SCL: Simplifying Distributed SDN Control Planes

Aurojit Panda[†] Wenting Zheng[†] Xiaohe Hu[‡] Arvind Krishnamurthy[◇] Scott Shenker[†⋆]
[†]UC Berkeley [‡]Tsinghua University [◇]University of Washington [⋆]ICSI

Abstract

We consider the following question: *what consistency model is appropriate for coordinating the actions of a replicated set of SDN controllers?* We first argue that the conventional requirement of strong consistency, typically achieved through the use of Paxos or other consensus algorithms, is conceptually unnecessary to handle unplanned network updates. We present an alternate approach, based on the weaker notion of eventual correctness, and describe the design of a simple coordination layer (SCL) that can seamlessly turn a set of single-image SDN controllers (that obey certain properties) into a distributed SDN system that achieves this goal (whereas traditional consensus mechanisms do not). We then show through analysis and simulation that our approach provides faster responses to network events. While our primary focus is on handling unplanned network updates, our coordination layer also handles policy updates and other situations where consistency is warranted. Thus, contrary to the prevailing wisdom, we argue that distributed SDN control planes need only be slightly more complicated than single-image controllers.

1 Introduction

Software-Defined Networking (SDN) uses a "logically centralized" controller to compute and instantiate forwarding state at all switches and routers in a network. However, behind the simple but ambiguous phrase "logically centralized" lies a great deal of practical complexity. Typical SDN systems use multiple controllers to provide availability in the case of controller failures.[1] However, ensuring that this collection of replicated controllers produce the desired behavior[2] requires an explicit consistency mechanism that coordinates the actions of the replicated controllers. Most distributed SDN controller designs have opted for consensus mechanisms such as Paxos (as used in ONIX [12]) and RAFT (as used in ONOS [2]), and recent work (*e.g.,* Ravana [10]) relies on even strong consistency guarantees.

But consistency is only a means to an end, not an end in itself. Operators and users care about certain properties or invariants of the forwarding state (that is, the forwarding entries residing in switches), and are not directly concerned about consistency among controllers. For example, they care whether the forwarding state enforces the desired isolation between hosts (by installing appropriate ACLs), or enables the desired connectivity between hosts (by installing functioning paths), or establishes paths traversing the specified set of middleboxes; operators and users do not care whether the controllers were in a consistent state when installing these forwarding entries.

With this invariant-oriented criterion in mind, we revisit the role of consistency in SDN systems. We analyze the consistency requirements for the two kinds of state that reside in controllers — *policy state* and *network state*[3] — and then argue that for network state the oft-used consensus-based mechanisms are both *conceptually inappropriate* and *practically ineffective*. This raises two questions: why should we care, and what in this argument is new?

Why should we care? Why not use the current consistency mechanisms even if they are not perfectly suited to the task at hand? The answer is three-fold.

First, consistency mechanisms are algorithmically complex and therefore often the source of bugs. To note a few examples, the early implementation of ONIX was plagued by bugs in its consistency mechanism; RAFT, heralded as an easier to understand and implement consistency mechanism, continues to struggle with fundamental bugs in the protocol [19]; and for other distributed controllers the consistency mechanisms are among the most complicated aspects of the program.[4] Thus, if we do not need sophisticated consistency mechanisms, it would be best to avoid them.

Second, consistency mechanisms impose limitations on availability. Typical consensus algorithms require a

[1] In some scenarios multiple controllers are also needed to scale controller capacity (*e.g.,* by sharding the state between controllers), but in this paper we focus on replication for reliability.

[2] By desired behavior we mean that the collective actions of a set of replicated controllers should be equivalent to what an infinitely scalable single controller would have done under similar circumstances.

[3] As we clarify later, network state describes the current network topology while policy state describes the properties or invariants desired by the operator (such as shortest path routing, or access control requirements, or middlebox insertions along paths).

[4] In fact, SDN application code for routing is typically 100s LOC, whereas the ZooKeeper code for implementing consensus primitives is 140K+ LOC.

majority of participants to be active and connected to make progress. Thus, consistency mechanisms would prevent networks from functioning properly under severe network conditions that partition the network because minority partitions would be unable to respond to network events.[5] While the lack of availability may be necessary in cases where strong notions of consistency are required, it seems unwise to pay this availability-penalty in cases where such consistency is not needed.

Third, consistency mechanisms impose extra latency in responding to events. Typically, when a link fails, a switch sends a notification to the nearest controller, which then executes a consensus protocol with the other controllers to ensure that a majority of them are aware of the event, and then one or more controllers respond with a new set of flow entries. While the first step (switch contacting a nearby controller) and last step (controllers updating flow entries in switches) are inherent in the SDN control paradigm, the intervening coordination step introduces extra delay. In some cases (such as a cluster on a single rack) the time required for this coordination may be negligible, but there are other cases (such as coordinating different datacenters across a WAN) where the time to execute a consensus mechanism will significantly delay the response to an event such as a link failure.

What is new here? Much work has gone into building distributed SDN systems (notably ONIX, ONOS, ODL, and Ravana),[6] and, because they incorporate sophisticated consistency mechanisms, such systems are significantly more complex than the *single-image controllers* (such as NOX, POX, Beacon, Ryu, and the like) that ushered in the SDN era.[7] In contrast, our reconsideration of the consistency requirements (or lack thereof) for SDN led us to design a simple coordination layer (SCL) that can transform any single-image SDN controller design into a distributed SDN system, as long as the controller obeys a small number of constraints. While our novelty lies mostly in how we handle unplanned updates to network state, SCL is a more general design that deals with a broader set of issues: different kinds of network changes (planned and unplanned), changes to policy state, and the consistency of the control plane (the so-called consistent updates problem). All of these are handled by the coordination layer, leaving the controller completely unchanged. *Thus, contrary to the prevailing wisdom, we argue that distributed SDN systems need only be slightly more complicated than single-image controllers.* In fact, they can not only be simpler than current distributed SDN designs, but have better performance (responding to network events more quickly) and higher availability (not requiring a majority of controllers to be up at all times).

2 Background

In building SDN systems, one must consider consistency of both the data and control planes. Consider the case of a single controller which has just received a policy or network update, and has computed new flow entries for all switches in the network. The controller sends messages to each switch updating their forwarding entries, but these updates are applied asynchronously. Thus, until the last switch has received the update, packets might be handled by a mixture of updated and non-updated switches, which could lead to various forms of invariant violations (*e.g.,* looping paths, missed middleboxes, or lack of isolation). The challenge, then, is to implement these updates in a way that no invariants are violated; this is often referred to as the *consistent updates* problem and has been addressed in several recent papers [11, 14, 17, 18, 22, 23].

There are three basic approaches to this problem. The first approach carefully orders the switch updates to ensure no invariant violations [14, 17]. The second approach tags packets at ingress, and packets are processed based on the new or old flow entries based on this tag [11, 22, 23]. The third approach relies on closely synchronized switch clocks, and has switches change over to the new flow entries nearly-simultaneously [18].

Note that the consistent updates problem exists even for a single controller and is not caused by the use of replicated controllers (which is our focus); hence, we do not introduce any new mechanisms for this problem, but can leverage any of the existing approaches in our design. In fact, we embed the tagging approach in our coordination layer, so that the single-image controller need not be aware of the consistent updates problem and can merely compute the desired flow entries.

The problem of control plane consistency arises when you have replicated controllers. It seems natural to require consistency between the controllers – in terms of their views of network and policy state – because they must collaboratively install forwarding state in switches, and any inconsistencies in controller state could result in inconsistent forwarding entries. This control plane consistency is typically achieved by treating the controllers as a replicated state machine (RSM). More specifically, systems like Onix [12] and Onos [2] use a consensus algorithm (such as Paxos or RAFT) to elect a leader among their controllers, and the leader is responsible for

[5] By minority partitions we mean a partition that contains a minority of controllers.

[6] There are many other distributed SDN platforms, but most (such as [16, 25]) are focused on sharding state in order to scale (rather than replicating for availability), which is not our focus. Note that our techniques can be applied to these sharded designs (*i.e.,* by replicating each shard for reliability).

[7] By the term “single-image” we mean a program that is written with the assumption that it has unilateral control over the network, rather than one explicitly written to run in replicated fashion where it must coordinate with others.

sequencing events and updating the dataplane. Before a leader can respond to an event, it must replicate it to a quorum of controllers (typically a majority) so that if the leader were to fail, the subsequent leader is aware of the last event.[8]

More recent work on Ravana [10] has looked at providing even stronger consistency guarantees. Ravana tries to ensure exactly-once semantics when processing network events. This stronger consistency requirement comes at the cost of worse availability, as exactly-once is impossible in the presence of failures [13].

