Renaissance: Self-Stabilizing Distributed SDN Control Plane

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

***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.**

I. 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.

Ensuring 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 [20]; 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 [21], and provided a self-stabilizing spanning tree algorithm for interconnecting bridges [22]. 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 solution that provides:

(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: *(i) 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. *(ii) 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 failures: We show that our control plane can even recover after the occurrence of transient failures. 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 SDN (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 in the presence of failures.

To ensure reproducibility and to facilitate research on improved and alternative algorithms, we will release the source code together with this paper. Due to the page limit, some of the proof details appear in [8].

Organization. We give an overview of our system and the components it interfaces in Section II. Our algorithm is presented (Section III), analyzed (Section IV), and validated (Section V). We discuss related work (Section VI) and then conclude (Section VII).

II. 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.

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 node $p_i \in P = P_C \cup P_S$. 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

[1]The word renaissance means ‘rebirth’ (French) and it symbolizes the ability to recover after the occurrence of transient faults.

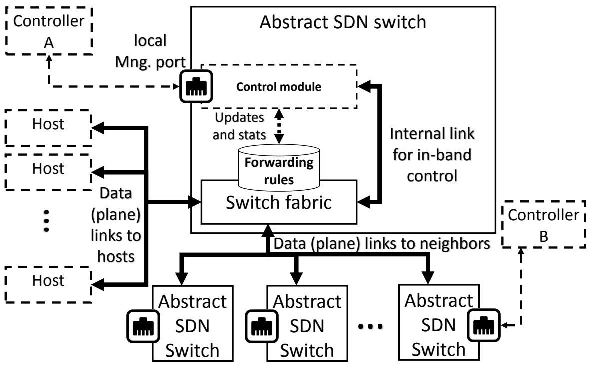

Fig. 1: Abstract SDN switch illustration.

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 to correctly forward their traffic. We do not assume anything about the hosts' network service, except for that their traffic may traverse the network.

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. Thus, control plane packets need to be logically distinguished from data plane traffic by some tag (or another deterministic discriminator).

Figure 1 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 following, we will assume a local topology discovery mechanism which reports to the controllers the availability of their direct neighbors. Also, we can assume reliable, bidirectional FIFO-communication channels (without reordering or omission) between communication endpoints at the transport layer [12].

A. Switches and Rules

Let p_j be a node and $N_c(j) \subseteq P$ (communication topology) be the set of directly attached neighboring nodes of p_j. At any given time, and for any given node $p_i \in P$, the set $N_o(j) \subseteq N_c(j)$ (operational topology) refers to p_j's directly connected nodes for which ports are currently available for packet forwarding.

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, κ is a bound on the number of concurrently failing links, $p_j \in N_c(i)$ 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 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*.

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.

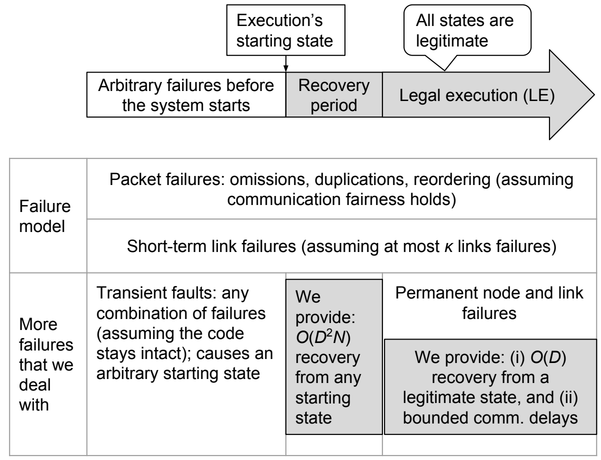

Fig. 2: Failures (white boxes) and recovery guarantees (gray boxes) of the proposed self-stabilizing SDN solution.

B. 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.

1) Topology Discovery: The local topology information in $N_o(i)$ (operational topology) is liable to rapidly change without notice. 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. This mechanism reports the set of nodes $N_c(i) \subseteq P$ (communication topology) which are directly connecting node $p_i \in P$ and node p_j, i.e., $p_j \in N_c(i)$.

2) Fault-resilient Flows: We consider fault-resilient flows which are a reminiscent of the flows in [18]. 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 link. The κ-fault-resilient flows are an enhancement of this redundancy for the case in which κ links fail, as described in [14].

C. Models

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 II-B1), 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 direct neighbor. Apart from this failure detector, we consider the control plane as an asynchronous system. We model the nodes as automata with input events that are either a packet reception or a periodic timer (at an unknown rate), as well as output events that send messages.

