Renaissance: Self-Stabilizing Distributed SDN Control Plane (Technical Report)

Marco Canini[1] Iosif Salem[2] Liron Schiff[3] Elad M. Schiller[2] Stefan Schmid[4]
[1] Université catholique de Louvain [2] Chalmers University of Technology
[3] GuardiCore Labs [4] University of Vienna & Aalborg University

Abstract

By introducing programmability, automated verification, and innovative debugging tools, Software-Defined Networks (SDNs) are poised to meet the increasingly stringent dependability requirements of today's communication networks. However, the design of fault-tolerant SDNs remains an open challenge.

This paper considers the design of dependable SDNs through the lenses of *self-stabilization*—a very strong notion of fault-tolerance. In particular, we develop algorithms for an in-band and distributed control plane for SDNs, called *Renaissance*, which tolerate a wide range of (concurrent) controller, link, and communication failures. Our self-stabilizing algorithms ensure that after the occurrence of an arbitrary combination of failures, (i) every non-faulty SDN controller can eventually reach any switch in the network within a bounded communication delay (in the presence of a bounded number of concurrent failures) and (ii) every switch is managed by at least one non-faulty controller.

We evaluate *Renaissance* through a rigorous worst-case analysis as well as a prototype implementation (based on OVS and Floodlight), and we report on our experiments using Mininet.

1 Introduction

Context and Motivation. Software-Defined Networks (SDNs) have emerged as a promising alternative to the error-prone and manually configured traditional communication networks. In particular, by outsourcing and consolidating the control over the data plane elements, SDNs support a programmatic verification and enable new debugging tools.

However, while the literature articulates well the benefits of the separation between control and data plane and the need for distributing the control plane (e.g., for performance and fault-tolerance), the question of how connectivity between these two planes is maintained (i.e., the communication channels from controllers to switches and between controllers) has not received much attention. Providing such connectivity, is critical for ensuring the availability and robustness of SDNs.

Guaranteeing that each switch is managed, at any time, by at least one controller is challenging especially if control is *in-band*, i.e., if control and data traffic is forwarded along the same links and devices and hence arrives at the same ports. In-band control is desirable as it avoids the need to build, operate, and ensure the reliability of a separate out-of-band management network. Moreover, in-band management can in principle improve the resiliency of a network, by leveraging a higher path diversity (beyond connectivity to the management port).

The goal of this paper is the design of a highly fault-tolerant distributed and in-band control plane for SDNs. In particular, we aim to develop a self-stabilizing software-defined network: An

SDN which recovers from controller, switch, and link failures, as well as a wide range of communication failures (such as packet omissions, duplications, or reorderings). As such, our work is inspired by Radia Perlman's pioneering work [21]: Perlman's work envisioned a self-stabilizing Internet and enabled today's link state routing protocols to be robust, scalable, and easy to manage. Perlman also showed how to modify the ARPANET routing broadcast scheme, so that it becomes self-stabilizing [22], and provided a self-stabilizing spanning tree algorithm for interconnecting bridges [23]. Yet, while the Internet core is "conceptually self-stabilizing", Perlman's vision remains an open challenge, especially when it comes to recent developments in computer networks, such as SDNs, for which we propose self-stabilizing algorithms.

Failure Model. We consider (i) fail-stop failures of controllers, (ii) link failures, and (iii) communication failures, such as packet omission, duplication, and reordering. In particular, our failure model includes up to κ link failures, for some parameter $\kappa \in \mathbb{Z}^+$. In addition, to the failures captured in our model, we also aim to recover from *transient faults*, i.e., any temporary violation of assumptions according to which the system and network were designed to behave, e.g., the corruption of the packet forwarding rules or malicious changes to the availability of links, switches, and controllers. We assume that (an arbitrary combination of) these transient faults can corrupt the system state in unpredictable manners. In particular, when modeling the system, we assume that these violations bring the system to an arbitrary state (while keeping the program code intact). Starting from an arbitrary state, the correctness proof of self-stabilizing systems [9, 11] has to demonstrate the return to correct behavior within a bounded period, which brings the system to a *legitimate state*.

The Problem. This paper answers the following question: How can all non-faulty controllers maintain bounded (in-band) communication delays to any switch as well as to any other controller? We interpret the requirements for provable (in-band) bounded communication delays to imply (i) the absence of out-of-band communications or any kind of external support, and yet (ii) the possibility of fail-stop failures of controllers and link failures, as well as (iii) the need for guaranteed bounded recovery time after the occurrence of an arbitrary combination of failures. The studied problem also considers the possibility of any transient violation of the assumptions according to which the system was designed to behave, which we call transient faults.

Our Contributions. We present an important module for dependable networked systems: a self-stabilizing software-defined network. In particular, we provide a (distributed) self-stabilizing algorithm for decentralized SDN control planes that, relying solely on in-band communications, recover (from a wide spectrum of controller, link, and communication failures as well as transient faults) by re-establishing connectivity in a robust manner. Concretely, we present a system, henceforth called *Renaissance*[1], which, to the best of our knowledge, is the first to provide:

1. *A robust efficient and decentralized control plane:* We maintain short, $O(D)$-length control plane paths in the presence of controller and link (at most κ many) failures, as well as, communication failures, where $D \leq N$ is the (largest) network diameter (when considering any possible network changes over time) and N is the number of nodes in the network. More specifically, suppose that throughout the recovery period the network topology was $(\kappa + 1)$-edge-connected and included at least one (non-failed) controller. We prove that starting from a legitimate state, i.e., after recovery, our self-stabilizing solution can:

[1]The word renaissance means 'rebirth' (French) and it symbolizes the ability of the proposed system to recover after the occurrence of transient faults that corrupt the system state.

- *Deal with fail-stop failures of controllers:* These failures require the removal of stale information (related to unreachable controllers) from the switch configurations. Cleaning up stale information avoids inconsistencies and having to store large amounts of history data.
- *Deal with link failures:* Starting from a legitimate system state, the controllers maintain an $O(D)$-length path to all nodes (switches and other controllers), as long as at most κ links fail. That is, after the recovery period the communication delays are bounded.

2. *Recovery from transient faults:* We show that our control plane can even recover after the occurrence of transient faults. That is, starting from an *arbitrary* state, the system recovers within time $O(D^2N)$ to a legitimate state. In a legitimate state, the number of packet forwarding rules per switch is at most N_C times the optimal, where N_C is (an upper bound on) the number of controllers.

While we are not the first to consider the design of self-stabilizing systems which maintain redundant paths also beyond transient faults, the challenge and novelty of our approach comes from the specific restrictions imposed by SDNs (and in particular the switches). In this setting not all nodes can compute and communicate, and in particular, SDN switches can merely forward packets according to the rules that are decided by other nodes, the controllers. This not only changes the model, but also requires different proof techniques, e.g., regarding the number of resets and illegitimate rule deletions.

In order to validate and evaluate our model and algorithms, we implemented a prototype of *Renaissance* in Floodlight using Open vSwitch (OVS), complementing our worst-case analysis. Our experiments in Mininet demonstrate the feasibility of our approach, indicating that in-band control can be bootstrapped and maintained efficiently and automatically, also in the presence of failures.

Organization. We give an overview of our system and the components it interfaces in Section 2 and introduce our formal model in Section 3. Our algorithm is presented in Section 4, analyzed in Section 5, and validated in Section 6. We then discuss related work (Section 7) before drawing the conclusions from our study (Section 8).

2 The System in a Nutshell

Our self-stabilizing SDN control plane can be seen as one critical piece of a larger architecture for providing fault-tolerant communications. Indeed, a self-stabilizing SDN control plane can be used together with existing self-stabilizing protocols on other layers of the OSI stack, e.g., self-stabilizing link layer and self-stabilizing transmission control protocols [12], which provide logical FIFO communication channels. To put things into perspective, we provide a short overview of the overall network architecture we envision. Our proposal includes new self-stabilizing components that leverage existing self-stabilizing protocols towards an overall network architecture that is more robust than existing SDNs.

We consider an architecture (Figure 1) that comprises mechanisms for local topology discovery and a logic for packet forwarding rule generation. We contribute to this architecture a self-stabilizing abstract switch as well as a self-stabilizing SDN control platform.

The network includes a set $P_C = \{p_1, \ldots, p_{N_C}\}$ of N_C *(remote) controllers*, and a set $P_S = \{p_{N_C+1}, \ldots, p_{N_C+N_S}\}$ of the N_S *(packet forwarding) switches*, where i is the unique identifier of

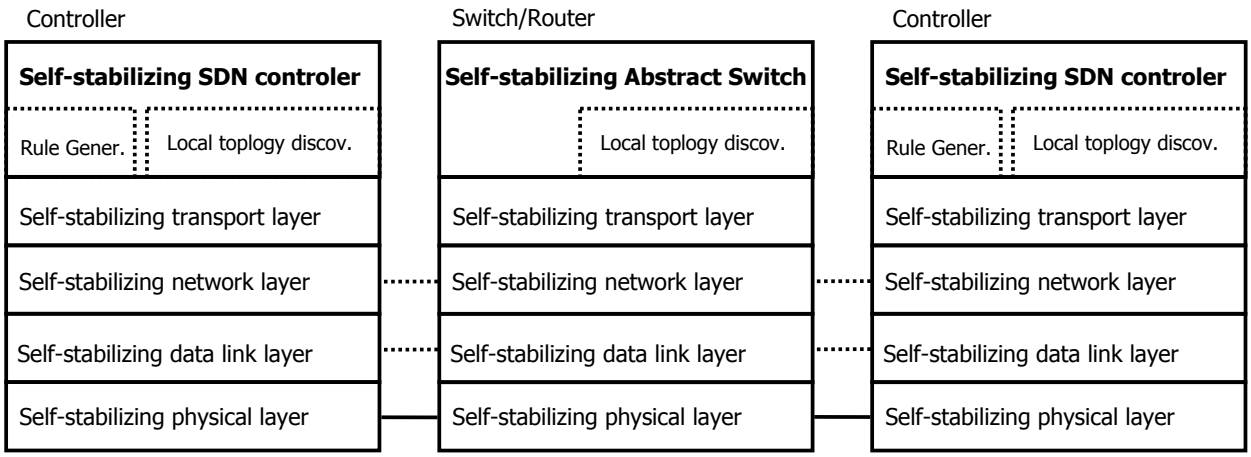

Figure 1: The system architecture, which is based on self-stabilizing versions of existing network layers. The external building blocks for rule generation and local topology discovery appear in the dotted boxes. The proposed contribution of self-stabilizing SDN controller and self-stabilizing abstract switch appear in bold.

node $p_i \in P = P_C \cup P_S$. We denote by $N_c(i) \subseteq P$ (communication neighborhood) the set of nodes which are directly connecting node $p_i \in P$ and node p_j, i.e., $p_j \in N_c(i)$. At any given time, and for any given node $p_i \in P$, the set $N_o(i)$ (operational neighborhood) refers to p_i's directly connected nodes for which ports are currently available for packet forwarding. The local topology information in $N_o(i)$ is liable to change rapidly and without notice. We denote the operational and connected communication topology as $G_o = (P, E_o)$, and respectively, as $G_c = (P, E_c)$, where $E_x = \{(p_i, p_j) \in P \times P : p_j \in N_x(i)\}$ for $x \in \{o, c\}$.

Each switch $p_i \in P_S$ stores a set of rules that the controllers install in order to define which packets have to be forwarded to which ports. In the out-of-band control scenario, a controller communicates the forwarding rules via a dedicated management port to the *control module* of the switch. In contrast, in an in-band setting, the control traffic is interleaved with the data plane traffic, which is the traffic between *hosts* (as opposed to controller-to-controller and controller-to-switch traffic): switches can be connected to hosts through data ports and may have additional rules installed in order to correctly forward their traffic. We do not assume anything about the hosts' network service, except for that their traffic may traverse any network link.

In an in-band setting, control and data plane traffic arrive through the same ports at the switch, which implies a need for being able to *demultiplex* control and data plane traffic: switches need to know whether to forward (data) traffic out of another port or (control) traffic to the control module. In other words, control plane packets need to be logically distinguished from data plane traffic by some tag (or another deterministic discriminator).

Figure 2 illustrates the switch model considered in this paper. Our self-stabilizing control plane considers a proposal for *abstract switches* that do not require the extensive functionality that existing SDN switches provide. An abstract switch can be managed either via the management port or in-band. It stores forwarding (match-action) rules. These rules are used to forward data plane packets to ports leading to neighboring switches, or to forward control packets (e.g., instructing the control module to change existing rules) to the local control module. Rules can also drop all the matched packets. The match part of a rule can either be exact match or optionally include

wildcards.

Maintaining the forwarding rules with in-band control is the key challenge addressed in this paper: for example, these rules must ensure (in a self-stabilizing manner) that control and data packets are demultiplexed correctly (e.g., using tagging). Moreover, it must be ensured that we do not end up with a set of misconfigured forwarding rules that drop *all* arriving (data plane and control plane) packets: in this case, a controller will never be able to manage the switch anymore in the future.

In the following, we will assume a local topology discovery mechanism which reports to the controllers the availability of their direct neighbors. Furthermore, we can assume reliable, bidirectional FIFO-communication channels (without reordering or omission) between communication endpoints at the transport layer [12].

2.1 Switches and Rules

Each switch $p_i \in P_S$ stores a set of forwarding rules which are installed by the controllers (servers) and define which packets have to be forwarded to which ports. In an out-of-band network, a controller communicates the forwarding rules via a dedicated management port to the *control module* of the switch. In contrast, in an in-band setting, the control traffic is interleaved with the dataplane traffic, and is communicated (possibly along multiple hops, in case of a remote controller) to a regular switch port. This implies that in-band control requires the switch to demultiplex control and data plane traffic. In other words, the dataplane of a switch cannot only be used to connect the switch ports internally but also to connect to the control module.

In this paper, we make the natural assumption that switches have a bounded amount of memory. Moreover, we assume that rules come in the form of match-action pairs, where the match can optionally include wildcards and the action part mainly defines a forwarding operation (cf. Figure 2).

More formally, suppose that $p_i \in P_S$ is a switch that receives a packet with $p_{src} \in P_C$ and $p_{dest} \in P$, as the packet source and destination, respectively. We refer to a *rule* (for packet forwarding at the switch) by a tuple $\langle k, i, src, dest, prt, j, metadata \rangle$, where p_k is the controller that created this rule, $prt \in \{0, \ldots, n_{prt}\} : n_{prt} \geq \kappa + 1$ is a priority that p_k assigns to this rule, $p_j \in N_c(i)$ is a port on which the packet can be sent whenever $p_j \in N_o(i)$, and $metadata$ is an (optional) opaque data value. Our self-stabilizing abstract switch considers only rules that are installed on the switches indefinitely, i.e., until a controller *explicitly* requests to delete them, rather than setting up rules with expiration *timeouts*.

We say that the rule $r = \langle k,\ i,\ src,\ dest,\ prt,\ j,\ metadata \rangle$ is *applicable* for a packet that reaches switch p_i and has source p_{src} and destination p_{dest}, when r is the rule with the highest prt (priority) that matches the packet's source and destination fields, and $p_j \in N_o(i)$, i.e., the link (p_i, p_j) is operational. We say that the set of rules of switch p_i, $rules(i)$, is *unambiguous*, if for every received packet there is at most one applicable rule. Thus, a packet can be forwarded if there exists only one applicable rule in the switch's memory. We assume an interface $myRules()$ which outputs the unambiguous rules that a controller $p_k \in P_C$ needs to install to a switch $p_j \in P_S$, based on p_k's knowledge of the network's topology. We require rules to be unambiguous and offer resilience against at most κ link failures (details appear in Section 2.2.2).

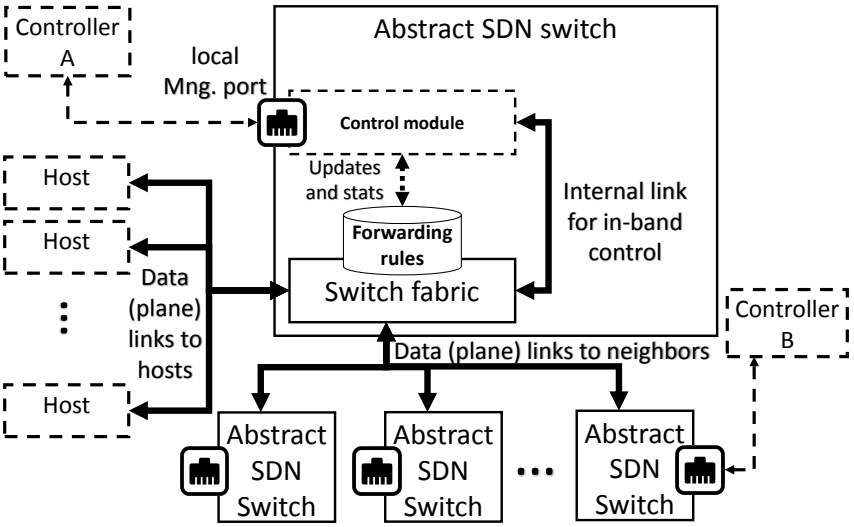

Figure 2: Abstract SDN switch illustration.

2.1.1 The Abstract Switch

The main task of switches is to forward traffic according to the rules installed by the controllers. In addition, switches provide basic functionalities for interacting with the controllers.

While OpenFlow, the de facto standard SDN switch implementation, as well as other suggestions (Forwarding Metamorphosis, P4, SNAP) provide innovative abstractions with respect to data plane functionality and means to implement efficient network services, there is less work regarding the control plane abstraction, especially with respect to fault tolerance.