While the existing distributed controller literature varies in mechanisms (Paxos, RAFT) and goals (from consensus to strict write-once semantics), there seems to be universal agreement on what we call the *consensus assumption*; this is the belief that consensus is the weakest form of coordination necessary for correctly implementing replicated controllers, and that these controllers must reach consensus before updating any forwarding entries. The consensus assumption follows naturally from the concept of a "logically centralized controller" as consensus is indeed the weakest consistency requirement that enables a collection of replicated controllers to act at all times as one logically centralized one.

However, we do not accept the consensus assumption for dealing with network updates and now argue that *eventual correctness* (which applies only after the controllers have taken action), not *consensus* (which is applied before the controllers have acted), is the most salient requirement for distributed controllers. Eventual correctness is merely the property that in any connected component of the network containing a controller, once there are no more updates, the controllers will eventually have the correct view of the network and policy state and that the forwarding entries will all be computed relative to that network and policy state. This seems like a minimal property to require, but consensus-based mechanisms cannot achieve it because they require a quorum of controllers to be up at all times.

So why are consensus-based algorithms used so widely? Traditional systems (such as data stores) that use consensus algorithms are "closed world", in that the truth resides within the system and no update can be considered complete until a quorum of the nodes have received the update; otherwise, if some nodes fail the system might lose all memory of that update. Thus, no actions should be taken on an update until consensus is reached and the update firmly committed. While policy state is closed-world, in that the truth resides in the system, network state is "open-world" in that the ground truth resides in the network itself, not in the controllers, *i.e.,* if the controllers think a link is up, but the link is actually down, then the truth lies in the functioning of the link, not the state in the controller. One can always reconstruct network state by querying the network. Thus, one need not worry about the controllers "forgetting" about network updates, as the network can always remind them. This removes the need for consensus *before* action, and the need for timeliness would suggest acting without this additional delay.

To see this, it is useful to distinguish between *agreement* (do the controllers *agree* with each other about the network state?) and *awareness* (is at least one controller *aware* of the current network state?). If networks were a closed-world system, then one should not update the dataplane until the controllers are in agreement leading to the consensus assumption arose. However, since networks are an open-world system, updates can and should start as soon as any controller is aware, without waiting for agreement. Waiting for agreement is unnecessary (since there is no worry that the update might be lost) and undesirable (since it slows down response to changes, reduces availability, and introduces complexity).

Thus, we should not unnecessarily delay updates while waiting for consensus. However, we should ensure that correctness is eventually achieved, in that controllers should eventually agree on the current network and policy state, and the forwarding entries should be computed relative to that state. The rest of this paper is devoted to describing a design that uses a simple coordination layer lying underneath any single-image controller (that obeys certain constraints) to achieve rapid and robust responses to network events, while achieving eventual correctness. To handle the broad spectrum of updates and policies, our design also includes mechanisms for consistent updates (in the dataplane) and policy updates (in the control plane).

3 Definitions and Categories

3.1 Network Model

We consider networks consisting of switches and end hosts connected by links, and controlled by a set of replicated controllers which are responsible for configuring the forwarding entries in all switches. We assume that all network components can fail (and possibly recover at some arbitrary time later) but do not exhibit byzantine behavior. We further assume (similar to standard SDN) that switches send controllers notifications when network events (*e.g.,* link failures) occur. Finally, we assume the use of in-band control channels that allow switches and controllers to communicate as long as the network is not partitioned. We describe the implementation of such channels in §6. We assume (as is standard in the distributed systems literature) that this channel is fair, in the

[8] This awareness is enforced by the consensus protocol and the requirement that all quorums intersect [7]. Moreover, this model assumes that controllers are deterministic, so one need only replicate the events and their order, not the actual resulting computations.

sense that a message sent infinitely often will be delivered infinitely often (*i.e.,* message losses are independent of message content). Moreover, we assume that when alive controllers and switches are responsive, in that they always respond to any messages received by them. Finally, we assume that when switches or controllers become unresponsive, their absence triggers a network update by either a neighboring switch or the operator, who notices an unresponsive controller and removes it from the controller group via a policy update (Appendix B).

We define *dataplane configuration* as the set of forwarding entries installed at all functioning switches in the network; and *network state* as the undirected graph (along with all the relevant metadata about link properties) whose nodes are (functioning) switches, hosts, and controllers in a network, and where edges are the functional links in the network. The *network configuration* represents the current network state and dataplane configuration. A *network policy*, which is specified by an operator, is a set of predicates over the network configuration: a policy is implemented by a network if and only if the network configuration satisfies that predicate.

Given these definitions, we state that a network is *correct* if and only if it implements all network policies. A network is rendered *incorrect* if a policy predicate becomes invalid as a consequence of one or more *network events*, which are changes either to network state or network policy. Controllers respond to network events by modifying the dataplane configuration in an attempt to restore correctness.

To implement these updates in a consistent manner (*i.e.,* to solve the consistent updates problem), the controllers use a *dataplane consistency mechanism*. The controllers also use a *control plane consistency mechanism* to ensure that after a network event all controllers eventually agree on the network configuration and that the data plane configuration converges to a correct configuration (*i.e.,* the one appropriate to the current network state and policy).

The consistency mechanisms can be evaluated based on how they ensure correctness following a network event. Their design is thus intimately tied to the nature of *policies* implemented by a network and the types of *events* encountered, which we now discuss.

3.2 Policy Classes

Operators can specify *liveness policies* and *safety policies* for the network. Liveness policies are those that must hold in steady state, but can be violated for short periods of time while the network responds to network events. Sample liveness properties include:

1. *Connectivity*: requiring that the data plane configuration is such that packets from a source are delivered to the appropriate destination.

2. *Shortest Path Routing*: requiring that all delivered packets follow the shortest path in the network.

3. *Traffic Engineering*: requiring that for a given traffic matrix, routes in the network are such that the maximum link utilization is below some target.

Note that all of these examples require some notion of global network state (*i.e.,* one cannot evaluate whether a path is shortest unless one knows the entire network topology). Note that such policies are inherently liveness, because, as discussed below, one cannot ensure that they always hold given the reliance on global state.

A *safety policy* must always hold, regardless of any sequence of network events. Following standard impossibility results in distributed systems [20], a policy is enforceable as a safety policy in SDN if and only if it can be expressed as a condition on a path (*i.e.,* a path either obeys or violates the safety condition, regardless of what other paths are available). Safety properties include:

1. *Waypointing*: requiring that all packets sent between a given source and destination traverse some middlebox in a network.

2. *Isolation*: requiring that some class of packets is never received by an end host (such as those sent from another host, or with a given port number).

In contrast to liveness properties that operate over more global state, these properties can be enforced using information carried with the packet using mechanisms borrowed from the consistent updates literature. Therefore, we extend the tagging approach from the consistent updates literature to implement safety policies in SCL. Our extension, and proofs showing that this is both necessary and sufficient, are presented in Appendix A.

3.3 Network Events

Next we consider the nature of network events, focusing on two broad categories:

1. *Unplanned network events*: These include events such as link failures and switch failures. Since these events are unplanned, and their effects are observable by users of the network, it is imperative that the network respond quickly to restore whatever liveness properties were violated by these topology events. Dealing with these events is the main focus of our work, and we address this in §5.

2. *Policy Changes*: Policy state changes slowly (*i.e.,* at human time scales, not packet time scales), and perfect *policy availability* is not required, as one could temporarily block policy updates without loss of *network availability* since the network would continue operating using the previous policy. Because time is not of the essence, we use two-phase commit when dealing with policy changes, which allows us to ensure that the policy state at all controllers is always consistent. We present mechanisms and analysis for such events in Appendix B.

Obviously not all events fall into these two categories, but they are broader than they first appear. Planned network events (*e.g.,* taking a link or switch down for maintenance, restoring a link or switch to service, changing link metadata) should be considered policy changes, in that these events are infrequent and do not have severe time constraints, so taking the time to achieve consistency before implementing them is appropriate. Load-dependent policy changes (such as load balancing or traffic engineering) can be dealt with in several ways: (i) as a policy change, done periodically on a time scale (*e.g.,* every five minutes) that is long compared to consistency mechanisms; (ii) as an unplanned network event, but the performance of the resulting flow entries should not depend critically on them being applied simultaneously; or (iii) as a dataplane mechanism (as in MATE [3]) where the control plane merely computes several paths and the dataplane mechanisms responds to load changes in real time without consulting the control plane. In Appendix B.2 we discuss these options in more detail.

