending [GHH+20]

1 **Input:** The set of OpenFlow update messages M

2 **Precondition: for each** $match_n \in MATCH$

3 $\forall(m_{del} \in M_{delete}) \wedge \forall(m_{add} \in M_{add}) : SdM(m_{del}, rp_i) \rightarrow SdM(m_{add}, rp_j) \mid rp_i \neq rp_j$

4 $\forall(m_{add} \in M_{add}) \wedge \forall(m_{del} \in M_{delete}) : SdM(m_{add}, rp_i) \rightarrow SdM(m_{del}, rp_j) \mid rp_i = rp_j$

5 $\forall(m_{add} \in M_{add}) : SdM(m_{add}, rp_i) \rightarrow SdM(m'_{add}, rp_j) \mid rp_i = INo \wedge rp_j = ENo$

6 **Variables initialization:**

7 $t_{cp} := 0$, $CI_{cp}[j] = \{\}\ \forall j : 1 \ldots N$

8 **Algorithm body:**

9 **for all** $m \in M \mid m \mathrel{\widehat{=}} match$ **do**

10 $t_{cp} := t_{cp} + 1$

11 $m = (cp, t_{cp}, OpenFlow\_message, rp_i)$

12 $SdM(m, CI_{cp})$ & $CI_{cp}[i] := (cp, t_{cp})$

Controller-to-switch message sending. Algorithm 1 handles sent events of update OpenFlow messages. As input, it takes all update messages calculated by the controller that should be communicated to the routing processes. As preconditions, the algorithm starts grouping the set of update messages M by *match* and preparing the order of disseminating of messages following the update policies introduced previously in Sections 5.1 and 5.2 (see lines 2-5). Then, it proceeds with sending per-match messages to the corresponding subset of routing processes (see lines 8-13). For each message send event, the logical clock of cp (denoted by t_{cp}) is incremented (see line 10), associating a timestamp to m (see line 11). Each message m is augmented by

encapsulating CI_{cp}. It contains information about the direct predecessors of m with respect to messages sent to the set P_{rp}. After sending a message 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 13) [GHH+20] (see line 12).

Algorithm 2: Switch-to-switch packet forwarding [GHH+20]

1 **Input:** Ingoing data packets PKT

2 **Variables initialization:**

3 $t_{rp_i} := 0$, $CI_{rp_i}[j] = \{\}$ $\forall j : 1 \ldots N$

4 **Algorithm body:**

5 $t_{rp_i} := t_{rp_i} + 1$

6 $pkt = (rp_i, t_{rpi}, header, data, rp_j)$

7 $FwdP(pkt, CI_{rp_i})$ & $CI_{rp_i}[j] := (rp_i, t_{rp_i})$

Switch-to-switch packet forwarding. This algorithm (see Algorithm 2) is responsible of forwarding the set of ingoing packets PKT. It takes PKT as input. Before forwarding a packet $pkt \in PKT$ to an rp_j, the logical clock of the sender rp_i (denoted by t_{rpi}) is incremented (see line 5) to associate a timestamp with pkt (see line 6). On forwarding a pkt to its next hop, the algorithm encapsulates pkt with the vector of control information CI_{rp_i}. It contains information about the direct predecessors of pkt with respect to OpenFlow messages/packets sent to rp_j. 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 [GHH+20] (see line 7).

Algorithm 3: Switch message/packet reception [GHH+20]

1 **Input:** OpenFlow messages sent from the controller and in-fly data packets MP

2 **Variables initialization:**

3 $Delivery_{rp_j}[i,k] = 0 \; \forall i,k : 1 \ldots N$, $CI_{rp_j}[i] = \{\} \; \forall i : 1 \ldots N$

4 **Algorithm body:**

5 $RecMP(mp = (p_i, t_i, content, p_j, CI_{mp}))$ % the content may be an OpenFlow message or a packet %

6 $wait \; (\forall k \; (k,x) \in CI_{mp}[j] \mid Delivery_j[k,j] \geq x)$ % the delivery condition of a $mp \in MP$ %

7 $DlvMP(mp)$ % the delivery of an $mp \in MP$ %

8 $Delivery_j[i,j] := t_{mp}$ % Actualization of the delivery matrix after the delivery of mp %

9 $\forall k \mid (k,y) \in CI_{mp}[i] \; Delivery_j[k,i] := max(Delivery_j[k,i], y)$ % Actualisation of the delivery matrix with respect to control information piggybacked with mp %

10 $CI_j[j] := (CI_j[j] \cup_{max} \{(i, t_{mp})\}) -_{max} CI_{mp}[j]$ % updating the vector of control information of rp_j with new delivery constraint for future messages sent from rp_j. %

11 $\forall p_k \in P \mid k \neq i,j$: $CI_j[k] := CI_j[k] \cup_{max} CI_{mp}[k]$ % updating the vector of control information of rp_j with delivery constraints related to other rps $\neq rp_i \wedge rp_j$ for future messages sent from rp_j. %

12 $\forall p_k \in P \mid k \neq j$: % garbage collection %

13 $\forall (l,x) \in CI_j[k]$

14 if $Delivery_j[l,k] \geq x$ then

15 delete (l,x) from $CI_j[k]$

Switch message/packet reception. The purpose of this Algorithm 3 is to ensure the delivery of the set of MP following the update policies established in the previous Section, guaranteeing updates free from connectivity inconsistencies, i.e transient forwarding loop-free and blackhole-free updates. As input, the algorithm takes

the set of MP [GHH+20].