We consider a slightly simplified switch model which does not include all the functionality one may find in a real SDN switch. In particular, the proposed abstract SDN switch only supports the *equal roles* approach (where multiple "equal" controllers manage the switch); the *master-slave setup* usually used by switches [19] is not relevant toward the design of our self-stabilizing distributed SDN control plane. We elaborate more on the interface in the following.

Configuration Queries (via a Direct Neighbor)

As long as the system rules and operational links support (bidirectional) packet forwarding between controller p_i and switch p_j, the abstract switch allows p_i to access p_j's configuration remotely, i.e., via the interfaces $manager(j)$ (query and update), $rules(j)$ (query and update) as well as $N_c(j)$ (query-only), where $manager(j) \subseteq P_C$ is p_j's set of assigned managers and $rules(j)$ is p_j's rule set. Also, a switch p_j, upon arrival of a query of a controller p_i, responds to p_i with the tuple $\langle j, N_c(j), manager(j), rules(j) \rangle$.

The abstract switch also allows controller p_i to query node p_j via p_j's direct neighbor, p_k as long as p_i knows p_k's local topology. In case p_j is a switch, p_i can also modify p_j's configuration (via p_j's abstract switch) to include a flow to p_i (via p_k) and then to add itself as a manager of p_j. We refer to this as the *query (and modify)-by-neighbor* functionality.

The Switch Memory Management

The number of rules and managers that each switch can store is bounded by $maxRules$ and $maxManagers$, respectively. The abstract switch has a way to deal with clogged memory by storing the rules and managers in a FIFO manner (say, using local counters that serve as timestamps in the meta-information ($metadata$) part of each rule). Whenever a controller accesses a switch, that switch refreshes these timestamps, i.e., all switch configuration items related to this controller. When the switch memory has more than $maxRules$ rules, the switch removes the rule that has the earliest timestamp so that a new rule can be added. Note that, as long as a switch has sufficient memory to store the rules of all controllers in P_C, the above mechanism does not need to remove any rule of controller $p_i \in P_C$ after the first time that p_i has refreshed its rules on that switch. Similarly, we assume that whenever the number of managers that a switch stores exceeds $maxManagers$, the last to be stored (or access) manager is removed so that a new manager can be added.

2.2 Building Blocks

Our architecture relies on a fault-tolerant mechanism for topology discovery. We use such a mechanism as an external building block. Moreover, we require a notion of resilient flows. We next discuss both these aspects.

2.2.1 Topology Discovery

We assume a mechanism for local neighborhood discovery. We consider a system that uses an (ever running) failure detection mechanism, such as the self-stabilizing Θ failure detector [4]: it discovers the switch neighborhood by identifying the failed/non-failed status of its attached links and neighbors. We assume that this mechanism reports the set of nodes which are directly connecting node $p_i \in P$ and node p_j, i.e., $p_j \in N_c(i)$.

2.2.2 Fault-resilient Flows

We consider fault-resilient flows which are a reminiscent of the flows in [18]. The definition of κ-fault-resilient flows considers the network topology G_c and assumes that G_c is not subject to changes. The idea is that the network can forward the data packets along the shortest routes, and use alternative routes in the presence of link failures, based on conditional forwarding rules [5]; these failover rules provide a backup for every edge and an enhancement of this redundancy for the case in which at most κ links fail, as we describe next.

Let $(p_{r_1}, \ldots, p_{r_n}) \in P^n$ be a directed path in the communication network G_c, where $n \in \{2, \ldots, |P|\}$. Given an operational network G_o, we say that $(p_{r_1}, \ldots, p_{r_n})$ is a *flow* (over a simple path) in G_o, when the rules stored in $p_{r_1}, \ldots, p_{r_n}$ relay packets from source p_{r_1} to destination p_{r_n} using the switches in the sequence $p_{r_2}, \ldots, p_{r_{n-1}}$ for packet forwarding (relay nodes). Let $G_o(k)$ be an operational network that is obtained from G_c by an arbitrary removal of k links. We say there is a k-fault-resilient flow from p_i to p_j in G_c when for any $k \leq \kappa$ there is a flow (over a simple path) from p_i to p_j in any $G_o(k)$. We note that when considering a communication graph, G_c, with a general topology, the construction of κ-fault-resilient flows is possible when $\kappa < \lambda(G_c)$, where $\lambda(G_c)$ is the edge-connectivity of G_c (i.e., the minimum number of edges whose removal can disconnect G_c).

3 Models

This section presents a formal model of the studied system (Figure 1), which serves as the framework for our correctness analysis of the self-stabilizing protocols (Section 5).

We model the control plane as a message passing system that has no notion of clocks (nor timeout mechanisms), as in the Paxos model [4, 17]. We borrow from [4, Section 6] a technique for local link monitoring (Section 2.2.1), which assumes that every abstract switch can complete *at least* one round-trip communication with any of its direct neighbors while it completes *at most* Θ round-trips with any other directly connected neighbor. Apart from this monitoring of link status, we consider the control plane as an asynchronous system. Note that once the system installs a κ-fault-resilient flow between controller $p_i \in P_c$ and node $p_j \in P \setminus \{p_i\}$, the network provides a communication channel between p_i and p_j that has a bounded delay (because we assume that there are never more than κ link failures). Moreover, these bounded delays are offered by the data plane while the control plane is still asynchronous as described above (since, for example, we assume no bound on the time it takes a controller to perform a local computation).

The self-stabilizing algorithms usually consist of a *do forever* loop that contains communication operations and validations that the system is in a consistent state as part of the transition decision. An iteration (of the *do forever* loop) is said to be *complete* if it starts in the loop's first line and ends at the last (regardless of whether it enters branches). As long as every non-failed node eventually completes its do forever loop, the proposed algorithm is oblivious to the rate in which this completion occurs (and the exact time consideration to be added later for the sake of fine-tuning performances).

3.1 The Communication Channel Model

We are given reliable end-to-end FIFO channels over capacitated links, as implemented, e.g., by [12], which guarantee no packet omission, duplication, and reordering. After the recovery period of the channel algorithm [12], it holds that, at any time, there is exactly one token $pkt \in \{act, ack\}$ in the channel that is either in transit from the sender $p_i \in P$ to the receiver $p_j \in P$, i.e., $channel_{i,j} = \{act\} \wedge channel_{j,i} = \emptyset$, or the token pkt is in transit from p_j to p_i, i.e., $channel_{i,j} = \emptyset \wedge channel_{j,i} = \{ack\}$. During the recovery period, it can be the case that the sender sends a message m_0 for which it receives a (false) acknowledgement ack_0 without having m_0 go through a complete round-trip. However, that can occur at most Δ_{comm} times, where $\Delta_{comm} \leq 3$ for the case of [12]. That is, once the sender sends message m_1 and receives its acknowledgement ack_1, the channel algorithm [12] guarantees that m_1 has completed a round-trip.

When node p_i sends a packet, $pkt \in \{act, ack\}$, to node p_j, the operation *send* inserts a copy of pkt to the FIFO queue that represents the above communication channel from p_i to p_j, while respecting the above token circulation constraint. When p_j receives pkt from p_i, node p_j delivers pkt from the channel's queue and transfers pkt's acknowledgement to the channel from p_j to p_i immediately after.

3.2 The Execution Model

For our analysis, we consider the standard *interleaving model* [11], in which there is a single (atomic) step at any given time. An input event can be either a packet reception or a periodic timer triggering p_i to resend. In our settings, the timer rate is completely unknown.

We model a node (switch or controller) using a state machine that executes its program by taking a sequence of *(atomic) steps*, where a step of a controller starts with local computations and ends with a single communication operation: either *send* or *receive* of a packet. A step of the abstract switch starts with a single message reception, continues with internal processing and ends with a single message send.

The *(node) state* p_i, denoted by s_i, consists of the values of all the variables of the node including its communication channels. The term *(system) state* is used for a tuple of the form $(s_1, s_2, \cdots, s_n, G_o)$, where each s_i is the state of node p_i (including messages in transit to p_i) and G_o is the operational network that is determined by the environment. We define an *execution (or run)* $R = c_0, a_0, c_1, a_1, \ldots$ as an alternating sequence of system states c_x and steps a_x, such that each state c_{x+1}, except the initial system state c_0, is obtained from the preceding state c_x by applying step a_x.

For the sake of simple presentation of the correctness proof, we assume that the abstract switch deals with one controller at a time, e.g., when requesting a configuration update or a query. Moreover, we assume that within a single atomic step, the abstract switch can receive the controller request, perform the update, and send a reply to the controller.

3.3 The Network Model

We assume that at any time there are no more than κ link failures. We model as a *transient fault* the events of a failure of more than κ links (or the addition of new links), switch fail-stop failures (or the addition of switches to the network), as well as any temporary violation of the assumptions according to which the system was designed to behave. Moreover, the fail-stop failure of node p_j is a transient fault that results in the removal of (p_i, p_j) from the network and p_j from $N_c(i)$, for every $p_i \in N_c(j)$. A transient fault can also corrupt the state of the nodes or the communication channels. We assume that such transient faults can only occur before the system starts running. Namely, during the system run, G_c does not change and it is $(\kappa + 1)$-edge connected.

We consider a system in which $maxRules$ is large enough to store all the rules that all controllers need to install to any given switch, and that $maxManagers \geq N_C$. We assume that $|P_C|$ and $|P_S|$ are known only by their upper bounds, i.e., $N_C \geq |P_C|$, and respectively, $N_S \geq |P_S|$. We use these bounds only for estimating the memory requirements per node, in terms of $maxRules$ and $maxManagers$, i.e., the maximum number of rules, and respectively, managers at any switch.

Suppose that a κ-fault-resilient flow from p_i to p_j is installed in the network. The term *primary path* refers to the path along which the network forwards packets from p_i to p_j *in the absence of failures*. We assume that $myRules()$ returns rules that encode κ-fault-resilient flows for a given network topology. The primary paths encoded by $myRules()$ are also the shortest paths in G_c (with the highest rule priority). A rule in $myRules()$ corresponding to k link failures (k-fault-resilient flow) has the $(k + 1)$-highest rule priority.

3.3.1 Communication Fairness

Due to the asynchronous nature of the system, we do not consider any bound on the communication delay, which could be, for example, the result of the absence of properly installed flows between the sender and the receiver. Nevertheless, when a flow is properly installed, the channel is not disconnected and thus we assume that sending a packet infinitely often implies its reception infinitely

often. We refer to the latter assumption as the *communication fairness* property. We make the same assumptions both for the link and transport layers.

3.3.2 Message Round-Trips

This work proposes a solution for bootstrapping in-band communication in SDNs. The correctness proof depends on the nodes' ability of exchanging messages during this bootstrapping. The proof uses the notation of message round-trip which we next define in detail. Let $p_i \in P_C$ be a controller and $p_j \in P \backslash \{p_i\}$ be a network node. Suppose that immediately after state c node p_i sends a message m to p_j, for which p_i awaits a response. At state c', that follows state c, p_j receives message m and sends a response message r_m to p_i. Then, at state c'', that follows state c', p_i receives p_j's response, r_m. In this case, we say that p_i has completed with p_j a round-trip of message m. Let P_i be the set of nodes with whom p_i completes a message round trip infinitely often in execution R. Suppose that immediately after the state c_{begin}, controller p_i takes a step that includes the execution of the first line of the do forever loop, and immediately after system state c_{end}, it holds that: (i) p_i has completed the iteration it has started immediately after c_{begin} (regardless of whether it enters branches) and (ii) every message m that p_i has sent to any node $p_j \in P_i$ during the iteration (that has started immediately after c_{begin}) has completed its round trip. In this case, we say that p_i's iteration (with round-trips) starts at c_{begin} and ends at c_{end}.

3.4 Self-Stabilization

We define the system's task by a set of executions called *legal executions* (LE) in which the task's requirements hold. That is, each controller p_i constructs a κ-fault-resilient flow to every node $p_j \in P$ (either a switch or a controller). We say that a system state c is *legitimate*, when every execution R that starts from c is in LE. A system is *self-stabilizing* [11] with relation to task LE, when every (unbounded) system execution reaches a legitimate state with relation to LE (cf. Figure 3).

3.4.1 Execution Fairness

We say that a system execution is *fair* when every step that is applicable infinitely often is executed infinitely often and fair communication is kept (both at the link and the transport layer). Note that only failing nodes ever stop taking steps and thus a violation of the fairness (communication or execution) assumptions implies the presence of transient faults, which we assume to happen only before the starting system state of any execution.

3.4.2 Asynchronous Frames

The first *(asynchronous) frame* in a fair execution $R = R' \circ R''$ is the shortest prefix R' of R, such that each controller starts and ends at least one complete iteration (with round-trips) during R' (see Section 3.3.2), where $\circ$ denotes an operation that concatenates two executions. The second frame in execution R is the first frame in execution R'', and so on.

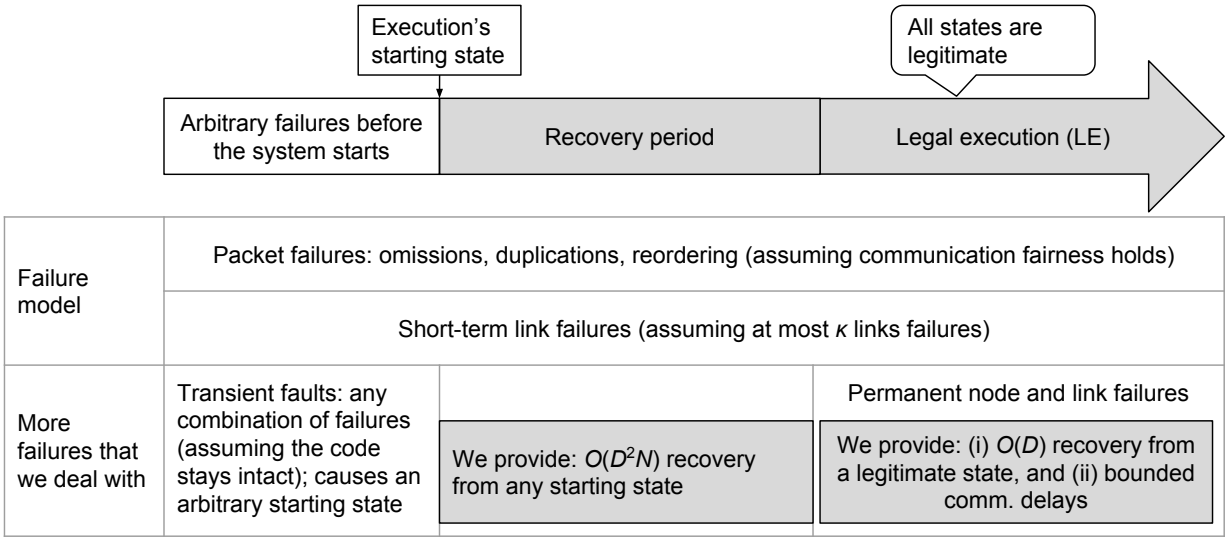

Figure 3: System execution, faults, and recovery guarantees of the proposed self-stabilizing algorithm.

3.4.3 Complexity Measures

The *stabilization time* (or recovery period) of a self-stabilizing system is the number of asynchronous frames it takes a fair execution to reach a legitimate system state when starting from an arbitrary one. Self-stabilizing systems require the use of bounded memory because real-world systems have only access to bounded memory. Moreover, the number of messages sent during an execution does not have an immediate relevance in the context of self-stabilization. The reason is that self-stabilizing algorithms can never terminate and stop sending messages, because if they did it would not be possible for the system to recover from transient faults (cf. [11, Chapter 2.3]). That is, suppose that the algorithm includes a predicate, such that when the predicate is true the algorithm forever stops sending messages. Then, a single transient fault can cause this predicate to be true in the starting state of an execution, from which the system can never recover. The latter holds because the algorithm will never send any messages and yet in the starting system state any variable that is not considered by the predicate can be corrupted.

4 Renaissance: A Self-Stabilizing SDN Control Plane

We present a self-stabilizing SDN control plane, called *Renaissance*, that enables each controller to discover the network, remove any stale information in the configuration of the discovered unmanaged switches (e.g., rules of failed controllers), and construct a κ-fault-resilient flow to any other node (switch or controller) that it discovers in the network.

4.1 High-level Description of the Proposed Algorithm

Algorithm 1 creates an iterative process of topology discovery that, first, lets each controller identify the set of nodes that it is directly connected to; from there, it finds the nodes that are directly connected to them; and so on. This network discovery process is combined with another process for

Algorithm 1: Self-stabilizing SDN, high-level code description for controller p_i.

1 **Local state:** $responses \subseteq \{m(j) : p_j \in P\}$ has the most recently received query replies;
2 $currTag$ and $prevTag$ are p_i's current and previous synchronization round, respectively;
3 **Interface:** $myRules(G, j, tag)$: returns the rules of p_i on switch p_j given a topology G on round tag;
4 **do forever begin**
5 Remove from $responses$ any reply from unreachable senders or not from round $prevTag$ or $currTag$. Also, remove from $responses$ any response from p_i and then add a record that includes the directly connected neighbors, $N_c(i)$;
6 **if** *responses includes a reply (with tag currTag) from every node that is reachable according to the accumulated local topology, G, in responses* **then**
7 Store $currTag$'s value in $prevTag$ and get a new and unique tag for $currTag$;
8 **foreach** *switch* $p_j \in P_S$ *and* p_j*'s most recently received reply* **do**
9 **if** *this is the start of a new synchronization round* **then**
10 Remove from p_j's configuration any manager p_k or rule of p_k that was not discovered to be reachable during round $prevTag$;
11 Add p_i in p_j's managers (if it is not already included) and replace p_i's rules in p_j with $myRules(G, j, tag)$;
12 **foreach** $p_j \in P$ *that is reachable from* p_i *according to the most recently received replies in responses* **do**
13 **send to** p_j (with tag $currTag$) an update message (if $p_j \in P_S$ is a switch) and query p_j's configuration;
14 **upon query reply** m **from** p_j **begin**
15 **if** *there is no space in responses for storing* m **then**
16 perform a C-reset by including in $responses$ only the direct neighborhood, $N_c(i)$
17 **if** m's tag equals to $currTag$ **then** include m in $responses$ after removing the previous response from p_j;
18 **upon arrival of a query (with a** $syncTag$**) from** p_j **begin**
19 **send to** p_j a response that includes the local topology, $N_c(i)$, and $syncTag$

bootstrapping communication between any controller and any node in the network, i.e., connecting each controller to its direct neighbors, and then to their direct neighbors, and so on, until it is connected to the entire reachable network.

