Received October 22, 2019, accepted November 10, 2019, date of publication November 25, 2019,
date of current version December 10, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2955554

FCR: Fast and Consistent Controller-Replication in Software Defined Networking

MAAZ MOHIUDDIN, MIA PRIMORAC, ELENI STAI, (Member, IEEE), AND JEAN-YVES LE BOUDEC, (Fellow, IEEE)
École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland

Corresponding author: Mia Primorac (mia.primorac@epfl.ch)

This work was supported in part by the SNSF National Research Programme Energy Turnaround (NRP 70) and in part by the Swiss Centre for Competence in Energy Research on the Future Swiss Electrical Infrastructure (Innosuisse SCCER Program).

ABSTRACT We consider the problem of coordination among replicated SDN controllers, where the challenge is to ensure a consistent view of the network while reacting to network events in a prompt manner. Existing solutions are either consensus-based, which achieve consistency at the expense of high latency; or eventual-consistency-based, which have low latency at the expense of severe limitations on the types of applications and policies implementable by the controller. We propose the Fast and Consistent Controller-Replication (FCR) scheme. FCR is based on a deterministic agreement mechanism that performs agreement on the input of controllers, instead of agreement on the output as done in consensus mechanisms. We formally prove that FCR provides the same guarantees in terms of implementable applications and network policies, as any deterministic single-image controller. Through simulation and implementation, we show that these guarantees can be implemented with little latency overhead, compared to eventual-consistency approaches, and can be achieved significantly faster than consensus-based approaches.

INDEX TERMS Software defined networking, control plane consistency, latency overhead, replicated SDN controllers.

I. INTRODUCTION

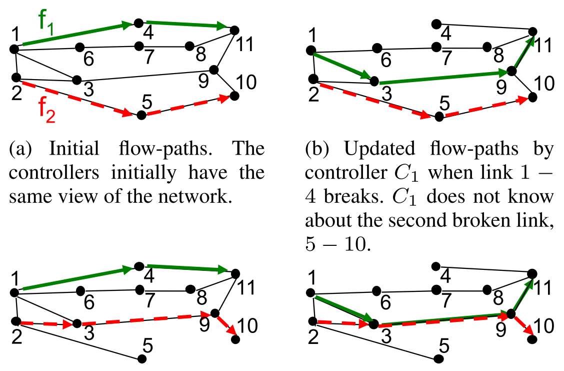
Software Defined Networking (SDN) aims to simplify the control and management of network infrastructures by relying on a "logically-centralized" controller with a global view of the network to manage the network state and to implement networking policies. In practice, a highly-available logically-centralized controller is implemented by replicating a single-image controller to address controller failures. This is orthogonal to using multiple controllers with sharded state for scalability, not studied in our work. The challenge with replication is to ensure control-plane consistency with low latency — and addressing this challenge is the focus of our paper.

In literature, two types of approaches are proposed for control-plane consistency for replicated single-image controllers: (1) high-latency, consensus-based approaches that guarantee (strong) consistency, *e.g.*, Onix [1], Ravana [2] and ONOS [3], and (2) low-latency and high-availability approaches, *e.g.*, SCL [4] that guarantees eventual consistency [5]. Eventual consistency ensures that in the absence of network events, all controllers will eventually have the correct view of the network (topology and desired policy), and that the forwarding rules installed in the network will eventually be same as those computed by a single-image controller using the same view. Replication schemes that guarantee eventual consistency do not wait for *agreement* between replicas; they proceed with update installation as soon as a replica is aware of an event. FCR obtains the best of both worlds, *i.e.,* it guarantees consistency (like consensus mechanisms), and has low latency (like eventually-consistent schemes).

A. THE CONTROL PLANE NEEDS CONSISTENCY

Developing SDN applications is simpler if one can rely on strong consistency guarantees, e.g., the Google's globally deployed software defined WAN B4 [6] uses Paxos for leader election for every controller functionality. Moreover, although eventual consistency is sufficient to provide some safety policies such as way-pointing and node isolation [4], it cannot guarantee a large number of safety policies that refer to more than one flow at a time such as edge isolation, node disjoint isolation, edge disjoint isolation [7], etc. These safety policies are required by applications that, e.g., need isolated resources (switches or physical links) in order to exchange privacy-sensitive data while sharing network infrastructure, as traffic patterns can reveal information.

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.



(a) Initial flow-paths. The controllers initially have the same view of the network.

(b) Updated flow-paths by controller C_1 when link $1-4$ breaks. C_1 does not know about the second broken link, $5-10$.

(c) Updated flow-paths by controller C_2 when link $5-10$ breaks. C_2 does not know about the second broken link, $1-4$

(d) Installed flow-paths by C_1 first and then C_2. Edge disjoint isolation of link $3-9$ is violated, while correct flow-paths do exist (1-6-7-8-11 and 2-3-9-10).

FIGURE 1. Example of edge disjoint isolation violation under an eventual consistent scheme. Edge disjoint isolation means that the paths of flows f_1, f_2 should not share a common link. The example network has 11 switches and 2 controllers.

We illustrate the problem with eventual consistency and one such safety policy: edge disjoint isolation. Consider the network state shown in Figure 1(a). Two flows, f_1 and f_2, start from switches 1 and 2, and terminate at switches 11 and 10, respectively. Two controllers, C_1 and C_2, initially have the same view of the network. They both implement shortest-path routing, as well as edge disjoint isolation safety policy, *i.e.,* the paths of flows f_1, f_2 should not share a common link. The initial routing paths of flows f_1 and f_2 are $1-4-11$ and $2-5-10$, respectively. Let two events occur: links $1-4$ (event e_1) and $5-10$ (event e_2) break almost simultaneously (Figure 1(d)). Due to different network delays, C_1 first learns only about e_1, and C_2 first learns about e_2. They do not ensure agreement, and promptly issue updates to modify the routing tables of the switches on the old and new paths. We assume that a switch uses the last installed update for a flow to serve packets of that flow. Moreover, in SCL [4], the controller uses the last known state from a switch to compute updates. As a result, C_1 and C_2 compute the new routes as shown in Figures 1(b) and 1(c), respectively. Each controller considers the edge disjoint isolation to be satisfied, but, an interleaving of the updates from C_1 and C_2 can violate the safety policy as shown in Figure 1(d). The state in Figure 1(d) could be reached if, first, C_1 installs its updates on all involved switches, and second, C_2 installs its updates, but the update from C_2 for switch 1 is delayed due to e.g., network losses. Conversely, if the controllers had a consistent view of the network, *i.e.,* were in agreement, they would have computed the paths: f_1 via $1-6-7-8-11$ and f_2 via $2-3-9-10$.

Existing eventually consistent schemes such as SCL [4] do not support reactive applications, such as NATs and firewalls, for which the first packet triggers an update installation and subsequent packets have to be handled by rules that causally follow from the first rule. Although labels obtained from vector clocks [8] can provide causal order, they only provide a partial order between updates from different controller replicas, as vector clocks might be incomparable. Installing only causally related updates from different replicas requires a total order among messages. In state-of-the art solutions that satisfy all safety policies or support all applications in an SDN, controller replicas typically obtain a total order by using high-latency consensus on the real-time path to agree on the logical clocks.

B. THE CONTROL PLANE NEEDS LOW LATENCY

After decades of optimizing dataplanes, recent research efforts reveal the growing need for high performance and tight latency guarantees of control planes [4], [9], [10]. FCR is our effort to make a step in the same direction.

The number of SDN use cases that require concrete control plane performance guarantees has been growing recently in security systems, virtual networks, as well as datacenter environments [9]. For instance, service chaining SDN applications [11] require fast reconfiguration to ensure network correctness. Furthermore, traffic engineering SDN applications, such as that of Google's B4 WAN [6], require frequent reconfiguration to improve network performance. In cases when controllers reside on a single rack the coordination delays are negligible even with Paxos, but coordination can significantly delay response to failures and other events when controllers are spread across a WAN.

C. CONTRIBUTIONS AND TRADE-OFFS

Main Contribution: We developed the Fast and Consistent Controller-Replication (FCR), a distributed coordination layer connecting single-image controller replicas and switches in SDN. FCR guarantees strong consistency, but presents low-latency; this combination makes it important and fills the literature gap between low latency eventual consistent schemes and high latency linearizable schemes. FCR, contrary to the eventual consistent schemes (SCL), supports reactive applications, e.g., NATs and firewalls.

To develop FCR we leveraged on the following design principles and mechanisms: (1) the SCL [4] architecture, which consists of controller proxies and switch proxies, (2) Quarts [12], a low-latency agreement mechanism used to achieve agreement among controller replicas, (3) intentionality clocks [13] to achieve a total ordering of events. Combining these and constructing an efficient (i.e., low latency, low bandwidth and highly available) and viable design is non-trivial.

Quarts [12] was designed for *pseudo-synchronous* distributed systems (see §III). Applying Quarts to SDNs, that are asynchronous, would require Quarts to be implemented inside the logic of the controller, which is impractical.

Contribution 1: FCR ports Quarts to SDNs with minimal modifications to SDN controllers and the switches.

Note that, due to the FLP impossibility result [14], no consensus algorithm can guarantee agreement and termination in an SDN. Thus, Quarts only guarantees agreement at the

TABLE 1. Comparison of replication schemes.

Convergence latency § VI-B	SCL $<$ FCR $<<$ Consensus
Availability § VI-B	SCL$>$ FCR $>>$ Consensus
Bandwidth overhead § VI-B	Consensus $<$ SCL $=$ FCR
Consistency guarantee § VI-A	SCL: *eventual consistency* FCR: *strong consistency* Consensus: *linearizability*

expense of termination. Some controller replicas can fail to terminate successfully, and are thereby prevented from issuing inconsistent updates. However, with Quarts the probability of unsuccessful termination is low, thus providing much higher availability than conventional consensus mechanisms, while maintaining low latency. An overview of this mechanism is presented in § III.

Naively applying Quarts to SDNs requires that controller replicas probe all switches for each triggered computation; this has a high bandwidth-overhead.

Contribution 2: FCR satisfies the probing requirements of Quarts while maintaining low bandwidth-overhead by using cached version of switches' state (see § III and § VII).