3.4 The Focus of Our Work

In this paper, we focus on how one can design an SDN control plane to handle unplanned network events so that liveness properties can be more quickly restored. However, for completeness, we also present (but claim no novelty for) mechanisms that deal with policy changes and safety properties. In the next section, we present our design of SCL, followed by an analysis (in Section 5) of how it deals with liveness properties. In Section 6 we describe SCL's implementation, and in Section 7 present results from both our implementation and simulations on SCL's performance. We delay until the appendices discussion of how SCL deals with safety properties.

4 SCL Design

SCL acts as a coordination layer for single-image controllers (*e.g.,* Pox, Ryu, etc.). Single image controllers are designed to act as the sole controller in a network, which greatly simplifies their design, but also means that they cannot survive any failures, which is unacceptable in production settings. SCL allows a single image controller to be replicated on multiple physical controllers thereby forming a distributed SDN control plane. We require that controllers and applications used with SCL meet some requirements (§4.1) but require no other modifications.

4.1 Requirements

We impose four requirements on controllers and applications that can be used with SCL:

(a) **Deterministic**: We require that controller applications be deterministic with respect to network state. A similar requirement also applies to RSM based distributed controllers (*e.g.,* ONOS, Onix).

(b) **Idempotent Behavior**: We require that the commands controllers send switches in response to any events are idempotent, which is straightforward to achieve when managing forwarding state. This requirement matches the model for most OpenFlow switches.

(c) **Triggered Recomputation**: We require that on receiving an event, the controller recomputes network state based on a log containing the sequence of network events observed thus far. This requires that the controller itself not retain state and incrementally update it, but instead use the log to regenerate the state of the network. This allows for later arriving events to be inserted earlier in the log without needing to rewrite internal controller state.

(d) **Proactive Applications**: We require that controller applications compute forwarding entries based on their picture of the network state. We do not allow reactive controller applications which respond to individual packet-ins (such as might be used if one were implementing a NAT on a controller).

4.2 General Approach

We build on existing single-image controllers that recompute state from a log of events every time recomputation is triggered (as in the triggered recomputation requirement above). To achieve eventual correctness, SCL must ensure (assuming no further updates or events) that in any network partition containing a controller eventually (i) every controller has the same log of events, and (ii) this log accurately represents the current state of the network. In traditional distributed control planes, these requirements are achieved by assuming that no more than a quorum (usually no more than one-half) of controllers can fail simultaneously, and using consensus to commit events before they are acted on. SCL ensures these requirements (without the restriction on having a controller quorum) through the use of two mechanisms:

Gossip: All live controllers in SCL periodically exchange their current event logs. This ensures that eventually the controllers agree on the log of events. However, because of our failure model, this gossip cannot by itself guarantee that logs reflect the current network state.

Periodic Probing: Even though switches send controllers notifications when network events (*e.g.,* link failures) occur, a series of controller failures and recoveries (or even a set of dropped packets on links) can result in situations where no live controller is aware of a network event. Therefore, in SCL all live controllers periodically send probe messages to switches; switches respond to these probe messages with their set of working links and their flow tables. While this response is not enough to recover all lost network events (*e.g.,* if a link fails and then recovers before a probe is sent then we will never

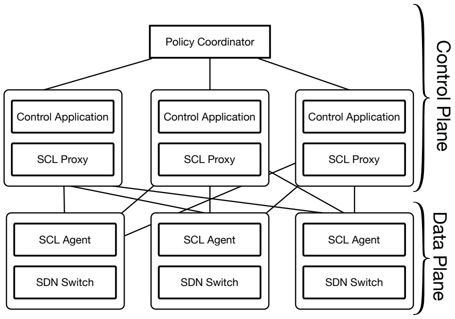

Figure 1: Components in SCL.

learn of the link failure), this probing mechanism ensures eventual awareness of the current network state and dataplane configuration.

Note that we explicitly design our protocols to ensure that controllers never disagree on policy, as we specify in greater detail in Appendix B. Thus, we do not need to use gossip and probe mechanisms for policy state.

Finally, in addition to these mechanisms, SCL requires that a *robust* communication channel be used for control plane communications, *i.e.,* the channel used by switches when notifying controllers of network events, by controllers when gossiping with each other, and by controllers for periodic probes. A *robust* communication channel is one where connectivity is maintained as long as the network is not partitioned, *i.e.,* there is a forwarding path between any two nodes not separated by a network partition. This robustness means that controllers will eventually receive gossip messages from any controllers that are not partitioned away, and will eventually learn of network events from any switch that is not located across a network partition. Note, we however *do not assume that this channel is lossless*, and so our mechanisms work even when messages from switches to controllers, and messages between controllers, might be lost due to failures, congestion or other causes.

4.3 Components

Figure 1 shows the basic architecture of SCL. We now describe each of the components in SCL, starting from the data plane.

SDN Switches: We assume the use of standard SDN switches, which can receive messages from controllers and update their forwarding state accordingly. We make no specific assumptions about the nature of these messages, other than that they are idempotent.

SCL-agent: The SCL-agent acts as a proxy between the control plane and switch's control interface. The SCL-agent is responsible for implementing SCL's robust control plane channels, for responding to the periodic probe messages, and for forwarding any switch notifications (*e.g.,* link failures or recoveries) to all live controllers in SCL. The SCL-agent, with few exceptions, immediately delivers any flow table update messages to the switch, and is therefore not responsible for providing any ordering or consistency semantics.

SCL-proxy: The SCL-proxy implements SCL's consistency mechanisms for both the control and data plane. To implement control plane consistency, the SCL-proxy (a) receives all network events (sent by the SCL-agent), (b) periodically exchanges gossip messages with other controllers; and (c) sends periodic probes to gain awareness of the dataplane state. It uses these messages to construct a consistently ordered log of network events. The SCL-proxy also keeps track of the network's dataplane configuration, and uses periodic-probe messages to discover changes. The SCL-proxy triggers recomputation at its controller whenever it observes a change in the log or dataplane configuration. This results in the controller producing a new dataplane configuration, and the SCL-proxy is responsible for installing this configuration in the dataplane. SCL implements data plane consistency by allowing the SCL-proxy to transform this computed dataplane configuration before generating switch update messages as explained in Appendix A.

Controller: As we have already discussed, SCL uses standard single image controllers which must meet the four requirement in §4.1.

These are the basic mechanisms (gossip, probes) and components (switches, agents, proxies, and controllers) in SCL. Next we discuss how they are used to restore *liveness policies* in response to *unplanned topology updates*. We later generalize these mechanisms to allow handling of other kinds of events later in Appendix B.

5 Liveness Policies in SCL

We discuss how SCL restores *liveness policies* in the event of *unplanned topology updates*. We begin by presenting SCL's mechanisms for handling such events, and then analyze their correctness.

5.1 Mechanism

Switches in SCL notify their SCL-agent of any network events, and SCL-agents forward this notification to all SCL-proxies in the network (using the control plane channel discussed in §6). Updates proceed as follows once these notifications are sent:

- On receiving a notification, each SCL-proxy updates its event log. If the event log is modified[9], the proxy triggers a recomputation for its controller.
- The controller updates its network state, and computes a new dataplane configuration based on this

[9] Since SCL-proxies update logs in response to both notifications from SCL-agents and gossip from other controllers, a SCL-proxy may be aware of an event before the notification is delivered.

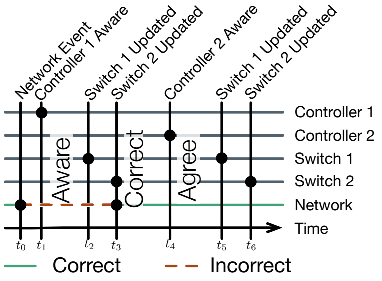

Figure 2: Response to an unplanned topology event in SCL.

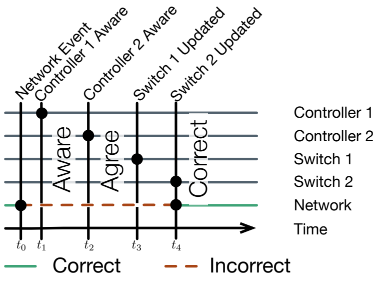

Figure 3: Response to an unplanned topology event when using consensus based controllers.