Each controller associates independently each iteration with a unique tag [2] that synchronizes a round in which the controller performs configuration updates and queries. Controller p_i also maintains the variables $currTag$ and $prevTag$ (line 2) of the round synchronization procedure, which starts when p_i queries all reachable nodes and ends when it receives replies from all of these nodes (cf. lines 6–7, as well as, Section 3). Upon receiving a query response, p_i runs lines 14–17 and replies to other controllers' queries in lines 19–18.

A controller $p_i \in P_C$ keeps a local state of query responses (cf. Section 2.1) from other nodes (line 1). These responses allow p_i to accumulate information about the network topology according to which the switch configurations are updated in each round. The following three basic functionalities of Algorithm 1 are provided by the do-forever loop in lines 4–13, which we detail below.

4.1.1 Establishing communication between every controller and every other node

A controller $p_i \in P_C$ can communicate and manage a switch $p_j \in P_S$ only after p_i has installed rules at all the switches on a path between p_i and p_j. This, of course, depends on whether there are no permanent link failures on the path. In order to discover these link failures, we use local mechanisms for failure detection at each node for querying about the status of every link (cf. Section 2.2.1). These mechanisms consider any permanent link failure as a transient fault and we assume that Algorithm 1 starts running only after the last occurrence of any transient fault (cf. Figure 3). Thus, as soon as there is a flow installed between p_i and p_j and there are no permanent failures on the primary path (Section 3), p_i and p_j can exchange massages that arrive *eventually*.

The above iterative process of network topology discovery and the process of rule installation consider κ-fault-resilient flows (cf. Section 2.2.2 and $myRules()$ function in Section 3). These flows are computed through the interface $myRules(G, j, tag)$ (line 3), where G is the input topology, p_j is the switch to store these rules, and tag is the tag of the synchronization round. Once the entire network topology is discovered, Algorithm 1 guarantees the installation of a κ-fault-resilient flow between p_i and p_j. Thus, once the system is in a legitimate state, the availability of κ-fault-resilient flows implies that the system is resilient to the occurrence of at most κ temporary link failures (and recoveries) and p_i can communicate with any node in the network within a bounded time.

4.1.2 Discovering the network topology and dealing with unreachable nodes

Algorithm 1 lets the controllers connect to each other via κ-fault-resilient flows. Moreover, Algorithm 1 can detect situations in which controller $p_k \notin P_C$ is not reachable from controller p_i (line 5). The reason is that p_i is guaranteed to (i) discover the entire network eventually, and (ii) communicate with any node in the network. This means that p_i eventually gets a response from every node in the network. Once that happens, the set of nodes that respond to p_i equals to the set of nodes that were discovered by p_i (line 6) and thus p_i can restart the process of discovering the network (line 7).

The start of a new round (in which p_i rediscovers the network) allows p_i to also remove information at the switches that is related to any unreachable controller $p_k \in P_C$, only when it has succeeded in discovering the network and bootstrapped communication. We note that, during new rounds (line 9), p_i removes information related to p_k from any switch p_j (line 10); whether this information is a rule or p_k's membership in p_j's management set. This stale information clean-up eventually brings the system to a legitimate state, as we will prove in Section 5.

Recall that we regard the long-term failure of links (or of more than κ links) as transient faults. After the occurrence of the last transient fault, the network returns to fulfill our assumptions about the topology G_c, i.e., G_c is $(\kappa + 1)$-edge connected. Then, Algorithm 1 brings the system back to a legitimate state (Section 5). The do-forever loop of Algorithm 1 completes by sending rule and manager updates to every switch that has a reply in $responses$, as well as querying every reachable node, with the current synchronization round's tag (lines 12–13).

4.1.3 Keeping the Switch Accessible

In order to render the network self-stabilizing, we ensure that initial misconfigurations are removed eventually. In particular, a switch may initially store rules which block any arriving packet, potentially making it unmanageable. Our algorithm solves this problem by interpreting any packet arriving at the proposed abstract switch (a simple extension of existing switch functionality) as a

Algorithm 2: Self-stabilizing SDN, code for controller p_i (Algorithm 1's details).

Symbols and operators: '$\bullet$' stands for 'any sequence of values', () is the empty sequence, $\circ$ (binary) is the sequence concatenation operator and $\bigcirc$ (unary) concatenates a set's items in an arbitrary order.
Constants: $N_c(i) \subseteq P$, p_i's directly connected nodes; $maxRules$ and $maxManagers$, maximum number of rules and managers, respectively; $maxResponses$: maximum size of the set $responses$;
Local state: $responses \subseteq \{m(j) : p_j \in P\}$ has the most recently received query replies $m(j)$, $p_j \in P$, where $m(j) := \langle j, N_c(j), manager(j), rules(j)\rangle$, $N_c(j)$ is p_j's neighborhood, $manager(j) \subseteq P_C$ has p_j's controllers, and $rules(j) \subseteq \{\langle k, j, src, dest, prt, z, tag\rangle : (p_k, p_j, p_z, p_{dest} \in P) \wedge p_{src} \in P_C \wedge prt \in \{0, \ldots, n_{prt}\} \wedge tag \in tagDomain\}$ is p_j's rule set;
$currTag$ and $prevTag$ are p_i's current, and respectively, previous synchronization round tags;
Interfaces: Section 2.2.2, Figure 4, as well as the following:
$myRules(G, j, tag)$: creates p_i's rules at switch p_j according to G with tag tag (cf.Section 2.2.2);
Macros: $res(x) = \{\langle \bullet, rules(j)\rangle \in responses : \forall_{r \in rules(j)}\, r = \langle \bullet, x\rangle\} \cup \{\langle i, N_c(i), \emptyset, \emptyset\rangle\}$;
$G(S) := (\{p_k : \exists_{\langle j, N_c(j), \bullet\rangle \in S} : (k = j \vee p_k \in N_c(j))\}, \{(j,k) : \exists_{\langle j, N_c(j), \bullet\rangle \in S} : (p_k \in N_c(j))\})$;
$fusion := res(currTag) \cup \{\langle k, \bullet, prevTag\rangle \in res(prevTag) : \langle k, \bullet, currTag\rangle \notin res(currTag)\}$;
$p_j \rightarrow_G p_k :=$ true if there is a path from p_j to p_k in G;

do forever begin

`/* Remove replies from unreachable senders or not from round` $prevTag$ `or` $currTag$`. */`

$responses \leftarrow \{\langle k, \bullet, rules\rangle \in responses : k \neq i \wedge (\exists_{x \in \{currTag, prevTag\}} \langle k, \bullet, rules\rangle \in res(x) \wedge p_i \rightarrow_{G(res(x))} p_k \wedge \langle i, \bullet, x\rangle \in rules)\} \cup \{\langle i, N_c(i), \emptyset, \emptyset\rangle\}$;

let $(newRound, msg) := (false, \emptyset)$; `/*` $newRound$ `and` msg `get their default values */`

`/* a new round with a new tag; remove responses with tag` $currTag$ `*/`

if $(\forall p_\ell : p_i \rightarrow_{G(res(currTag))} p_\ell \implies \langle \ell, \bullet\rangle \in res(currTag)$ **then**

  $(newRound, prevTag) \leftarrow (true, currTag)$; $currTag \leftarrow nextTag()$;

  $responses \leftarrow responses \setminus \{\langle j, \bullet\rangle \in res(currTag) : p_j \in P\}$;

`/* The reference tag,` $referTag$`, is` $currTag$ `when a topology change is discovered */`

if $G(fusion) = G(res(prevTag))$ **then let** $referTag := prevTag$ **else let** $referTag := currTag$;

foreach $p_j \in P_S : \langle j, Ngb, Mng, Rul\rangle \in res(referTag)$ **do** `/* manage switch` p_j`'s rules */`

  `/*` p_i `is switch` p_j`'s manager; remove unreachable managers on new rounds and nodes with no rules */`

  let $M := \{p_k \in Mng : (\exists_{r \in Rul}\, r = \langle k, \bullet\rangle) \wedge (\neg newRound \vee p_i \rightarrow_{G(res(prevTag))} p_k)\} \cup \{p_i\}$;

  $msg \leftarrow msg \cup \{(p_j, \langle \text{`}delMngr\text{'}, k\rangle) : p_k \in (Mng \setminus M)\} \cup \{(p_j, \langle \text{`}addMngr\text{'}, i\rangle)\}$;

  `/* Remove any` p_j`'s rule that is associated with an unreachable node,` p_k `*/`

  $msg \leftarrow msg \cup \{(p_j, \langle \text{`}delAllRules\text{'}, k\rangle) : (\exists_{r \in Rul}\, r = \langle k, \bullet\rangle) \wedge p_k \notin M\}$;

  `/*` p_i `refreshes all of its rules at switch` p_j `according to` $referTag$ `*/`

  $msg \leftarrow msg \cup \{(p_j, \langle \text{`}updateRules\text{'}, myRules(G(res(referTag)), j, currTag)\rangle)\}$;

`/* Send the prepared messages to all reachable nodes in an aggregated form */`

foreach $p_j : p_i \rightarrow_{G(fusion)} p_j$ **do**

  send $(\langle \text{`}newRound\text{'}, currTag\rangle) \circ [\bigcirc_{m:(p_j,m) \in msg}(m)] \circ (\langle \text{`}query\text{'}, currTag\rangle)$ **to** p_j;

upon query reply $m := \langle j, \bullet, rls\rangle$ **from** p_j **begin**

`/*` p_i `tests that` m`'s tag matches` $prevTag$ `and there is room to store` m `*/`

if $|responses \cup \{m\}| > maxResponses$ **then** $responses \leftarrow \{\langle i, N_c(i), \emptyset, \emptyset\rangle\}$; `/* C-reset */`

if $(\exists_{r \in rls}\, r = \langle \bullet, currTag\rangle)$ **then** $responses \leftarrow (responses \setminus \{\langle j, \bullet\rangle\}) \cup \{m\}$;

upon arrival of $(\bullet \circ (\langle \text{`}query\text{'}, tag\rangle))$ **from** p_j **do send** $\langle i, N_c(i), \perp, \{\langle j, i, \perp, \perp, \perp, \perp, tag\rangle\}\rangle$ **to** p_j

control packet by default (if not matched otherwise). In particular, we leverage the limited size of the switch memory and disallow wildcarding on a specific field used for control traffic, rendering it impossible to disallow all control traffic given the limited number of rules (cf. Section 2.1).

4.2 Refining the Model: Variables, Building Blocks, Interfaces

With these intuitions in mind and before presenting our algorithm in detail, we introduce some more notation, interfaces, and building blocks. Throughout this section we refer to Algorithm 2 (the detailed version of Algorithm 1).

Local Variables Each controller's state includes *responses* (line 3), which is the set of the most recent query replies, and the tags $currTag$ and $prevTag$, which are p_i's current, and respectively, previous synchronization round tags. Each response $m(j) : p_j \in P$ can arrive from either a switch or another controller and it has the form $\langle j, N_c(j), manager(j), rules(j)\rangle$. The code denotes by $N_c(j)$ the neighborhood of p_j, by $manager(j) \subseteq P_C$ the controllers of p_j, and by $rules(j) \subseteq \{\langle k, j,\ src,\ dest,\ prt,\ z, tag\rangle : (p_k, p_j, p_z, p_{dest} \in P) \wedge (p_{src} \in P_C) \wedge prt \in \{0, \ldots, n_{prt}\} \wedge tag \in tagDomain\}$ the rule set of p_j. We assume that the size of *responses* is bounded by $maxResponses \geq 2(N_C + N_S)$, hence the local state has bounded size (the factor of 2 is due to responses from the rounds $prevTag$ and $currTag$).

An Internal Building Block: Round Synchronization An SDN controller accesses the abstract switch in synchronized rounds. Each round has a unique tag that distinguishes the given round from its predecessors. We assume access to a self-stabilizing algorithm that generates unique *tags* of bounded size from a finite domain of tags, $tagDomain$. The algorithm provides a function called $nextTag()$ that, during a legal execution, returns a unique tag. That is, immediately before calling $nextTag()$ there is no tag anywhere in the system that has the returned value from that call. Given two tags, t_1 and t_2, we require that $t_1 = t_2$ holds if, and only if, they have identical values. We use these tags for synchronizing the rounds in which the controllers perform configuration updates and queries. Namely, in the beginning of a round, controller $p_i \in P_C$ generates a new tag and stores that tag in the variable $currTag \leftarrow nextTag()$. Controller p_i then attempts to install at every reachable switch $p_j \in P_S$ a special meta-rule $\langle i, j, \perp, \perp, n_{prt}, \perp, t_{metaRule}\rangle$, which includes, in addition to p_i's identity, the tag $t_{metaRule} = currTag$ and has the lowest priority (before making any configuration update on that switch). It then sends a query to all (possibly) reachable nodes in the network and combines that query with the tag $t_{query} = currTag$. The response to that query from other controllers $p_j \in P_C$ includes the query tag, t_{query}. The response to the query from the switch $p_k \in P_S$ includes the tag $t_{metaRule}$ of the most recently installed meta-rule that p_k has in its configuration. The controller p_i ends its current round once it has received a response from every (possibly) reachable node in the network and that response has the tag of $currTag$.

We note the existence of self-stabilizing algorithms, such as the one by Alon et al. [2], that in fair executions (that are legal with respect to the self-stabilizing end-to-end communication protocol) provide unique tags within a number of synchronization rounds that is bounded (by a constant whenever the execution is legal with respect to the self-stabilizing end-to-end communication protocol). We refer to that known bound by Δ_{synch} and note that during a legal execution of the round synchronization algorithm, it holds that controller p_i receives only a response message m that matches $currTag$, i.e., it discards any message with different tag. Moreover, since during legal executions $nextTag()$ returns only unique tags, m and its acknowledgment are guaranteed to form a complete round-trip. Note that we do not require $nextTag()$ to support concurrent calls since every controller manages its own synchronization rounds; one round at a time.

Command type	Command	Switch p_j's control module action
new round	$\langle$'$newRound$', $t_{metaRule}\rangle$	updates current synchronization tag of the switch
update command	$\langle$'$delMngr$', $k\rangle$	deletes p_k from $manager(j)$
	$\langle$'$addMngr$', $k\rangle$	adds p_k in $manager(j)$
	$\langle$'$delAllRules$', $k\rangle$	deletes all rules of p_k
	$\langle$'$updateRules$', $newRules\rangle$	replaces all rules of p_i with $newRules$
query command	$\langle$'$query$', $t_{query}\rangle$	sends query response $m(j)$ to p_i

Figure 4: Abstract switch p_j's control module interface, for each controller $p_i \in P_C$.

Interfaces Controller p_i can send requests or *queries* to any other node p_j (which could be either another controller or a switch). We detail the switch interface below and illustrate it in Figure 4.

The controllers send command batches, which are sequences of commands. The special meta-data command $\langle$'$newRound$', $t_{metaRule}\rangle$ is always the first command and updates the special meta-rule to store $t_{metaRule}$. We use it for starting a new round (where $t_{metaRule} = t$ is the round's tag). This starting command could be followed by a number of commands, such as $\langle$'$delMngr$', $k\rangle$ for the removal of controller p_k from the management of switch p_j, $\langle$'$addMngr$', $k\rangle$ for the addition of controller p_k from the management of switch p_j, and $\langle$'$delAllRules$', $k\rangle$ for the deletion of all of p_k's rules from the configuration of switch p_j, where $p_k \in P_C \setminus \{p_i\}$. The rules' update is done via $\langle$'$updateRules$', $newRules\rangle$ and it is the second last command. This update replaces all of p_i's rules at switch p_j (except for the special meta-rule) with the rules in $newRules$. These commands are to be followed by the round's query $\langle$'$query$', $t_{query}\rangle$, where $t_{query} = t$ is the query's tag. The switch p_j replies to a query by sending $m = \langle j, N_c(j),\ manager(j),\ rules(j)\rangle$ to p_i, such that the rule set includes also the special meta-rule $\langle i, \bullet, t\rangle \in rules(j)$. Whenever $p_j \in P_C$ is another controller, response to a query is simply $\langle i, N_c(i), \perp, \{\langle j, i, \perp, \perp, \perp, \perp, t_{query}\rangle\}\rangle$ (line 28). Note that controller p_j simply ignores all other types of commands. We use the interface function $myRules(G, j, tag)$ (Section 2.2.2) for creating the packet forwarding rules that controller p_i installs at switch p_j when p_i's current view on the network topology is G in round tag (line 6).