To achieve the total order of messages without using consensus, FCR uses intentionality clocks [13] that were designed for real-time cyber-physical systems. Intentionality clocks are logical clocks similar to Lamport clocks [15] and vector clocks [8], and are applied to distributed systems with a centralized (possibly replicated) controller and reactive distributed agents. An SDN system does not conform to this model.

Contribution 3: FCR adapts intentionality clocks to SDNs, by appropriately modifying the conditions under which their values are incremented (discussed in § III).

We formally prove that FCR can be used to implement all the safety policies like the underlying single-image controller. Finally, we prove that FCR adds bounded latency-overhead for agreement.

Table 1 qualitatively highlights the advantages and trade-offs of FCR and where it belongs in the design space compared to the state-of-the-art solutions. FCR outperforms Consensus in latency and availability at the cost of bandwidth overhead and a strong consistency instead of linearizability guarantee - which is though a strong-enough guarantee for the SDN applications. Furthermore, FCR provides strong instead of eventual consistency guarantee compared to SCL, at the cost of slightly higher latency and lower availability. We evaluate FCR in § VI through a discrete-event simulation to study availability, response and convergence time, and consistency. We compare the performance of FCR to that of SCL [4] and consensus-based schemes [2], [3] in datacenter and ISP topologies. We find that, with FCR, the median convergence time after an event is $0.5\times$ that of Ravana [2] and consensus [3]. The latency improvement is more profound at the 99^{th} percentile of convergence time, with FCR being $170\times$ faster than consensus and Ravana. When compared to the eventually-consistent SCL, the median latency of FCR is twice that of SCL while the tail latency is comparable. However, unlike SCL, FCR can also implement (1) NATs and firewalls, and (2) safety policies that refer to more than one flow (see §VI).

We show the applicability of our solution through a proof-of-concept implementation (§VII), built by extending SCL's source code [16]. We compare our implementation against the original SCL one and show that our solution has comparable response and convergence time.

II. SYSTEM MODEL

We consider multiple replicas of a single-image controller (e.g., POX [17], Ryu [18]) that communicate and install updates on switches.

The network can drop, delay and reorder packets. Links in the network can fail at any time. One-way propagation delay between any two end-points is bounded by δ_n - everything beyond is considered to be a delay fault or a loss. We assume that each controller is susceptible to crash and delay faults [19] - it can stop functioning or be intermittently slow in issuing updates. Thus, we have a crash-recover fault-model [20]. The controllers are assumed to be non-susceptible to arbitrary Byzantine faults [21].

We require the following from the single-image controller:

Controller applications are deterministic. If given the same network state, each replica will compute the same set of updates. This requirement is present in all the previous works [2], [4], [12].

Controllers trigger computation upon receiving an event. The controllers recompute the state using messages from the switches as well as a cached version of switches' state.

We relax some constraints of SCL. In addition to proactive controller applications that compute updates based on the network state, we enable reactive controller applications that respond to individual packet-ins, e.g., NAT and firewall.

As normally done in SDN deployments, we assume that switches communicate only with controllers and not with each other. Idempotent behavior is required when a switch installs updates, i.e., installing the same update twice has the same result as installing it once. FCR ensures control-plane consistency by performing agreement. However, for dataplane consistency, we rely on existing mechanisms for consistent-update installation, included in the design for completeness, but we claim no novelty for it. Specifically, we integrate the labeling-based mechanism proposed in [22].

III. BACKGROUND AND CHALLENGES

In this section, we describe Quarts and intentionality clocks, the two building blocks of our design, borrowed from the literature on real-time cyber-physical systems [12], [13]. We discuss the requirements on distributed systems for applying these mechanisms and how FCR realizes them in an SDN.

Quarts [12] is an agreement protocol for replicated controllers in pseudo-synchronous cyber-physical systems. These systems comprise of a central controller that receives inputs from distributed agents, in well-defined rounds, such that all agents are required to send inputs at roughly the same time. These inputs are of the form (r, i, m), where r is a label of the round number common to all messages within the round, i is a unique identifier of the agent, and m is the value. The total number of agents and their IDs are known *a priori*. If two controller replicas receive messages (r, i_1, m_1) and (r, i_2, m_2) that have the same round label r, and the same ID $i_1 = i_2$, they must have also the same value, *i.e.*, $m_1 = m_2$. Consequently, each controller replica has a vector v of uniquely identified messages. Also, each controller replica has state that is labeled with a round label r, and is used along with these messages to perform computation.

Quarts works in two phases, collection and voting. In the collection phase, each controller replica (1) queries its peers for the missing message values from switches in the current round and (2) responds to received queries if it knows the corresponding message value. In this way, each replica tries to maximize the size of its vector v. After collection, each replica sends a *digest* to all other replicas in order to communicate the available inputs for the round. The digest comprises a state label and a vector of zeros and ones, where ones are corresponding to the IDs present in v, at that replica. Finally, each replica performs deterministic voting on the received digests from all replicas. Voting is based on plurality [23]: a digest appearing more often than others is chosen. Ties break with a predefined static priority on digests.

The collection and voting phases last for a bounded time, $2\delta_n$ and $3\delta_n$, respectively. After a bounded-delay of $5\delta_n$, at each replica, Quarts outputs the following: (1) A vector v' of uniquely identified messages chosen after agreement, and (2) a boolean flag *success*. For all replicas on which *success* is set to *True* for a given label r, the vectors v' are exactly the same. Each index in v' is either empty (holds no message), or holds the same message across all replicas with a *True* flag. Thus, all successful replicas have the same set of chosen messages, thereby ensuring control-plane consistency.

Quarts was previously deployed in systems where the distributed agents send events to the controller, either periodically or when polled, with the events labeled with a round number. This behavior does not naturally correspond to that of switches, which are subject to asynchronous events. In order to apply Quarts in such as setting, we require that switches label events with an intentionality clock (see below) derived from the latest messages received from the controller. These labels are used by the controller to cluster events from different switches that happen roughly at the same time into one control round. This contrasts to the previous application of Quarts in the sense that within a typical control round, only a subset of the switches send events.

As only the switches with an event communicate with the SDN controller, the controller does not have the most recent state from other switches. As described in § II, the SDN controller uses the last known state. Obtaining recent state from switches after each event has high bandwidth overhead. Instead, FCR uses a cached version of the state from all switches, called a Baseline Network Snapshot (BNS), that is updated consistently across all controller replicas using off-the-shelf consensus - but on the *non-real-time path*. However, due to arbitrary delays in computation, two controller replicas might use two different BNSs in computation. Preventing this behavior without any additional latency overhead is non-trivial. To address this issue, FCR adds the BNS as the state input to Quarts, as described in § IV-C.

Reactive controller applications such as NAT require a total order among updates sent to the switches. This also requires labeling of events and updates in the SDN. We adapt the intentionality clock from [13]; this is a scalar logical clock, like Lamport clock, designed for systems with centralized controller and reactive agents. Unlike Lamport clocks, where each agent increments its logical clock by one on receiving a message, intentionality clocks are only incremented by one when the controller performs a computation.

In systems with reactive distributed agents, the central controller drives the clock ahead. However, in SDNs, the switches not only react to updates from controllers, but also to network events, whereby they send the events to the controllers. If the switch does not increment its logical clock and sends the new event with the old clock value, it might violate the requirement of Quarts that for the same r, messages must have the same value. We modified intentionality clocks by allowing for switches to increment (by one) their logical clock as soon as they experience an event instead of incrementing the intentionality clock when the controller computes (see §IV-B). As FCR uses scalar logical clocks, any two labels are comparable. Messages with the same label and same ID are considered to be equivalent, *i.e.,* it is the same message, as discussed in §V. Thus, a total order among all messages is achieved.

IV. FCR DESIGN

FCR acts as a coordination layer for single-image controllers (e.g., POX, Ryu). FCR consists of two components: the Quarts Switch Proxy (QSP) and the Quarts Controller Proxy (QCP). There is one QCP on each controller and one QSP on each switch. In Figure 1, there are 11 QSPs and 2 QCPs.

The number of QCPs (`n_qcp`) and QSPs (`n_qsp`) are assumed to be constant and known to all the QCPs. Additionally, in order to distinguish between messages from different controllers and switches, each QSP and QCP is given a unique ID. Addition and removal of both switches and controllers, and planned policy changes are all assumed to be infrequent; they are handled by a policy coordinator, using a non time-critical mechanism such as two-phase commit [24] between the policy coordinator and the controller replicas.

A QSP receives OpenFlow [25] events from its switch and relays the state of the switch to the QCPs in the form of *Status* messages (*Status*). The events can be either network events, that depend on the topology of the network (e.g. a change of a port status), or packet-ins. In the example in Figure 1, the QSPs on switches 1 and 4 send a *Status* message to the QCPs after event e_1, and those on switches 5 and 10 do so after e_2. Due to network losses, only the QCP of C_1 receives a message from the QSPs of switches 1 and 4 for the event e_1, while the QCP of C_2 receives a message from the QSPs of 5, 10 for the event e_2.

The operation of QCP is summarized as follows:

- On non-real-time path, periodically probe the QSPs. Maintain a consistent BNS on QCPs using consensus.
- On real-time path, perform Quarts before computing and issuing updates to QSPs.
- If some switch-states are missing while computing the updates, use their state in the BNS decided by Quarts.

Each QCP stores a BNS with a corresponding label. The BNS is regularly updated using an out-of-band mechanism that employs consensus and BNSs with the same label are the same. When a QCP receives a *Status* message from a switch, it initiates Quarts. Specifically, it enters the collection phase of Quarts and attempts to collect all the *Status* message for this round as discussed in §III. The collection phase is followed by the voting phase. There are two possible outcomes of the voting phase.

Success: In the example in Figure 1, assume that after the collection phase, the QCPs of C_1 and C_2 have the *Status* messages from all QSPs that received an event, i.e., 1, 4, 5, 10. Also, assume that both store the BNS with label 4. Then, their digests have value 4 for the BNS label, 1 for switches 1, 4, 5, 10 and zero for the rest of the switches, i.e., they are written as 4.10011000010. The QCPs at C_1 and C_2 exchange digests and choose their common digest. They both have success set to `True` because (i) they have all the required *Status* messages in the digest to perform a successful computation and (i) they store BNSs with the same label. For all switches with zero value in the digest, their state is assigned according to the (common) BNS. Thus, they relay the *Status* messages to C_1, C_2 as OpenFlow messages that are then used to compute and issue updates. In this way, the edge-isolation property will be ensured (e.g., they decide f_1 via $1-6-7-8-11$ and f_2 via $2-3-9-10$).

