Ravana: Controller Fault-Tolerance in Software-Defined Networking

Naga Katta, Haoyu Zhang, Michael Freedman, Jennifer Rexford
Princeton University
{nkatta, haoyuz, mfreed, jrex}@cs.princeton.edu

ABSTRACT

Software-defined networking (SDN) offers greater flexibility than traditional distributed architectures, at the risk of the controller being a single point-of-failure. Unfortunately, existing fault-tolerance techniques, such as replicated state machine, are insufficient to ensure correct network behavior under controller failures. The challenge is that, in addition to the application state of the controllers, the switches maintain hard state that must be handled consistently. Thus, it is necessary to incorporate switch state into the system model to correctly offer a "logically centralized" controller.

We introduce Ravana, a fault-tolerant SDN controller platform that processes the control messages transactionally and exactly once (at both the controllers and the switches). Ravana maintains these guarantees in the face of both controller and switch crashes. The key insight in Ravana is that replicated state machines can be extended with lightweight switch-side mechanisms to guarantee correctness, without involving the switches in an elaborate consensus protocol. Our prototype implementation of Ravana enables *unmodified* controller applications to execute in a fault-tolerant fashion. Experiments show that Ravana achieves high throughput with reasonable overhead, compared to a single controller, with a failover time under 100ms.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network Operating Systems*

General Terms

Design, Reliability

Keywords

Software-Defined Networking, Fault-Tolerance, Replicated State Machines, OpenFlow.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SOSR 2015, June 17 - 18, 2015, Santa Clara, CA, USA.
Copyright 2015 ACM 978-1-4503-3451-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2774993.2774996

1 Introduction

In Software-Defined Networking (SDN), a logically centralized *controller* orchestrates a distributed set of switches to provide higher-level networking services to end-host applications. The controller can reconfigure the switches (though *commands*) to adapt to traffic demands and equipment failures (observed through *events*). For example, an SDN controller receives events concerning topology changes, traffic statistics, and packets requiring special attention, and it responds with commands that install new forwarding rules on the switches. Global visibility of network events and direct control over the switch logic enables easy implementation of policies like globally optimal traffic engineering, load balancing, and security applications on commodity switches.

Despite the conceptual simplicity of centralized control, a single controller easily becomes a single point of failure, leading to service disruptions or incorrect packet processing [1, 2]. In this paper, we study SDN fault-tolerance under crash (fail-stop) failures. Ideally, a fault-tolerant SDN should behave the same way as a fault-free SDN from the viewpoint of controller applications and end-hosts. This ensures that controller failures do not adversely affect the network administrator's goals or the end users. Further, the right abstractions and protocols should free controller applications from the burden of handling controller crashes.

It may be tempting to simply apply established techniques from the distributed systems literature. For example, multiple controllers could utilize a distributed storage system to replicate durable state (*e.g.*, either via protocols like two-phase commit or simple primary/backup methods with journaling and rollback), as done by Onix [3] and ONOS [4]. Or, they could model each controller as a replicated state machine (RSM) and instead consistently replicate the set of *inputs* to each controller. Provided each replicated controller executes these inputs deterministically and in an identical order, their internal state would remain consistent.

But maintaining consistent controller state is only part of the solution. To provide a logically centralized controller, one must also ensure that the *switch state* is handled consistently during controller failures. And the semantics of switch state, as well as the interactions between controllers and switches, is complicated. Broadly speaking, existing systems do not reason about switch state; they have not rigorously studied the semantics of processing switch events and executing switch commands under failures.

Yet one cannot simply extend traditional replication techniques to include the network switches. For example, running a consensus protocol involving the switches for every event would be prohibitively expensive, given the demand for high-speed packet processing in switches. On the other hand, using distributed storage

	Total Event Ordering	Exactly-Once Events	Exactly-Once Commands	Transparency
Consistent Reliable Storage	✓	✗	✗	✗
Switch Broadcast	✗	✓	✗	✓
Replicated State Machines	✓	✗	✗	✓
Ravana	✓	✓	✓	✓

Table 1: Comparing different solutions for fault-tolerant controllers

to replicate controller state alone (for performance reasons) does not capture the switch state precisely. Therefore, after a controller crash, the new master may not know where to resume reconfiguring switch state. Simply reading the switch forwarding state would not provide enough information about all the commands sent by the old master (*e.g.*, PacketOuts, StatRequests).

In addition, while the system could roll back the controller state, the switches cannot easily "roll back" to a safe checkpoint. After all, what does it mean to rollback a packet that was already sent? The alternative is for the new master to simply repeat commands, but these commands are not necessarily idempotent (per §2). Since an event from one switch can trigger commands to other switches, simultaneous failure of the master controller and a switch can cause inconsistency in the rest of the network. We believe that these issues can lead to erratic behavior in existing SDN platforms.