We denote the operational and connected communication topology as $G_o = (P, E_o)$, and respectively, as $G_c = (P, E_c)$, where for $x \in \{o, c\}$, $E_x = \{(p_i, p_j) \in P \times P : p_j \in N_x(i)\}$. We assume that, during the system run, 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) as well as switch fail-stop failures (or the addition of switches to the network). 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 $N_c(i)$. 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 (cf. Figure 2).

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 $myRules()$ maintains rules for κ-fault-resilient flows; their primary paths are also the shortest paths in G_c.

We define the system's task by a set of system runs called *legal executions* (LE) in which the task's requirements hold. I.e., each controller p_i constructs a κ-fault-resilient flow to every node $p_j \in P$ (either a switch or controller). We say that a system state c is *legitimate*, when every execution R (run) 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 (Figure 2).

III. 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 that it discovers.

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 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.

A. Variables, Building Blocks, Interfaces

Before presenting our algorithm, we introduce some notation, interfaces, and building blocks.

Local Variables. Each controller's state includes $responses$ (line 2), which is the set of the most recent query replies, and the tags (line 3), which are p_i's current ($currTag$) and previous ($prevTag$) synchronization round tags. Node p_j's response $m(j) : p_j \in P$ 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 (cf. Section II-A). We assume that the size of $responses$ is bounded by $maxResponses \geq 2(N_C + N_S)$.

The Round Synchronization Mechanism. 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 $\langle i, j, \bot, \bot, n_{prt}, \bot, t_{metaRule}\rangle$, 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. Once p_i has received a response with $currTag$ from all reachable nodes, it ends that round.

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 a self-stabilizing end-to-end protocol) provide unique tags within a number of synchronization rounds that is bounded (by a constant whenever the execution is legal with respect to a self-stabilizing end-to-end protocol).

Interfaces. Controller p_i can send requests or *queries* to any other node p_j. The controllers send command batches, which are sequences of commands. The special metadata command $\langle `newRound', t_{metaRule}\rangle$ is always the first command. We use it for starting a new round (where $t_{metaRule} = t$ is the round's tag). This start 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 p_j's configuration, where $p_k \in P_C \setminus \{p_i\}$. The rules' update, done via $\langle `updateRules', newRules\rangle$, replaces all of p_i's rules at switch p_j (except for the special meta rule).

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 metarule $\langle i, \bullet, t\rangle \in rules(j)$. Whenever $p_j \in P_C$ is another controller, the response to a query is simply $\langle i, N_c(i), \bot, \{\langle j, i, \bot, \bot, \bot, \bot, t_{query}\rangle\}\rangle$ (line 25). Note that controller p_j ignores all other types of commands. We use the interface function $myRules(G, j, tag)$ (Section II-B2) 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 (line 4).

B. Description of Algorithm 1

Each controller associates independently each iteration with a unique tag that synchronizes a round in which the controller performs configuration updates and queries. Controller p_i maintains the variables $currTag$ and $prevTag$ (line 3) of the round synchronization procedure, which starts when p_i queries all reachable

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

1 **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.

2 **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 $rules(j) \subseteq \{\langle k, j, src, dest, prt, z, tag\rangle : (p_k, p_j, p_z, p_{dest} \in P) \land p_{src} \in P_C \land prt \in \{0, \ldots, n_{prt}\} \land tag \in tagDomain\}$ is p_j's rule set;

3 $currTag$ and $prevTag$ are p_i's current, and respectively, previous synchronization round tags;

4 **Interfaces (sections II-B2 and III-A):** $myRules(G, j, tag)$: creates p_i's rules at switch p_j according to G with tag tag;

5 **Macros:** $res(x) = \{\langle \bullet, rules(j)\rangle \in responses : \forall_{r \in rules(j)}\, r = \langle \bullet, x\rangle\} \cup \{\langle i, N_c(i), \emptyset, \emptyset\rangle\}$;

6 $G(S) := (\{p_k : \exists_{\langle j, N_c(j), \bullet\rangle \in S} : (k = j \lor p_k \in N_c(j))\}, \{(j, k) : \exists_{\langle j, N_c(j), \bullet\rangle \in S} : (p_k \in N_c(j))\})$;

7 $fusion := res(currTag) \cup \{\langle k, \bullet, prevTag\rangle \in res(prevTag) : \langle k, \bullet, currTag\rangle \notin res(currTag)\}$;