An $mp \in MP$ piggybacks delivery constraints encapsulated into CI_{mp}. The receiver $rp_j \in P_{rp}$ lookups into CI_{mp} if there is an other/others mp'(s) that have to be delivered before mp. Recalling that there is a distinction between the reception of a message mp (see line 3) and its delivery (see Line 5). On the one hand, the reception is the fact that a $p_j \in P$ is notified about the reception event, i.e. such event has no influence on the global state of the network. On the other hand, the delivery of an mp to a process $p_j \in P$ implies that the mp in question is received and all previous delivery constraints on p_j were satisfied (see Line 6). Therefore, once the delivery constraints are satisfied, mp is delivered to rp_j (see Line 7). The following is to update control information saved into the data structures, i.e. the delivery matrix (see line 8-9) and the vector of control information (see lines 9-15), of rp_j. The delivery matrix $Delivery_j$ of rp_j is updated indicating that the message sent from p_i with sequence number is equal to t_{mp}, is already delivered to rp_j (see line 8). Furthermore, $Delivery_j$ matrix is also updated with respect to messages/packets mps delivered to process p_i (see line 9). Respecting the vector of control information CI_j, the algorithm updates it based on the control information piggybacked from CI_{mp}. This is, firstly, by updating it with new delivery constraints for future messages sent from rp_j: $CI_j[j]$ is updated by adding (i, t_{mp}), the control information related to the last delivered mp. This is ensured by using the $\cup_{max}$ operator[1] (see Algorithm 4). Also, already satisfied and transitive dependencies are deleted from $CI_j[j]$ using the $-_{max}$[2]. operator (see Algorithm 5) (see line 10). Secondly, $CI_j[k]$, where k represents all processes p_k except the sender and receiver process, are updated basing on $CI_{mp}[k]$. This is in order to maintain the causal order delivery of messages sent from rp_j and whose are causally dependent on messages sent

[1]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 [PRS97].

[2]The $-_{max}$ operator deletes the delivery constraints already known to be satisfied ($TP2$) from the current set of message delivery constraints ($TP1$) [PRS97]

by p_i to p_j. Therefore, $CI_i[k]$ is updated by adding the delivery constraints piggybacked by CI_{mp} (see Line 11). Finally, garbage collection on CI_j is applied, and this to reduce the communication overhead when ensuring the causal ordering of messages. Hence, based on the $Delivery_j$ matrix, CI_j is updated to only contain only recent delivery constraints needed for the delivery of future messages (see lines 12-15) [GHH+20].

Algorithm 4: : The operator $\cup_{max}$: $(TP1 \cup_{max} TP2)$.

```
1 Input: set of tuples TP1, set of tuples TP2, set of tuples TP
2 change := true
3 TP := TP1 ∪ TP2      (TP1 and TP2 contain the delivery constraints)
4 while (change) {
5     change := false
6     if (i, x) ∈ TP and (i, y) ∈ TP and (x < y)
7       { TP := TP − {(i, x)}
8       change := true } }
9 return(TP)
```

Algorithm 5: : The operator $-_{max}$: $(TP1 -_{max} TP2)$.

```
1 Input: set of tuples TP1, set of tuples TP2, set of tuples TP
2 change := true
3 TP := TP1
4 while (change) {
5     change := false
6     if (i, x) ∈ TP and (i, y) ∈ TP2 and (x ≤ y)
7       { TP := TP − {(i, x)}
8       change := true } }
9 return(TP)
```

5.3.3 Proof of correctness

In the previous subsection, we introduced an update mechanism as a solution for the connectivity inconsistency problems formalized in the previous Chapter. We now demonstrate that the proposed mechanism is transient forwarding loop-free as well as transient forwarding blackhole-free.

The update mechanism is transient forwarding loop-free. Based on the Definition 8, a transient forwarding loop occurs due to the violation of the causal order delivery of the relevant OpenFlow update messages. Consequently, a packet or a flow of packets enters into a loop between the underlying switches. To prove that the mechanism is transient forwarding loop-free, it is sufficient to demonstrate that there is no data packet flow that holds with the conditions specified in Definition 8. To accomplish the proof, we focus on the packet *pkt* that triggers the transient forwarding loop pattern. The following theorem proves this observation [GHH+20].

Theorem 1 *The update mechanism guarantees that* $\nexists$ *a data packet* $pkt_k \in pflow_m \mid pflow_m \widehat{=} match_n$*, such that (i)* $SdM(m') \rightarrow FwdP(pkt_k)$ *and (ii)* $DlvMP(pkt_k, t_i) \rightarrow DlvMP(m, t'_i)$ *where* $m, m' \widehat{=} match_n$ *and* $SdM(m) \rightarrow SdM(m')$*.*

Assuming that the Algorithm 1 and the Algorithm 2 store knowledge of the latest message/packet $mp \in MP$ sent from a 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 about the structure, see the structure description subsection). For this proof, we consider this additional instruction in both mentioned algorithms: $ForDelivered_i[i, j] = t$. This hypothetical instruction builds 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 [GHH+20]. 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 1 *This is proven by contradiction. Let us assume that there is a packet* $pkt_k \in pflow_m$*, such that:*

1. $SdM(m) \rightarrow FwdP(pkt_k)$

   According to Definition 8, 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 [PRS97], 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.*

The update mechanism is transient forwarding blackhole-free on Existing nodes. Following Definition 9, a transient forwarding blackhole on an Existing node (ENo) occurs due to a violation of FIFO order on the delivery of relevant update messages. Thus, a $pkt_k \in PKT$ enters into a transient forwarding blackhole on a ENo. The update mechanism presented in the previous Section prevents that pkt_k to match the conditions 1 and 2 (specified in Definition 9), giving raise the violation of the FIFO order. The following theorem provides this observation.