4.3 Algorithm Details

Algorithm 2 presents the proposed solution with a greater degree of details than Algorithm 1. Algorithm 2 is centered around a *do forever* loop, which starts by removing stale information from *responses* (line 12). This removal action includes refreshing information related to controller p_i, which deletes information about any node that is not reachable from p_i. The reachability test uses the currently known information about the network topology, G (line 8), and the relation $\rightarrow_G$ (line 10) that tells whether node p_j is reachable from controller p_i in G, given the information in *responses*.

Algorithm 2 accesses the switch configurations in synchronization rounds. Lines 13–16 manage the start (and end) of synchronization rounds. When a new round starts, i.e., the condition of the if-statement of line 14 holds, controller p_i marks the start of a new round ($newRound_i = true$), updates the values of the tags $prevTag_i$ and $currTag_i$ and clears any record with tag $currTag$ of the responses stored in $responses_i$ (line 15 and 16).

Algorithm 2 refreshes (and reconstructs) the information about remote nodes (controllers and switches including the ones that are directly attached to it) by sending queries (line 24) and updating the set of stored responses (line 27). Notice that controller p_i also responds to query requests coming

from other controllers (line 28). Algorithm 2 uses this information for completing the information about the switches that are directly connected to a remote controller (and thus the other fields in the response messages are the empty sets).

The heart of Algorithm 2 includes the updates of every switch $p_j \in P_S$ (line 18 to 21). For every switch p_j (line 18), controller p_i considers p_j's stored response $\langle j, Ngb_i, Mng_i, Rul_i \rangle$ for which it prepares a set of commands to be stored in the set msg_i (lines 13, 20, 21, 22 and 24). To that end, p_i first calculates the set of managers that p_j should have in the following manner. If this iteration of the do forever loop (lines 11 to 24) is the first one for the round $currTag_i$, the value of $newRound_i$ is *true* (line 15); this leads p_i to remove any controller p_k that is not reachable according to $G(res(prevTag))$ (lines 19 to 21). Whenever the iteration is not the first one, p_i merely asserts that it is a manager of p_j.

Controller p_i removes any rules of an unreachable controller p_k (line 21) and updates all of its rules at switch p_j (line 22) using the interface function $myRules()$ (line 22) and the reference tag, *referTag* (line 17). The proposed algorithm selects *referTag*'s value to be $prevTag$ during legal executions. During recovery periods, the discovered topology can differ from that one that is stored with the tag $prevTag$. In that case, the algorithm selects $currTag$ as the reference tag. After preparing these commands to all the switches, controller p_i prepares query commands to all reachable nodes (including both controllers and switches) and then sends all prepared commands to their designated destinations. Note that each of these configuration updates are done via a single message that aggregates all commands for a given destination (line 24).

We note that when a query response arrives at p_i, before the update of the response set (line 27), p_i checks that there is sufficient storage space for the arriving response (line 26). If space is lacking, p_i performs what we call a 'C-reset'. Note that p_i stores responses only for the current synchronization round, $currTag$.

5 Correctness Proof

We prove the correctness of Algorithm 2 by showing that when the system starts in an arbitrary state, it reaches a legitimate state (Definition 1) within a bounded period of $((\Delta_{comm} + \Delta_{synch}) + 2)D+1)[((\Delta_{comm}+\Delta_{synch})D+1)\cdot N_S+N_C+1]$ frames (Theorem 2). Moreover, we show that when starting from a legitimate state, the system satisfies the task requirements and it is also resilient to a bounded number of failures (lemmas 6 and 7).

We refer to the values of variable X at node p_i (controller or switch) as X_i, i.e., the variable name with a subscript that indicates the node index. Similarly, we refer to the return values of function f at controller p_k as f_k.

Definition 1 (Legitimate System State)**.** *State $c \in R$ is legitimate with respect to Algorithm 2 when $\forall p_i \in P_C$ and $\forall p_k \in P \setminus \{p_i\}$ the following holds.*

1. *$\langle k, N_c(k), manager(k), rules(k) \rangle \in responses_i$ if, and only if, $N_c(k)$, $manager(k)$, and $rules(k)$ are p_k's neighborhood, managers, and respectively, set of packet forwarding rules (line 3) as well as $p_i \rightarrow_G p_k$ (line 10). Moreover, for the case of controller $p_k \in P_C$, the task does not require p_k to have any managers or rules, i.e., $manager(k) = \emptyset$ and $rules(k) = \emptyset$.*

2. *Any controller is the manager of every switch and only these controllers can be the mangers of any switch, i.e., $p_i \in P_C \wedge p_k \in P_S \iff p_i \in manager(k)$.*

3. *The rules installed in the switches encode κ-fault-resilient flows between controller p_i and node p_k in the network G_c (Section 2.2.2).*

4. *The end-to-end protocol (Section 3.1) as well as the round synchronization protocol (Section 2.2.1) between p_i and p_k are in a legitimate state.*

5.1 Overview

The proof of Theorem 2, starts by establishing bounds on the number of rules that each switch needs to store (Lemma 1). The proof arguments are based on the bounded network size and the memory management scheme of the abstract switch (Section 2.1.1), which guarantees that, during a legal execution, all non-failing controllers are able to store their rules (Lemma 1). The bounded network size also helps to bound, during a legal execution, the amount of memory that each controller needs to have (Lemma 2). This proof also bounds the number of C-resets that a controller might take (line 26) during the period in which the system recovers from transient faults. This is line 16 in Algorithm 1. Note that this bound on the number of C-resets is important because C-resets delete all the information that a controller has about the network state.

C-resets are not the only disturbing actions that might occur during the recovery period. The system cannot reach a legitimate state before it removes stale information from the configuration of every switch. Note that failing controllers cannot remove stale information that is associated with them and therefore non-failing controllers have to remove this information for them. Due to transient faults, it could be the case that one controller can remove information that is associated with another non-failing controller. We refer to these 'mistakes' as illegitimate deletion of rules or managers (Section 5.3). Note that illegitimate deletions occur when the (stale) information that a controller has about the network topology differ from the actual network topology, G_c. Moreover, due to stale information in the communication channels, any given controller might aggregate (possibly stale) information about the network more than once and thus instruct more than once the switch to delete illegitimately the rules of other controllers.

Theorem 1 bounds the number of these illegitimate deletions. It does so by counting the number of possible steps in which a controller might have stale information about the network and that stale information leads the controller to perform an illegal deletion. The proof arguments start by considering a starting state in which the controller is just about to take a step that instructs the switches to perform an illegitimate deletion. The proof then argues that between any two resets, the controller has to aggregate information about the network in such a way that it preserves that it has the complete network topology. But, this can only happen after receiving a reply from every node in the preserved topology (Claim 5.1). By induction on the distance k between controller $p_i \in P_c$ and node $p_j \in P \setminus \{p_i\}$, the proof shows that the information that p_i has about p_j is correct within $k \cdot (\Delta_{comm} + \Delta_{synch} + 1) + 1$ times in which p_i instruct the switches to perform an illegitimate deletion, because there is a bounded number of stale information in the communication channel between p_i and p_j (Lemma 3). Thus, the total number of illegitimate deletions is at most $D \cdot (\Delta_{comm} + \Delta_{synch} + 1) + 1$.

The proof demonstrates recovery from transient faults by considering a period in which there are no C-resets and no illegitimate deletions (Section 5.4). In such a period, all the controllers construct κ-fault-resilient flows to any other node in the network (Lemma 4). This part of the proof is again by induction on the distance k between controller $p_i \in P_c$ and node $p_j \in P \setminus \{p_i\}$. The induction shows that, within $((\Delta_{comm} + \Delta_{synch}) + 2)k$ frames, p_i discovers correctly its k-

distance neighborhood and establishes a communication channel between p_i and p_j. This means that within $((\Delta_{comm} + \Delta_{synch}) + 2)D$ frames in which there are no C-resets and no illegitimate deletions, the system reaches a legitimate state (Lemma 5).

The above allows Theorem 2 to show that within $((\Delta_{comm} + \Delta_{synch}) + 2)D + 1)[((\Delta_{comm} + \Delta_{synch})D + 1) \cdot N_S + N_C + 1]$ frames in R, there is a period of $((\Delta_{comm} + \Delta_{synch}) + 2)D + 1)$ frames in which there are no C-resets and no illegitimate deletions and thus the system reaches a legitimate state. Lemma 6 shows that, when starting from a legitimate state an then letting a single link in the network to be added or remove from G_c, the system recovers within $O(D)$ frames. The arguments here consider that number of frames it takes for each controller to notice the change and to update all the switches. By similar arguments, Lemma 7 shows that after the addition or removal of at most $N_C - 1$ controllers, the system recovers in a legitimate system state within $O(D)$ frames.

5.2 Analysis of Memory Requirements

Lemmas 1 and 2 bound the needed memory at every node during a legal execution. Recall that we assume that the switches implement a mechanism for dealing with clogged memory (Section 2.1.1), such that once controller $p_i \in P_C$ refreshes its rules on a given switch, that switch never removes p_i's rules.

Lemma 1 considers an event that can delay recovery, i.e., the removal of a rule at a switch due to lack of space. Lemma 1 bounds the needed memory for every switch, and thus relates to events that can delay recovery, i.e., the removal of a rule at a switch due to lack of space.

Lemma 1 (Bounded Switch Memory)**.** *(i) Suppose that R is a legal execution of Algorithm 2. A switch needs to let no more than $maxManagers \geq N_C$ controllers to manage it and (2) no more than $maxRules \geq N_C \cdot (N_C + N_S - 1) \cdot n_{prt}$ packet forwarding rules.*

Proof. Let $p_j \in P_S$ be a switch.
Number of managers. Recall that we assume that $maxManagers \geq N_C \geq |P_C|$, i.e., the bound is large enough to store all managers (once all stale information is removed in a FIFO manner that is explained in Section 2.1.1). During a legal execution R of Algorithm 2, every controller accesses every switch repeatedly (line 24). This way, every $p_i \in P_C$, is always among the N_C most recently installed controllers at $p_j \in P_S$.
Number of rules. Recall that a rule is a tuple of the form $\langle k, i, src, dest, prt, j, tag \rangle$, where $p_k \in P_C$ is the controller that created this rule, $p_i \in P_S$ is the switch that stores this rule, $p_{src} \in P_C$ and $p_{dest} \in P$ are the source, and respectively, the destination of the packet, prt is the packet's priority, $p_j \in P$ is the relay node (i.e., the rule's action field) and tag is the synchronization round tag.

To show that there are no more than $N_C \cdot (N_C + N_S - 1) \cdot n_{prt}$ rules that a switch needs to store, recall that each of the N_C controllers $p_{src} \in P_C$ constructs κ-fault-resilient flows to every node $p_{dest} \in P \setminus \{p_{src}\}$ in the network. Thus, switch $p_i \in P_S$ might be a hop on the κ-fault-resilient flow between p_{src} and p_{dest}. That is, there are at most $N_C \cdot (N_C + N_S - 1)$ such flows that pass via p_i, because for each of the N_C possible flow sources p_{src}, there are exactly $(N_C + N_S - 1)$ destinations p_{dest}. Each such flow stores at most $n_{prt} \geq \kappa + 1$ rules at p_i, i.e., one for each priority. Note that, during a legal execution, each switch $p_i \in P_S$ stores at most one tag per $p_{src} \in P_C$ (line 24). □

Lemma 2 considers an event C-reset, which can delay recovery.

Lemma 2 (Bounded Controller Memory)**.** *(1) Let $a_x \in R$ be the first step in which controller p_i runs lines 25–27 (upon query reply). For every state in R that follows step a_x, node p_i stores no more than maxResponses replies in the set $responses_i$. (2) Suppose that R is a legal execution. Controller $p_i \in P_C$ needs to store, in the set $responses_i$, no more than $maxResponses \geq 2 \cdot (N_C + N_S)$ items. (3) Suppose that R is any execution, which may start in an arbitrary state. Controller p_i performs a C-reset at most once in R, i.e., takes a step $a_{x'} \in R$ that includes the execution of line 26, in which the if-statement condition is true.*

Proof. **Part (1).** We note that p_i modifies $responses_i$ only in line 12 and line 16 in the do-forever loop (lines 11–24), and in lines 26 and 27 in the query reply procedure (lines 25–27). In line 12 and line 16, the size of $responses_i$ either decreases (possible only at the first step that p_i executes line 12 or line 16) or stays the same. Thus, the rest of this proof focuses only at lines 26 and 27, where the set $responses_i$ increases due to the addition of an incoming reply (line 27).

Let $a_{x'}$ be the first step in R, in which controller p_i executes lines 25–27 due to a message m_j that p_i receives from node p_j. By line 26, if $|responses_i \cup \{m_j\}| > maxResponses$ holds, then p_i performs a C-reset, i.e., sets $responses_i \leftarrow \{\langle i, N_c(i), \emptyset, \emptyset\rangle\}$, which implies that $|responses_i| = 1$ after the execution of line 26. Hence, after the execution of line 27 in step $a_{x'}$, $|responses_i| < maxResponses$ holds for the state $c_{x'+1}$, which follows $a_{x'}$ immediately. Similarly, since the size of $responses_i$ increases only when p_i executes line 27, for every step $a_{x''}$ and the system state $c_{x''+1}$ that appears in R after $c_{x'+1}$, it is true that $|responses_i| \leq maxResponses$ holds in $c_{x''+1}$, due to line 26. Thus, for every system state that follows the first step $a_{x'} \in R$, it holds that $|responses_i| \leq maxResponses$.
Part (2). Line 12 removes from $responses_i$ any response that its synchronization round tag is not in the set $\{prevTag_i, currTag_i\}$ and line 27 does not add to $responses_i$ a response that its synchronization round tag is not $currTag_i$. Moreover, line 16 makes sure that when finishing one synchronization round and then transitioning to the next one, $responses_i$ includes responses only with synchronization round tags that are $prevTag_i$. Therefore, there are no more than two synchronization round tags that could be simultaneously present in $responses_i$. Moreover, line 12 also removes any response from an unreachable node, because item 1 of Definition 1 holds in any system state of a legal execution. This further limits the set $responses_i$ to includes response from at most $N_C + N_S$ nodes. Therefore, $|responses_i| \leq 2 \cdot (N_C + N_S)$.
Part (3). Suppose that p_i does perform a C-reset during R. Once that happens, parts (i) and (ii) of this proof imply that this can never happen again. □

5.3 Bounding the Number of Illegitimate Deletions

We consider another kind of event that might delay recovery (Definition 2) and prove that it can occur a bounded number of times. Recall that Δ_{comm} is the number of frames in which the end-to-end protocol stabilizes (Section 3.1) and Δ_{synch} the number of frames in which the round synchronization mechanism stabilizes (Section 4.2).

Definition 2 (Illegitimate deletions)**.** *A switch p_j performs an illegitimate deletion when it removes a non-failing controller $p_\ell \in P_C$ from its manager set (or its rules), due to a command that it received from another controller $p_k \in P_C$.*

Theorem 1 (Bounded number of illegitimate deletions)**.** *Let $a_{x_k} \in R$ be the k-th step in which controller $p_i \in P_C$ executes lines 15–16 during execution R. Suppose that R includes at least $((\Delta_{comm} + \Delta_{synch})D + 1)$ such a_{x_k} steps, where D is the network diameter. Let R' be a prefix*

of $R = R' \circ R''$ that includes the steps $a_1, \ldots, a_{x_{(\Delta_{comm}+\Delta_{synch})D+1}} \in R'$ and R'' be the matching suffix. Controller p_i does not take steps $a_{s'_k} \in R''$ that send a message m_k to $p_j \in P_S$, such that p_j performs an illegitimate deletion (Definition 2) upon receiving m_k.

Proof. This proof uses Claim 5.1 and Lemma 3. Theorem 1 follows by the case of $k \geq D$ for Lemma 3 and then applying Part (ii) of Claim 5.1.