8 $p_j \to_G p_k$:= true if there is a path from p_j to p_k in G;

9 **do forever begin**

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

10 $responses \leftarrow \{\langle k, \bullet, rules\rangle \in responses : k \neq i \land (\exists_{x \in \{currTag, prevTag\}} \langle k, \bullet, rules\rangle \in res(x) \land p_i \to_{G(res(x))} p_k \land \langle i, \bullet, x\rangle \in rules)\} \cup \{\langle i, N_c(i), \emptyset, \emptyset\rangle\}$;

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

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

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

13 $(newRound, prevTag) \leftarrow (true, currTag)$; $currTag \leftarrow nextTag()$;

14 $responses \leftarrow responses \setminus \{\langle j, \bullet\rangle \in res(currTag) : p_j \in P\}$;

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

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

16 **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 a manager; remove unreachable managers on new rounds and nodes with no rules */

17 **let** $M := \{p_k \in Mng : (\exists_{r \in Rul}\, r = \langle k, \bullet\rangle) \land (\neg newRound \lor p_i \to_{G(res(prevTag))} p_k)\} \cup \{p_i\}$;

18 $msg \leftarrow msg \cup \{(p_j, \langle \text{'}delMngr\text{'}, k\rangle) : p_k \in (Mng \setminus M)\} \cup \{(p_j, \langle \text{'}addMngr\text{'}, i\rangle)\}$;

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

19 $msg \leftarrow msg \cup \{(p_j, \langle \text{'}delAllRules\text{'}, k\rangle) : (\exists_{r \in Rul}\, r = \langle k, \bullet\rangle) \land p_k \notin M\}$;

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

20 $msg \leftarrow msg \cup \{(p_j, \langle \text{'}updateRules\text{'}, myRules(G(res(referTag)), j, currTag)\rangle)\}$;

21 **foreach** $p_j : p_i \to_{G(fusion)} p_j$ **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;

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

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

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

25 **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

nodes and ends when it receives replies from all of these nodes (cf. lines 6 and 8, 12–14, as well as, Section II-C). When a query response arrives at p_i, before the update of the response set (line 24), p_i checks that there is sufficient storage space for the arriving response (line 23). If space is lacking, p_i performs what we call a 'C-reset'. Note that p_i stores responses only for the current round, $currTag$. Controller p_i replies to other controllers' queries in line 25.

Controller $p_i \in P_C$ keeps a local state of query responses (cf. Section II-A) from other nodes (line 2). The responses accumulate information about the network topology according to which the switch configurations are updated in each round. Algorithm 1's do-forever loop (lines 9–21) provides these functionalities.

Establishing communication between every controller and every node in the network. 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 link state monitoring (cf. Section II-B1). The algorithm considers 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 2). 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 II-C), p_i and p_j can exchange massages eventually.

The above iterative process of network topology discovery and the process of rule installation consider κ-fault-resilient flows (cf. Section II-B2 and $myRules()$ function in Section II-C). These flows are computed through the interface $myRules(G, j, tag)$ (line 4). Once the entire 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 these 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.

Topology discovery 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 10). 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. 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 12) and thus p_i can restart the process of discovering the network (lines13–14).

The start of a new (rediscovery) round, allows p_i to also remove information at the switches that is related to any unreachable controller $p_k \in P_C$ (assuming that it had succeeded in discovering the network and bootstrapped communication). During new rounds, p_i removes information related to p_k from any switch p_j (lines 17–19); 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 IV. The do-forever loop of Algorithm 1 completes by sending rule (line 20) 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 21–21). Note that each of these configuration updates are done via a single message that aggregates all commands for a given destination.

Keeping the Switch Accessible. In order to render the network self-stabilizing, we ensure that initial mis-configurations are removed eventually. 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 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 II-A).

IV. Correctness Proof

We prove Algorithm 1's correctness by showing that (i) when the system starts in an arbitrary state, it reaches a legitimate state within a bounded period (Theorem 1), i.e., Figure 2's recovery period is bounded. Also, (ii) when starting from a legitimate state and letting a bounded number of failures to occur, the system returns to a legitimate state within a bounded period.

Refined model. We measure the recovery period of Algorithm 1 in terms of frames. The first *(asynchronous) frame* of run R is the shortest prefix R' of R in which every controller starts and ends at least one complete round-trip with every node of its discovered topology. The second frame in R is the first frame in R'', which is R's suffix that starts after R', and so on. We denote by Δ the recovery time of an end-to-end channel [12] and the self-stabilizing algorithm for unique label generation [2].