Failure: In the case of two controllers, if after collection C_1 and C_2 have different partial digests, no *Status* messages will be relayed to controllers and no updates will be issued. Note that for more than two controllers, the QCPs, the partial digests of which win the plurality vote, may still compute and install updates. If a QSP does not receive an update in response to its events, either due to network losses or due to the controllers not computing because of failed agreement, the QSP will trigger a new round of agreement by sending a new *Status* message after a timeout of T_{ret}.

Algorithms 1 and 2 describe the design of QCP and QSP, respectively. The contents of FCR messages are presented in §IV-A. To ensure total order among these messages, they are labeled using intentionality clocks, described in §IV-B.

A. FCR MESSAGES & FUNCTIONS

FCR uses four types of messages for communication between a QSP and a QCP, namely *Probe*, *Status*, *Sync_Status* and *Update*. Each message except *Probe* has a label l. The other fields in the individual messages are described below.

• *Probe* is used by a QCP to query a QSP for the current state of its switch. This is used for periodically updating the BNS that is used in computations for switches with a zero-entry in the chosen Quarts digest, i.e., without recent events.

• Status$<l, i, m>$ message is sent by a QSP to advertise the state of its switch after an event. i is the ID of the QSP and m is the message body. A controller receives a *Status* and uses it to recreate the state of the network, and computes updates. This approach is similar to SCL's [4]. Although SCL allows only for network events that signal topology changes, FCR also allows for packet-ins (PACKET_IN in OpenFlow [25]) that require a per-packet update from the controllers.

A *Status* message contains an FCR header, the QSP logical clock, the status of all the switch ports (*up* or *down*) and the list of events since the last received update, including packet-ins. We do not require sending the entire content of the flow tables as it can be recomputed by the controllers. When another event happens before the current update is installed, all subsequent packet-ins are appended to the previous *Status* message (Algorithm 2 lines 9, 11) to create a new *Status* message. Also, old network events are potentially overwritten by new ones. After an update installation the packet events will not be included in the next *Status* message.

• *Sync_Status*$<l, i, m>$ is used by a QSP to advertise the state of the corresponding switch, as a response to a probe from a QCP. It bears the same description with the *Status* message.

• *Update*$<l, ack, m>$ is a routing update to be installed on the switch, sent by a QCP to a QSP. *ack* indicates to the switch if its *Status* message was used in computation of this update, in which case, ack $=$ `True`, else it is `False`. m is the body of the message that contains the actual routing updates.

In addition to these messages, the QCPs also exchange other messages for agreement, as described in §IV-C.

The main actions of a QCP are (i) sending probe messages at boot time and after a timeout of T_{probe} to all QSPs (Algorithm 1 line 8), (ii) initiating Quarts for achieving agreement on the input among QCPs upon the reception of a *Status* message from a QSP (Algorithm 1 line 17), (iii) sending the agreed input (output of Quarts) to the controller that then computes the updates (Algorithm 1 line 37), and (iv) sending the *Update* messages, with the controller's updates, to the concerned QSPs (Algorithm 1 line 46).

The main actions of a QSP are (i) sending a *Status* message to QCPs when receiving an event from its switch (Algorithm 2 line 6), (ii) sending a *Sync_Status* message as response to a *Probe* received from a QCP (Algorithm 2 line 16), (iii) sending *Status* messages to QCPs if there exist events for which it has not received an update before a timeout of T_{ret} (Algorithm 2 line 19), and (iv) receiving the *Update* messages and sending only the valid ones to the switch (Algorithm 2 line 23).

B. ORDERING FCR MESSAGES

Ensuring control-plane consistency requires that all the QCPs have the same message ordering. QSPs export their state using *Status* and *Sync_Status* messages. The state changes when (1) an event occurs at a QSP, or (2) a QSP installs an update, whereby it acknowledges pending events.

FCR implements modified intentionality clocks [13] to capture the notion of a snapshot in an SDN. Each QSP and QCP maintain a local logical-clock, C. Each outgoing message is tagged with a label. The label is obtained as the value of the local logical-clock right before sending the message. The logical clock is strictly monotonically increasing by the following rules: (1) each QSP increments its logical clock by one, when it receives an event from the switch or when it experiences a timeout (Algorithm 2 lines 7 and 20). (2) On receiving a message, the logical-clock at each recipient is updated to the maximum of the local logical-clock and the received label. This is important for synchronizing labels among switches and controllers.

Note that in the original intentionality clocks, the logical clock is incremented by the controller replicas only, *i.e.,* QCPs, because the distributed agents (equivalent to QSPs) are assumed to be reactive. In the case of FCR, if event e_2 occurs after e_1 on the same switch, then the corresponding *Status* messages will have different labels and the label of e_2 must be higher. This property is called local causality. If a controller computes two updates for two different switches in response to the same snapshot, then they will have the same label because they belong to the same snapshot. Furthermore, if two controllers compute updates for the same switch in response to the same snapshot, then the updates will have the same label. Thus, the labels of outgoing messages from a QSP (*Status*) follow the same causal order as the events on its switch. This also enables FCR to support reactive controller applications such as NAT and firewall. The labels ensure that message ordering at the controllers is same as the causal order at each switch. To ensure that the total order is maintained across QSP reboots, QSPs augment the label with an epoch that is incremented at each QSP reboot. The epoch is stored in a persistent storage.

Algorithm 2 illustrates that QSPs always update the current state of the switch, exported in *Status* messages, together with the logical-clock (lines 11 and 20). This ensures that all outgoing messages from a QSP with the same label have the same body - a key requirement of Quarts (see §III).

C. AGREEMENT AT QCP

On the real-time path, QCPs use Quarts [12] for agreement that is initiated with the received *Status* message (Algorithm 1, line 22), the value of the local logical-clock and the local label of the BNS. Quarts is round based, where a round is indicated by a label (e.g., C in Algorithm 1). Quarts terminates after a maximum delay of $5\delta_n$, and returns `success`, an updated S_Crt and the chosen BNS label C^*_{bns} at each QCP. To this end, it uses a deterministic voting function based on plurality voting (§ III, [23]). If, at a QCP, two digests are tied with the same number of votes, then the digest with the highest (predefined) priority is chosen. If `success` = `True`, the chosen set of values (updated S_Crt and BNS label C^*_{bns}) is forwarded to the respective controllers for computation (Algorithm 1 line 44). Note that a possible outcome of Quarts (in Algorithm 1 line 22) is `success` = `False`. In this case, the QCP will not forward the *Status* messages to the controller (Algorithm 1 line 23) and it will prepare for the next round. Conversely, if `success` is `True` on two QCPs in the same round, then Quarts [12] guarantees that their S_Crt and C^*_{bns} are identical.

Recall that the controllers recompute the network state from the received inputs. Thus, if S_Crt does not have the *Status* message from a switch, the state of this switch is assumed to be as in the chosen BNS, i.e., *BNS* with label C^*_{bns} (Algorithm 1, lines 40-44). A switch with an empty value in the BNS is treated as a network partition. Moreover, since the controllers are deterministic (see §II), when two controllers compute updates with the same set of *Status* messages, the resulting *Update* messages will be identical.

Note that we use Quarts as described in [12], without any modifications to the collection and voting phase. In [12], the QCPs perform agreement only once, as it was designed for real-time systems, where performing agreement on messages after their real-time deadline expired is superfluous. This however, is not true in SDNs. If a QCP does not reach agreement after a first attempt, more attempts for agreement can be performed. Since exchanging *Status* messages between the QCPs increases the chance of them having the whole set of messages, more collection rounds result in a higher chance of success in the subsequent voting.

In Algorithm 1, for ease of presentation, the QCP does not process *Status* messages if the controller is busy computing. However, this can be optimized for even lower latency, by performing agreement among QCPs for a higher label while the controller computes updates for an older label.

In order to maintain the most recent baseline snapshot of the network (BNS), FCR uses a routine consisting of (1) a probing mechanism and (2) a conventional consensus algorithm such as Paxos [26] to consistently install the new BNS (Algorithm 1, lines 9-15). This routine is triggered when a QCP starts, in order to bootstrap the initial BNS, and triggered every T_{probe} to update the BNS. As the consensus algorithm is not a contribution of this paper, we use the function `update_baseline_snapshot` as an off-the-shelf utility. This function includes the conventional consensus algorithm and resets the `probe_timer` to T_{probe} on each QCP, on termination of the consensus protocol. In this way, all probes are pseudo-periodic (QCPs that were not involved in a consensus trigger re-synchronization) with a period T_{probe}.

Note that this mechanism is not latency-critical. Therefore, the latency due to the consensus algorithm does not impact the response or convergence time of FCR. The price to pay for using an out-of-band mechanism is that for a QSP that did not have a *Status* message in $\mathbf{S_{Crt}}$, QCPs might not use the most recent state. However, due to the high success rate of Quarts, it is likely that all QCPs have the most recent state for that QSP by having processed its previous state change in a previous iteration of Quarts.