Claim 5.1. *(i) The condition in the if-statement of line 14 holds if, and only if, $V_{reported} = V_{reporting}$, where $V_{reported} = \{p_k : \exists_{\langle j, N_c(j), \bullet, rls\rangle \in responses_i}((k = j \vee p_k \in N_c(j)) \wedge \exists\langle i, j_k, \bullet, currTag_i\rangle \in rls)\} \cup \{\langle i, N_c(i), \emptyset, \emptyset\rangle\}$ and $V_{reporting} = \{p_j : \langle j, \bullet, rls\rangle \in responses_i \wedge (\exists\langle i, j_k, \bullet, currTag_i\rangle \in rls)\}$. (ii) Suppose that every node p_j in G_c has sent a response $\langle j, \bullet\rangle$ to p_i. Suppose that p_i stores these responses in $responses_i$ together with p_i's report about its directly connected neighborhood, $\langle i, N_c(i), \emptyset, \emptyset\rangle$, cf. lines 7 and 12. In this case, the condition in the if-statement of line 14 holds.*

Proof of Claim 5.1. **The proof of Part (i).** The condition in the if-statement of line 14 is $(\forall p_\ell : p_i \rightarrow_{G(res_i(currTag_i))} p_\ell \implies \langle \ell, \bullet\rangle \in res_i(currTag_i)$. When $V_{reported} = V_{reporting}$ holds, the following two claims also hold by the definition of these sets (and vice versa): (a) p_i's response is in $responses_i$, and (b) for every node p_j that was queried with tag $currTag_i$, such that before the query either p_j had a response in $responses_i$ or a direct neighbor of p_j had a response in $responses_i$, there exists a response from p_j in $responses_i$ with rules that have the tag $currTag_i$. Hence, the condition in the if-statement of line 14 is true.
The proof of Part (ii). This is just a particular case in which $P = V_{reported} = V_{reporting}$. □

Lemma 3. *Let $p_{j_k} \in P$ be a node that is at distance k from p_i in G_c, such that $p_{j_0}, p_{j_1}, \ldots, p_{j_k}$ is any shortest path from p_i to p_{j_k} and $p_{j_0} = p_i$. Let $c_{x_y} \in R$ be the system state that immediately follows step $a_{x_y} \in \{a_{x_1}, \ldots, a_{x_{k\cdot(\Delta_{comm}+\Delta_{synch})+1}}\} \subset R'$.*

1. *Let $\ell > k \cdot \Delta_{comm} + 1$. The system state c_{x_ℓ} is legal with respect to the end-to-end protocol of the channel between p_i and p_{j_k}, and it holds that $m = \langle j_k, \bullet\rangle$ is a message arriving from p_{j_k} through the channel to p_i, which is an acknowledgment for p_i's message to p_{j_k}.*

2. *Let $\ell > k \cdot (\Delta_{comm} + \Delta_{synch}) + 1$. The system state c_{x_ℓ} is legal with respect to the round synchronization protocol between p_i and p_{j_k}. That is, for any message $m = \langle j_k, \bullet, rls\rangle$ that arrives from the channel from p_{j_k} to p_i, it holds that $m \in responses_i \wedge \exists_{r \in rls}\, r = \langle i, j_k, \bullet, currTag_i\rangle$. Moreover, message m is an acknowledgement of a message m' that p_i has sent to p_{j_k} and together m' and m form a completed round-trip.*

Proof of Lemma 3. We note that the first step, a_{x_1} could occur due to the fact that the system starts in an arbitrary state in which the condition of the if-statement of line 14 holds, hence the addition of 1 in $k \cdot (\Delta_{comm} + \Delta_{synch})$. The proof is by induction on $k > 0$. That is, we consider the steps in $a_{x_y} \in \{a_{x_1}, \ldots, a_{x_{k\cdot(\Delta_{comm}+\Delta_{synch})+1}}\}$.
The base case of $k = 1$. Claim 5.1 says that the condition in the if-statement of line 14 holds if, and only if, $V_{reported} = V_{reporting}$, where $\{\langle i, N_c(i), \emptyset, \emptyset\rangle\} \subseteq V_{reported}$ (line 7). Therefore, for any $\ell > 1$, we have that $a_{x_\ell} \in \{a_{x_2}, \ldots, a_{x_{k\cdot(\Delta_{comm}+\Delta_{synch}+1)+1}}\}$ implies that $\{\langle i, N_c(i), \emptyset, \emptyset\rangle\} \subseteq V_{reporting}$ holds immediately before a_{x_ℓ}.

Claim 5.2. *Between $a_{x_{k-1}}$ and a_{x_k}, a message $\langle j_k, \bullet, rls\rangle : \exists_{r \in rls}\, r = \langle i, j_k, \bullet, currTag_i\rangle$ arrives from the channel from $p_{j_k} \in N_c(i)$ to p_i, which p_i stores in $responses_i$, where $k \geq 1$.*

Proof of Claim 5.2. During the step $a_{x_{k-1}}$, controller p_i removes any response $\langle j_k, \bullet, rls\rangle : \exists_{r \in rls}\, r = \langle i, j_k, \bullet, currTag_i\rangle$ (line 16) and the only way in which $\langle j_k, \bullet, rls\rangle : \exists_{r \in rls}\, r = \langle i, j_k, \bullet, currTag_i\rangle$ holds immediately before a_{x_k} is the following. Between $a_{x_{k-1}}$ and a_{x_k}, a message arrives through the channel from $p_{j_k} \in N_c(j_{k-1}) : j_0 = i$ to p_i, which p_i stores in $responses_i$ (line 27). This is true because no other line in the code that accesses $responses_i$ adds that message to $responses_i$ (cf. lines 12, 16, and 27). □

The proof of Part (1). It can be the case the p_i sends a message for which it receives a (false) acknowledgement from p_{j_1}, i.e., without having that message go through a complete round-trip. However, by Δ_{comm}'s definition (Section 3.1), that can occur at most Δ_{comm} times.
The proof of Part (2). It can be the case that p_i receives message m from p_{j_1} for which the following condition does not hold in c_{j_1}: $m = \langle \bullet, rls\rangle \in responses_i \wedge \exists_{r \in rls}\, r = \langle i, j_k, \bullet, currTag_i\rangle$. However, by Δ_{synch}'s definition (Section 2.2.2), that can occur at most Δ_{synch} times. The rest of the proof is implied by the properties of the round synchronization algorithm (Section 2.2.2).
The induction step. Suppose that, within more than $(\Delta_{comm}k+1)$ and $((\Delta_{comm}+\Delta_{synch})k+1)$ synchronization rounds from R's starting state, the system reaches a state in which conditions (1), and respectively, (2) hold with respect to some $k \geq 1$. We show that in $c_{x_{\Delta_{comm}(k+1)+1}}$ and $c_{x_{(\Delta_{comm}+\Delta_{synch})(k+1)+1}}$, conditions (1), and respectively, (2) hold with respect to $k+1$.
The proof of Part (1). Claim 5.1 says that the condition in the if-statement of line 14 holds if, and only if, $V_{reported} = V_{reporting}$. By the induction hypothesis, condition (2) holds with respect to k in $c_{x_{(\Delta_{comm}+\Delta_{synch})k+1}}$ and therefore $A(k+1) \cup \{\langle i, N_c(i), \emptyset, \emptyset\rangle\} \subseteq V_{reported}$, where $A(k) = \{\langle j_{k'}, N_c(j_{k'}), \bullet, rls\rangle : 1 < k' \leq k \wedge \exists_{r \in rls}\, r = \langle i, j_{k'}, \bullet, currTag_i\rangle\}$. Therefore, $a_{x_{(\Delta_{comm}+\Delta_{synch})(k+1)+2}} \in a_{x_2} \ldots a_{x_{k\cdot(\Delta_{comm}+\Delta_{synch}+1)+1}}$ implies that $A(k+1) \cup \{\langle i, N_c(i), \emptyset, \emptyset\rangle\} \subseteq V_{reporting}$ holds in the system state that appears in R immediately before $a_{x_{(\Delta_{comm}+\Delta_{synch})(k+1)+2}}$. Claim 5.2 implies the rest of the proof.
The proof of Part (2). The proof here follows by similar arguments to the ones that appear in the proof of item (2) of the base case. □

□

Part (iii) of Lemma 2 and Theorem 1 imply Corollary 1.

Corollary 1. *Any execution R of Algorithm 2 includes no more than N_C C-resets (Lemma 2) and $((\Delta_{comm} + \Delta_{synch})D + 1) \cdot N_S$ illegitimate deletions (Theorem 1).*

5.4 Recovery From Transient Faults

In this section we prove that Algorithm 2 is self-stabilizing. Lemma 4 shows that (under some conditions, such as reset freedom) controller p_i eventually discovers the local topology of a switch p_{j_k} that is at distance k from p_i in the graph G_c. This means that p_i has all the information that its needs for constructing (at least) a 0-fault-resilient flow to p_{j_k} and discover any switch $p_{j_{k+1}} \in N_c(p_{j_k})$ that is at distance $k+1$ from p_i. Then, Lemma 5 shows that, within a bounded number of frames, no stale information exists in the system. Theorem 2 combines Corollary 1 and Lemma 5 to show that, within a bounded number of frames, the system reaches a legitimate state from which only a legal execution may continue.

We start by giving some necessary definitions. Let G_i be the value of $G(referTag_i)$ (line 8 and line 17) that controller $p_i \in P_C$ computes in a step $a_x \in R$. We say that there is a path

between $p_i \in P$ and $p_j \in P$, when there exist $p_{j_0}, p_{j_1}, \ldots, p_{j_k} \in P$, such that (1) $p_{j_0} = p_i$, (2) $p_{j_k} = p_j$, (3) $p_{j_1}, \ldots, p_{j_{k-1}} \in P_S$, and (4) the rules installed by a controller $p_\ell \in P_C$ at the switches in $p_{j_1}, \ldots, p_{j_{k-1}}$ (and also p_i or p_j if they are also switches) forward packets from p_i to p_j as well as from p_j to p_i (when the respective links are operational). We say that two nodes $p_i \in P$ and $p_j \in P$ *can exchange packets*, when there is a path between p_i and p_j. Moreover, we say that the rules installed in the switches $p_s \in P_S$ *facilitate κ-fault-resilient flows between p_i and p_j*, if at the event of at most κ link failures there exists a path between p_i and p_j. Let p_x and p_y be two nodes in P and recall that we assume that every node $p_z \in P$ has a fixed ordering of its neighbors, i.e., $N_c(z) = \{p_{i_1}, \ldots, p_{i_{|N_c(z)|}}\}$. We define the *first shortest path* between p_x and p_y to be the shortest path between p_x and p_y that includes the nodes with minimum indices according to the neighborhood orderings (among all the shortest paths between these two nodes).

Lemma 4. *Let $p_i \in P_C$ be a controller and $p_{j_k} \in P$ be a node in P that is at distance k from p_i in G_c, such that $p_{j_0}, p_{j_1}, \ldots, p_{j_k}$ is the first shortest path from p_i to p_{j_k} and $p_{j_0} = p_i$ in G_c. Suppose that C-resets (Lemma 2) and illegitimate deletions (Theorem 1) do not occur in R. For every $k \geq 0$, and any system state that follows the first $((\Delta_{comm} + \Delta_{synch}) + 2)k$ frames from the beginning of R, the following hold.*

1. *$\langle j_k, N_c(j_k), manager_i(j_k), rules_i(j_k)\rangle \in res_i(prevTag_i)$, where $N_c(j_k)$, $manager_i(j_k)$, and $rules_i(j_k)$ are p_{j_k}'s neighborhood, managers, and rules that p_i has received from p_{j_k}, respectively. Moreover, for the case of controller $p_{j_k} \in P_C$, it holds that $manager(j_k) = \emptyset$ and $rules(j_k) = \emptyset$.*
2. *$p_i \in manager_{j_k}(j_k)$.*
3. *the rules in $rules_{j_0}(j_0), rules_{j_1}(j_1), \ldots, rules_{j_k}(j_k)$ facilitate packet exchange between p_i and p_{j_k} along $p_{j_0}, p_{j_1}, \ldots, p_{j_k}$ (when the respective links are operational).*
4. *The end-to-end protocol as well as the round synchronization protocol between p_i and p_{j_k} are in a legitimate state.*

Proof. The proof is by induction on k.
The base case. Claims 5.3, 5.4, and 5.5 imply that the lemma statement holds for $k = 1$.

Claim 5.3. *Within one frame from R's beginning, the system reaches a state in which condition (1) is fulfilled with respect to p_i and any node that is in p_i's distance-1 neighbors in G_c.*

Proof of Claim 5.3. During the first frame (with round-trips) of R, controller p_i starts and completes at least one iteration in which it sends a query (line 24) to every node $p_{j_1} \in P$ that is in p_i's distance-1 neighborhood in G_c (this includes both switches, as we explain in Section 2.1.1, as well as other controllers, which respond according to line 28). Moreover, during that first frame, p_{j_1} receives that query and replies to p_i (lines 25-27) within one step (Section 3.2). Thus, the first part of condition (1) is fulfilled, because controller p_i then adds (or updates) the latest (query) replies that it received from these neighbors to $responses_i$. The second part of condition (1) is implied by the first part of condition (1) and by line 28. □

Claim 5.4. *Within two frames from the beginning of R, the system reaches a state in which conditions (2) and (3) are fulfilled with respect to p_i and any node that is in p_i's distance-1 neighbors in G_c.*

Proof of Claim 5.4. This proof uses Claim 5.3 to prove this claim by first showing that within one frame from the beginning an execution in which condition (1) holds, the system reaches a state in which conditions (2) and (3) are fulfilled with respect to p_i and any node $p_j \in N_c(i)$. This indeed implies that conditions (2) and (3) are fulfilled within two frames of R for p_i's direct neighbors.

Let R^* be a suffix of R such that in R^*'s stating system state, it holds that condition (1) is fulfilled with respect to p_i and any node that is in p_i's distance-1 neighbors in G_c. During the first frame (with round-trips) of R^*, controller p_i starts and completes at least one iteration (with round-trips) in which it is able to include p_i in p_j's manager set, $manager_j(j)$ (line 19 to 21) and to install rules at $p_j \in N_c(i)$ (line 22). We know that this installation is possible, because p_i is a direct neighbor of $p_j \in N_c(i)$ (Section 2.1.1). Once these rules are installed, the packet exchange between p_i and $p_j \in N_c(i)$ is feasible. This implies that conditions (2) and (3) are fulfilled within one frame of R^* (and two frames of R) for p_i's direct neighbors. □

Claim 5.5. *Within* $((\Delta_{comm} + \Delta_{synch}) + 2)$ *frames from the beginning of* R*, the system reaches a state in which condition (4) is fulfilled with respect to* p_i *and any node that is in* p_i*'s distance-*1 *neighbors in* G_c*.*

Proof of Claim 5.5. Since conditions (2) and (3) hold within two frames with respect to $k = 1$, controller p_i and p_{j_1} can maintain an end-to-end communication channel between them because the network part between p_i and p_{j_1} includes all the needed flows. By Δ_{comm}'s definition (Section 3.1), within Δ_{comm} frames, the system reaches a legitimate state with respect to the end-to-end protocol between p_i and p_{j_1}. Similarly, by Δ_{synch}'s definition (Section 2.2.2), within Δ_{synch} frames, the system reaches a legitimate state with respect to the round synchronization protocol between p_i and p_{j_1}. Thus, condition (4) holds within $((\Delta_{comm} + \Delta_{synch}) + 2)$ frames from R's beginning. □

The induction step. Suppose that, within $((\Delta_{comm} + \Delta_{synch}) + 2)k$ frames from R's starting state, the system reaches a state $c_x \in R$ in which conditions (1), (2), (3) and (4) hold with respect to k. We show that within $(\Delta_{comm} + \Delta_{synch}) + 2$ frames from c_x, the system reaches a state in which the lemma's statements hold with respect to $k + 1$ as well.

Showing that, within one frame from c_x, processor p_i knows all of its distance-$(k+1)$ neighbors. This part of the proof starts by showing that within one frame from c_x, execution R reaches a state, such that $p_i \rightarrow_{G_i} p_j$ holds for every distance-$(k+1)$ neighbor of p_i in G_c. The system state c_x encodes (packet forwarding) rules that allow p_i to exchange packets with its distance-k neighbors in G_c (since by the induction hypothesis, conditions (3) and (4) hold with respect to k in c_x). Moreover, p_i stores in $res(prevTag_i)$ replies from p_i's distance-k neighbors in G_c (since by the induction hypothesis, condition (1) holds for k in c_x). The latter implies that p_i knows, as part of G_i in c_x, all of its distance-$(k+1)$ neighbors, $\{p_k : \exists\langle j, N_c(j), \bullet\rangle \in res_i(prevTag_i) \wedge (k = j \vee k \in N_c(j, prevTag_i))\}$, since every reply of a distance-k neighbor, p_{j^*}, in G_c (which $res_i(prevTag_i)$ stores in c_x) includes p_{j^*}'s neighborhood.

Condition (1) holds with respect to $k+1$ within $((\Delta_{comm}+\Delta_{synch})+2)k+1$ frames. Using the above we show that, within one frame from c_x, controller $p_i \in P_C$ queries all of its distance-$(k+1)$ neighbors (line 24), receives their replies, and stores them in $responses_i$ (lines 25–27), i.e., $\langle j_{k+1}, N_c(j_{k+1}), manager_i(j_{k+1}), rules_i(j_{k+1})\rangle \in res_i(currTag_i)$ for every distance-$(k+1)$ neighbor $p_{j_{k+1}}$ of p_i in G_i. Recall that c_x encodes rules that let p_i to forward packets with its distance-k neighbors in G_c (condition (3) holds for k in c_x). By the query-by-neighbor functionality (Section 2.1.1), every such distance-k neighbor reports on its direct neighbors (that include p_i's

distance-$(k+1)$ neighbors), which implies that it forwards the query message to p_i's distance-$(k+1)$ neighbor as well as the reply back to p_i. Therefore, within $((\Delta_{comm}+\Delta_{synch})+2)k+1$ frames, the system reaches a state, $c_{x'}$, in which condition (1) holds with respect to $k+1$.

Conditions (2) to (3) hold with respect to $k+1$ within $((\Delta_{comm}+\Delta_{synch})+2)k+2$ frames. The next step of the proof is to show that within one frame from $c_{x'}$, the system reaches the state $c_{x''}$ in which conditions (2) and (3) hold with respect to $k+1$ (in addition to the fact that condition (1) holds). By the functionality for querying (and modifying)-by-neighbor (Section 2.1.1) and for every switch p_j that is a distance-$(k+1)$ neighbor of p_i in G_c, it holds that between $c_{x'}$ and $c_{x''}$: (a) p_i adds itself to the manager set $manager(j)$ of p_j (line 19 to 21), and (b) p_i installs its rules in p_j's configuration (line 22). (We note that for the case p_j is another controller, there is no need to show that conditions (2) and (3) hold.)

Condition (4) holds for $k+1$ within $((\Delta_{comm}+\Delta_{synch})+2)(k+1)$ frames. The proof is by similar arguments to the ones that appear in the proof of Claim 5.5.

Thus, conditions (1), (2), (3), and (4) hold for $k+1$ within $((\Delta_{comm}+\Delta_{synch})+2)(k+1)$ frames in R and the proof is complete. □

Lemma 5 bounds the number of frames before the system reaches a legitimate system state.