Theorem 2 *The update mechanism guarantees that ∄ a data packet* $pkt_k \in pflow_m \mid pflow_m \widehat{=} match_n$*, such that (i)* $SdM(m) \rightarrow SdM(m')$ *and (ii)* $DelMP(m', rp_b) \rightarrow DelMP(m, rp_b) \mid DelMP(m', rp_b) \rightarrow DelMP(pkt, rp_b)$ *and* $DelMP(pkt_k, rp_b) \rightarrow DelMP(m, rp_b)$ *where* $m, m' \widehat{=} match_n$.

A FIFO order ensures that messages by the same sender, in our case the controller $cp \in P_{cp}$, are delivered in the order that they were sent to a routing process receiver $rp \in P_{rp}$. That is, if cp sends a message $m \in M$ before it sends a message $m' \in M$, then rp delivers m' until it has previously received m. Therefore, to prove that the update mechanism is transient forwarding blackhole-free on ENos, it is sufficient to prove that ensuring the causal order of update events implies ensuring the FIFO order of the same. In other words, we should prove that the FIFO order is a particular case of causal order, ensured by the proposed update mechanism.

Proof 2 *Let* p *be the the first statement of this proof:* $m, m' \in M$ *are two update messages such that* $SdM(m, cp) \rightarrow SdM(m', cp)$ *where the routing process receiver of both* m *and* m' *is an ENo denoted by* $rp_b \in P_{rp}$*, and* q *to be the second statement: the delivery of both messages on* rp_b *is as following* $DelMP(m, rp_b) \rightarrow DelMP(m', rp_b)$*. Assuming that the FIFO order expressed in this case by* $p \implies q$ *is a specific case of the causal order.*

p *is true as it is considered as precondition when triggering the update mechanism (see Algorithm 1). It was be proven in [PRS97] that a causal ordering algorithm guarantees* q *when* p *is true following these two cases:*

1. $m \rightarrow m'$ *where* $p \in P$ *is the sender process of both, denoted by* p'.
2. $m \rightarrow m'$ *where the sender processes are respectively* $p \in P$ *and* $p' \in P \mid p \neq p'$*, and there is a finite sequence of message* $M' = \{m_1, m_2, ..., m_{n-1}, m_n\}$, $m_n = m'$ *with* $n >= 2$ *that causally relate* m *and* m'.

Hence, the first case p' correspond to the statement p guaranteeing q. Therefore, $p' \implies q$, meaning that FIFO order is a particular case of causal order.

The update mechanism is transient forwarding blackhole-free on Inserted nodes. During an update, a transient forwarding blackhole on an Inserted node (INo) denoted by $rp_b \in P_{rp}$, occurs, following the Definition 10, when rp_b delivers a packet $pkt \in PKT$ from its direct predecessor before delivering the corresponding update message, installing a new rule to forward pkt to its next hope. Thus, pkt enters into a transient forwarding blackhole on rp_b. The proposed update policy for this forwarding blackhole pattern is to i) establish a causal dependencies between (see Section 5.2.1) the subset of messages $M_{add} = \{m_{add_1}, m_{add_2}, ..., m_{add_n}\}$ sent to all INos, and any $pflow_m = \{pkt_1, pkt_2, ..., pkt_n\}$ that enters to rp_b and matches M_{add}. ii) To ensure the causal order delivery of the update events based on the update mechanism proposed in the previous Section. Therefore, no pkt enters into a transient forwarding blackhole on an INo. The following theorem formalizes the latter mentioned.

Theorem 3 *The update mechanism guarantees that $\nexists$ a data packet $pkt_k \in pflow_m \mid pflow_m \mathrel{\widehat{=}} match_n$, such that (i) $SdM(m, rp_r) \rightarrow FwdP(pkt_k, rp_r)$ and (ii) $DelMP(pkt_k, rp_b) \rightarrow DelMP(m, rp_b) \mid m \in M_{add}$ where $m \mathrel{\widehat{=}} match_n$.*

Proof 3 *This is proven by contradiction. Let us assume that there is a packet $pkt_k \in pflow_m$, such that:*

1. $SdM(m, rp_b) \rightarrow FwdP(pkt_k, rp_b)$

   This causal dependency is accorded based on the update policy on INos.

2. $DlvMP(pkt_k) \rightarrow DlvMP(m)$

Following the condition 1 and 2, $ForDelivery_{cp}[cp, rp_b] > Delivery_{rp_b}[cp, rp_b]$ when rp_b receives pkt_k. However, and based on the proof of [PRS97], this cannot occur as

the algorithm allows the delivery of pkt_k *to* rp_b *only when* $ForDelivery_{cp}[cp, rp_i] <= Delivery_{rp_b}[cp, rp_b]$*, which contradicts the initial assumption.*

5.4 Scope and limitations

Scope of this work. The aspects that were covered are summarized through the following points.