Ravana. In this paper, we present Ravana, an SDN controller platform that offers the abstraction of a fault-free centralized controller to control applications. Instead of just keeping the controller state consistent, we handle the *entire event-processing cycle* (including event delivery from switches, event processing on controllers, and command execution on switches) as a *transaction*—either all or none of the components of this transaction are executed. Ravana ensures that transactions are totally ordered across replicas and executed exactly once across the entire system. This enables Ravana to correctly handle switch state, without resorting to rollbacks or repeated execution of commands.

Ravana adopts replicated state machines for control state replication and adds mechanisms for ensuring the consistency of switch state. Ravana uses a two-stage replication protocol across the controllers. The master replica decides the total order in which input events are received in the first stage and then indicates which events were processed in the second stage. On failover, the new master resumes transactions for "unprocessed" events from a shared log. The two stages isolate the effects of a switch failure on the execution of a transaction on other switches—a setting unique to SDN. Instead of involving all switches in a consensus protocol, Ravana extends the OpenFlow interface with techniques like explicit acknowledgment, retransmission, and filtering from traditional RPC protocols to ensure that any event transaction is executed *exactly once* on the switches.

While the various techniques we adopt are well known in distributed systems literature, ours is the first system that applies these techniques comprehensively in the SDN setting to design a correct, faul-tolerant controller. We also describe the safety and liveness guarantees provided by the Ravana protocol and argue that it ensures *observational indistinguishability* (per §4) between an ideal central controller and a replicated controller platform. Our prototype implementation allows unmodified control applications, written for a single controller, to run in a replicated and fault-tolerant manner. Our prototype achieves these properties with low overhead on controller throughput, latency, and failover time.

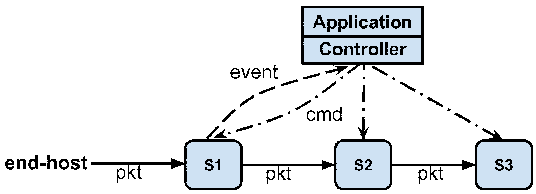


Figure 1: SDN system model

Contributions. Our fault-tolerant controller system makes the following technical contributions:

- We propose a two-phase replication protocol that extends replicated state machines to handle consistency of *external* switch state under controller failures.
- We propose extensions to the OpenFlow interface that are necessary to handle controller failures like RPC-level ACKs, retransmission of un-ACKed events and filtering of duplicate commands.
- We precisely define correctness properties of a logically centralized controller and argue that the Ravana protocol provides these guarantees.
- We present a prototype of a transparent Ravana runtime and demonstrate our solution has low overhead.

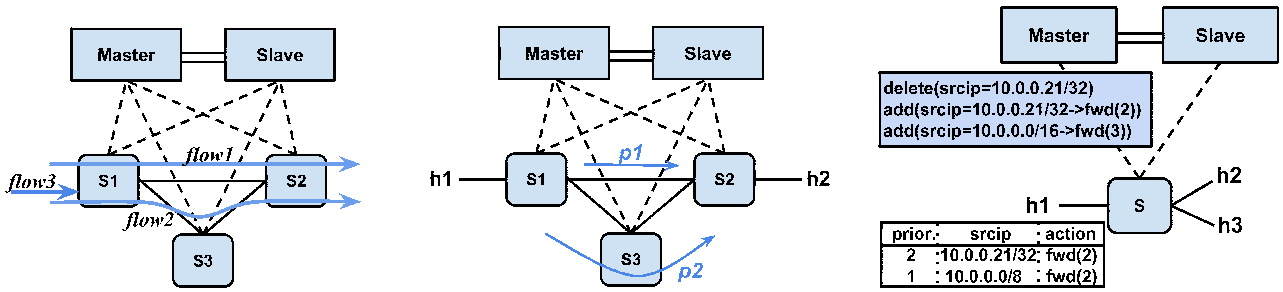
2 Controller Failures in SDN

Figure 1 shows the normal execution of an SDN in the absence of controller failures. Recent versions of OpenFlow [5], the widely used control channel protocol between controllers and switches, have some limited mechanisms for supporting multiple controllers. In particular, a controller can register with a switch in the role of *master* or *slave*, which defines the types of events and commands exchanged between the controller and the switch. A switch sends all events to the master and executes all commands received from the master, while it sends only a limited set of events (for example, `switch_features`) to slaves and does not accept commands from them.

Combining existing OpenFlow protocols with traditional techniques for replicating controllers does not ensure correct network behavior, however. This section illustrates the reasons with concrete experiments. The results are summarized in Table 1.