Lemma 5. *Let $R = R' \circ R''$ be an execution of Algorithm 2 that includes a prefix, R', of $(\Delta_{comm}+\Delta_{synch})+2)D+1$ frames that has no occurrence of C-resets or illegitimate deletions. (1) Any system state in R'' is legitimate (Definition 1). (2) Let $a_x \in R''$ be a step that includes the execution of the do-forever loop that starts in line 12 and ends in line 24. During that step a_x, the value of msg_i, which p_i sends to $p_j \in P$ in line 24, does not include the record $\langle$'delMngr', $\bullet\rangle$ nor the record $\langle$'delAllRules', $\bullet\rangle$, i.e., no deletions, whether they are illegitimate or not, of managers or rules. (3) No controller p_i takes a step in R'' during which the condition of line 26 holds, which implies that p_i performs no C-reset during R''.*

Proof. When comparing the conditions of Definition 1 and the ones of Lemma 4, we see that Lemma 4 guarantees that within $(\Delta_{comm}+\Delta_{synch})+2)D$ frames the system reaches a state $c_{almostSafe} \in R'$ in which all the conditions of Definition 1 hold except condition 2 with respect to controllers $p_j \notin P_C$ that do not exist in the system (and their rules that are stored by the switches). From condition 1 of Definition 1, we have that at each controller $p_i \in P_C$, it holds that $G(res(currTag_i)) = G(fusion_i) = G_c$. This implies that p_i can identify correctly any stale information related to p_j and remove it from configuration of every switch (see line 18 to 22) that is in the system during the round that follows $c_{almostSafe}$, which takes one frame because condition 1 of Definition 1 holds. This means that within $(\Delta_{comm}+\Delta_{synch})+2)D+1$ frames the system reaches a legitimate state in which all the conditions of Definition 1 hold and thus R'' is a legal execution, i.e., the first part of the lemma holds. Part (2) of this lemma is implied by the fact that there is no controller $p_j \notin P_C$ that the controller $p_i \in P_C$ needs to remove from the configuration of any switch during the legal execution R''. Part (3) is implied by Part (3) of Lemma 2 and the fact that R'' is a legal execution. □

Theorem 2 (Self-Stabilization)**.** *Within $((\Delta_{comm}+\Delta_{synch})+2)D+1)[((\Delta_{comm}+\Delta_{synch})D+1)\cdot N_S+N_C+1]$ frames in R, the system reaches a state $c_{safe} \in R$ that is legitimate (Definition 1). Moreover, no execution that starts from $c_{safe} \in R$ includes a C-reset nor illegitimate deletion of managers or rules.*

Proof. In this proof, we say that an execution R_{adm} is admissible when it includes at least $((\Delta_{comm}+\Delta_{synch})+2)D+1$ frames and no C-reset nor an illegitimate deletion. Let R be an execution of Algorithm 2. Let us consider R's longest possible prefix R', such that R' does not include any sub-execution that is admissible, i.e., $R = R' \circ R''$. Recall that by Corollary 1 the prefix R' has no more than $((\Delta_{comm}+\Delta_{synch})D+1)\cdot N_S + N_C$ C-resets or illegitimate deletions. By the pigeonhole principle, the prefix R' has no more than $((\Delta_{comm}+\Delta_{synch})+2)D+1)[((\Delta_{comm}+\Delta_{synch})D+1)\cdot N_S + N_C + 1]$ frames. By Lemma 5, R'' does not include C-resets nor deletions of managers or rules, and the system has reached a safe state, which is c_{safe}. □

5.5 Returning to a legitimate state after topology changes

This part of the proof considers executions in which the system starts in a state c', that is obtained by taking a system state c_{safe} that satisfies the requirements for a legitimate system state (Definition 1), and then applying a bounded number of failures and recoveries. We discuss the conditions under which no packet loss occurs when starting from c', which is obtained from c_{safe} and (i) the events of up to r link failures and up to ℓ link additions (Lemma 6), as well as, (ii) the events of up to r controller failures and up to ℓ controller additions (Lemma 7).

Lemma 6. *Suppose that c' is obtained from a legitimate system state c_{safe} by the removal of at most r links and the addition of at most ℓ links (and no further failures), and R is an execution of Algorithm 2 that starts in c'. It holds that no packet loss occurs in R as long as $r \leq \kappa$ and $\ell \geq 0$. For the case of $r \leq \kappa \wedge \ell \geq 0$ recovery occurs within $O(D)$ frames, while for the case of $r > \kappa$ bounded communication delays can no longer be guaranteed.*

Proof. We consider the following cases.

The case of $r \leq \kappa$ and $\ell = 0$. Suppose that a single link e has failed, i.e., it has been permanently removed from G_c, in a state c' that follows a legitimate system state c_{safe}. Say that e is included either in a primary path Π_0 in $G_o(0)$ or in one of the alternative paths of Π_0, Π_k in $G_o(k)$, where $k > 0$, for a controller p_i (cf. definitions of the function $myRules()$ and the graphs $G_o(k)$ in Section 2.2.2). For every such case, since e's failure occurs after a legitimate state, communication is maintained when at most $\kappa - 1$ links (other than e) are non-operational. Let s be the index in $\{0, 1, \ldots, \kappa\}$ for which $e \in \Pi_s$. Due to the construction of the paths Π_k, $k \in \{0, 1, \ldots, \kappa\}$, in the computation of the function $myRules()$ in p_i, if $s = 0$, then each alternative path Π_k before e's failure is now considered as path Π_{k-1}, for $k \in \{1, \ldots, \kappa\}$. Otherwise, if $s \neq 0$, the paths Π_k remain the same for $k \in \{0, \ldots, s-1\}$ and each path Π_k is now considered as the alternative path Π_{k-1} for $k \in \{s+1, \ldots, \kappa\}$. In both cases, a new path Π_κ is computed and installed in the switches if that is possible due to the edge-connectivity of G_c, and if that is not the case, the rules installed in the network's switches facilitate $(\kappa - 1)$-fault-resilient flows between every controller and every other node in the network. The recovery time is at most 1 frame (if e belongs to some path Π_k), since the removal of link e occurs after a legitimate state and all nodes in the network can be reached by every controller $p_i \in P_C$.

Note that if e is not part of any flow, then its failure has no effect in maintaining bounded communication delays. By extension of the argument above, bounded communication delays can be maintained when at most κ link failures occur. That is, in the worst case when exactly κ link failures occur, bounded communication delays are maintained due to the existence of the κ^{th} alternative paths and the assumption that no further failures occur in the network.

The case of $r = 0 \wedge \ell > 0$. A link addition can violate the first shortest path optimality, thus in this case all paths should be constructed from scratch. Since, the link addition occurs after a legitimate state, no stale information exist in the system, and no resets or illegitimate deletions occur. Hence, by Lemma 4 (for $k = D$) within $2D$ frames it is possible to (re-)build the κ-fault containing flows throughout all nodes in the network and reach a legitimate system state (since the edge-connectivity cannot decrease with link additions).
The case of $r \leq \kappa$ and $\ell > 0$. Note that by the first case, bounded communication delays are maintained, since $r \leq \kappa$. Since ℓ links are added in G_c, the controllers require $O(D)$ frames to install new paths (by Lemma 4), even though the connectivity of G_c might be less than $\kappa + 1$ (but for sure at least 1). Hence, bounded communication delays are guaranteed in this case, given that no more failures occur.
The case of $r > \kappa$. In this case, we do not guarantee bounded communication delays. This holds, due to the fact that the removal of more than κ edges might break connectivity in G_c, which makes the existence of alternative paths for $r > \kappa$ link failures impossible. □

Lemma 7. *Suppose that c' is obtained from a legitimate system state c_{safe} by the removal of at most r nodes and the addition of at most ℓ nodes (and no further failures), and R is an execution of Algorithm 2 that starts in c'. It holds that no packet loss occurs in R if, and only if, G_c remains connected (and $N_C \geq 1 \wedge N_S \geq 1$), and in this case the network recovers within $O(D)$ frames. For the case of $r > 0 \wedge \ell = 0$ bounded communication delays can no longer be guaranteed.*

Proof. We study the following cases.
The case of $r > 0$ and $\ell = 0$. The removal of a switch p_j is equivalent to the removal of all the links that are adjacent to p_j. Since the edge-connectivity is at least $\kappa + 1$, the minimum degree of every node in G_c is at least $\kappa + 1$. Thus, a switch removal (equiv. removal of at least $\kappa + 1$ links) would violate the assumption of at most κ link failures, possibly violating connectivity or affecting all the alternative paths between two endpoints in the network. In this case, Algorithm 2 can only guarantee that the controllers will install $\tilde{\kappa}$-fault-resilient flows, where $0 \leq \tilde{\kappa} \leq \kappa$.

The case of removing a controller p_i can be handled by Algorithm 2 if we assume that the communication graph G_c stays (at least) $(\kappa + 1)$-edge-connected after removing p_i. In that case, each controller $p_{i'}$ can discover the removal of p_i and delete it from $responses_{i'}$ in 1 frame, and then, in the subsequent frame, $p_{i'}$ can delete p_i's rules from $rules_j(j)$ and p_i from $manager_j(j)$, for every switch p_j. Hence, within 2 frames the system recovers to a legitimate state, since the existing rules of the other controllers stay intact.
The case of $r = 0$ and $\ell > 0$. We assume that if controller or switch additions occur (including their adjacent links) after a legitimate state, the new node is initialized with empty memory. That is, $responses_i$ is empty if a new controller p_i is added, and $manager_j(j) = rules_j(j) = \emptyset$ if a new switch p_j is added. Note that the new node should not violate the assumption of G_c's edge-connectivity being at least $\kappa + 1$. In both cases, and similarly to link additions, the first shortest path optimality might be violated and hence (as in the case of link additions) a period of $2D$ frames is needed (Lemma 4) to (re-)build the κ-fault-resilient flows (since no stale information exist, and no resets or illegal deletions occur).
The case of $r > 0$ and $\ell > 0$. Let G'_c be G_c after the removal of at most r nodes and the addition of at most ℓ nodes. If G'_c is $\tilde{\kappa}$-edge-connected, where $1 < \tilde{\kappa} \leq \kappa$, then bounded communication delays in the occurrence of at most $\tilde{\kappa}$ link failures can be guaranteed by following the arguments of Section 5.4 for $\kappa = \tilde{\kappa}$. □

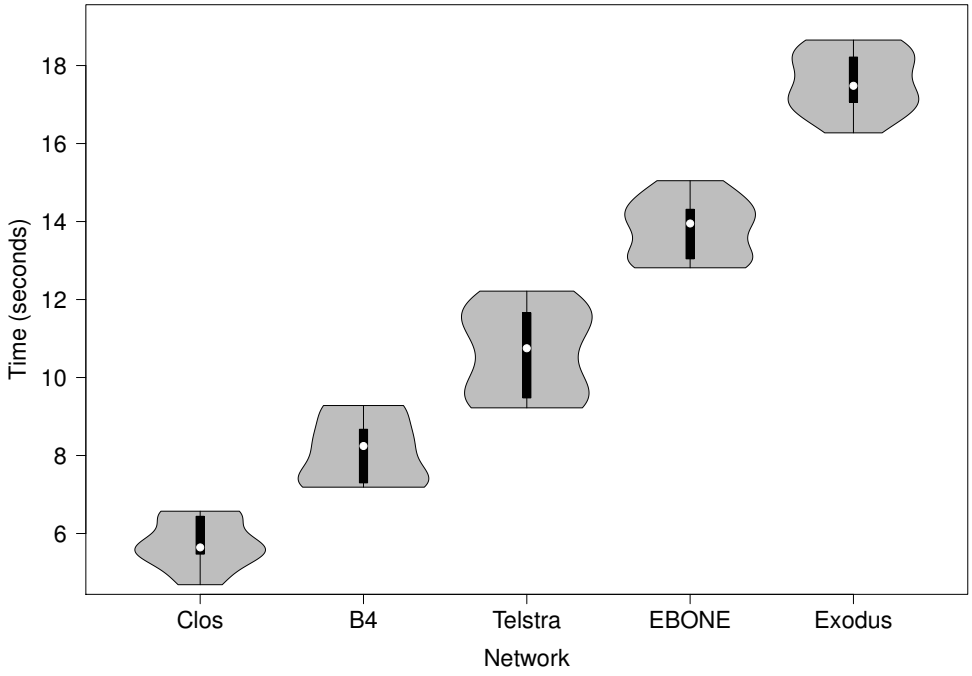

Figure 5: Bootstrap time for the networks using 3 controllers. The network diameters are 4, 5, 8, 10 and 11 (left to right order).

6 Evaluation

In order to evaluate our approach, and in particular, to complement our theoretical worst-case analysis and study the performance in different settings, we implemented a prototype using Open vSwitch (OVS) and Floodlight. In the following, we first present the setup of our experiments. Subsequently, we report on the main insights from our emulations in Mininet. In particular, we empirically evaluate the time to bootstrap an SDN (after the occurrence of transient faults), the recovery time (after the occurrence of benign failures), as well as the throughput during the link failure. We observe that *Renaissance*'s bootstrap recovery time from benign failures depends mostly on the network diameter and the number of messages depends mostly on the number of nodes. *Renaissance*'s recovery is not effected that much by the number of controllers and is not effecting the network's ability to update its flows.

6.1 Setup

We study the following scenarios. We consider a spectrum of different topologies (varying in size and diameter), including B4 (Google's inter-datacenter WAN based on SDN), Clos datacenter networks and Rocketfuel networks (namely Telstra, EBONE, and Exodus). The link status detectors (for switches and controllers) are parametrized with frequency $\theta = \Delta(G) \cdot 3$, where $\Delta(G)$ is the maximum node degree. If not stated otherwise, the controllers issue query requests and install flows once per second. Paths are computed according to Breadth First Search (BFS) and we use OpenFlow fast-

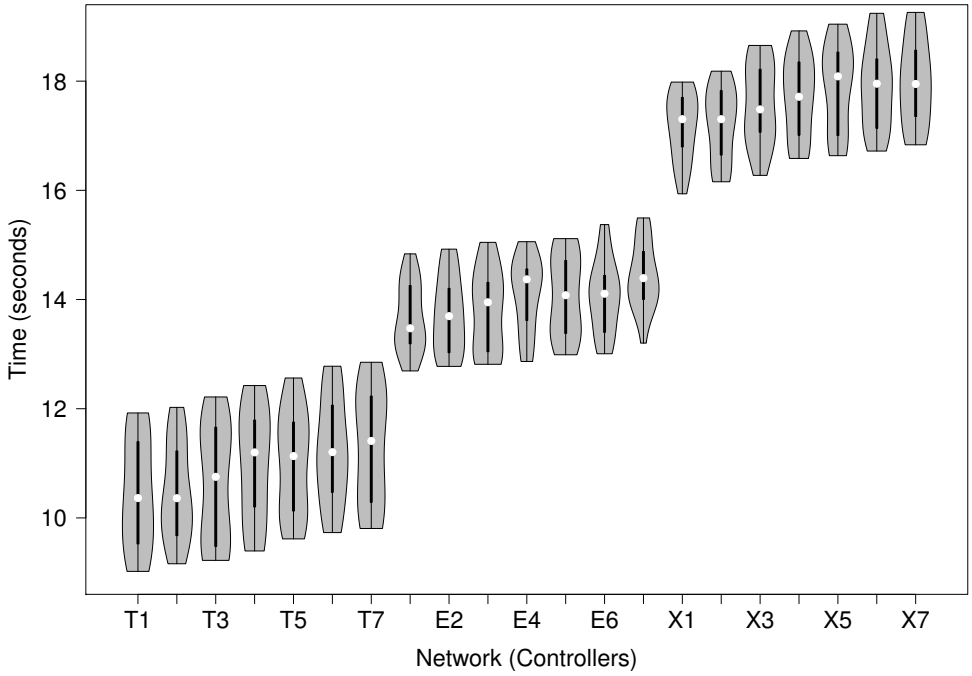

Figure 6: Bootstrap time for Telstra (T), EBONE (E) and Exodus (X) for 1 to 7 controllers.

failover groups for backup paths. The hosts for traffic and RTT evaluation are placed such that the distance between them is as large as the network diameter. The experiments were conducted using PCs with Ubuntu 16.04.1 OS, Intel Core i7-2600K CPU at 3.40GHz (8CPUs) with 16GB RAM.

6.2 Results

We structure our evaluation of *Renaissance* around the main questions related to the SDN bootstrap, recovery times, and overhead, as well as regarding the throughput during failures. For illustrating our data in Figures 5–6 and 8–13 we use the, rather informative, violin plots [14]. In these plots, we indicate the median with a white dot. The first and third quartiles are the endpoints of a thick black line (hence the white dot representing the median is a point on the black line). The thick black line is extended with thin black lines to denote the minimum and maximum of all the data (as the whiskers of box plots). Finally, the vertical boundary of each surface denotes the kernel density estimation (same on both sides) and the horizontal boundary only closes the surface.
How efficiently can *Renaissance* bootstrap an SDN (resp. handle transient faults)? We first study how fast we can establish a stable network starting from empty switch configurations. Towards this end, we measure how long it takes until all controllers in the network reach a legitimate state in which each controller can communicate with any other node in the network (by installing packet-forwarding rules on the switches). For the smaller networks (B4 and Clos), we use 3 controllers, and for the Rocketfuel networks (Telstra, EBONE and Exodus), we use up to 7 controllers.