- Updates in SDNs are modeled at the event level according to the distributed and asynchronous nature of SDNs.
- Inconsistency connectivity invariant violations are identified and formally defined
- A mechanism to support connectivity consistency update for SDNs are designed.

Limitations of our approach. The limitations of our update approach are summarized through the following points.

- To bound the scope of this research, our network update was modeled to support a non-finite delay and no loss of messages and packets.
- The proposed update mechanism does not prevent the triggering of permanent forwarding blackholes. The mechanism should be extended to cover the network update policy defined in Section 5.2.2.

5.5 Chapter summary

This chapter introduced asynchronous SDN update policies, supporting connectivity consistency during SDN updates which includes transient forwarding loop-free and forwarding blackhole. These policies are based on establishing causal dependencies between relevant update events defined in the previous chapter. An update mechanism

that ensures these causal dependencies was presented. Finally, it was demonstrated that creates causal dependencies between relevant events is sufficient to ensure transient forwarding loop and transient forwarding blackhole. However, it is not the case for the permanent forwarding blackhole.

Chapter 6

Discussion

The problem of inconsistent connectivity updates was intensively attacked using different methods and techniques. The different solutions, as presented in Chapter 3, can be grouped based on four approaches: ordered update, n-phase commit update, timed update, and causal update approaches. In this Chapter, we analytically discuss our approach comparing it with the proposed approaches. Qualitative evaluation and comparison are led based on the following criteria: the execution model adopted to perform updates, the prior knowledge to consider when updating, the degree of the flexibility of updates and the computation costs. The execution update model consists of the system properties in which one attempts to design SDN updates – whether asynchronous or synchronous. Prior knowledge on updating consists of the pre-required information a control plane needs to launch a consistent update. The degree of flexibility of updates concerns whether an update mechanism permits that in-fly packets may be forwarded based on both initial forwarding policy and the final one, and not only one of them. As to metrics of update computation cost, amount of resources required to perform updates is measured and compared based on memory requirement and update rounds. The Table 6.1 summarizes the qualitative evaluation based on the mentioned metrics.

Execution update model. All previous related work established synchronous execution models to perform updates. Taking the timed update approach [MM16, ME16]. In this approach, updates on forwarding devices are synchronized to be processed simultaneously. To achieve the last mentioned, two synchronous system properties are assumed: i) clocks of forwarding devices are synchronized , and ii) upper bound on message delivery is defined. Actually, each forwarding device receives messages/packet

flows at an interval time $[PT - \varepsilon, PT + \varepsilon]$ where ε represents the clock synchronization error. Two delivery time intervals of events may overlap, knowing that the execution order of update events is not trivial to ensure update consistency. Hence, one cannot find out or force the execution order of events by using mechanisms for synchronizing physical time. Therefore, this is why such approach may result to an inconsistent update during the transition update phase, i.e. during updates.

On the other hand, system properties, based on which an SDN update is modeled in this work, represents a major difference compared to the state-of-the-art. Indeed, and as a fist system property, we assume that i) forwarding devices' clocks may not be accurate and can be out of synchronization. Furthermore, as a second system property, we assume that ii) update messages can be delayed for arbitrary period of times. Thus, there is no known upper limit on update message transmission delay between network devices. Forwarding devices receive messages/packet flows without any constraints on transmission time, which is a natural characteristic of a typical distributed system. However, the order of the delivery of relevant update events is constrained to be a causal order based on the established network policies (see Chapter 5). As a result, unlike the timed update, our asynchronous update approach permits to ensure connectivity consistent updates including during the transition update phase.

The factor behind this result is that we provide an asynchronous-based solution for an inherently asynchronous system. This is important because how SDN updates are modeled in this work is more suitable for real-world scenarios since it relaxes strong assumptions about forwarding devices clocks synchronization and upper bound on message delivery. On the other hand, the timed update approach provides a synchronous-based solution for an inherently asynchronous system. Such approach does not prevent unpredictable variations of command execution time on network switches caused by a delay of an update message.

Prior knowledge. The control plane is the brain of an SDN network and is responsible to calculate sequences of update operations. In some of the proposed solutions, the control plane needs to collect and maintain information about the network as an input to perform consistent updates.

In regards to the ordered update approach proposed by [FMW16], algorithms calculating sequences of update operations are centralized into the controller side. An update is performed based on different rounds, i.e. steps. Incrementally, and during each round, algorithms perform a per-round consistent update by executing a subset of operation on a subset of forwarding devices. To do so, the update mechanism extracts some constraints/rules and then verifies that their update rounds match the extracted constraints in the network forwarding graphs. In such approach, the complete network forwarding graph is considered as a prior knowledge to set up updates. Also, it should always be up-to-date after performing each update in order to ensure the consistency of future updates.

Concerning the timed update approach [MM16, ME16], and as discussed previously, the controller requires that clocks of forwarding devices should always be perfectly synchronized to be able to simultaneously perform update on the underlying forwarding devices.

As to the two-phase commit update approach [RFR+12, KRW13], the proposed update algorithm needs the network topology as a pre-requirement. In fact, the algorithm should distinguish between the forwarding devices that belongs to initial and final forwarding paths. We should highlight that our formalization of the forwarding blackhole problem is based on a per-forwarding devices classification only for analysis purpose, and our proposed solution does not need such classification as input to the update mechanism.

In our approach, the only prior knowledge is to be aware of the list of network

devices involved in an update. This is to be able to sort update messages by their match fields and to arrange the dissemination of messages from the controller following some predefined order (see Algorithm 1).