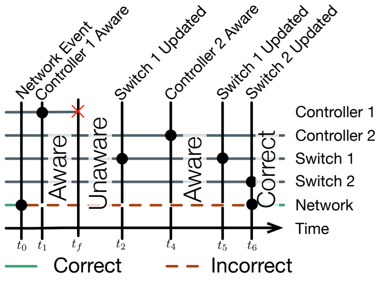

Figure 4: Response to one unplanned topology event in the presence of controller failure.

updated state. It then sends a set of messages to install the new dataplane configuration which are received by the SCL-proxy.

- The SCL-proxy modifies these messages appropriately so safety policies are upheld (Appendix A) and then sends these messages to the appropriate switch SCL-agents.
- Upon receiving an update message, each switch's SCL-agent immediately (with few exceptions discussed in Appendix B) updates the switch flow table, thus installing the new dataplane configuration.

Therefore, each controller in SCL responds to unplanned topology changes without coordinating with other controllers; all the coordination happens in the coordination layer, which merely ensures that each controller sees the same log of events. We next show how SCL achieves correctness with these mechanisms.

5.2 Analysis

Our mechanism needs to achieve eventual correctness, *i.e.,* after a network event and in the absence of any further events, there must be a point in time after which policies always hold. This condition is commonly referred to as *convergence* in the networking literature, which requires a period of quiescence (*i.e.,* no further events) for networks to converge. Note that during quiescence, no new controllers are added because adding a controller requires either a network event (when a partition is repaired) or a policy change (when a new controller is inserted into the network), both of which violate the assumption of quiescence. We observe that *eventual correctness* is satisfied once the following properties hold during the quiescent period:

i. The control plane has reached *awareness* (*i.e.,* at least one controller is aware of the current network configuration).
ii. The controllers have reached *agreement* (*i.e.,* they agree on network and policy state).
iii. The dataplane configuration is correct (the flow entries are computed with the correct network and policy state).

To see this consider these conditions in order. Because controllers periodically probe all switches, no controller (that remains up) can remain ignorant of the current network state forever. Recall that we assume that switches are responsive (while up, if they are probed infinitely often they respond infinitely often) and the control channel is fair (if a message is sent infinitely often, it is delivered infinitely often), so an infinite series of probes should produce an infinite series of responses. Once this controller is aware of the current network state, it cannot ever become unaware (that is, it never forgets the state is knows, and there are no further events that might change this state). Thus, once condition 1 holds, it will continue to hold during the quiescent period.

Because controllers gossip with each other, and also continue to probe network state, they will eventually reach agreement (once they have gossiped, and no further events arrive, they agree with each other). Once the controllers are in agreement with each other, and with the true network state, they will never again disagree in the quiescent period. Thus, once condition 1 and 2 hold, they will continue to hold during the quiescent period.

Similarly, once the controllers agree with each other and reality, the forwarding entries they compute will eventually be inserted into the network (when they probe a switch, if the forwarding entries they see are not consistent with what they have calculated, they will try to install the correct state). And once the forwarding entries agree with those computed based on the true network state, they will stay in agreement throughout the quiescent period. Thus, once condition 1, 2, and 3 hold, they will continue to hold during the quiescent period. This leads to the conclusion that if the network becomes quiescent it will eventually become correct, with all controllers aware of the current network state, and all switches having forwarding entries that reflect that state.

We now illustrate the difference between SCL's mechanisms and consensus based mechanisms through the use of an example. For ease of exposition we consider a small network with 2 controllers (quorum size is 2) and 2 switches, however our arguments are easily extended to larger networks. For both cases we consider the control plane's handling of a single link failure. We assume that the network had converged prior to the link failure.

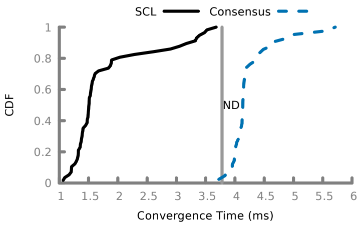

Figure 5: Fat Tree: CDF of time taken for routes to converge after single link failure. There are cases that exceed ND because of queuing in the network.

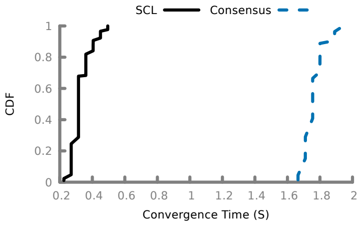

Figure 6: AS1221: CDF of time taken for routes to converge after single link failure.

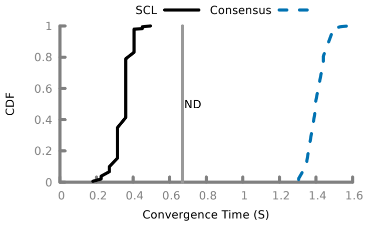

Figure 7: AS1239: Time taken for routes to converge after single link failure.

First, we consider SCL's handling of such an event, and show a plausible timeline for how this event is handled in Figure 2. At time t_0 a network event occurs which renders the network incorrect (some liveness properties are violated) and the control plane unaware. At time t_1 Controller 1 receives a notification about the event, this makes the control plane *aware* of the event, but the control plane is no longer in *agreement*, since Controller 2 is unaware. Controller 1 immediately computes updates, which are sent to the switch. Next at time t_3 both switches have received the updated configuration, this render the network *correct*. Finally, at time t_4, Controller 2 receives notification of this event, rendering the control plane in *agreement*. The network is thus converged at t_4. Note, since SCL does not use consensus, this is not the only timeline possible in this case. For example, another (valid) timeline would have the controllers reaching *agreement* before the dataplane has reached *correctness*. Finally, we also note that even in cases where a notification is lost, gossip would ensure that the network will reach agreement in this case.

Next, we show how such an event is handled in a consensus based controller framework in Figure 3. Without loss of generality, we assume that Controller 1 is the leader in this case. Similar to above, the control plane becomes aware at time t_1. Next, at time t_2, Controller 2 becomes aware of the event through the consensus mechanism, the controllers therefore reach *agreement*. Once agreement has been reached Controller 1 computes updates, and the network is rendered correct at time t_4. Consensus mechanisms require that any configuration changes occur after agreement, hence this is the only plausible timeline in this case. Finally, observe that when consensus mechanisms are used the dataplane configuration is updated once (at t_3 and t_4), while SCL issues two sets of updates, one from each controller. This is a necessary consequence of our design.

Finally, we consider a case where a controller, *e.g.,* Controller 1, fails while the event is being handled. Because we assumed that our network had 2 controllers, consensus algorithms cannot make progress in this case. We therefore focus on SCL's response, and show a timeline in Figure 4. In this case the control plane is temporarily aware of the network event at time t_1, but becomes unaware when Controller 1 fails at time t_f. However, since switches notify all live controllers, Controller 2 eventually receives the notification at time t_4, rendering the control plane *aware* and in *agreement*. At this time Controller 2 can generate updates for both Switch 1 and 2 rendering the network correct. Note, that even in cases where the Controller 2's notification is lost (*e.g.,* due to message losses), it would eventually be aware due to periodic probing.

6 Implementation

Our implementation of SCL uses Pox [15], a single-image controller. To simplify development of control applications, we changed the *discovery* and *topology* modules in POX so they would use the log provided by the SCL-proxy in SCL. We also implemented a SCL-agent that works in conjunction with OpenVSwitch [21], a popular software switch. In this section we present a few implementation details about how we implement robust channels (§6.1); and a few optimizations that we found important for our implementation (§6.2).

6.1 Robust Channel

Most existing SDN implementations require the use of a separate control network (referred to as out-of-band control). In contrast, our implementation of SCL reuses the same network for both control and data packets. The use of such an *in-band control* network is essential for implementing *robust* channels required by SCL (where we assume the control channel functions between any two connected nodes). In our design, the SCL-agent is responsible for implementing control channels. It does so by first checking if it has previously seen a received message, and flooding (*i.e.,* broadcasting) the control message if it has not. Flooding the message means that each message goes through all available paths in the network, ensuring that in the absence of congestion all control messages are guaranteed to be delivered to all nodes in the same network partition. This meets our requirements for *robust channels* from §4.2. We show in the evaluation that this broadcast based robust channel consumes a small percentage (< 1Mbps) of the network bandwidth. Furthermore, since SCL's correctness does not depend on reliable delivery of control plane messages, we also limit the amount of bandwidth allocated for control messages (by prioritizing them) and rely on switches to drop any traffic in excess of this limit, so that even under extreme circumstances the control traffic is limited to a small fraction of the total bandwidth.

6.2 Optimizations

Our implementation also incorporates three optimization to reduce bandwidth usage. Our optimizations generally aim to reduce the size of messages, but occasionally lead to an increase in the number of messages.