We are indeed able to bootstrap in *any* of the configurations studied in our experiments. In

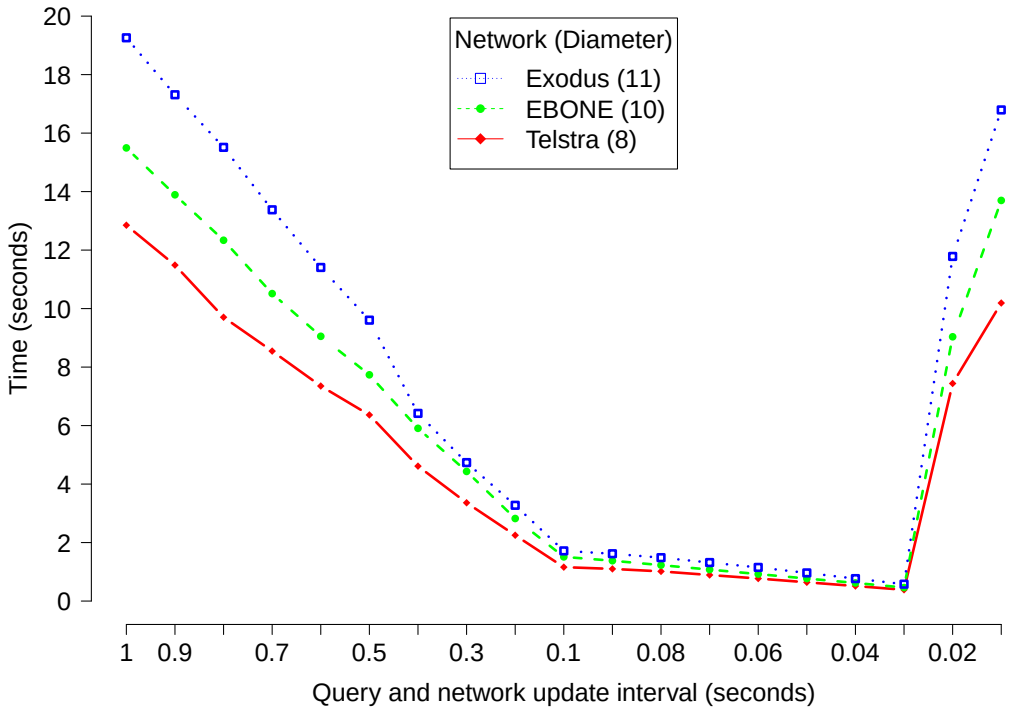

Figure 7: Bootstrap time for Telstra, EBONE and Exodus using 7 controllers, as a function of query intervals.

terms of performance, as expected, the stabilization time is proportional to the network diameter (Figure 5). It also depends on the number of controllers (Figure 6): more controllers result in slightly longer stabilization times as there will be more flows needed for each node in the network.

Note that the shown stabilization times only provide qualitative insights: they are, up to a certain point, proportional to the frequency at which controllers request configurations and install flows (Figure 7). Besides bootstrap *time*, we are also interested in the *communication overhead* of bootstrapping the network. Concretely, we measure the maximum number of controller messages, taking 3 controllers for the smaller networks B4 and Clos, and 7 controllers for the Rocketfuel networks Telstra, EBONE and Exodus in these experiments. While the communication overhead naturally depends on the network size (and controller locations), Figure 8 suggests that when normalized, the overhead is similar for different networks.

How efficiently does *Renaissance* recover in the presence of benign failures (link failure and node crashes)? In order to study the recovery from benign failures, we distinguish between different types of benign failures: fail-stop failures of controllers, permanent switch-failures, and permanent link-failures. The experiments start from a legitimate system state, to which we introduce the failures.

In the *fail-stop* failure experiments (figures 9 and 10), we disconnect a single controller that is initially chosen at random, and measure the recovery time (from benign failures). The procedure is repeated for the same controller for each measurement. We also measure disconnecting 1-6 controllers simultaneously for the Rocketfuel (Telstra, EBONE and Exodus) networks, while running 7 controllers. The multiple controllers chosen for disconnection are also initially chosen at random, and the same ones are used when repeating the procedure for each measurement.

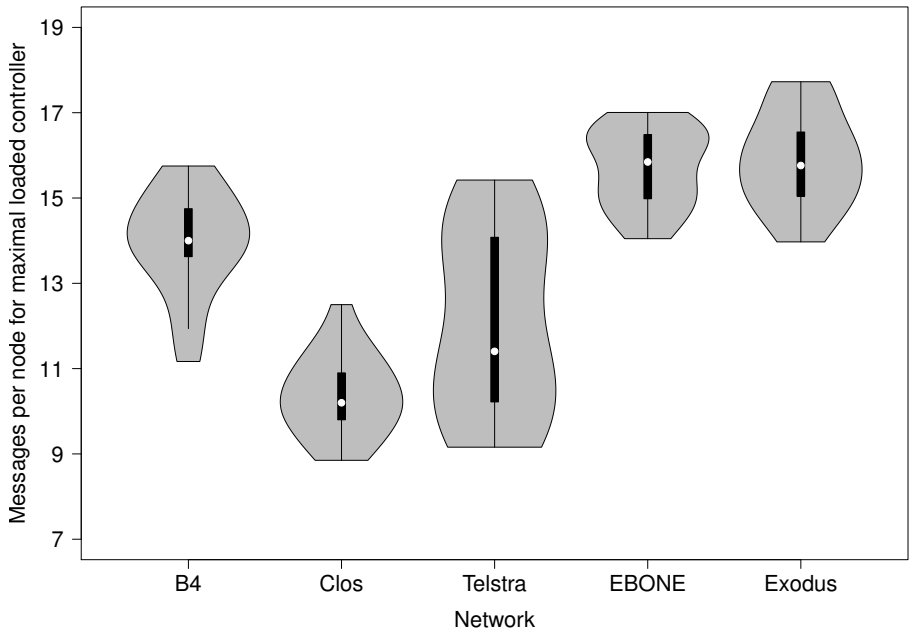

Figure 8: Communication complexity per node needed from a max loaded global controller to reach a stable network.

The experiments for *permanent switch-failures* (Figure 11) are performed similarly, however, we only disconnect a random switch from a set of switches that do not disconnect the network. For the experiments for *permanent link-failures* (figures 12 and 13), we disconnect either a single link that has maximal distance from all the controllers in the network (the procedure is repeated for the same link for each measurement), or, in case of multiple link failures, we choose failed links at random (making sure we do not disconnect the entire network).

We find that the recovery time (from benign failures) is roughly linear in the number of nodes (Figure 9). The diameter also affects the time, but only to a smaller extent. For example, B4, which has a larger network diameter but is smaller compared to Clos, has a smaller average recovery time (from benign failures).

The number of failed controllers (in Figure 10) does not affect the time much, as expected. The times in Figure 11 are similar to the ones in Figure 9.

We observe that the time in Figure 12 looks linear and similar to the one in Figure 9, but with smaller time units for some of the networks (B4 and Clos): permanent link-failures do not necessarily require the controller to configure all the nodes in the network in order to recover (from benign failures), as opposed to fail-stop failures. However, the bigger networks (Telstra, EBONE and Exodus) take a little longer than those of Figure 9.

Figure 13 looks similar to Figure 10 in that there is a slight difference in time for the multiple link-failures: since the failures occur simultaneously and the controllers detect them approximately at the same time.

Performance and Transient Behavior. Besides connectivity, we are also interested in performance metrics such as throughput and message loss *during stabilization*, that is, recovery from

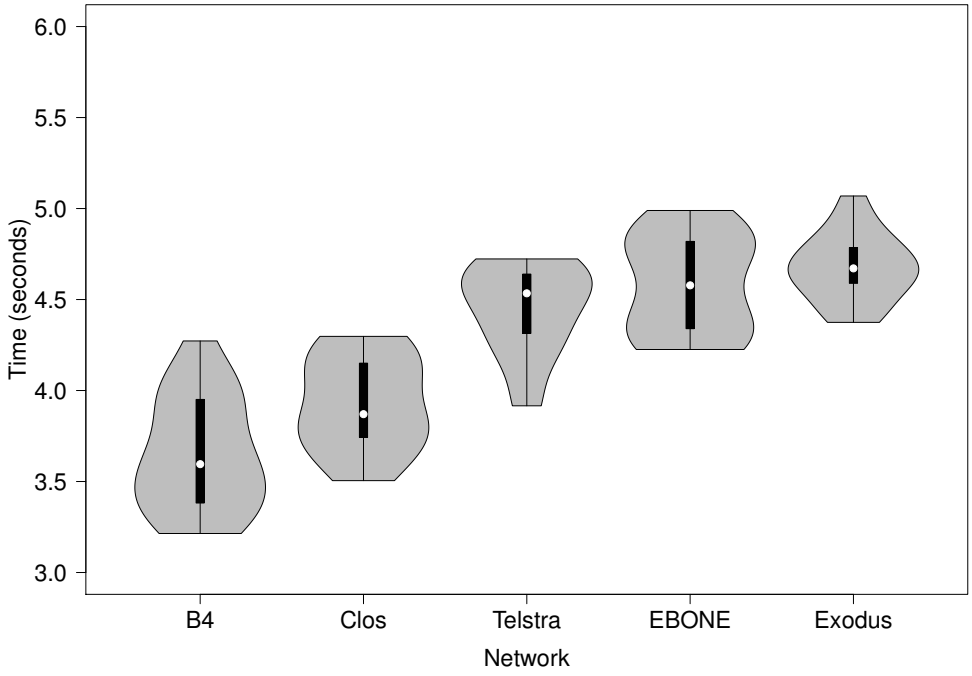

Figure 9: Recovery time after fail-stop failure for a controller.

transient faults that bring the system to an arbitrary starting state. In particular, while the recovery time (from benign failures) is quite fast, involving the control plane is time-consuming and can lead to packet reorderings and congestion. Accordingly, we employ local fast failover mechanisms in our approach.

In the following, we measure the TCP throughput between two hosts (placed at maximal distance from each other), in the presence of a link-failure located as close to the middle of the primary path as possible. To generate traffic, we use Iperf. A specific link to fail is chosen such that it enables a backup path between the hosts.

The maximum link bandwidth is set to 1000 Mbits/s. We conduct throughput measurements during a period of 30 seconds. The link-failure occurs after 10 seconds, and we expect a throughput drop due to the traffic being rerouted to a backup path, which causes TCP's congestion control mechanism to reduce the transmission rate when packet loss or reorderings occur. We note that our prototype utilizes packet tagging for incremental update [25]. This helps the system to guarantee avoidance of another drop in throughput as a result of the new paths being installed from the controllers in order to repair flows.

We can see in Figure 14 that *one* throughput valley occurs indeed (to around 750 - 800 mbits/s). This is interesting, because, as we confirmed in additional experiments (not shown here), a naive approach which does not account for the multiple control planes and recovery from benign failures, results in repeated rerouting and hence repeated performance drops: The throughput increases after the traffic is rerouted to a backup path but drops once again after a few seconds, when the network recovers from benign failures. Using per-packet consistent paths and tagging forces the

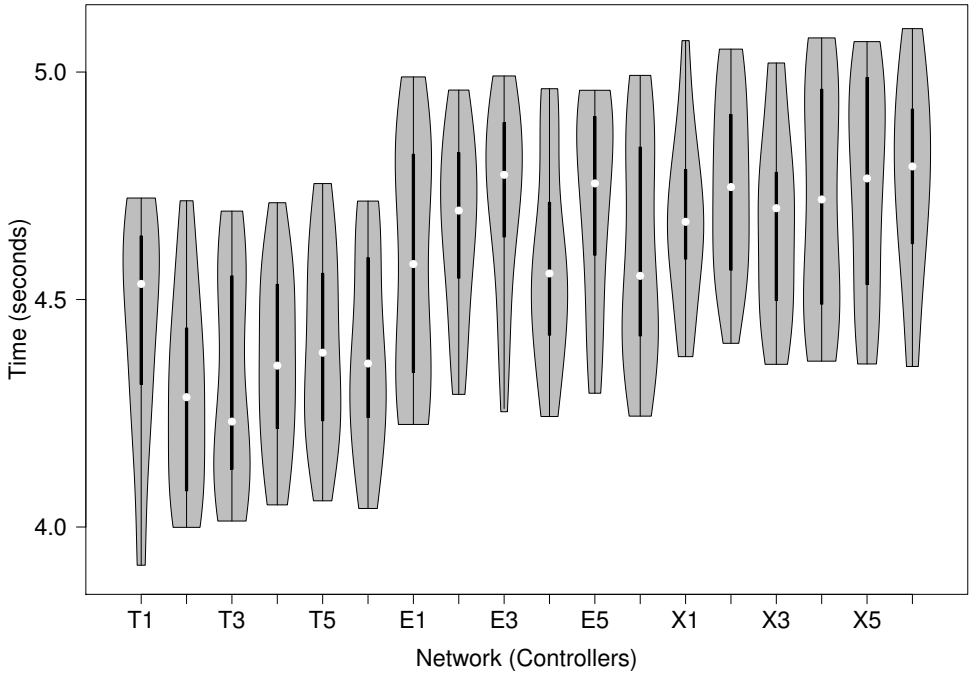

Figure 10: Recovery time after fail-stop failure for 1-6 controllers in Telstra (T), EBONE (E) and Exodus (X).

packets to only use the new primary path once all the necessary rules have been installed on the switches, which can reduce packet loss and re-transmissions of packets.

For comparison, Figure 15 shows the throughput over time without recovery: only the backup paths are used in these experiments, and no new primary paths are calculated or used after the link-failure at the 10th second. The result is very similar to Figure 14: there is a strong correlation between these two methods (Figure 14 vs Figure 15) in terms of performance, see Table 1:

Network	Correlation
Clos	0.94
B4	0.95
Telstra	0.92
EBONE	0.96
Exodus	0.94

Table 1: Correlation coefficient of the average throughput for the experiments in Figure 14 and Figure 15.

In order to gain more insights, we investigate the number of re-transmissions (after the link-failure) for Telstra, EBONE and Exodus network topologies. We observe that around 10% of the packets sent at the 11th second (after the link-failure) are re-transmissions (Figure 16).

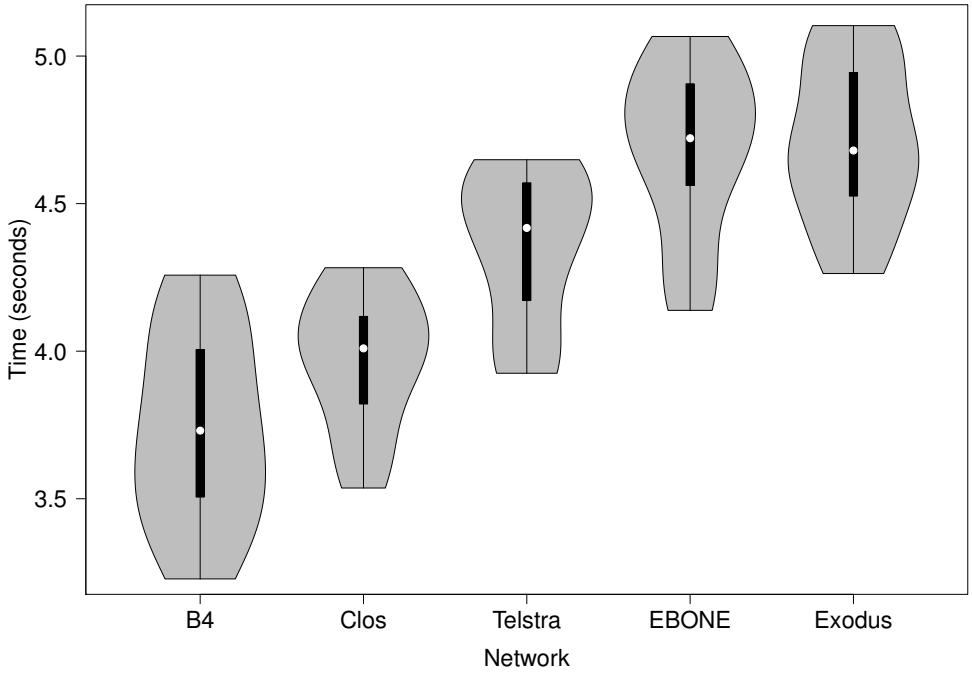

Figure 11: Recovery time after permanent switch-failure.

7 Related Work

The design of distributed SDN control planes has been studied intensively in the last few years [3, 7, 10, 13, 16, 29, 31], both for performance and robustness reasons. While we are not aware of any existing solution for our problem (supporting an in-band and distributed network control), there exists interesting work on bootstrapping connectivity in an OpenFlow network [15, 30] (that does not consider self-stabilization). In contrast to our paper, Sharma et al. [30] do not consider how to support multiple controllers nor how to establish the control network. Moreover, their approach relies on switch support for traditional STP and requires modifying DHCP on the switches. We do consider multiple controllers and establish an in-band control network in a self-stabilizing manner. Katiyar et al. [15] suggest bootstrapping a control plane of SDN networks, supporting multiple controller associations and also non-SDN switches. However, the authors do not consider fault-tolerance. We provide a very strong notion of fault-tolerance, which is self-stabilization.