Algorithm 1 QCP Design

```
1  $C \leftarrow 0$;                                   // Logical clock used to label events
2  $C_{bns}, C^*_{bns} \leftarrow 0$;                  // Logical clocks of BNSs
3  **S_Crt** $\leftarrow$ [ ];                         // Vector of current status messages
4  **BNS** $\leftarrow$ [ ];                           // Vectors for baseline snapshot
5  Set probe_timer to $T_{probe}$;                     // Timer to send a Probe message
6  ACK $\leftarrow$ [ ];                               // ACKs to be sent to the switches
7  controller_free $\leftarrow$ True; compute $\leftarrow$ False; qcp_free $\leftarrow$ True;
8  **on** boot or probe_timer fires
9      Send *Probe* to all QSPs;
10     **repeat**
11         **on** reception of *Sync_Status*<$l$, $i$, $m$>
12             **BNS**[$i$] $\leftarrow (l, m)$; $C_{bns} \leftarrow \max(C_{bns}, l)$;
13         **end**;
14     **until** *timer* $2\delta_n$ *expires*;
15     **BNS** $\leftarrow$ update_baseline_snapshot(**BNS**, $C_{bns}$);
16 **end**;
17 **on** reception of *Status*<$l$, $i$, $m$> and qcp_free = True
18     **if** $C < l$ **then**
19         $C \leftarrow l$;                           // New status; update clock
20         **S_Crt** $\leftarrow$ [ ]; **S_Crt**[$i$] $\leftarrow m$;
21         qcp_free $\leftarrow$ False;
22         success, **S_Crt**, $C^*_{bns}$ $\leftarrow$ Quarts(**S_Crt**, $C$, $C_{bns}$);
23         **if** *success* **then**
24             **for** $1 \leq i \leq n\_qsp$ **do**
25                 **if S_Crt**[$i$] $=\perp$ **then**
26                     ACK[i] $\leftarrow$ False;
27                 **else**
28                     ACK[i] $\leftarrow$ True;
29                 **end**
30             **end**
31             compute $\leftarrow$ True;
32         **else**
33             qcp_free = True;
34         **end**
35     **end**
36 **end**;
37 **on** controller_free = True and compute = True
38     controller_free $\leftarrow$ False;
39     compute $\leftarrow$ False;
40     **for** $1 \leq i \leq n\_qsp$ **do**
41         **if S_Crt**[$i$] $=\perp$ *and* **BNS**[$i$].$l = C^*_{bns}$ **then**
42             **S_Crt**[$i$] $\leftarrow$ **BNS**[$i$].$m$;
43     **end**
44     Send **S_Crt** to the controller;
45 **end**;
46 **on** reception of computed updates from the controller
47     **for** *each update m for QSP i* **do**
48         Send *Update*< $C$, $ACK[i]$, $m$> to QSP $i$;
49     **end**
50     qcp_free $\leftarrow$ True; controller_free $\leftarrow$ True;
51 **end**;
```

Algorithm 2 QSP Design

```
1  C $\leftarrow$ reads epoch from persistent storage;
2  $S \leftarrow \perp$;                              // Current status
3  acked $\leftarrow$ True;                          // Was the last event acked?
4  Set event_timer to $T_{ret}$;                     // Timer to send a Status message
5  packet_events $\leftarrow$ [ ];                   // Unacknowledged packet events
6  **on** event $e$ received from the switch
7      $C \leftarrow C + 1$;
8      **if** *e is a packet event* **then**
9          Add $e$ to packet_events;
10     **end**
11     $S \leftarrow$ Current status $\cup$ packet_events;
12     *acked* $\leftarrow$ *False*;
13     Send *Status*< C, id, S>to all QCPs;
14     Set event_timer to $T_{ret}$;                 // Send Status if fires before Update
15 **end**;
16 **on** receive *Probe*
17     Send *Sync_Status*< $C$, id, S>to the QCP;
18 **end**;
19 **on** event_timer fires and acked = False
20     $C \leftarrow C + 1$; $S \leftarrow$ Current status $\cup$ packet_events;
21     Send *Status*< C, id, S>to all QCPs; Set event_timer;
22 **end**;
23 **on** receive *Update*<l, ack, m>
24     **if** $C \leq l$ **then**
25         $C \leftarrow l$;
26         **if** *ack* **then**
27             Send $m$ to the switch;
28             packet_events $\leftarrow$ [ ];
29             acked $\leftarrow$ True; Cancel event_timer;
30         **end**
31     **end**
32 **end**;
```

D. DATAPLANE CONSISTENCY

The agreement mechanism described earlier guarantees control-plane consistency. However, to enforce the safety policies, the updates also need to be installed consistently on all the switches, i.e., dataplane consistency is needed. For consistent update-installation, we rely on prior work [22]: a packet is tagged with a label by the first switch in its data path and all subsequent switches serve it with a rule with the same label. When installing the updates, QSPs use the label in the *Update* messages sent by the QCP to tag the rules as belonging to a particular snapshot of the network.

Additionally, Algorithm 2 line 26, illustrates that an *Update* is only forwarded to a switch if it was computed with the most recent state of the switch. Although we have not found any safety policies that could be violated by not performing this check before installing the update, it is certainly a good practice for liveness. For instance, a QSP could be instructed to forward packets on a port that is down. Without the check, the QSP the would install such an update and acknowledge the pending event (`acked` = true), thereby violating connectivity (a liveness policy).

V. FORMAL GUARANTEES

We provide guarantees for FCR for control-plane consistency and safety policies. The proofs are in Appendix A.

A. CONTROL-PLANE GUARANTEES

Consistency is said to hold for label r if and only if the updates with label r sent by QCPs to QSPs have the same value for the same QSP.

Theorem 1 (FCR Consistency): The design of QCP presented in Algorithm 1 guarantees consistency in the presence of any number of delay- or crash-faulty replicas.

In addition to providing control-plane consistency, it is desirable to have a low-latency overhead due to an agreement mechanism among QCPs. The latency overhead of a QCP is defined as the time spent by the QCP in Algorithm 1, line 22.

Theorem 2 (Bounded Latency-Overhead): The latency overhead of a non-faulty QCP is less than or equal to $5\delta_n$.

B. SAFETY POLICY GUARANTEES

Safety policies must never be violated by a controller. We define a safety policy s as *enforceable* if there exists a single-image controller C that never violates s. Then, s is said to be enforceable by C or C enforces s. Let S_c be the set of the enforceable safety policies by the single-image controller, C. Similarly, we define the set of enforceable policies by a logically centralized controller obtained by replicating C with FCR as S_{FCR}. From prior literature on distributed systems [7], we know that when the control plane is sharded across different controllers, it is impossible to enforce safety policies in S_c. However, we do not consider sharding in this paper, thereby the impossibility result does not apply. We prove the following in Appendix A-C and compare the safety guarantees with SCL in A-D.

Theorem 3 (FCR Safety): For any single-image controller C, each $s \in S_c$ is enforceable by a logically-centralized controller obtained by replicas of C using FCR, i.e., $S_{FCR} = S_c$, under the model described in Section II.

Next, we prove that FCR ensures that events are not lost. This is an important liveness property showing that consistent updates will be eventually computed. For each unacknowledged event at a switch, computation of updates will be re-triggered by *Status* messages from its QSP, until an update is received. We remind that as FCR requires agreement on *Status* messages before deciding on updates for switches, when agreement is not successful, no updates are sent to the switches. A network event that overwrites another network event is denoted as converse event, e.g., a port up event overwrites a previous port down event.

Theorem 4 (No Lost Events): Each event is eventually acknowledged unless a converse event happens.

Moreover, in order to correctly implement reactive controller applications such as NAT and firewall, it is important to ensure that all packets in the flow (after the first) are handled by updates that causally follow after the first update. This requires that any two updates received at a switch (possibly from different controllers) are comparable, i.e., there must exist a total order among updates. The labeling mechanism and control-plane consistency in FCR guarantees that two updates u_1 and u_2 with labels l_1 and l_2, respectively, received at a switch, are identical if $l_1 = l_2$. Alternatively, if $l_1 < l_2$, then u_1 is time-wise before u_2, proving total order. Consequently, we conclude that FCR can be used to implement reactive controller applications.

VI. PERFORMANCE EVALUATION

We use a discrete-event simulation to compare safety and performance properties of FCR with other replication schemes [2], [3], [26], [27]. The dataplane is modeled by a graph of switches and the flows, and the control plane is modeled as `n_qcp` $(= g)$ independent controllers with their computation time drawn according to the combined delay- and crash-fault model of [12], [19]. This fault model is an adaption of the Gilbert-Elliot model [28], where the controller is either in the state of normal operation N or in the crashed state C. When in state N, a controller can either finish its computation within a deadline τ or be delayed with probability θ_d by exponentially distributed delay time. When in state C, the controller is crashed and does not issue updates. The controller enters this state with probability θ_c and returns to state N after a mean-time-to-repair (MTTR) of 30 s. We take $\tau = 10$ ms as the mean update computation time of a non-faulty controller. In FCR, if a switch does not receive updates for an event in less than $T_{ret} = 1$ s since the last *Status* message was sent, its QSP sends the event in a new *Status* to all QCPs to re-initiate the computation of the updates.

We use an out-of-band communication network between the dataplane and control plane that is modeled as probabilistic synchronous [29]. The out-of-band network will drop or delay messages, with a probability p. Messages that are delivered have a maximum delay of δ_n. For the comparative evaluations, we study the following replication schemes:

Ravana [2]: Ravana guarantees consistency and uses primary-standby replication [27]. Before responding to an event, the primary replica synchronizes with the standby replicas using view-stamped replication [30]. The latency overhead in Ravana is caused by the state-synchronization mechanism. We implement the "totally ordered events" flavor of Ravana from [2] to compare with FCR that also provides total ordering among events.

Passive: We implement the "weakest" flavor of Ravana [2], which provides lower latency, at the cost of eventual consistency. The state synchronization between the primary and standbys is performed using an unreliable mechanism, where the standbys might be out-of-sync for some time. Also, when the primary is detected as faulty by one or more backups, a new primary is elected using a leader election mechanism that employs consensus, which adds latency.

SCL [4]: SCL is an active replication scheme that provides eventual consistency and does not perform agreement. As a result, the response time of SCL is the lowest among all the replication schemes. State conflicts among the controller replicas are resolved via a periodic-gossip mechanism, and safety policies might be violated during the period of conflict.

Consensus [3]: We simulate ONOS [3] and use Fast Paxos [26] that is optimized for lower latency. ONOS guarantees consistency, but suffers from higher response time due to consensus latency.

The first set of experiments evaluates safety (§ VI-A). FCR is proven to satisfy safety at all times (§ V); the same is true for Ravana and Consensus. However, eventual consistent schemes (SCL, Passive) do not guarantee safety. For a safety policy, namely edge disjoint isolation, we study the probability of its violation under eventual consistency, on the topology shown in Figure 1(a). The second set of experiments evaluates liveness and latency metrics (§ VI-B) on two datacenter fat-tree topologies [31] with 8 and 16 port switches referred to as ft8 and ft16, respectively, as well as an ISP topology. We measure the response time of the control plane, the convergence time of the dataplane and unavailability.