Degree of flexibility of updates. Previous approaches that designed consistent update mechanisms are considered rigid. This is because, during updates, in-fly packets either traverse the initial forwarding path or the final one, but never both. This is may delay updates as the execution of updates over forwarding devices is variable from one to another in term of the timing of message execution. Thus, an involved forwarding device that cannot update, due to the delay of an update message, prevents packets from continuing traversing the final forwarding path until the reception of the delayed message.

On the contrary there are only two works that propose flexible consistent updates which are the causal update approach [LSM19] that proposed Suffix Causal Consistency (SCC) and our approach. On the one hand, SCC introduces update algorithms based on Lamport timestamp. Note that SCC and our approach are quite different. In fact, SCC uses timestamps to tag packets reflecting the rules that correspond to each forwarding device. On the other hand, our approach uses timestamps to establish causal order between relevant 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 consistent updates. However, SCC allows that an in-fly packet *pkt* starts flying based on the initial forwarding path. Then, if *pkt* reaches a forwarding devices which has already updated its forwarding table, then *pkt* will be redirected to traverse the final forwarding path to reach its destination. As described in Chapter 3, the fourth step of the proposed algorithm treats this case. In fact, the controller calculates and installs extra temporarily forwarding rules, redirecting such packets to the recent forwarding path to reach their destination. The question here is how many extra rules would be

required per policy update? Indeed, this generates extra overhead related to controller and forwarding device memories, as well bandwidth as this requires message exchange between the controller and switches.

On the other hand, based on our proposal, a packet may start flying from the initial path and then be forwarded to the destination based on the final path without the need to install extra rules to direct it to the final path. Thus, only the rules calculated to perform updates are required. An example scenario of execution of updates is presented in [GHH+20] (see Figure 6 in Section 8). Taking the forwarding path which corresponds to $match_1$. A packet $pkt \in PKT$ may start flying from S_1 to S_2 based on the initial forwarding policy. Then, pkt reaches S_2 after it updates. Hence, S_2 re-forwards pkt to S_1. Following our updates mechanism, S_1 delivers pkt only after the delivery of the update message that delete the rule r forwarding packet from S_1 to S_2, preventing the trigger of a forwarding loop. Also, our update mechanism ensures that the delivery of r is ensured only after the delivery of the message installing the rule r' forwarding pkt from S_1 to S_n, preventing the occurrence of a forwarding blackhole. Finally, S_n ends by forwarding pkt to the Destination node.

Computational costs. Despite all the advantage provided by the different update approaches, they do not prevent extra resources required to set up their consistent update algorithms.

Regarding the solutions based on the two-phase commit update [RFR+12, KRW13], during updates, involved forwarding devices should be able to maintain rules of both forwarding policies (the initial and the final ones). In fact, if a switch maintains n forwarding rules, thus, and at the transition phase of update, the same switch should be able to store up to n^2 rules. This may end by congesting the memories of switches. Even worse, this approach may not be applicable when the number of rules bypasses the memory limit size of a switch. On the other hand, update number round is constant

as each update needs two rounds.

As to ordered update approaches [LRFS14, LwZ+13, FMW16], they calculate updates based on complete network forwarding graphs. In fact, the algorithms take forwarding graph properties as input to validate if each i-round matches or not the calculated update constraints. Computationally, the cost lies on the number of flows F, switches V , links E, requiring $O(FVE)$ which is costly in a large scale network. Concerning the round number, in the worst case, the algorithms require $O(n)$-rounds where n is the number of new edge to be added. This augments the number of interactions between the controller and forwarding devices, and therefore incurs a bandwidth overhead: in each round, forwarding devices should acknowledge once the new forwarding rules are installed to allow the controller to initiate the next update round. Furthermore, extra computation time overhead is generated as the controller should stop working while the switches acknowledge the achievement of each round.

As to the timed update approaches [MM16, ME16], and to simultaneously perform network updates among involved forwarding devices, authors based on ReversePTP [MM14], a clock synchronization protocol for SDNs. To synchronous the update operations, each forwarding device should periodically send a synchronous message to the controller containing its local timestamp. Thus, $O(n)$ messages should be sent periodically to the controller. Also, the later should maintain $O(n)$ timestamps related to the different forwarding devices. This serves to calculate the timestamp conversion between the controller's clock time and the forwarding devices' clock time, implementing a time-based protocol that allows forwarding devices to be synchronized [MM14]. The process of synchronizing switch' clocks is not transient during updates; however, it is permanent. Thus, the bandwidth and memory overhead generated are permanent too.

Concerning the work based on SCC [LSM19], the algorithm parts Backward closure and Send-back rules are calculated in function of the forwarding paths taken by the two

flow equivalence classes, i.e. the old and the new forwarding paths of flows. Assuming that c^{old} and c^{new} be the number of the old and the new equivalence classes, respectively, and then, iteratively, Backward closure algorithm should compare every old forwarding rule, belonging to the set of the old rules R^{old}, against every new forwarding rule, belonging to the set of new rules R^{new}. Thus, Backward closure cost $O(c^{old}c^{new} \times |R^{old}| \times |R^{new}|)$ steps. In the same manner, Send-back algorithm compares every pairs of the set R^{old}. Thus, it costs $O(c^{old}c^{new} \times |R^{old}|^2)$. Hence, the running time of the whole algorithm is $O(c^{old}c^{new} \times (|R^{old}| \times |R^{new}| + |R^{old}|^2))$-steps [LSM19].