To the best of our knowledge, our paper is the first to present a comprehensive model and rigorous approach for the design of in-band decentralized control planes providing self-stabilizing properties. As such, our approach complements much ongoing, often more applied, related research. In particular, our control plane can be used together with and support distributed systems such as ONOS [3], ONIX [16], ElastiCon [10], Beehive [31], Kandoo [13], STN [7] to name a few. Our paper also provides missing links for the interesting work by Akella and Krishnamurthy [1], whose switch-to-controller and controller-to-controller communication mechanisms rely on strong primitives, such as consensus protocols, consistent snapshot and reliable flooding, which are not currently available

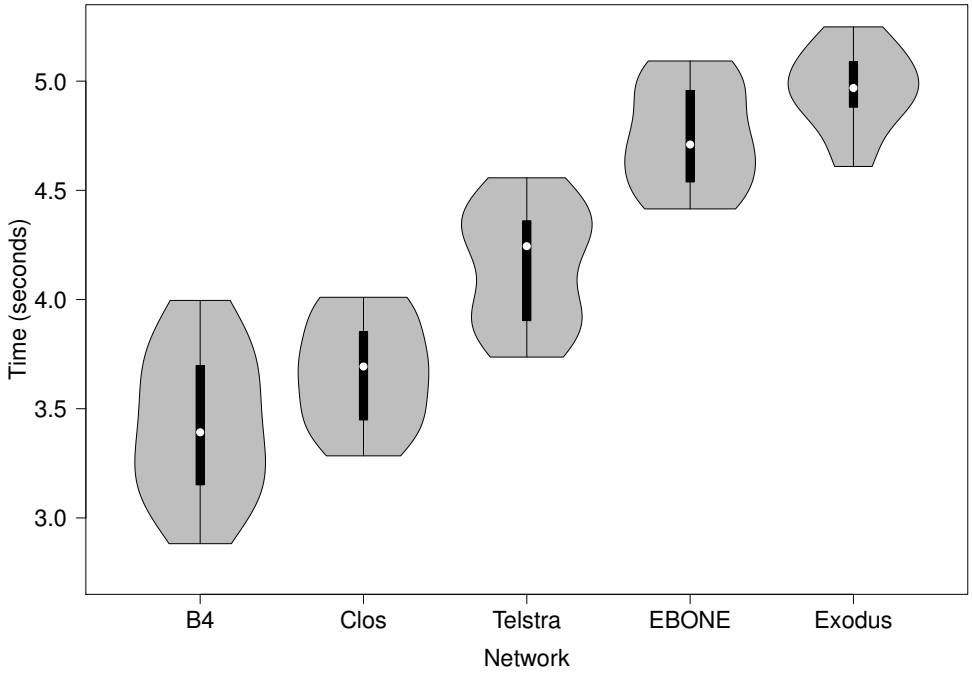

Figure 12: Recovery time after permanent link-failure.

in OpenFlow switches.

We also note that our approach is not limited to a specific technology, but offers flexibilities and can be configured with additional robustness mechanisms, such as warm backups, local fast failover [24], or alternatives spanning trees [6, 20].

Our paper also contributes to the active discussion of which functionality can and should be implemented in OpenFlow. DevoFlow [8] was one of the first works proposing a modification of the OpenFlow model, namely to push responsibility over most flows to switches and adding efficient statistics collection mechanisms. SmartSouth [26] shows that in recent OpenFlow versions, interesting network functions (such as anycast or network traversals) can readily be implemented in-band. More closely related to our paper, [27] shows that it is possible to implement atomic read-modify-write operations on an OpenFlow switch, which can serve as a powerful synchronization and coordination primitive also for distributed control planes; however, such an atomic operation is not required in our system: a controller can claim a switch with a simple write operation. In this paper, we presented a first discussion of how to implement a strong notion of fault-tolerance, namely a self-stabilizing SDN [9, 11].

We are not the first to consider self-stabilization in the presence of faults that are not just transient faults (see [11], Chapter 6 and references therein). Thus far, these self-stabilizing algorithms consider networks in which all nodes can compute and communicate. In the context of the studied problem, some nodes, i.e., the switches, can merely forward packets according to rules that are decided by other nodes, the controllers. To the best of our knowledge, we are the first to demonstrate a rigorous proof for the existence of self-stabilizing algorithms for an SDN control plane.

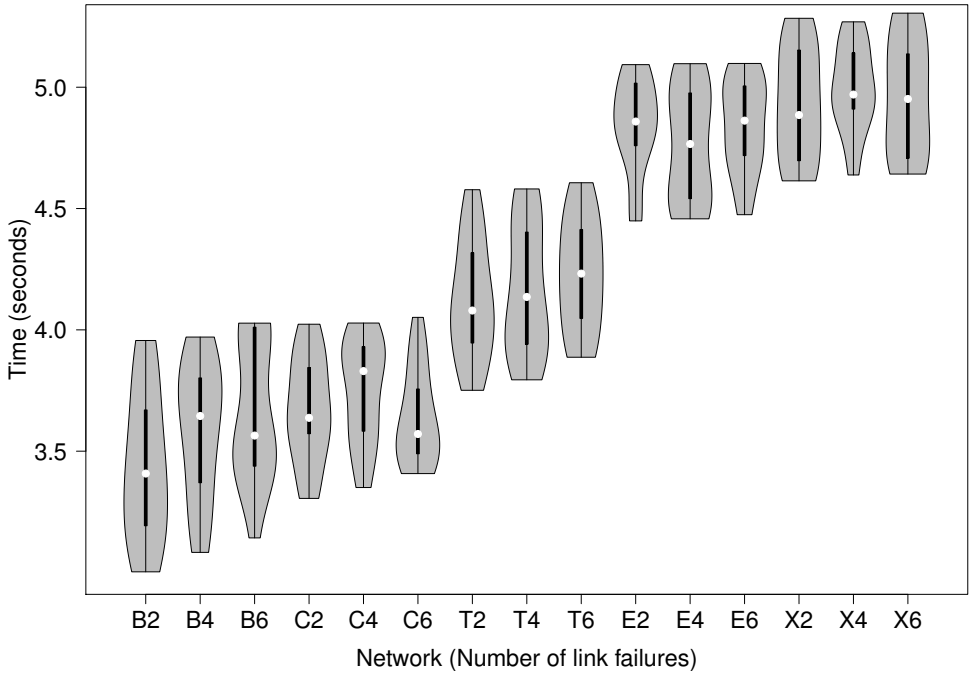

Figure 13: Recovery time after multiple (2,4,6) permanent link-failures at random for B4 (B), Clos (C), Telstra (T), EBONE (E) and Exodus (X).

This proof uses a number of techniques, such as the one for assuring a bounded number of resets and illegitimate rule deletions, that were not used in the context of self-stabilizing bootstrapping of communication (to the best of our knowledge).

Bibliographic Note. We reported on preliminary insights on the design of in-band control planes in two short papers on *Medieval* [27,28]. However, Medieval is not self-stabilizing because its design depends on the presence of non-corrupted configuration data, e.g., related to the controllers' IP addresses, which goes against the idea self-stabilization. In contrast, in this paper we provide a rigorous algorithm and proof of self-stabilization. The design of *Renaissance* also departs from Medieval's "white-listing philosophy" when it comes to traffic forwarding to the controller, and instead goes for a black listing approach to block traffic which should stay in the dataplane explicitly.

8 Conclusion

While the benefits of the separation between control and data planes have been studied intensively in the SDN literature, the important question of how to connect these planes has received less attention. This paper presented a first model and an algorithm, as well as a rigorous analysis and proof-of-concept implementation of a self-stabilizing SDN control plane called *Renaissance*.

We understand our paper as a first step, and believe it opens several interesting directions for future research. In particular, while we have deliberately focused on the more challenging in-band

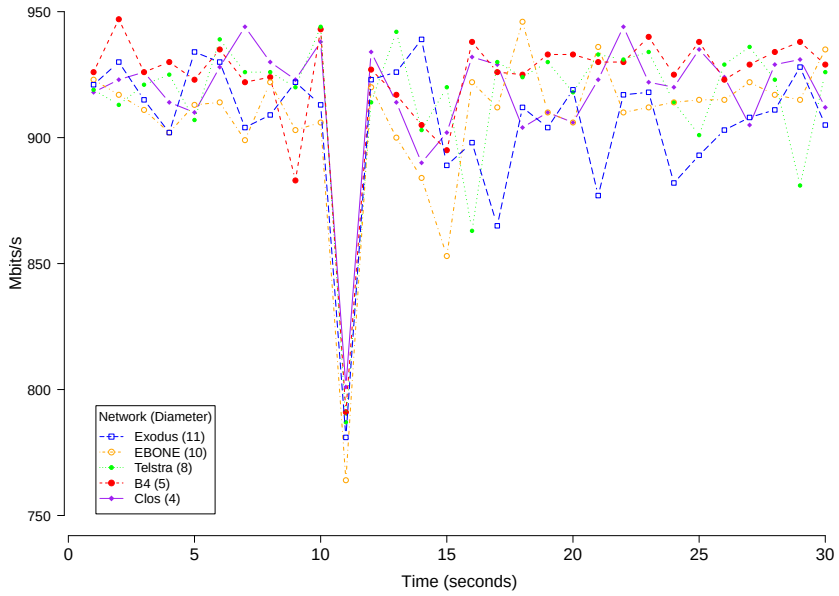

Figure 14: Throughput for the different networks using network updates with tags.

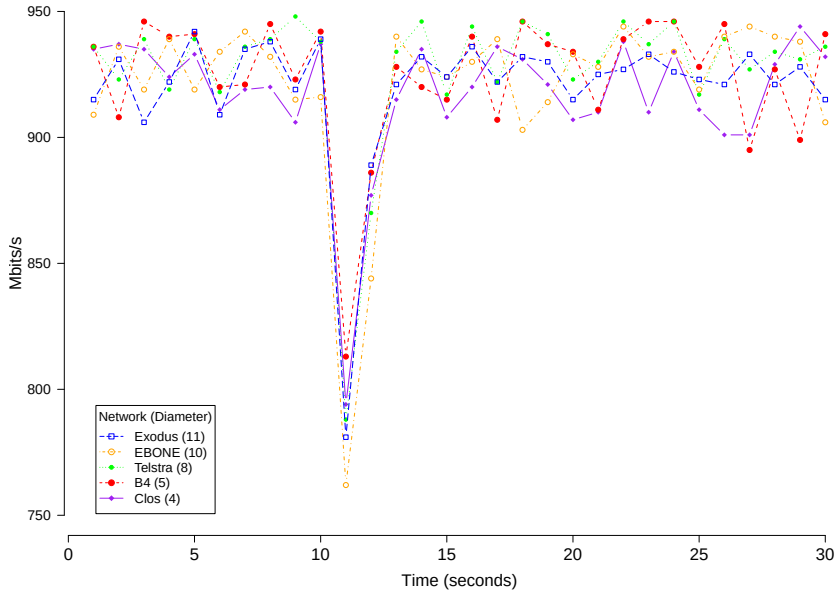

Figure 15: Throughput for the different networks using no recovery after link-failure.

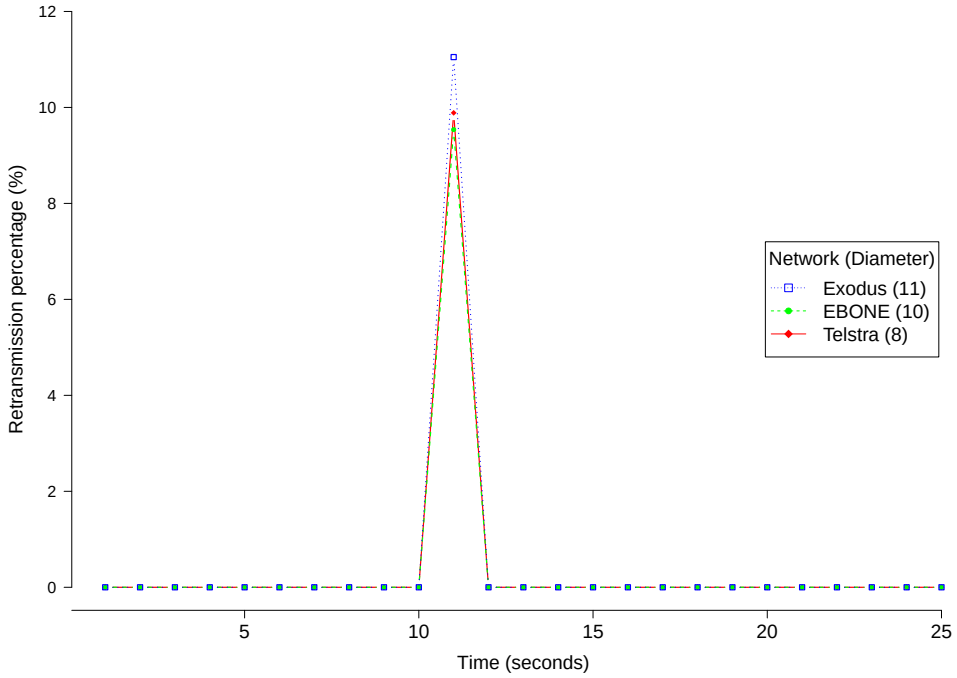

Figure 16: Retransmission percentage rate for packets sent at each second.

control scenario only, we anticipate that our approach can also be used in networks which combine both in-band and out-of-band control, e.g., depending on the network sub-regions. Moreover, while fundamental, our model is still simple and could be extended, e.g., to account for the dynamics of control and data plane traffic, e.g., by adjusting the failure detector model accordingly or to establish the backup routing paths for control traffic by considering the data traffic dynamics. Finally, while our prototype experiments demonstrate feasibility of our approach and show promising results, it remains to conduct a more rigorous practical evaluation.

Acknowledgments. Part of this research was supported by the Danish Villum Fonden blokstipendier project *Reliable Computer Networks (ReNet)*. This research is (in part) supported by European Union's Horizon 2020 research and innovation programme under the ENDEAVOUR project (grant agreement 644960). We are grateful to Michael Tran for developing the prototype and for the many discussions. We are also thankful to Emelie Ekenstedt for helping to improve the presentation.

References

[1] Aditya Akella and Arvind Krishnamurthy. A Highly Available Software Defined Fabric. In *HotNets*, 2014.

[2] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien

Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *J. Comput. Syst. Sci.*, 81(4):692–701, 2015.

[3] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru M. Parulkar. ONOS: towards an open, distributed SDN OS. In Aditya Akella and Albert G. Greenberg, editors, *Proceedings of the third workshop on Hot topics in software defined networking, HotSDN '14, Chicago, Illinois, USA, August 22, 2014*, pages 1–6, 2014.

[4] Peva Blanchard, Shlomi Dolev, Joffroy Beauquier, and Sylvie Delaët. Practically self-stabilizing paxos replicated state-machine. In *Proc. 2nd International Conference on Networked Systems (NETYS)*, pages 99–121, 2014.

[5] Michael Borokhovich, Liron Schiff, and Stefan Schmid. Provable data plane connectivity with local fast failover: introducing openflow graph algorithms. In *Proc. 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 121–126, 2014.

[6] Michael Borokhovich, Liron Schiff, and Stefan Schmid. Provable Data Plane Connectivity with Local Fast Failover: Introducing OpenFlow Graph Algorithms. In *HotSDN*, 2014.

[7] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *INFOCOM*, 2015.

[8] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. In *SIGCOMM*, 2011.

[9] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[10] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.

[11] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[12] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks. In *Proc. International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 133–147, 2012.

[13] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proc. ACM HotSDN*, 2012.

[14] Jerry L Hintze and Ray D Nelson. Violin plots: a box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.

[15] Rohit Katiyar, Prakash Pawar, Abhay Gupta, and Kotaro Kataoka. Auto-configuration of SDN switches in sdn/non-sdn hybrid network. In *Proceedings of the Asian Internet Engineering Conference, AINTEC 2015, Bangkok, Thailand, November 18-20, 2015*, pages 48–53. ACM, 2015.

[16] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[17] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[18] Junda Liu, Baohua Yang, Scott Shenker, and Michael Schapira. Data-driven network connectivity. In *Proc. ACM HotNets*, page 8, 2011.

[19] Open Networking Foundation. OpenFlow Switch Specification Version 1.3.4.

[20] Merav Parter. Dual Failure Resilient BFS Structure. In *PODC*, 2015.

[21] Radia Perlman. *Interconnections (2Nd Ed.): Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[22] Radia J. Perlman. Fault-tolerant broadcast of routing information. *Computer Networks*, 7:395–405, 1983.

[23] Radia J. Perlman. An algorithm for distributed computation of a spanningtree in an extended LAN. In William Lidinsky and Bart W. Stuck, editors, *SIGCOMM '85, Proceedings of the Ninth Symposium on Data Communications, British Columbia, Canada, September 10-12, 1985*, pages 44–53. ACM, 1985.

[24] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *HotSDN*, 2013.

[25] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: change you can believe in! In Hari Balakrishnan, Dina Katabi, Aditya Akella, and Ion Stoica, editors, *Tenth ACM Workshop on Hot Topics in Networks (HotNets-X), HOTNETS '11, Cambridge, MA, USA - November 14 - 15, 2011*, page 7. ACM, 2011.

[26] Liron Schiff, Michael Borokhovich, and Stefan Schmid. Reclaiming the Brain: Useful OpenFlow Functions in the Data Plane. In *HotNets*, 2014.

[27] Liron Schiff, Petr Kuznetsov, and Stefan Schmid. In-Band Synchronization for Distributed SDN Control Planes. *SIGCOMM Comput. Commun. Rev.*, 46(1), January 2016.

[28] Liron Schiff, Stefan Schmid, and Marco Canini. Ground control to major faults: Towards a fault tolerant and adaptive sdn control network. In *Proc. IEEE/IFIP DSN Workshop on Dependability Issues on SDN and NFV (DISN)*, 2016.

[29] Stefan Schmid and Jukka Suomela. Exploiting locality in distributed sdn control. In *Proc. ACM SIGCOMM HotSDN*, 2013.

[30] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. In-Band Control, Queuing, and Failure Recovery Functionalities for OpenFlow. *IEEE Network*, 30(1), January 2016.

[31] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple Distributed Programming in Software-Defined Networks. In *Proc. ACM SOSR*, 2016.