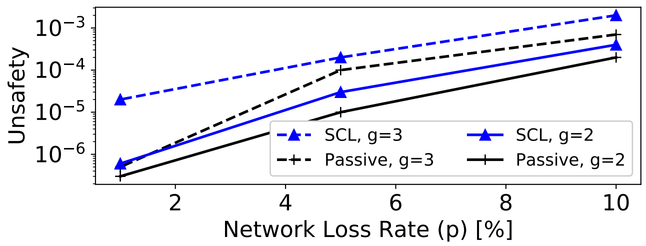


FIGURE 2. **Unsafety in SCL and Passive schemes. Unsafety of FCR is 0.**

A. SAFETY POLICY RESULTS

We quantify unsafety with SCL and Passive for the edge disjoint isolation policy for flows f_1 and f_2 in Figure 1. In each experiment, we create 2 simultaneous link events: breaking of link $4-11$ and link $5-10$. When the updates from controllers are installed, we check if the paths violate edge disjoint isolation. Unsafety is defined as the fraction of iterations in which the policy is violated. We use $\theta_d = 1E-3$, $\theta_c = 1E-4$, $\delta_n = 1\ ms$; and vary g as 2 or 3, and p as 1%, 5% or 10%.

Figure 2 shows the unsafety of SCL and Passive as a function of network loss rate p for different number of replicas g. We see that, for both schemes, unsafety increases with p. For SCL, unsafety with two replicas is $6E-7$ at $p = 1\%$ and $4E-4$ at $p = 10\%$, whereas for Passive it is $3E-7$ at $p = 1\%$ and $4E-4$ at $p = 10\%$. The unsafety of Passive is lower than that of SCL because it does unreliable state synchronization instead of no-agreement in SCL. Even-though the number of violations for SCL and Passive is low, the probability of disagreement increases sharply with more replicas (Figure 2) and network size. Note that this topology is a minimal example to illustrate violations - their number would be higher in real-world topologies like [6]. The unsafety of FCR is proven to be zero for all values of p and g, according to Theorem 3.

Finding 1: The eventual-consistent schemes, SCL and Passive, have a positive probability of safety violation, thus being inappropriate for safety critical applications. This probability increases significantly with the increase of network losses and with the number of replicas that aim at higher availability.

B. LIVENESS POLICY AND LATENCY RESULTS

We study the price to pay, in terms of liveness and latency properties, for guarantying safety with FCR. To quantify the latency properties, we measure the *response time*, i.e., the time between the occurrence of an event at a switch and the installation of the new update on that switch. During this interval, a new event might occur on this or another switch, *i.e.,* the network snapshot is modified. This will trigger a computation of new updates. Therefore, installation of an update at a switch, as well as partial installation of updates, *i.e.,* at a subset of the switches involved after a network event, do not necessarily imply that the required liveness (shortest path) policy is satisfied. We say that the dataplane has *converged* if both safety and liveness policies are satisfied after an event. Thus, *convergence time* is time between occurrence of an event and the convergence of the dataplane. We measure *unavailability* as the fraction of time during which the shortest path liveness policy is violated for at least one flow.

In this section, we use two sets of parameters: `normal` and `aggressive`. Parameters λ_f and λ_r represent mean-time-to-failure (MTTF) and mean-time-to-repair (MTTR), respectively. In the `normal` setup, we use $p = 1E-3$, $\theta_c = 1E-4$, $\theta_d = 1E-3$, $\lambda_f = 24$ hrs, $\lambda_r = 12$ hrs. With these values of λ_f and λ_r, in ft16 with 3072 links, there is an event every 15 s. In the `aggressive` setup, we use $p = 1E-2$, $\theta_c = 1E-3$, $\theta_d = 1E-2$, $\lambda_f = 12$ hrs, $\lambda_r = 6$ hrs. We vary the number of controller replicas g and network latency δ_n.

We use ft16 with the `normal` parameter set, $g = 2$ and round-trip time of $1\ ms$, i.e., $\delta_n = 0.5\ ms$ as a basic scenario to highlight the main findings. We do sensitivity studies by varying δ_n, g, different topologies and fault profiles. Figure 3 shows the empirical cumulative distribution function (ECDF) of response times and convergence times, along with results on unavailability for different schemes. Scenario 1 in Table 2 illustrates that the median response time with FCR is $0.6\times$ that of Ravana and $0.4\times$ that of Consensus.

Finding 2: The median response and convergence time with FCR is lower than that of consistency-guaranteeing schemes and higher than that of eventually-consistent schemes.

To visualize the tail improvement, we show in Figure 4, on the log scale, the complementary CDF (CCDF) of response and convergence times. From Scenario 1 in Table 2, we see that the median and the tail response times of FCR are $2.34\times$ and $1.16\times$ that of SCL, respectively. In contrast, the tail response time of Ravana, Passive and Consensus is $1.56\times$, $2.09\times$, $1.98\times$ that of FCR, respectively. Their tail convergence time is *two orders of magnitude* ($\sim 170\times$) larger than that of FCR. This makes FCR clearly a better choice for networks that require safety, but, also aim at low tail-latency, e.g., in datacenters [32].

Finding 3: The tail response and convergence time with FCR is comparable to that of eventually-consistent schemes and drastically lower than consistency-guaranteeing schemes.

The improvement in response time due to FCR is attributed to the unbounded latency overhead of some form of consensus mechanism used in Passive, Ravana and Consensus. Also, the underlying agreement algorithm of FCR, has a higher probability of reaching an agreement than Consensus. Due to its lower response and convergence time, FCR has an availability three orders of magnitude higher than Passive, Ravana, and Consensus, as shown in Figure 3(c). The unavailability of FCR ($1.97E-06$) is comparable ($2.5\times$) to that of SCL ($6.9E-06$); this highlights the efficacy of FCR in maximizing liveness while providing safety guarantees. The drastically high ($> 10\times$) availability of FCR and SCL when compared to other protocols is also the reason for the low tail-latencies measured at 99th percentile. Note that the tail measured at 99.99th percentile and the maximum response

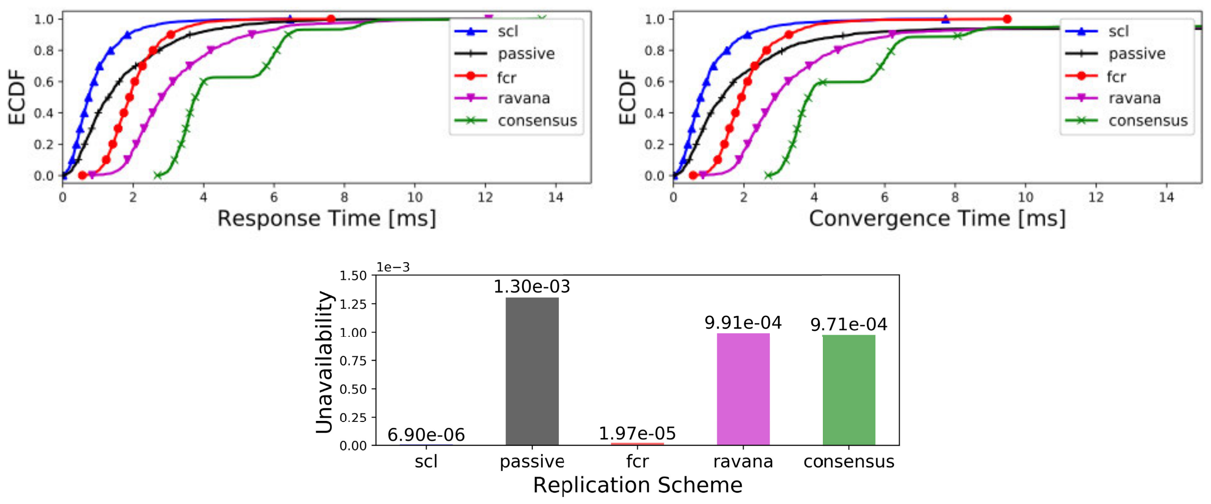


FIGURE 3. Representative scenario (Basic): ft16 with $g = 2$, $\delta_n = 0.5$ *ms* and parameter setup `normal`.

TABLE 2. Median and tail (at 99th percentile) response and convergence times for different scenarios. The column Setup shows the difference in each scenario from basic setup seen in Figure 3.

		Median and Tail Response Time [ms]					Median and Tail Convergence Time [ms]				
#	Setup	SCL	FCR	Passive	Ravana	Consensus	SCL	FCR	Passive	Ravana	Consensus
1	Basic	0.73, 3.86	1.71, 4.48	1.3, 7.01	2.82, 9.35	3.77, 8.91	0.77, 5.01	1.72, 4.73	1.39, 815.79	2.89, 759.48	3.84, 855.36
2	$g = 3$	0.57, 2.56	1.38, 3.34	1.27, 6.84	2.81, 8.29	3.65, 8.44	0.59, 2.71	1.39, 3.41	1.41, 858.44	2.91, 861.6	3.69, 791.75
3	$\delta_n = 0.1\ ms$	0.55, 3.18	0.74, 3.72	1.05, 7.39	1.38, 6.61	2.19, 7.66	0.6, 3.71	0.74, 3.93	1.13, 825.56	1.42, 759.94	2.21, 829.39
4	$\delta_n = 1\ ms$	1.05, 3.89	2.93, 5.82	1.48, 6.44	4.7, 11.49	7.01, 12.92	1.09, 4.53	2.94, 6.17	1.53, 688.12	4.82, 764.15	7.05, 854.38
5	$g = 3, \delta_n = 1\ ms$	0.83, 2.84	2.43, 4.35	1.62, 7.77	4.65, 9.64	7.03, 12.37	0.86, 2.99	2.43, 4.39	1.71, 880.37	4.73, 875.17	7.08, 786.05
6	ft8	0.79, 3.39	1.72, 4.31	1.27, 6.59	2.78, 7.84	3.72, 9.12	0.82, 3.88	1.73, 4.38	1.35, 749.94	2.83, 617.33	3.75, 813.46
7	`as1221`	0.76, 3.72	1.96, 6.30	1.33, 7.18	2.83, 8.33	3.77, 11.13	0.88, 6.07	2.07, 9.24	1.53, 887.37	2.98, 922.55	3.98, 971.8
8	g=3, `aggressive`	0.59, 2.27	1.52, 4.09	1.27, 7.27	2.83, 7.76	3.65, 8.79	0.63, 2.93	1.58, 5.04	1.49, 912.17	2.96, 967.87	3.72, 865.85