Our approach establish a causal order-based policies to guarantee transient connectivity consistency during updates. Our update mechanism requires piggybacking and store control information on forwarding devices, resulting in memory and bandwidth overhead. However, the causal order message/packet algorithm was designed based on the IDR (see Definition 5). Based on IDR, only direct dependency information between messages/packets with respect to the destination process(es) is 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 is because there is no concurrent message sent from the controller to a forwarding device [GHH+20]. Furthermore, each message only piggybacked control information which only correspond to its match field. An example scenario of execution of updates of two distinct forwarding paths where each one correspond to different match *match* is presented in [GHH+20] (see Section 8). In Table A1, presented in Appendix A in [GHH+20], showing control information vector CI_{cp} of the controller process, we can observe that messages of $match_2$ do not piggyback control information related to messages of $match_1$. As to control information of vectors of forwarding devices, in fact, a component of a $CI_{rp_i}[j]$ can have at most n tuples for each process, i.e., for each vector component. This occurs when a forwarding device 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 [GHH+20].

Concerning memory overhead, a routing process rp_i needs to store a vector $CI_{rp_i}[j]$ and a matrix $Delivery_i$ of N x N, which then requires $O(n^2)$ integers to be stored.

Approaches	**Execution model**	**Prior knowledge**	**Degree of flexibility of updates**	**Computational costs**
N-phase commit update [RFR+12, KRW13]	Synchronous	Network topology	Rigid	For the 2-phase update, each switch should store n^2 rules during the transition phase. $O(1)$-round
Ordred update [LRFS14, LwZ+13] [FMW16]	Synchronous	Complete network forwarding graph	Rigid	$O(FVE)$ is needed to calculate an update. $O(n)$-rounds where n is the number of involved switches.
Timed update [MM16, ME16]	Synchronous	Perfect clock synchronization of switches	Rigid	$O(n)$ messages should be disseminated periodically to the controller, and it should maintain $O(n)$ timestamps. $O(1)$-round.
Causal update [LSM19]	Synchronous	The sets of forwarding rules to keep, to delete and to add	Flexible	The algorithm needs $O(c^{old}c^{new}\times$ ($\|R^{old}\|\times$ $\|R^{new}\|+$ $\|R^{old}\|^2$))-steps
Our approach [GHH+19, GHH+20]	Asynchronous	Set of involved switches and messages	Flexible	$O(n^2)$ control information to be piggybacked with each message/packet. $O(n^2)$ integers to be stored in each switch. $O(1)$$-round

Table 6.1: A qualitative comparison of our approach with the-state-of-the-art

Chapter 7

Conclusion and future work

7.1 Conclusion

This research aimed to design a consistent connectivity update mechanism for Software-Defined Networks (SDNs) in order to enforce the correctness of data plane when network policies change, maintaining an asynchronous communication between network entities. To conduct our approach, the following two methodological steps were the key to achieve our investigation results.

1. **A formal model of SDN updates.** The system is modeled to support updates at the event level assuming the following system properties which align with the distributed and asynchronous nature of SDNs:

   - Clocks of network entities are not accurate and then they are out of synchronization.
   - Communication between network entities is asynchronous. Thus, update messages and data packets may be transmitted and received by entities at any time intervals.
   - No global time reference is required to perform updates.
   - Update messages and data packets can be delayed for arbitrary period of times. Thus, there is no known upper limit on messages and data packets transmission delay between network entities.

2. **Formally defining patterns of invariant violation leading to inconsistency connectivity.** The two invariant violations, forwarding loops and forward-

ing blackhole, leading to inconsistency connectivity are characterized by defining the patterns under which these phenomenons may occur during updates. Each defined pattern represents an abstraction and a generalization of the phenomenon in question. On the one hand, the introduced abstractions reduce the complexity to understand the phenomenon by highlighting the relevant cause behind its triggering and by hiding the irrelevant details. On the other hand, the generalizations allows to catch any occurrence of these phenomenons.

Work findings. The findings of this work can be briefly summarized by the following points.

- The provided network modelization represents an asset of the presented update mechanism as it neither needs to synchronous clocks of network entities nor needs to define upper bound on the delivery of messages and data packets to perform updates guaranteeing transient connectivity consistency.

- Definitions of the invariant violations forwarding loop and forwarding blackhole, leading to inconsistent connectivity updates, are introduced capturing any of their occurrence during updates.

- To solve the problem from an asynchronous point of view, an update mechanism is proposed to break any occurrences of transient forwarding loop and transient forwarding blackhole.

- An important founding is that our update mechanism, and to ensure transient forwarding loop-free updates, takes in input the set of update messages and the set of the involved network entities to be updated. However, in [FMW16], which is one of the most recent works studying how to update SDN in a loop-free manner, requires the complete forwarding graph to calculate update constraints and the

operation steps in which update can be performed. Furthermore, in the worst case, the update mechanism of [FMW16] requires $O(n)$-steps where n is the number of network forwarding entities whereas our update mechanism requires $O(1)$-step, that is, a constant number of steps.

- Another important founding is that our consistent update mechanism permits that packet flows, which interleave with the set up of update messages, traverse both forwarding paths, i.e. the initial one (before updating) and the final one (after updating). This promotes the availability of the existence of a forwarding path for these data packets which should traverse both of forwarding paths during updates. However, the related work in the field permits that these packet flows traverse only the initial or the final forwarding path to preserve consistency. This may reduce the availability of a forwarding path for these packet flows during updates, and therefore, it may perturb their delivery to their destinations.