Definition 1 (Legitimate System State)**.** *We say that $c \in R$ is Algorithm 1's legitimate state when: (1) the controllers have the correct information about the nodes in the network, (2) any controller is the manager of every switch and only these controllers can be the managers of any switch, (3) the rules installed in the switches encode κ-fault-resilient flows between all controllers and nodes, and (4) the end-to-end and round synchronization protocols are in a legitimate state.*

Theorem 1 bounds the recovery period of Algorithm 1. Due to the page limit, we present a proof sketch for Theorem 1. The detailed proof appears in [8].

Theorem 1. *Within $((\Delta+1)D+1)[(\Delta D+1)\cdot N_S+N_C]$ frames in run R of Algorithm 1, the system reaches a state $c_{safe} \in R$ that is legitimate (Definition 1).*

Proof sketch: **Claim 1.** *Each switch needs to store a bounded number of rules. Each controller needs to store a bounded number of responses and performs at most one C-reset (line 22).* The proof arguments are based on the bounded network size and the memory management scheme of the abstract switch (Section II-A), which guarantees that, during a legal execution, all non-failing controllers can store their rules. The bounded network size also helps to bound, during a legal execution, the amount of memory that each controller needs to have. This proof also bounds the number of C-resets that a controller might take during the period in which the system recovers from transient faults. Note this bound importance; C-resets delete all the information that a controller has. □

The system cannot reach a legitimate state before it removes stale information from every switch configuration. 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 a controller removes erroneously information about another non-failing controller. We refer to these 'mistakes' as illegitimate deletions of rules and note that they occur when the (stale) information that a controller has about the network topology differs from the actual network topology, G_c. 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 the switches again to delete illegitimately rules of other controllers.

Claim 2. *There is a bounded number of steps in which a controller instructs the switches to perform illegitimate deletions (due to stale information).* Consider a starting state in which the controller is just about to take a step that instructs the switches to perform an illegitimate deletion. We argue that between any two such instructions, the controller has to aggregate information about the network in a way that it preserves (erroneously) the complete network topology. This can only happen after receiving a reply from every node in the (erroneously) preserved topology. 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 $k \cdot (\Delta+1)+1$ times in which p_i instructs 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. Thus, the total number of times that a controller instructs an illegitimate deletion is at most $D \cdot (\Delta + 1) + 1$. □

Claim 3. *Algorithm 1 recovers from transient faults.* Consider a period in which there are no C-resets and no illegitimate deletions. In such a period, all the controllers construct κ-fault-resilient flows to any other network node. 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 $(\Delta + 1)k$ frames, p_i discovers correctly its distance-k neighborhood and establishes a communication channel between p_i and p_j. This means that within $(\Delta + 1)D$ frames in which there are no C-resets and no illegitimate deletions, the system reaches a legitimate state c_{safe}. Moreover, within $((\Delta+1)D+1)[(\Delta D+1) \cdot N_S + N_C]$ frames, R has a period of $(\Delta+1)D+1$ frames in which there are no C-resets and no illegitimate deletions and thus the system reaches c_{safe}. □ ■

Our proof also shows that, when starting from a legitimate state and then letting a single link in the network to be added or removed from G_c, the system recovers within $O(D)$ frames. The arguments here consider the number of frames it takes for each controller to notice the change and to update all the switches. By similar arguments, we show that, within $O(D)$ frames, the system recovers after the addition or removal of at most $N_C - 1$ controllers in a legitimate system state.

V. Evaluation

We evaluate our approach and study *Renaissance*'s performance. We implemented a prototype using Open vSwitch (OVS) and Floodlight. The prototype source code will be released together with this paper.

Setup. The experiments are conducted using PCs with Ubuntu 16.04.1 OS, Intel Core i7-2600K CPU at

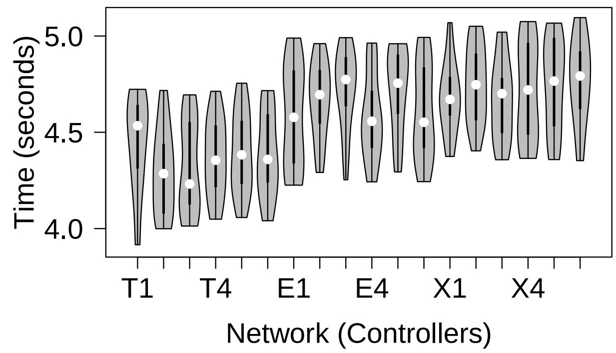

Fig. 3: Bootstrap time for Telstra (T), EBONE (E) and Exodus (Ex) for 1 to 7 controllers.