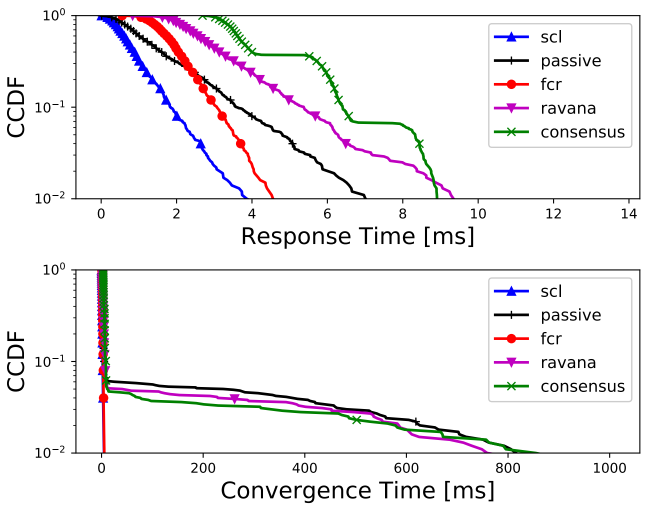


FIGURE 4. CCDF of response and convergence times for the basic scenario.

time and convergence time for SCL and FCR also show a similar trend. Findings 1 and 2 are true also for other scenarios (see Table 2). Next, we study the variation in the latency improvement of FCR with different parameters.

Finding 4: The latency improvement of FCR increases significantly with more replicas, decreases with larger network latency, and is not affected by network size.

From Scenarios 1 and 2 in Table 2, we see that the response and convergence times of most schemes reduce with more replicas. Adding replicas makes the controller more "available", in the sense that the controller becomes more reactive [33]. As FCR can reach an agreement with fewer available replicas, its probability of agreement increases faster than in Consensus and Ravana, by adding more replicas. Hence, with $g = 3$, the latency improvement increases on average by 2.3× for tail response time and by 245× (170× for $g = 2$) for tail convergence time. A similar trend is seen in Scenarios 4 and 5. Note that for SCL an increase of the number of replicas may reduce latency and convergence time, but it renders safety violation more likely (Figure 2). Thus, with SCL, for a given probability of safety violation, the latency and convergence time cannot be decreased by increasing the number of replicas. On the contrary, with our proposed FCR, increasing the number of replicas can ensure significantly low values of latency and convergence time without any impact on the guaranteed zero unsafety.

In Scenarios 3 and 4, we vary the one-way network latency δ_n to 0.1 *ms* and 1 *ms*, respectively, from 0.5 *ms* in the basic scenario. As FCR performs agreement, its latency is limited by the latency of the out-of-band control network. Thus, we find that the average convergence time of FCR increases as δ_n increases. However, the increase is much lower than that of Ravana and Consensus because, on average, the agreement

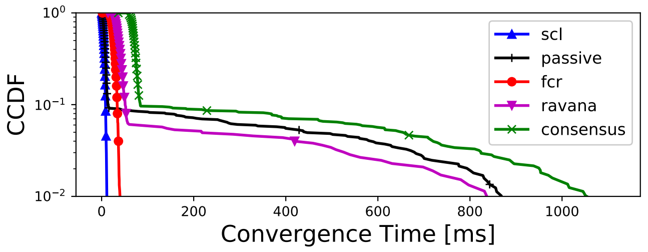


FIGURE 5. CCDF of convergence time for AS 1221 with parameter set `normal`, $g = 2$ and $\delta_n = 10$ *ms*.

mechanism of FCR exchanges less traffic than Consensus and Ravana. As a result, for $\delta_n = 0.1$ *ms*, we see that the tail-latency improvement with FCR, with respect to Consensus and Ravana, is 1.9× for response time and 202× for convergence time, on average. For $\delta_n = 1$ *ms*, these drop to 1.73× and 131×, respectively.

We also find that at lower δ_n, FCR closely follows SCL, whereas their difference is larger at high values of δ_n. This is because SCL uses less the out-of-band network while responding to events as it does not perform agreement. Hence, SCL's response time is very close to δ_n. As seen earlier, the performance of FCR can be further improved to be closer to SCL with more controller replicas. When more replicas cannot be used, a marginally higher convergence time is a small price to pay for guaranteed control-plane consistency, i.e., safety. Alternatively, for SDN controllers that do not require control-plane consistency and do not implement applications based on packet events (e.g., NATs and firewalls), SCL could be used instead of FCR, for lower latency.

As FCR performs agreement on `n_qsp` number of *Status* messages, its performance depends on network size. We see from Scenarios 1 and 6 that, as the network size increases, FCR's convergence time increases. The same is observed in Consensus, albeit for a different reason: more events might occur while controllers are agreeing, thus increasing the convergence time. The latency improvement of FCR with respect to Consensus is similar for both network sizes.

The performance of FCR is bottlenecked by the performance of Quarts [12]. Thus, the convergence time of FCR depends on the success of Quarts, which depends on the fault parameters θ_c and θ_d, and the network loss rate p. Scenario 8, with the `aggressive` setup, shows the impact of these parameters. We see that the performance of all schemes is affected by the increased fault-rate; and the low-latency schemes, FCR and SCL, are affected the most. This can be remedied by appropriately dimensioning the degree of replication when the controller is expected to be faulty. The performance of Quarts for a wider range of fault profiles is studied in [12].

1) ISP TOPOLOGY

In order to study the performance of FCR in a non-datacenter topology, we simulated the AS 1221 ISP topology, mapped in the Rocket-Fuel Project [34], with 4367 switches with $\delta_n = 10$ *ms* in Scenario 7. Figure 5 shows that although the tail convergence time of FCR is increased when compared to SCL due to large one-way latency, the large improvement over Consensus, Ravana and Passive is still preserved. Moreover, as ISP networks have multiple, redundant paths of different costs, there is a higher chance of eventually consistent schemes violating safety policies, such as edge disjoint isolation, than in datacenter networks.

2) BANDWIDTH OVERHEAD AND SCALABILITY

FCR has the same bandwidth overhead as SCL [4] as it uses the same probing strategy. For example, on the largest topology we tested, AS 1221, 98% of the time (measured on 500 *ms* intervals) bandwidth overhead is under 1 *Mbps*. This is smaller than 0.0001% of link capacity if we assume 10 *Gbps* links.

3) NETWORK PARTITIONS

In the presence of network partitions, FCR might sacrifice availability in some cases, but it never sacrifices consistency. Two cases of network partitions are possible: (1) One or more switches are isolated from all the controllers and cannot send or receive packets. The controllers will proceed normally without events from the isolated switches. Availability for the still-reachable switches will not be hindered. (2) A network partition occurs between controllers. In this case, the updates may still be installed on the switches, if there exists a partition with enough QCPs for the plurality-based voting of Quarts to succeed.

4) AVAILABILITY VS. BANDWIDTH

The reduced availability (especially in case (2)) is a result of our design choice – we choose to keep the bandwidth overhead low at the cost of availability. For networks where the bandwidth budget allows it, one could increase availability in general by adding a probing phase after each event (instead of probing periodically).

Every QCP with a *Status* or *Sync_Status* from all QSPs with the right label proceeds with computing and installing updates after the probing phase – without communicating with other QCPs first. The corresponding controller computes with the ground truth for its current label, thus providing availability. The same does not apply to QCPs with a partial digest. See Appendix B for more details.

VII. SYSTEM IMPLEMENTATION

We evaluate our proof-of-concept implementation of FCR against the original SCL implementation used in [4]. We run the two systems in the same environment in Mininet 2.2.1 [35]: 20 switches and 16 hosts are connected via 48 1 *Gbps* links (with the default Mininet behavior) forming a 4-port fattree. Both schemes support in-band and out-of-band communication. We evaluated the out-of-band setup commonly used in datacenters [36]. We use a single machine with two Intel Xeon E5-2680 (Haswell) processors with a total of 24 cores and 48 hyper-threads running at 2.5 *GHz*, and 256 *GB* of main memory. We run Ubuntu

LTS 16.04.2 distribution with the Linux kernel version 4.4.0. FCR is implemented using POX controllers [17] with shortest-path application in order to make a fair comparison with SCL implementation. The controllers are paired with their corresponding QCPs via OpenFlow 1.0 [37]. We implemented Quarts in C++ based on algorithm from Saab *et al.* [12] and exported it as a Python module in order to connect it with POX. Our implementation of Quarts is easy to plug-in via intuitive API calls. The only parameters we had to set was the network delay upper bound δ_n, which we empirically measured and set to 10 *ms*, and the addresses of all the controllers.

We use unmodified Open vSwitch [38] (version 2.5.2) that communicates with the QSP layer, also via OpenFlow 1.0. *Status* messages are constructed from the switch state since sending the full content of flow tables is not necessary in FCR. For reactive applications such as NAT, not evaluated in this work, reading the list of pending packet-ins would be required as well. Note that a QCP needs to differentiate updates received from controllers as a result of two successive computations, in order to label them correctly (Algorithm 1). To this end, it uses the flag `controller_free` to check if the single-image controller finished computing, before sending new *Status* messages, thereby ensuring that the received updates correspond to the last *Status* messages. The QCP sets the flag to `True` upon receiving a special OpenFlow message, `handle_QUEUE_GET_CONFIG_REQUEST`, with which the controller signals it finished computing.

We analyze the response and convergence times (defined in §VI) of both FCR and SCL across 200 random single-link failures. The time between consecutive events is one minute. This scenario corresponds to the evaluation of SCL in [4]. In Figure 6 we show the results with two and three single-image controller replicas. We find that FCR and SCL are comparable in both response and convergence times, since the voting in FCR is likely to succeed due to good network conditions. Notice that while the response time of FCR is slightly higher than SCL due to the Quarts delay, the convergence time of FCR is slightly better than SCL due to control-plane consistency in FCR ensuring that different controllers do not install conflicting updates. As a result, the dataplane converges faster with FCR. Authors in [4] show that for the same 4-port fattree topology ONOS is expected to be 1.24× and 1.63× slower than SCL in tail response and convergence time, respectively. In our setting similar results are expected for ONOS vs. FCR. Note that the latency improvement observed in §VI is much smaller because the high latency with consensus (ONOS) is observed in cases of network partitions, not studied in the implementation evaluation.