First, we observe that each switch's response to periodic probes includes a copy of its flow tables along with link state information. Switch flow table sizes can range from several 1000-100K flow entries, and in a naive implementation these messages would rapidly consume most of the bandwidth available for control channels. However, in the absence of controller failures (*i.e.,* in the common case) it is unlikely that a controller's version of the switch's flow table differs from reality. Therefore, when sending a periodic probe request to a switch s, the controller's SCL-proxy includes the hash of their view of s's flow tables in the request. The SCL-agents sends a flow table as a part of their response if and only if this hash differs from the hash for the actual flow table. This dramatically reduces the size of probe responses in the common case.

Second, we require that SCL-proxies converge on a single network event log. However, here again sending the entire log in each Gossip message is impractical. To address this we use a log truncation algorithm, whereby SCL-agents no longer include log entries about which all active controllers agree. This works as follows: (a) SCL-proxies always include new or modified log entries (since a previous Gossip round) in gossip message; (b) any time a SCL-proxy finds that all active controllers agree on a log entry it marks that entry as committed and stops including it in the log; and (c) SCL-proxies can periodically send messages requesting uncommitted log entries from other controllers. The final mechanism (requesting logs) is required when recovering from partitions.

Finally, similar to other SDN controllers, SCL does not rewrite switch flow tables and instead uses incremental flow updates to modify the tables. Each SCL-proxy maintains a view of each the current network configuration, and uses this information to only send updates when rules need to change. Note, in certain failure scenarios, *e.g.,* when a controller fails and another simultaneously recovers, this might incur increase time for achieving correctness, however our correctness guarantees still hold.

Topology	Switches	Links	Diameter	ND	CD
AS1221	83	236	0.63 S	0.77 S	0.72 S
AS1239	361	1584	0.54 S	0.67 S	0.54 S
Fat Tree	320	3072	2.52 ms	3.78 ms	1.336 ms

Table 1: Properties of the topologies for which we run evaluation.

As we show later in §7 these optimizations are enough to ensure that we use no more than 1Mbps of bandwidth for control messages in a variety of networks.

7 Evaluation

As mentioned earlier, the standard approach to replicating SDN controllers is to use consensus algorithms like Paxos and RAFT to achieve consistency among replicas. The design we have put forward here eschews such consensus mechanisms and instead uses mostly-unmodified single-image controllers and a simple coordination layer. Our approach has two clear advantages:

Simplicity: Consistency mechanisms are complicated, and are often the source of bugs in controller designs. Our approach is quite simple, both conceptually and algorithmically, leading to a more robust overall design whose properties are easy to understand.

Eventual Correctness: This seems like the most basic requirement one could have for distributed controllers, and is easily achieved in our design, yet consensus mechanisms violate it in the presence of partitions.

Given these advantages, one might ask why might one choose current consensus-based approaches over the one we have presented here? We investigate three possible reasons: response times, impact of controller disagreement, and bandwidth overhead. Our analysis uses the network model presented in §7.1, which allows us to quantitatively reason about response times. We then use simulations to gain a deeper understanding of how the design operates. We use simulation rather than our implementation because we we can more closely control the environment and monitor the results.

Finally, in §7.4 we use CloudLab [1] to compare convergence times for SCL and ONOS [2] when responding to link failures in a datacenter topology. Our results demonstrate that the effects observed in simulation also hold for our implementation.

7.1 Response to Network Events

The primary metric by which SDN controller designs should be evaluated is the extent to which they achieve the desired properties or invariants. For example, if we ignore the presence of partitions (where consensus mechanisms will always perform poorly), then we can ask how

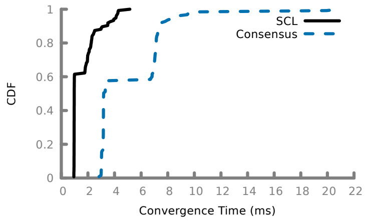

Figure 8: Fat Tree: CDF of time taken for the network (switches and controllers) to fully agree after a single link failure.

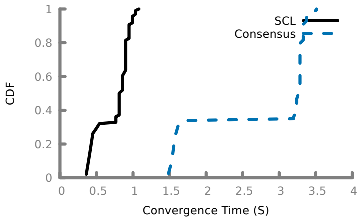

Figure 9: AS1221: CDF of time taken for the network (switches and controllers) to fully agree after a single link failure.

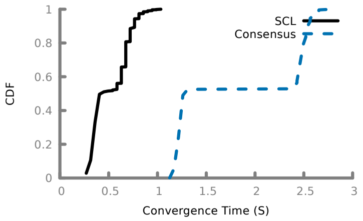

Figure 10: AS1239: CDF of time taken for the network (switches and controllers) to fully agree after a single link failure.

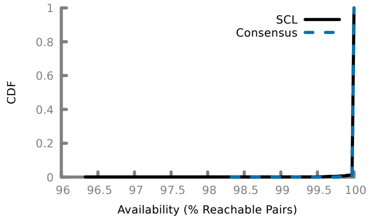

Figure 11: Fat Tree: CDF of availability showing percentage of physically connected host pairs that are reachable based on routing rules.

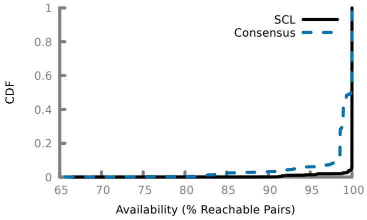

Figure 12: AS 1221: CDF of availability showing percentage of physically connected host pairs that are reachable based on routing rules.

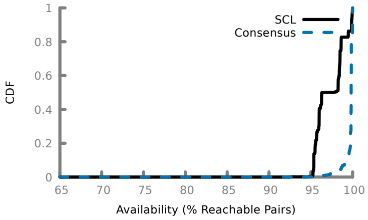

Figure 13: AS1239: CDF of availability showing percentage of physically connected host pairs that are reachable based on routing rules.

quickly a design responds to network events. We can express these delays using the notation that: $lat(x, y)$ is the network latency between points x and y; network nodes c are controllers and network nodes s are switches; and CD is the time it takes the consensus algorithm to converge.

For SCL, the total response delay for an event detected by switch s is given by the sum of: (a) the time it takes for the event notification to reach a controller ($lat(s, c)$ for some c) and (b) the time for controller updates to reach all affected switches s' ($lat(c, s')$). Thus, we can express the worst case delay, which we call ND, as:

$$ND \quad = \quad \max_{s} \max_{s'} \min_{c} \quad [lat(s, c) + lat(c, s')]$$

which represents the longest it can take for a switch s' to hear about an event at switch s, maximized over all s, s'.

When consensus-based approaches are used we must add the consensus time CD to ND (and in fact ND is an underestimate of the delay, because the delay is $[lat(s, c) + lat(c, s')]$ for a given c that is the leader, not the controller that minimizes the sum).[10]

At this simple conceptual level, three things become apparent: (i) typically SCL will respond faster than consensus-based systems because they both incur delay ND and only consensus-based systems incur CD, (ii) as the number of controllers increases, the relative performance gap between SCL and consensus-based approaches will increase (CD typically increases with more participants, while ND will decrease), and (iii) SCL has better availability (SCL only requires one controller to be up, while consensus-based designs require at least half to be up).

We further explore the first of these conclusions using simulations. In our simulations we compare SCL to an idealized consensus algorithm where reaching consensus requires that a majority of controllers receive and acknowledge a message, and these messages and acknowledgments do not experience any queuing delay. We therefore set CD to be the median round-trip time between controllers. We run our simulation on three topologies; two AS topologies (AS 1221 and 1239) as measured by the RocketFuel project [24], and a datacenter topology (a 16-ary fat-tree). We list sizes and other properties for these topologies in Table 1. Our control application computes and installs shortest path routes.

Single link failure We first measure time taken by the network to converge back to shortest path routes after a link failure. Note that here we are not measuring how long it takes for all controllers and switches to have a consistent view of the world (we return to that later), just when are shortest routes installed after a failure. For each topology, we measured the convergence time by failing random links and plot CDFs in Figure 5, 6, 7. These results show that SCL clearly restores routes faster than even an idealized Paxos approach, and typically faster than the worst-case bound ND (which would require the failure to be pessimally placed to achieve).

But one can also ask how long it takes to actually achieve full agreement, where one might think consensus-based algorithms would fare better. By full agreement we mean: (a) the dataplane has converged so

[10] But if the consensus protocol is used for leader election, then ND is actually an underestimate of the worst-case delay, as there is no minimization over c.