3.40GHz (8CPUs) with 16GB RAM. 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 requests and install flows once per second. Paths are computed according to Breadth First Search (BFS) and we use OpenFlow fast-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.

How efficiently can *Renaissance* bootstrap an SDN (resp. handle transient failures)? 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 terms of performance, as expected, the stabilization time is proportional to the network diameter and also depends on the number of controllers (Figure 3): 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 4).

How efficiently does *Renaissance* recover in the presence of benign failures (link and node failures)? We consider different types of benign failures: controller fail-stop, permanent switch-failure, and permanent link-failure. The experiments start from a legitimate system

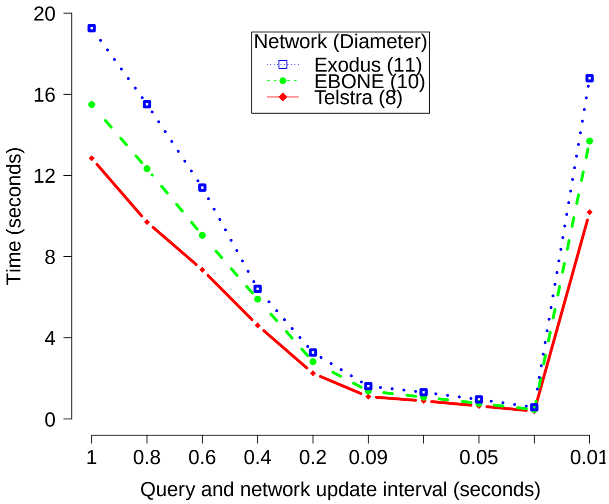

Fig. 4: Bootstrap time for Telstra, EBONE and Exodus using 7 controllers, as a function of query intervals.

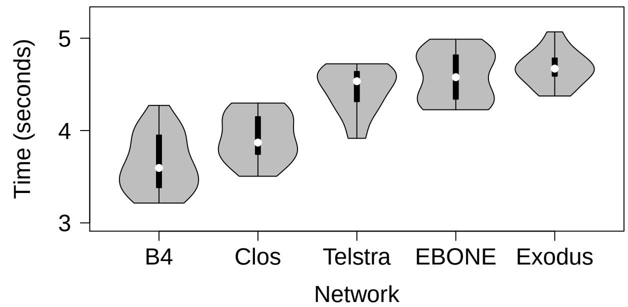

Fig. 5: Recovery time after fail-stop failure of a controller.

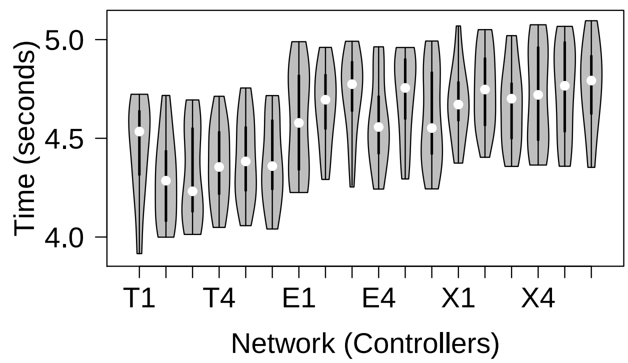

Fig. 6: recovery time after fail-stop failure for 1-6 controllers in Telstra (T), EBONE (E) and Exodus (Ex).

state, to which we apply these failures.

In the *fail-stop* failure experiments (figures 5 and 6), 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.

For the experiments for *permanent link-failures* (Figure 7), 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 (after a fail-stop failure of a controller) is roughly linear in the number of nodes (Figure 5). 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. The number of failed controllers (in Figure 6) does not affect the time much, as expected.

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 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 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. 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 occurs. We note that our prototype utilizes packet tagging for incremental update [24]. This helps to avoid another drop in throughput as a result of the new paths being installed in order to repair flows while destroying the old backup paths.