VIII. RELATED WORK

Numerous related works evaluate the trade-off between the consistent global network view and latency of SDN controller applications [39], [40]. Approaches that focus on

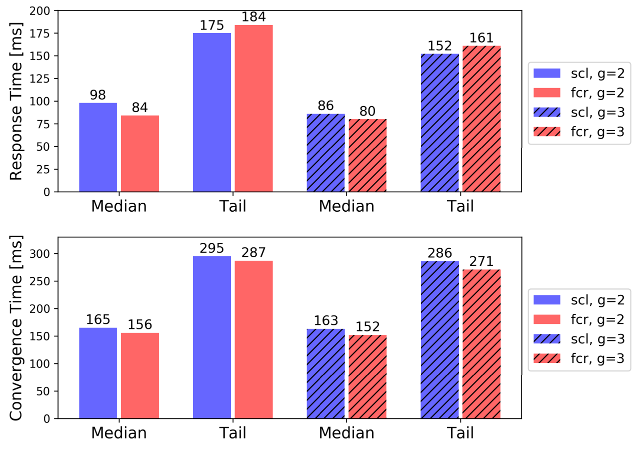


FIGURE 6. Median and tail (at 99th percentile) of 4-port fattree with 2 and 3 replicas.

ensuring control plane consistency impose high delays in responding to network events. Hyperflow [41] passively synchronizes network-wide views of OpenFlow controllers. ONOS [3] uses Paxos - known for its high latency and complicated implementation [4], [33]. Approaches that instead use Raft (e.g. ONIX) [1], [42] have similar latency issues. Ravana comes in multiple flavors that offer different levels of consistency guarantees. The strongest one offers, exactly-once event processing and exactly-once execution of commands [2], but at a high latency price. Alternatively, approaches like SCL chose the opposite trade-off in favor of low latency, as we saw in §I. The trade-offs in SCL [4], Ravana [2] and Consensus [3] have been extensively studied in §VI.

In passive replication schemes such as Ravana, view-stamped replication [30] and primary-backup replication [27], the primary replica is a single-point of failure. Hence, when it fails, the switches timeout and resend the request event. This adds to the tail-latency. Moreover, such schemes perform a fail-over to the backup by detecting the primary as faulty. As perfect failure-detection is proven to be impossible in asynchronous systems [43], passive-replication schemes either have to expend large delays in ensuring smooth failover or are susceptible to inconsistencies due to multiple simultaneous primary replicas. Additionally, when the primary replica is delayed intermittently, the delay faults go undetected and availability is lowered as seen in [19].

The need for fast control planes has been recognized by a number of researchers. Molero *et al.* [10] implement the entire control plane using programmable ASICs. Chen and Benson [9] tackle the problem by better scheduling of TCAM resources in SDN switches. Either approach could be used in combination with FCR for ultra low latency.

An orthogonal problem to control-plane consistency is the consistency of updates under their asynchronous installation on the switches [22], [44]–[47]. For that purpose FCR uses packet labeling [22], [47] (detailed in §IV-D).

IX. CONCLUSION

We presented FCR, a coordination layer for replicated single-image controllers that guarantees control-plane consistency with low-latency. Through simulation, we find that FCR can provide up to 170× improvement in tail latency compared to other consistency-guaranteeing mechanisms. We formally prove that FCR enforces all safety policies that are enforceable by the underlying single-image controller. FCR uses Quarts for agreement among replicated-controllers, which supports both stateless and stateful controllers. In this work, FCR is applied to stateless controller applications, and stateful applications are left for future work.

APPENDIXES

APPENDIX A

PROOFS OF THEOREMS OF SECTION V

A. PROOF OF THEOREM V.1

Proof: From Lemma A.5 in [48], Quarts ensures that if two QCPs, Q_i and Q_j, return `success = True` for label C (Algorithm 1 line 22), then they have the same $\mathbf{S_{Crt}}$ and same C^*_{bns}. Moreover, as the function `update_baseline_snapshot` uses consensus, we have a consistent BNS at all QCPs for the same BNS label. Therefore, for QSPs with a zero in the chosen digest, every QCP will chose the same state. Furthermore, as controllers are deterministic, agreement on the inputs used for updates' computation is sufficient to guarantee that the resulting updates received by Q_i and Q_j from the respective controllers are identical. Lastly, the value of the label does not change while the controller is computing, i.e., while `qcp_free =False` (Algorithm 1 lines 17, 21 and 50). Hence, the outgoing updates have the same label. □

B. PROOF OF THEOREM V.2

Proof: The duration of Quarts at a QCP is at most $5\delta_n$ (From Theorem V.2 in [12]). □

C. PROOF OF THEOREM V.3

Proof: As s is enforceable by C, the set of updates for label r, U^0_r, computed by C, does not violate s. Let U^i_r be the set of updates computed by the i^{th} replica of C, C_i. Control-plane consistency (Theorem 1) implies that $U^i_r = U^0_r$ for any C_i. Furthermore, as updates are assumed to be idempotent, an installation of U^i_r for each C_i has the same result as an installation of U^0_r. □

D. SAFETY GUARANTEES OF FCR VS. SCL

Following the discussion in Section V for the safety guarantees, let us denote the set of enforceable policies when replicating C with SCL as S_{scl}. From SCL [4], we have that S_{scl} includes a policy s if and only if it can be expressed entirely as a condition on exactly one path, i.e., the path violates or obeys the policy regardless of other existing paths in the network. For example, way-pointing states that the packets of a flow should follow a path that includes a given set of switches in the network. On the contrary, the definition of safety policies in S_c can concern multiple flows, hence multiple paths. For example, edge disjoint isolation for flows f_1 and f_2 states that their paths p_1 and p_2 should not share a link. Therefore, $S_{scl} \subset S_c$. Since, $S_{FCR} = S_c$, FCR enforces more safety policies than those enforced by SCL.

From the example of edge disjoint isolation in Figure 1, we notice that safety policies in S_c require control-plane consistency. Specifically, the controllers C_1, C_2 that are replicas of C, compute two different sets of paths for flows f_1, f_2, each set satisfying the safety policy. However, the true paths followed come from different controllers, and this interleaved set of paths does not satisfy the safety policy. As a result, in the absence of control-plane consistency, there needs to be a mechanism so that all switches serve all flows by using the updates from the same "chosen" controller; this is in fact a consensus problem and is faced with the same drawbacks of doing consensus in control-plane.

E. PROOF OF THEOREM V.4

This result derives from the fact that the QSP (i) sets the `acked` field to `False` when an event occurs (Algorithm 2 line 12), (ii) tracks if the last event emerged at the switch has been acknowledged (variable *acked* at line 19 of Algorithm 2) and (iii) installs updates only if they have been computed accounting for the current state of the switch (checked at the receive Update function of Algorithm 2, line 26). The proof is as follows.

Proof: Let event e_1 occur at switch 1 at a time when the logical clock of its QSP is $C = l_1$.

Then, at Algorithm 2 line 12, the flag *acked* is set to `False`.

Let $l_2 \geq l_1$ be the value of the logical-clock at the QSP of switch 1, at the first time when the *acked* flag is set to `True` at Algorithm 2 line 29.

Thus, the switch 1 received an update with label l_2 and with the field *ack* set to `True`.

Hence, there exists a QCP which succeeded in Quarts for label l_2 such that **S_Crt**[1] $\neq \perp$ for label l_2.

Therefore, switch 1 sent a *Status* message, S_{l_2}, with label l_2.

However, for all labels within $\{l_1 + 1, \ldots, l_2 - 1\}$, e_1 is still unacknowledged, since *acked* = `False`. Thus, either a converse event occurred or still $S_{l_2} \ni e_1$.

The last statement completes the proof. □

APPENDIX B

FCR WALK-THROUGH

In this section, for the ease of understanding of the readers, we discuss how FCR operates using the topology of Figure 1 under the typical scenario and a list of atypical scenarios:

1) Typical success scenario.
2) Multiple switch events.
3) QSP's logical clock is lower than the received update label.

The network in Figure 1 consists of two controllers, C_1 and C_2, their corresponding QCPs, QCP_1 and QCP_2, and 11 switches, switch 1 to switch 11, as well as their corresponding QSPs, QSP_1 to QSP_{11}.

Scenario 1: An event happens on switch 1 and the internal logical clock of QSP_1 increments to l. QSP_1 sends a *Status* message with label l to both QCPs. Assume that both QCP_1 and QCP_2 have internal logical clocks smaller than l; therefore they begin Quarts with the received *Status* message. In Quarts collection phase, each QCP queries its peers for the *Status* from all other QSPs. As no other event had occurred, the collection results in a digest [1 0 0 0 0 0 0 0 0 0 0] and the same digest is chosen in the voting phase. Both QCPs immediately proceed with computing and installing the updates with the label l.

Scenario 2: Like in Scenario 1 an event happens on switch 1 and the internal logical clock of QSP_1 increments to l. QSP_1 sends a *Status* message with label l to all the QCPs. While the QCPs are doing collection for event from QSP_1, they receive events from other switches in the form of *Status* messages. Now, one of the following scenarios can occur.

- First: QCP_1 and QCP_2 have the same digest after collection. In this case, the voting on both QCPs succeeds, and they proceed with computation and installation of updates on all QSPs like in Scenario 1.
- Second: QCP_1 and QCP_2 have the different digests after collection and QCP_1 has full digest [1 1 1 1 1 1 1 1 1 1 1]. Note that this scenario is unlikely as it requires events happening on all the switches within a short time period. In this case, QCP_1 will succeed in the voting phase due to the absolute priority of full digest in Quarts' priority voter. Alternatively, QCP_2 will fail in the voting phase. Therefore, QCP_1 will proceed to computation and installation of updates, whereas QCP_2 is prevented from computation. However, QCP_2 may compute updates in the next successful round after l when it retrieves either the full-digest or the same digest like the majority. In this simple example there are only two controllers - therefore the majority is both of them.
- Third: QCP_1 and QCP_2 have different digests after collection and neither has a full digest. In this case, the voting phase will fail on both the QCPs and they are both prevented from computing updates. The event with label l remains unacknowledged. No new events happen in the meantime and the event timer fires on the QSPs. The internal logical clocks are incremented to $l+1$ and new *Status* messages with the label $l+1$ are sent to all the QCPs starting a new round of Quarts.