It is important to highlight that the case of permanent forwarding blackhole needs further an in-depth study. As it was discussed in Chapter 4 and 5, it cannot establish causal dependencies between the relevant updates events to guarantee their causal order delivery.

Thesis impact. Network community may refer to this work to leverage from two outputs:

- The already existed and future update mechanisms could verify the connectivity consistency correctness of their algorithms against the provided formal definitions of the phenomenon.

- The update mechanism is ready to be extended to support other consistency properties. The system model, based on which the algorithm is designed, does not make hard assumptions like e.g. upper bound on message/packet delivery

or notation of shared and globally synchronous clock between participant nodes. This is give support to the designer because he will feel free from such system constraints.

7.2 Future work

During the progress of this research work, it was noticed that the proposed theory can be extended to resolve other problems in the distributed systems research area. Furthermore, and to better understand the implications of our results, future studies could address the implementation of our mechanism to consider other performance metrics. Thereby, the future directions of this work may include:

- **The establishment of delivery order for concurrent events.** It was defined in this work that a permanent forwarding blackhole occurs, during updates, due to a concurrent access of two identified type of events to a Forgotten node, defined in our work to be a node that belong only to the initial forwarding path. Then, we showed that establishing causal dependencies between the underlying events is not possible for this invariant violation pattern.

  In this sense, a future research could be focused on the study of extending temporal causal dependencies between events to consider other domains of relationship between events like the degree of closeness of events in terms of distance [CH14]. The accent could be made on how to combine temporal/logical and distance domains to identify when each relevant event is created. How ancient one event is compared to another could be a conditional metric to decide about the delivery of concurrent events within the permanent forwarding blackhole pattern.

- **The extension of the scope of this investigation to consider a finite delay and/or the loss of messages and packets.** To delimit the scope of

this research work, it was assumed infinite delay and no loss of update messages and packets. Indeed, ensuring temporal constraints is out of scope in the present work.

The proposed solution of this work can be extended to support limited delay and message/packet loss by including the principle of Delta-Causal. In Delta-Causal [HDGF09], each message/packet has associated a delta lifetime that determines the maximal possible delay of messages/packets. Thus, a future work direction could be on studying the extension of the SDN model and the update mechanism of this work to support these assumptions.

- **The study of other inconsistency update problems.** In this work, we demonstrate that establishing causal dependencies is sufficient to cover transient connectivity problems. Another future direction of this investigation could be to continue studying and defining other network invariant violation patterns related to policy and capacity inconsistency. Such study will allow to answer whether ensuring causal dependencies will be sufficient to cover other consistent network update properties in SDNs.

- **The implementation of the update mechanism to consider other performance metrics.** To evaluate the performance of the update mechanism, evaluation metric, on the one hand, were measured based on space complexity. On the other hand, time complexity was calculated based on the required round per update, i.e. order of update steps. As yet, Mininet, which is one of the most today's used SDN emulator, does not offer performance fidelity to measure the time needed to implement updates. In this context, authors of [YJ17] have proposed a container-based virtual time system for software-defined network emulation. This system has enhanced the fidelity of the Mininet environment to bridge the gap

between research ideas and real-world network applications in terms of time. In this sense, future directions of this work can be focused on the implementation of the proposed update mechanism with such emulator.

Bibliography

[ADSW16] Saeed Amiri, Szymon Dudycz, Stefan Schmid, and Sebastian Wiederrecht. Congestion free rerouting of flows on dags. *CoRR*, pages 1–13, 2016.

[ALMS16] Saeed Amiri, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Transiently consistent sdn updates: Being greedy is hard. In *In International Colloquium on Structural Information and Communication Complexity*, volume 9988, pages 391–406, 2016.

[ARA06] Shaikh Aman, Dube Rohit, and Varma Anuja. Avoiding instability during graceful shutdown of multiple ospf routers. *IEEE/ACM Trans.Netw.*, 3(14):532–542, 2006.

[BBCC14] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2), 2014.

[CH14] Jose Roberto Perez Cruz and Saúl E. Pomares Hernández. Temporal data alignment and association for event-streaming in ubiquitous environments based on fuzzy-causal dependencies. In *24th International Conference on Electronics, Communications and Computing*, pages 127–134, 2014.

[DLS16] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't touch this: Consistent network updates for multiple policies. *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 1:133–143, 2016.

[DSH+10] A. Doria, J. Hadi Salim, R. Haas, W. Wang, L. Dong, and R. Gopal. Forwarding and control element separation (forces) protocol specification, 2010.

[FB07] Pierre Francois and Olivier Bonaventure. Avoiding transient loops during the convergence of link-state routing protocols. *IEEE/ACM Transactions on Networking*, 15(6):1280–1932, 2007.

[FLMS18] Klaus-Tycho Foerster, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Loop-free route updates for software-defined networks. *IEEE/ACM Transactions on Networking*, 26(1):328–341, 2018.

[FLSW18] Klaus-Tycho Foerster, Thomas Luedi, Jochen Seidel, and Roger Wattenhofer. Local checkability, no strings attached: (a)cyclicity,reachability, loop free updates in SDNs. *Theoretical Computer Science*, 709:48–63, 2018.