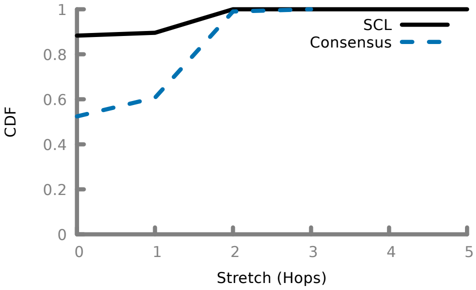

Figure 14: AS 1221: CDF of path inflation

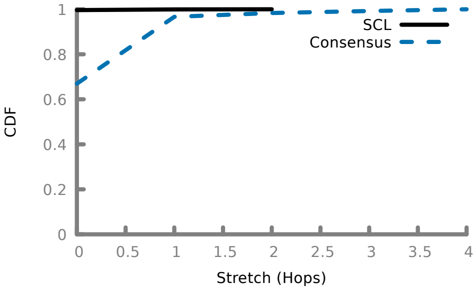

Figure 15: AS1239: CDF of path inflation.

Traffic Type	Fat Tree	AS 1221	AS1239
Overall	63.428Kbps	205.828 Kbps	70.2 Kbps
Gossip	46.592 Kbps	3.2 Kbps	28.6 Kbps
Link Probing	16.329 Kbps	3.88 Kbps	36.4 Kbps
Routing Table Probing	0.0Kbps	248.4 Kbps	4.8Kbps

Table 2: Control Bandwidth usage for different topologies

that packets are routed along the shortest path, (b) all controllers have received notification for the link failure event and (c) every controller's logical configuration corresponds with the physical configuration installed in the data plane. For each topology, we measured convergence time by failing random links and plot CDFs in Figure 8, 9, 10. Thus, even when measuring when the network reaches full agreement, SCL significantly outperforms idealized Paxos. Note that what consensus-based algorithms give you is knowing *when* the controllers agree with each other, but it does not optimize how quickly they reach agreement.

Continuous link failure and recovery To explore more general failure scenarios, we simulate five hours with an ongoing process of random link failure and recoveries. The time between link failures in the network (MTBF) is 30 seconds, and each link's mean-time-to-recovery (MTTR) is 15 seconds; these numbers represent an extremely aggressive failure scenario. We look at two metrics: connectivity and stretch.

For connectivity, we plot the percentage of available[11] host pairs in Figures 11,12, 13. SCL and Paxos offer comparable availability in this case, but largely because the overall connectivity is so high. However, even when hosts are connected, the paths between them might be suboptimal. To investigate this we plotted the stretch of all connected host-pairs (comparing the shortest path to the current path given by routing), as shown in Figures 14, 15. We did not observe any stretch in the datacenter topology, this can be explained by the large number of equal cost redundant paths in these topologies. As one can see, SCL significantly outperforms ideal Paxos in terms of path stretch in the other two topologies.

7.2 Impact of Controller Disagreement

While the previous convergence results clearly indicate that SCL responds faster than consensus-based algorithms, one might still worry that in SCL the controllers may be giving switches conflicting information. We first observe that such an update race can happen only under rare conditions: controllers in SCL only update physical configuration in response to learning about new network events. If a single link event occurs, each controller would proactively contact the switch only after receiving this event and would install consistent configuration. A race is therefore only observable when two network events occur so close together in time so that different controllers observe them in different order, which requires that the time difference between these network events is smaller than the diameter of the network. But even in this case, as soon as all controllers have heard about each event, they will install consistent information. So the only time inconsistent information may be installed is during the normal convergence process for each event, and during such periods we would not have expected all switches to have the same configuration.

Thus, the impact of controller disagreement is minimal. However, we can more generally ask how often do controllers have an incorrect view of the network. In Figures 16, 17, 18 we show, that under the continuous failure/recover scenario described above, the CDF of how many links are incorrectly seen by at least one controller (where we periodically sample both the actual state of the physical network and the network state as seen by each controller). As we can see, typically there are very few disagreements between controllers and the physical state, and that SCL outperforms ideal Paxos in this regard. We also measured how often the event logs between controllers differed in SCL, and they agreed 99.4% of the time.

7.3 Bandwidth Usage

SCL uses broadcast for all control plane communication. This has several advantages: it avoids the need for bootstrapping (*i.e.,* running a routing algorithm on the control plane, which then allows the controllers to manage

[11] By which we mean host pairs that are physically connected and can communicate using the current physical configuration

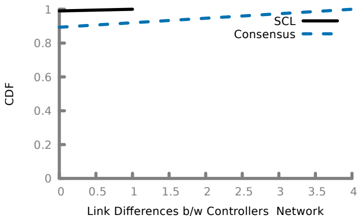

Figure 16: CDF of number of times network state disagreed with controller's network state on a fat tree.

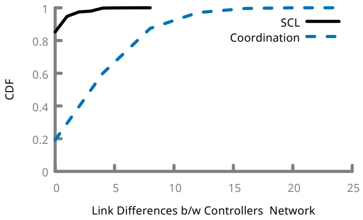

Figure 17: AS 1221: CDF of number of times network state disagreed with any controller's network state.

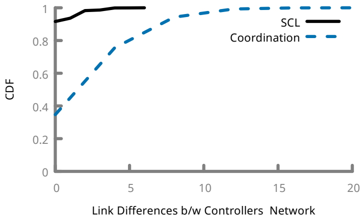

Figure 18: AS1239: CDF of number of times network state disagreed with any controller's network state.

the data plane), is extremely robust to failures, and provides automatic alignment between data plane and control plane connectivity. This last one is particularly important, because otherwise one would have to handle special cases such as where controllers within a given data plane partition might not be able to talk to each other. Such cases would unavoidably complicate the controller logic, and make eventual correctness harder to achieve.

However, one might think that this requires too much bandwidth to be practical. Here we consider this question by running a simulation where the mean time between link failures (MTBF) is 10 minutes, and the mean time for each link to recover is 5 minutes. The control traffic depends on the control plane parameters, and here we set SCL's gossip timer at 1 second and network state query period at 20 seconds. Table 2 shows the control plane's bandwidth consumption for the topologies we tested. On average we find that SCL uses a modest amount of bandwidth. We also looked at instantaneous bandwidth usage (which we measured in 0.5 second buckets) and found that most of the time peak bandwidth usage is low: for the fat tree topology 99.9% of the time bandwidth is under 1Mbps, for AS1221 98% of the time bandwidth is under 1Mbps, and for AS1239 97% of the time bandwidth is under 1Mbps.

One might ask how the bandwidth used on a given link scales with increasing network size. Note that the dominant cause of bandwidth usage is from responses to link and routing table probing messages. The number of these messages grows as the number of switches (not the number of controllers), and even if we increase the number of switches three orders of magnitude (resulting in a network with about 100K switches, which is larger than any network we are aware of) the worst of our bandwidth numbers would be on the order of 200Mbps, which is using only 2% of the links if we assume 10Gbps links. Furthermore, the failure rate is extremely high; we would expect in a more realistic network setting that SCL could scale to even large networks with minimal bandwidth overhead.

System	First Update			Correct		
	Median	Min	Max	Median	Min	Max
SCL	117.5ms	91ms	164ms	268.5ms	201ms	322ms
ONOS	159.5ms	153ms	203ms	276.5ms	164ms	526ms

Table 3: Time taken before a controller begins reacting to a network event (First Update) and before the network converges (Correct).

7.4 Response Time of the SCL Implementation

The primary metric by which we compare our implementation to existing distributed controller frameworks is response time. We begin by showing the improvements achieved by our implementation of SCL when compared to a traditional distributed controller in a datacenter network. We ran this evaluation on CloudLab [1], where the dataplane network consisted of 20 switches connected in a fat tree topology (using 48 links). Each of the dataplane switches ran OpenVSwitch (version 2.5.0). In the control plane we used 3 replicated controllers. To fit into the CloudLab environment we modified SCL to use an out-of-band control channel: each SCL-proxy established a TCP connection with all SCL-agents and forwarded any network events received from the switch on all of these connections. For comparison we used ONOS (version 1.6.0). Both controllers were configured to compute shortest path routes.

In our experiments we measured the time taken for paths to converge to their correct value after a single link failure in the network. We measured both the time before a controller reacts to the network event (First Update in the table) and before all rule updates are installed (Correct). We repeated this test 10 times and report times in milliseconds elapsed since link failures (Table 3). We find that SCL consistently responds to network events before ONOS; this is in line with the fact that ONOS must first communicate with at least one other controller before installing rules. In our experiments, we found that this could induce a latency of up to 76ms before the controller issues its first updates. In the median case, SCL also achieved correctness before ONOS, however the gap is smaller in this case. Note, however, the latencies in this setup are very small, so we expected to find a small performance gap.