Scenario 3: One more time, an event happens on switch 1 and the internal logical clock of QSP_1 increments to l. An event that happened on switch 1 triggered updates with the logical clock l. QSP_3, did not receive its updates and its internal logical clock stayed behind at the value $l-1$ while all other QSPs moved on to value l. The next event is triggered on the switch 7 and labeled with $l+1$. Even though QSP_3 effectively "skipped" a round, when it receives an update with label $l+1$ in the next round it will update its internal logical clock to $l+1$. A subsequent *Status* message from QSP_3 after an event will have a label $l+2$. Updates will be recomputed in the controllers and the operation of the system will continue normally.

REFERENCES

[1] 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 *Proc. OSDI*, vol. 10, 2010, pp. 1–6.

[2] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, Art. no. 4.

[3] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an open, distributed SDN OS," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 1–6.

[4] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "SCL: Simplifying distributed SDN control planes," in *Proc. NSDI*, 2017, pp. 329–345.

[5] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[6] 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 *Proc. ACM SIGCOMM Conf.*, Hong Kong, Aug. 2013, pp. 3–14, doi: 10.1145/2486001.2486019.

[7] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "CAP for networks," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, New York, NY, USA, 2013, pp. 91–96. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491186

[8] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," *Austral. Comput. Sci. Commun.*, vol. 10, no. 1, pp. 56–66, Feb. 1988.

[9] H. Chen and T. Benson, "The case for making tight control plane latency guarantees in SDN switches," in *Proc. Symp. SDN Res. (SOSR)*, New York, NY, USA, 2017, pp. 150–156. [Online]. Available: http://doi.acm.org/10.1145/3050220.3050237

[10] E. C. Molero, S. Vissicchio, and L. Vanbever, "Hardware-accelerated network control planes," in *Proc. 17th ACM Workshop Hot Topics Netw. (HotNets)*, Redmond, WA, USA, Nov. 2018, pp. 120–126, doi: 10.1145/3286062.3286080.

[11] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Seattle, WA, USA, Apr. 2014, pp. 543–546. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/fayazbakhsh

[12] W. Saab, M. Mohiuddin, S. Bliudze, and J.-Y. Le Boudec, "Quarts: Quick agreement for real-time control systems," in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Automat.*, Sep. 2017, pp. 1–8.

[13] W. Saab, M. Mohiuddin, S. Bliudze, and J.-Y. L. Boudec, "Ordering events based on intentionality in cyber-physical systems," in *Proc. 9th ACM/IEEE Int. Conf. Cyber-Phys. Syst.*, Apr. 2018, pp. 107–118.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[16] SCL Contributors. *SCL (Simple Coordination Layer) Implementation*. Accessed: Jan. 10, 2019. [Online]. Available: https://github.com/NetSys/scl

[17] J. McCauley. *The POX Network Software Platform*. Accessed: Jan. 10, 2019. [Online]. Available: https://noxrepo.github.io/pox-doc/html/

[18] Ryu. *Component-Based SDN Framework*. Accessed: Jan. 10, 2019. [Online]. Available: https://osrg.github.io/ryu/index.html

[19] M. Mohiuddin, W. Saab, S. Bliudze, and J.-Y. Le Boudec, "Axo: Masking delay faults in real-time control systems," in *Proc. 42nd Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Oct. 2016, pp. 4933–4940.

[20] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *IEEE Trans. Softw. Eng.*, vol. SE-9, no. 3, pp. 219–228, May 1983.

[21] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[22] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proc. 10th ACM Workshop Hot Topics Netw. (HotNets-X)*, 2011, pp. 7:1–7:6.

[23] D. M. Blough and G. F. Sullivan, "A comparison of voting strategies for fault-tolerant distributed systems," in *Proc. 9th Symp. Reliable Distrib. Syst.*, Oct. 1990, pp. 136–145.

[24] J. N. Gray, "Notes on data base operating systems," in *Operating Systems*. Berlin, Germany: Springer, 1978, pp. 393–481.

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[26] L. Lamport, "Fast paxos," *Distrib. Comput.*, vol. 19, no. 2, pp. 79–103, 2006.

[27] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distrib. Syst.*, vol. 2, no. 1, pp. 199–216, May 1993.

[28] E. O. Elliott, "Estimates of error rates for codes on burst-noise channels," *Bell Syst. Tech. J.*, vol. 42, no. 5, pp. 1977–1997, Sep. 1963.

[29] D. Dzung, R. Guerraoui, D. Kozhaya, and Y.-A. Pignolet, "Never say never—Probabilistic and temporal failure detectors," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2016, pp. 679–688.

[30] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. 7th Annu. ACM Symp. Princ. Distrib. Comput.*, 1988, pp. 8–17.

[31] C. G. Requena, F. G. Villamón, M. E. G. Requena, P. J. L. Rodríguez, and J. D. Marín, "RUFT: Simplifying the fat-tree topology," in *Proc. 14th IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2008, pp. 153–160.

[32] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[33] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Berlin, Germany: Springer, 2006.

[34] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with Rocketfuel," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 133–145, 2002.

[35] Mininet. *An Instant Virtual Network on Your Laptop (or Other PC)*. Accessed: Jan. 10, 2019. [Online]. Available: http://mininet.org

[36] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, and U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network," *Commun. ACM*, vol. 59, no. 9, pp. 88–97, 2016. [Online]. Available: http://doi.acm.org/10.1145/2975159

[37] *OpenFlow*. Accessed: Jan. 10, 2019. [Online]. Available: https://en.wikipedia.org/wiki/OpenFlow

[38] *Open vSwitch*. Accessed: Jan. 10, 2019. [Online]. Available: http://www.openvswitch.org

[39] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: State distribution trade-offs in software defined networks," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw.*, 2012, pp. 1–6.

[40] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 136–141, Feb. 2013.

[41] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for openflow," in *Proc. Internet Netw. Manage. Conf. Res. Enterprise Netw.*, 2010, pp. 1–6.

[42] H. Howard, M. Schwarzkopf, A. Madhavapeddy, and J. Crowcroft, "Raft refloated: Do we have consensus?" *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 12–21, Jan. 2015.

[43] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

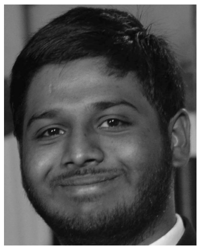
[44] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2012, pp. 323–334.

[45] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res. (SOSR)*, 2015, pp. 21:1–21:14.

[46] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. 12th ACM Workshop Hot Topics Netw. (HotNets)*, 2013, pp. 20:1–20:7.

[47] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 49–54.

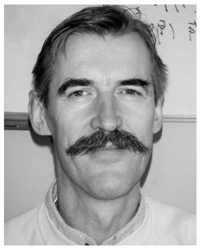
[48] W. Saab, M. Mohiuddin, S. Bliudze, and J.-Y. Le Boudec, "Quarts: Quick agreement for real-time control systems," in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Automat. (ETFA)*, 2017, pp. 1–8.



MAAZ MOHIUDDIN received the bachelor's degree in electrical engineering with a minor in computer science from the Indian Institute of Technology Hyderabad, in 2013, and the Ph.D. degree in computer science from EPFL, in 2018, under the supervision of Prof. J.-Y. Le Boudec. He is currently a Software Engineer with Cisco Systems, Switzerland. His main research interests are fault-tolerant computing and communication in real-time cyber-physical systems. He has coauthored an article presented at WFCS 2015 that received the Best Paper Award.

MIA PRIMORAC received the M.Sc. degree in computing from the University of Zagreb, Croatia, in 2014. She is currently pursuing the Ph.D. degree with EPFL. In 2013, she worked as a Research Intern with the Operating Systems Laboratory, EPFL. In 2016, she worked as a Research Intern with the Networking Platforms Lab, Intel Labs, Hillsboro, OR, USA. She is also a member of the Data Center Systems and Network Architecture Laboratory, EPFL. In 2017, she received the Best Paper Award for her work on latency measurements of network functions published at SIGCOMM KBNets'17. Her research interest includes understanding and mitigating latency variability in data centers.

ELENI STAI received the Diploma degree in electrical and computer engineering from the National Technical University of Athens (NTUA), Greece, in 2009, the B.Sc. degree in mathematics from the National and Kapodistrian University of Athens, Greece, in 2013, and the M.Sc. degree in applied mathematical sciences and the Ph.D. degree in electrical engineering from NTUA, in 2014 and 2015, respectively. She is currently a Postdoctoral Researcher with the Laboratory for Communications and Applications (LCA2), EPFL. She has received the Chorafas Foundation Best Ph.D. Thesis Award, the Thomaidis Foundation Best M.Sc. Thesis Award, and the Best Paper Award in ICT 2016. Her main research interests include network design and optimization, smart-grid control and applications, and data analytics on networks. She has coauthored the book *Evolutionary Dynamics of Complex Communications Networks* (Taylor and Francis Group, CRC Press).



JEAN-YVES LE BOUDEC received the Agrégation in mathematics the from École Normale Supérieure de Saint-Cloud, Paris, in 1980, and the Ph.D. degree from the University of Rennes, France, in 1984. From 1984 to 1987, he was with INSA/IRISA, Rennes. In 1987, he joined the Bell Northern Research, Ottawa, ON, Canada, as a member of scientific staff with the Network and Product Traffic Design Department. In 1988, he joined the IBM Zurich Research Laboratory, where he was the Manager of the Customer Premises Network Department. In 1994, he became an Associate Professor with EPFL, where he is currently a Professor. He has coauthored a book on *Network Calculus*, which serves as a foundation for deterministic networking, an introductory textbook on *Information Sciences*. He is the author of the book *Performance Evaluation*. His research interests are in the performance and architecture of communication systems and smart grids.