[FMW16] Klaus-Tycho Foerster, Ratul Mahajan, and Roger Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *IFIP Networking Conference*, pages 1–9, 2016.

[FSV16] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. Survey of consistent network updates. *IEEE Communications Surveys & Tutorials*, 21(2):1435–1461, 2016.

[FSYM14] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey Mogul. Enforcing network-wide policies in thepresence of dynamic middlebox actions

using flowtags. In *USENIX Conference onNetworked Systems Design and Implementation*, pages 533–546, 2014.

[FW16] Klaus-Tycho Forster and Roger Wattenhofer. The power of two in consistent network updates: Hard loop freedom, easyflow migration. *In Proceedings of the 2016 25th International Conference on Computer Communication andNetworks (ICCCN)*, pages 1–9, 2016.

[GHH+19] Amine Guidara, Saúl E. Pomares Hetnandez, Lil María X. Rodríguez Henríquez, Hatem Hadj Kacem, and Ahmed Hadj Kacem. A study of the forwarding blackhole phenomenon during software-defined network updates. In *2019 Sixth International Conference on Software Defined Systems (SDS)*, pages 186–193, 2019.

[GHH+20] Amine Guidara, Saúl E. Pomares Hetnandez, Lil María X. Rodríguez Henríquez, Hatem Hadj Kacem, and Ahmed Hadj Kacem. Towards causal consistent updates in software-defined networks. *Appl. Sci.*, 10(6), 2020.

[HDGF09] Saúl E. Pomares Hernández, L. Dominguez, E. Rodriguez Gomez, and J. Fanchon. An efficient delta-causal algorithm for real-time distributed systems. *Appl. Sci.*, pages 1711–1718, 2009.

[JMD14] Yosr Jarraya, Taous Madi, and Mourad Debbabi. A survey and a layered taxonomy of software-defined networking. *Communications Surveys Tutorials, IEEE*, 16(4):1955–1980, 2014.

[KRV+15] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. In *Proceedings of the IEEE*, number 1, pages 14–76, 2015.

[KRW13] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *The Second ACMSIGCOMM Workshop on Hot Topics in Software Defined Networking*, volume 49–54, 2013.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Magazine Communications of the ACM*, pages 558–565, 1978.

[LMS15] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Scheduling loop-free network updates: It's good to relax! In *In Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 13–22, 2015.

[LRFS14] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. 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*, HotNets-XIII, pages 15:1–15:7, New York, NY, USA, 2014. ACM.

[LSM19] T.A. Liu S., Benson and Reiter M.K. Efficient and safe network updates with suffix causal consistency. In *In Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[LwZ+13] Hongqiang Harry Liu, Xin wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David A. Maltz. zupdate: Updating data center networkswith zero loss. *ACM SIGCOMM Computer Communication Review*, 43:411–422, 2013.

[ME16] Moses Y. Mizrahi E.S.T. Timed consistent network updates in software-defined networks. *IEEE/ACM Trans. Netw.*, pages 1–14, 2016.

[Mil91] David L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991.

[MM14] Tal Mizrahi and Yoram Moses. Using ReversePTP to distribute timein software defined networks. *International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication, ISPCS*, 2014.

[MM16] Tal Mizrahi and Yoram Moses. Software defined networks: It's about time. *In Proceedings of the 35th Annual IEEEInternational Conference on Computer Communications*, 2016.

[MW13] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *In Proceedings of the12th ACMWorkshop on Hot Topics in Networks*, number 20, pages 1–7, 2013.

[NT17] Canini M. Nguyen T.D., Chiesa M. Decentralized consistent updates in SDN. In *The Symposium on SDN Research*, pages 21–33, 2017.

[ONF12] ONF. Software-defined networking: The new norm for networks. Technical report, Open Networking Foundation, 2012.

[ONF15] ONF. Openflow switch specification, version 1.5.1. Technical report, Open Networking Foundation, 2015.

[PD13] Ben Pfaff and Bruce Davie. The Open vSwitch Database Management Protocol. RFC 7047, December 2013.

[PH15] Saúl Eduardo Pomares Hernández. The Minimal Dependency Relation for Causal Event Ordering in Distributed Computing. *Applied Mathematics & Information Sciences*, 9(1):pp.57–61, January 2015.

[PRS97] Ravi Prakash, Michel Raynal, and Mukesh Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *J. Parallel Distrib. Comput.*, 41(2):190–204, March 1997.

[RFR+12] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[SAJ+14] Marc Suñé, Victor Alvarez, Tobias Jungel, Umar Toseef, and Kostas Pentikousis. An openflow implementation for network processors. *Proceedings of the 2014 Third European Workshop on Software Defined Networks*, pages 123–124, 2014.

[Son13] Haoyu Song. Protocol-oblivious forwarding: unleash the power of sdn through a future-proof forwarding plane. *HotSDN '13: Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132, 2013.

[VC16] Stefano Vissicchio and Luca Cittadini. Flip the (flow) table: Fast lightweight policy-preserving sdn updates. In *IEEE INFOCOM*, pages 1–9, 2016.

[YJ17] Jiaqi Yan and Dong Jin. A lightweight container-based virtual time system for software-defined network emulation. *Journal of Simulation*, 11(3):253–266, 2017.

[ZLLC14] Wei Zhou, Li Li, Min Luo, and Wu Chou. REST API design patterns for SDN northbound API. In *Proceedings of the 2014 28th International Con-*

ference on Advanced Information Networking and Applications Workshops, 2014.