8 Conclusion

Common wisdom holds that replicated state machines and consensus are key to implementing distributed SDN controllers. SCL shows that SDN controllers eschewing consensus are both achievable and desirable.

References

[1] A. Akella. Experimenting with next-generation cloud architectures using cloudlab. *IEEE Internet Computing*, 19:77–81, 2015.

[2] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. Onos: towards an open, distributed sdn os. In *HotSDN*, 2014.

[3] A. Elwalid, C. Jin, S. H. Low, and I. Widjaja. Mate: Mpls adaptive traffic engineering. In *INFOCOM*, 2001.

[4] S. Ghorbani and M. Caesar. Walk the Line: Consistent Network Updates with Bandwidth Guarantees. In *HotSDN*, 2012.

[5] J. Gray. Notes on data base operating systems. In *Advanced Course: Operating Systems*, 1978.

[6] J. He, M. Bresler, M. Chiang, and J. Rexford. Towards robust multi-layer traffic engineering: Optimization of congestion control and routing. *IEEE Journal on Selected Areas in Communications*, 25:868–880, 2007.

[7] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[9] S. Kandula, D. Katabi, B. S. Davie, and A. Charny. Walking the tightrope: responsive yet stable traffic engineering. In *SIGCOMM*, 2005.

[10] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *SOSR*, 2015.

[11] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *HotSDN*, 2013.

[12] 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.

[13] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.

[14] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. In *HotNets*, 2013.

[15] J. Mccauley. POX: A Python-based OpenFlow Controller. http://www.noxrepo.org/pox/about-pox/.

[16] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker. Extending sdn to large-scale networks. In *Open Networking Summit*, 2013.

[17] J. McClurg, N. Foster, and P. Cerný. Efficient synthesis of network updates. *CoRR*, abs/1403.5843, 2015.

[18] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates. In *SOSR*, 2015.

[19] D. Ongaro. Bug in Single-Server Membership Changes. raft-dev post 07/09/2015, https://goo.gl/a7hKXb, 2015.

[20] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker. Cap for networks. In *HotSDN*, 2013.

[21] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[22] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[23] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. consistent Updates for Software-Defined Networks: Change You Can Believe In! In *HotNets*, 2011.

[24] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. In *SIGCOMM*, 2002.

[25] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *INM/WREN*, 2010.

A Safety Policies in SCL

Next we look at how SCL ensures that *safety policies* always hold. From our definitions, safety policies must never be violated, even in the presence of an arbitrary sequence of network events. Unlike liveness policies, we cannot rely on the control plane to enforce these policies. We therefore turn to data plane mechanisms for enforcing these policies.

As noted earlier, safety policies cannot have any dependence on what other paths are available in the network (disallowing policies affecting optimality, *e.g.,* shortest path routing) or on what other traffic is in the network (disallowing policies which ensure traffic isolation). The safety policies listed in §3.2 meet this requirement: waypointing requires that all chosen paths include the waypoint, while isolation limits the set of endpoints that valid paths can have. Next, we present our mechanism for enforcing safety policies, and then provide an analysis showing that this mechanism is both necessary and sufficient.

A.1 Mechanism

Safety policies in SCL constrain the set of valid paths in the network. We assume the control applications only generate paths that adhere to these constraints. Such policies can therefore be implemented by using dataplane consistency mechanisms to ensure that packets only follow paths generated by a control application.

Our mechanisms for doing this extends existing work on consistent updates [4, 22]. However in contrast to these works (which implicitly focus on planned updates), our focus is explicitly restricted to unplanned topology updates. The primary aim of this mechanism is to ensure that each packet follows exactly one controller generated path. Similar to prior work, we accomplish this by associating a label with each path, and tagging packets on ingress with the label for a given path. Packets are then routed according to this label, ensuring that they follow a single path. Packets can be tagged by changing the VLAN tag, adding an MPLS label, or using other packet fields.

In SCL, the SCL-proxy is responsible for ensuring that packets follow a single path. Our mechanism for doing this is based on the following observation: since we assume controllers are *deterministic*, the path (for a single packet) is determined entirely by the current network state and policy. The SCL-proxy therefore uses a hash of the network state and policy as a policy label (Π). When a controller sends its SCL-proxy a flow table update, the SCL-proxy modifies each match predicate in the update. The match predicates are updated so that a packet matches the new predicate if and only if the packet is tagged with Π and if it would have matched the old predicate (*i.e.,* given a match-action rule r with match m, the SCL-proxy produces a rule r' which will only match packets which are tagged with Π and match m). The SCL-proxy also augments the update to add rules which would tag packets on ingress with an appropriate tag. The SCL-proxy then sends these rules to the appropriate switch (and its SCL-agent). Once these rules are installed, packets are forwarded along exactly one path from ingress to egress, and are dropped when this is not

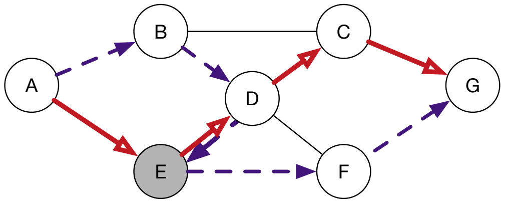

Figure 19: Example where waypointing is violated when using inconsistent paths.

possible.

Since, the labels used by SCL are based on the network state and policy, controllers in agreement will use the same label, and in steady state all controllers will install exactly one set of rules. However, during periods of disagreement, several sets of rules might be installed in the network. While these do not result in a correctness violation, they can result in resource exhaustion (as flow table space is consumed). Garbage collection of rules is a concern for all consistent update mechanisms, and we find that we can apply any of the existing solutions [11] to our approach. Finally, we observe that this mechanism need not be used for paths which do not enforce a safety policy. We therefore allow the controller to mark rules as not being safety critical, and we do not modify the match field for such rules, reducing overhead. We do not require that single-image controllers mark packets as such, this mechanism merely provides an opportunity for optimization when control applications are SCL aware.

A.2 Analysis

Next we show that this mechanism is both necessary and sufficient for ensuring that safety policies hold. Since we assume that controllers compute paths that enforce safety policies, ensuring that packets are only forwarded along a computed path is *sufficient* to ensure that safety policies are upheld. We show necessity by means of a counterexample. We show that a packet that starts out following one policy compliant path, but switches over partway to another policy compliant path can violate a safety policy. Consider the network in Figure 19, where all packets from A to G need to be waypointed through the shaded gray node E. In this case, both the solid-red path and the dashed purple path individually meet this requirement. However, the paths intersect at both D and E. Consider a case where routing is done solely on packet headers (so we do not consider input ports, labels, etc.). In this case a packet can follow the path A—B—D—C—G, which is a combination of two policy compliant paths, but is not itself policy compliant. Therefore, it is *necessary* that packets follow a single path to ensure safety policies are held. Therefore our mechanism is both *necessary* and *sufficient*.

A.3 Causal Dependencies

Safety policies in reactive networks might require that causal dependencies be enforced between paths. For example, reactive controllers can be used to implement stateful firewalls, where the first packet triggers the addition of flow rules that allow forwarding of subsequent packets belonging to the connection. Ensuring correctness in this case requires ensuring that all packets in the connection (after the first) are handled by rules that are causally after the rules handling the first packet. Our mechanism above does not guarantee such causality (since we do not assume any ordering on the labels, or on the order in which different controllers become aware of network paths). We have not found any policies that can be implemented in proactive controllers (which we assume) that require handling causal dependencies, and hence we do not implement any mechanisms to deal with this problem. We sketch out a mechanism here using which SCL can be extended to deal with causal dependencies.

For this extension we require that each SCL-proxy maintain a vector clock tracking updates it has received from each switch, and include this vector clock in each update sent to a SCL-agent. We also require that each SCL-agent remember the vector clock for the last accepted update. Causality can now be enforced by having each SCL-agent reject any updates which happen-before the last accepted update, *i.e.,* on receiving an update the SCL-agent compares the update's vector clock v_u with the vector-clock for the last accepted update v_a, and rejects any updates where $v_u \preceq v_a$. The challenge here is that the happens-before relation for vector clocks is a partial order, and in fact v_u and v_a may be incomparable using the happens-before relation. There are two options in this case: (a) the SCL-agent could accept incomparable updates, and this can result in causality violations in some cases; or (b) the SCL-agent can reject the incomparable update. The latter is safe (*i.e.,* causality is never violated), however it can render the network unavailable in the presence of partitions since controllers might never learn about events known only to other controllers across a partition.