We can see in Figure 8 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 recov-

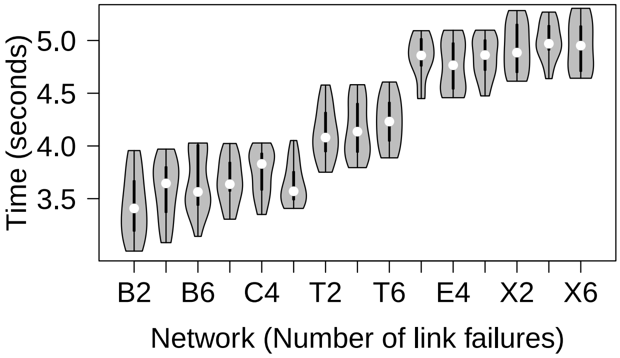

Fig. 7: Recovery time after multiple (2,4,6) permanent link-failures at random for B4 (B), Clos (C), Telstra (T), EBONE (E) and Exodus (Ex).

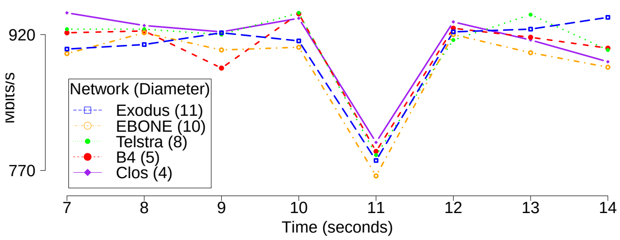

Fig. 8: Throughput for the different networks.

ery 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 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.

VI. Related Work

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 [28], Kandoo [13], STN [7]. 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 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 [23], or alternatives spanning trees [6], [19].

Furthermore, there exists interesting work on bootstrapping connectivity in an OpenFlow network [15], [27] (that does not consider self-stabilization). In contrast to our paper, Sharma et al. [27] 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.

We are not the first to consider self-stabilization in the presence of faults that are not just transient faults (see [11], Chapter 6). Thus far, self-stabilizing algorithms consider networks in which all nodes can compute and communicate. In the context of the studied problem, some nodes (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. This proof uses a number of techniques that are unique to the area, such as the one for assuring a bounded number of resets and illegitimate deletions. We reported on preliminary insights in two short papers on *Medieval* [25], [26]. However, Medieval is not self-stabilizing because its design depends on the presence of non-corrupted configuration data.

VII. Conclusion

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 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. In order to facilitate future research, we will release the prototype source code together with this paper.

References

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

[2] N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *J. Comput. Syst. Sci.*, 81(4):692–701, 2015.

[3] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. M. Parulkar. ONOS: towards an open, distributed SDN OS. In A. Akella and A. 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] P. Blanchard, S. Dolev, J. Beauquier, and S. Delaët. Practically self-stabilizing paxos replicated state-machine. In *Proc. 2nd International Conference on Networked Systems (NETYS)*, pages 99–121, 2014.

[5] M. Borokhovich, L. Schiff, and S. 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] M. Borokhovich, L. Schiff, and S. Schmid. Provable Data Plane Connectivity with Local Fast Failover: Introducing OpenFlow Graph Algorithms. In *HotSDN*, 2014.

[7] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *INFOCOM*, 2015.

[8] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid. Self-stabilazing sdns. Technical report, Department of Computer Science and Engineering, Chalmers University of Technology, Rännvägen 6B, S-412 96 (Göteborg) Sweden, 06 2017. Technical report 2017:04 and also to appear soon on arXiv with this submission's title.

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

[10] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.

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

[12] S. Dolev, A. Hanemann, E. M. Schiller, and S. 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] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proc. ACM HotSDN*, 2012.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[15] R. Katiyar, P. Pawar, A. Gupta, and K. 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] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

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

[18] J. Liu, B. Yang, S. Shenker, and M. Schapira. Data-driven network connectivity. In *Proc. ACM HotNets*, page 8, 2011.

[19] M. Parter. Dual Failure Resilient BFS Structure. In *PODC*, 2015.

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

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

[22] R. J. Perlman. An algorithm for distributed computation of a spanningtree in an extended LAN. In W. Lidinsky and B. 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.

[23] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *HotSDN*, 2013.

[24] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: change you can believe in! In H. Balakrishnan, D. Katabi, A. Akella, and I. 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.

[25] L. Schiff, P. Kuznetsov, and S. Schmid. In-Band Synchronization for Distributed SDN Control Planes. *SIGCOMM Comput. Commun. Rev.*, 46(1), Jan. 2016.

[26] L. Schiff, S. Schmid, and M. 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.

[27] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester. In-Band Control, Queuing, and Failure Recovery Functionalities for OpenFlow. *IEEE Network*, 30(1), January 2016.

[28] S. H. Yeganeh and Y. Ganjali. Beehive: Simple Distributed Programming in Software-Defined Networks. In *Proc. ACM SOSR*, 2016.