B Policy Changes in SCL

All the mechanisms presented thus far rely on the assumption that controllers in the network agree about *policy*. The *policy coordinator* uses 2-phase commit [5] (2PC) to ensure that this holds. In SCL, network operators initiate policy changes by sending updates to the *policy coordinator*. The policy coordinator is configured with the set of active controllers, and is connected to each controller in this set through a reliable channel (*e.g.,* established using TCP). On receiving such an update, the policy coordinator uses 2PC to update the controllers as follows:

1. The policy coordinator informs each controller that a policy update has been initiated, and sends each controller the new policy.
2. On receiving the new policy, each controller sends an acknowledgment to the policy coordinator. Controllers also start start queuing network events (*i.e.,* do not respond to them) and do not respond to them until further messages are received.
3. Upon receiving an acknowledgement from all controllers, the policy coordinator sends a message to all controllers informing them that they should switch to the new policy.
4. On receiving the switch message, each controller starts using the new policy, and starts processing queued network events according to this policy.
5. If the policy coordinator does not receive acknowledgments from all controllers, it sends an abort message to all controllers. Controllers receiving an abort message stop queuing network events and process both queued events and new events according to the old policy.

Two phase commit is not live, and in the presence of failures, the system cannot make progress. For example, in the event of controller failure, new policies cannot be installed and any controller which has received a policy update message will stop responding to network events until it receives the switch message. However, we assume that policy changes are *rare* and performed during periods when an administrator is actively monitoring the system (and can thus respond to failures). We therefore assume that either the policy coordinator does not fail during a policy update or can be restored when it does fail, and that before starting a policy update network administrators ensure that all controllers are functioning and reachable. Furthermore, to ensure that controllers are not stalled from responding to network events forever, SCL controllers gossip about *commit* and *abort* messages from the 2PC process. Controllers can commit or abort policy updates upon receiving these messages from other controllers. This allows us to reduce our window of vulnerability to the case where either (a) the policy coordinator fails without sending any commits or aborts, or (b) a controller is partitioned from the policy coordinator and all other controllers in the system.

The first problem can be addressed by implementing fault tolerance for the policy coordinator by implementing it as a replicated state machine. This comes at the cost of additional complexity, and is orthogonal to our work.

The second problem, which results from a partition preventing the policy coordinator from communicating with a controller cannot safely be solved without repairing the partition. This is not unique to SCL, and is true for all distributed SDN controllers. However, in some cases, restoring connectivity between the policy coordinator and all controller might not be feasible. In this case network operators can change the set of active controllers known to the policy agent. However it is essential that we ensure that controllers which are not in the active set cannot update dataplane configuration, since otherwise our convergence guarantees do not hold. Therefore, each SCL agent is configured with a set of *blacklisted* controllers, and drops any updates received from a controller in the blacklisted set. We assume that this blacklist can be updated through an out of band mechanism, and that operators blacklist any controller before removing them from the set of active controllers. Re-enabling a blacklisted controller is done in reverse, first it is added to the set of active controllers, this triggers the 2PC mechanism and ensures that all active controllers are aware of the current configuration, the operator then removes the controller from the blacklist.

Finally, note that SCL imposes stricter consistency requirements in responding to policy changes when compared to systems like ONOS and Onyx, which store policy using a replicated state machine, this is a trade-off introduced due to the lack of consistency assumed when handling network events. This is similar to recent work on consensus algorithms [7] which allow trade-offs in the number of nodes required during commits vs the number of nodes required during leader election.

B.1 Planned Topology Changes

Planned topology changes differ from unplanned ones in that operators are aware of these changes ahead of time and can mitigate their effects, *e.g.,* by draining traffic fron links that are about to be taken offline. We treat such changes as policy changes, *i.e.,* we require that operators change their policy to exclude such a link from being used (or include a previously excluded link), and implement them as above.

B.2 Load-Dependent Update

Load-dependent updates, which include policies like traffic engineering, are assumed by SCL to be relatively infrequent, occurring once every few minutes. This is the frequency of traffic engineering reported in networks such as B4 [8], which aggressively run traffic engineering. In this paper we focus exclusively on traffic engineering, but note that other load-dependent updates could be implemented similarly.

We assume that traffic engineering is implemented by choosing between multiple policy compliant paths based on a traffic matrix, which measures demand between pairs of hosts in a network. Traffic engineering in SCL can be implemented through any of three mechanisms, each of which allows operators to choose a different point in the trade-off space: (a) one can use techniques like TeXCP [9] and MATE [3] which perform traffic engineering entirely in the data plane; (b) operators can update traffic matrices using periodic probing, gossip or other techniques, which allows traffic engineering to be treated similar to unplanned topology updates; or (c) operators can update traffic matrices using the policy update mechanisms.

Dataplane techniques including TeXCP can be implemented in SCL unmodified. In this case, control plane applications must produce and install multiple paths between hosts, and provide mechanisms for a switch or middlebox to choose between these paths. This is compatible will all the mechanisms we have presented thus far; the only additional requirement imposed by SCL is that the field used to tag paths for traffic engineering be different from the field used by SCL for ensuring dataplane consistency. These techniques however rely on local information to chose paths, and might not be sufficient in some cases.

Updates to the traffic matrix can also be treated similar to unplanned topology updates, and might be triggered by responses to probe messages (which include switch flow tables, and hence demands on each flow), or gossip messages. Since such a matrix might vary often, we cannot assume that it converges to a single value across controllers. Instead we require that any traffic engineering function implemented using this mechanism is locally smooth (*e.g.,* [6]), *i.e.,* paths computed by controllers do not change significantly in response to small changes in the traffic matrix.

When treated as policy changes, each traffic matrix is submitted to a *policy coordinator* which uses 2PC to commit the parameters to the controller. In this case, we allow each update process to use a different coordinator, thus providing a trivial mechanism for handling failures in policy coordinators. However, similar to the policy update case, this mechanism does not allow load-dependent updates in the presence of network partitions.

C Ordering Log Messages

Ensuring that controllers reach agreement requires the use of a mechanism to ensure that event ordering at each controller is eventually the same (§5). In our analysis we assumed the used of some ordering function that used metadata added by SCL-agents to network events to produce a total order of events that all controllers would agree on. Here we describe that mechanism in greater detail. SCL-agents in SCL augment network events with a *local* (*i.e.,* per SCL-agent) sequence number. This sequence number allows us to ensure that event ordering in controller logs always corresponds to the causal order at each switch; we call this property *local causality*. SCL-agent failures might result in this sequence number being reset, we therefore append an epoch count (updated when a switch boots) to the sequence number, and ensure that sequence numbers from a SCL-agent are monotically increasing despite failures. We assume the epoch is stored in stable storage and updated each time a switch boots. We also assume that each switch is assigned an ID, and that switch IDs impose a *total order* on all switches. IDs can be assigned based on MAC addresses, or other mechanisms.

Our algorithm for ensuring consistent ordering depends on the fact that *gossip messages* include the position of each event in the sender's log. Our algorithm then is simple: when notified of an event by the data plane (this might be due to a direct event notification or because of periodic probing), each SCL-proxy inserts the event at the end of the log. If such an insertion violates local causality, SCL swaps events until local causality is restored, *e.g.,* if a controller is notified of an event e from a switch s with sequence number 5, but the log already contains another event e', from the same switch with sequence number 6, then these events are swapped so that e occupies the log position previously occupied by e' and e' appears at the end of the log. It is simple to see that the number of swaps needed is bounded. When a SCL-proxy receives a gossip message, it checks to see if its position for event disagrees with the position for the sender of the gossip message. If there is a disagreement, we swap messages to ensure that the message sent by a switch with a lower ID appears first. Again it is easy to see that the number of swaps required is bounded by the number of positions on which the logs disagree.

Assuming that the network being controlled goes through periods of change (where a finite number of new network events occur) followed by periods of quiescence (where no new network events occur), and that quiescent periods are long enough to reach agreement, then the only events reordered are those which occur within a single period of change. Since we assumed that only a finite number of new events occur within a period of change, there's only a finite number of event swaps that need to be carried out to ensure that the log ordering is the same across all controllers. Furthermore, any event that is not in its final position must be swapped during a round of gossip. This is because otherwise all controllers must agree on the message's position, and hence it will never subsequently be changed. This implies that all controllers must agree upon log ordering within a finite number of gossip rounds. Note, our assumption about quiescent periods is required by all routing algorithms, and is not unique to SCL.