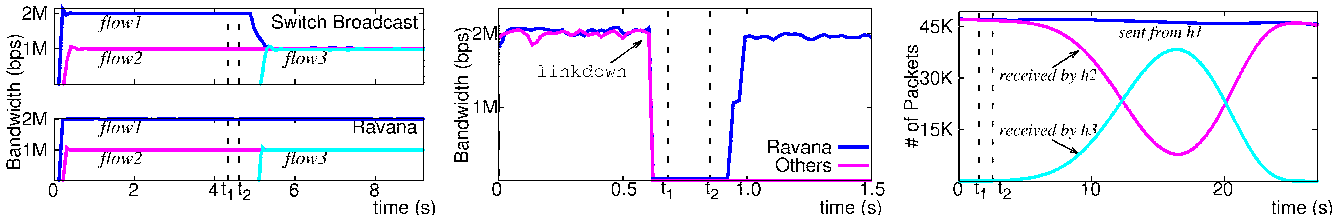
2.1 Inconsistent Event Ordering

OpenFlow 1.3 allows switches to connect to multiple controllers. If we directly use the protocol to have switches broadcast their events to every controller replica independently, each replica builds application state based on the stream of events it receives. Aside from



(a) Total Event Ordering (b) Exactly-Once Event Delivery (c) Exactly-Once Command Execution

Figure 2: Examples demonstrating different correctness properties maintained by Ravana



(a) Bandwidth Allocation (b) `linkdown` Under Failures (c) Routing Under Repeated Commands

Figure 3: Experiments for the three examples shown in Figure 2. t_1 and t_2 indicate the time when the old master crashes and when the new master is elected, respectively. In (c), delivery of commands is slowed down to measure the traffic leakage effect.

the additional overhead this places on switches, controller replicas would have an inconsistent ordering of events from different switches. This can lead to incorrect packet-processing decisions, as illustrated in the following example:

Experiment 1: In Figure 2a, consider a controller application that allocates incoming flow requests to paths in order. There are two disjoint paths in the network; each has a bandwidth of 2Mbps. Assume that two flows, with a demand of 2Mbps and 1Mbps respectively, arrive at the controller replicas in different order (due to network latencies). If the replicas assign paths to flows in the order they arrive, each replica will end up with 1Mbps free bandwidth but on different paths. Now, consider that the master crashes and the slave becomes the new master. If a new flow with 1Mbps arrives, the new master assigns the flow to the path which it thinks has 1Mbps free. But this congests an already fully utilized path, as the new master's view of the network diverged from its actual state (as dictated by the old master).

Figure 3a compares the measured flow bandwidths for the switch-broadcast and Ravana solutions. Ravana keeps consistent state in controller replicas, and the new master can install the flows in an optimal manner. Drawing a lesson from this experiment, a fault-tolerant control platform should offer the following design goal:

Total Event Ordering: Controller replicas should process events in the same order and subsequently all controller application instances should reach the same internal state.

Note that while in this specific example, the newly elected master can try to query the flow state from switches after failure; in general, simply reading switch state is not enough to infer sophisticated application state. This also defeats the argument for transparency because the programmer has to explicitly define how application state is related to switch state under failures. Also, information about PacketOuts and events lost during failures cannot be inferred by simply querying switch state.

2.2 Unreliable Event Delivery

Two existing approaches can ensure a consistent ordering of events in replicated controllers: (i) The master can store shared application state in an *external* consistent storage system (e.g., as in Onix and ONOS), or (ii) the controller's *internal* state can be kept consistent via replicated state machine (RSM) protocols. However, the former approach may fail to persist the controller state when the master fails during the event processing, and the latter approach may fail to log an event when the master fails right after receiving it. These scenarios may cause serious problems.

Experiment 2: Consider a controller program that runs a shortest-path routing algorithm, as shown in Figure 2b. Assume the master installed a flow on path $p1$, and after a while the link between $s1$ and $s2$ fails. The incident switches send a `linkdown` event to the master. Suppose the master crashes before replicating this event. If the controller replicas are using a traditional RSM protocol with unmodified OpenFlow switches, the event is lost and will never be seen by the slave. Upon becoming the new master, the slave will have an inconsistent view of the network, and cannot promptly update the switches to reroute packets around the failed link.

Figure 3b compares the measured bandwidth for the flow $h1 \rightarrow h2$ with an unmodified OpenFlow switch and with Ravana. With an unmodified switch, the controller loses the link failure event which leads to throughput loss, and it is sustained even after the new master is elected. In contrast, with Ravana, events are reliably delivered to all replicas even during failures, ensuring that the new master switches to the alternate path, as shown by the blue curve. From this experiment, we see that it is important to ensure reliable

Property	Description	Mechanism
At least once events	Switch events are not lost	Buffering and retransmission of switch events
At most once events	No event is processed more than once	Event IDs and filtering in the log
Total event order	Replicas process events in same order	Master serializes events to a shared log
Replicated control state	Replicas build same internal state	Two-stage replication and deterministic replay of event log
At least once commands	Controller commands are not lost	RPC acknowledgments from switches
At most once commands	Commands are not executed repeatedly	Command IDs and filtering at switches

Table 2: Ravana design goals and mechanisms

event delivery. Similarly, event repetition will also lead to inconsistent network views, which can further result in erroneous network behaviors. This leads to our second design goal:

Exactly-Once Event Processing: All the events are processed, and are neither lost nor processed repeatedly.

2.3 Repetition of Commands

With traditional RSM or consistent storage approaches, a newly elected master may send repeated commands to the switches because the old master sent some commands but crashed before telling the slaves about its progress. As a result, these approaches cannot guarantee that commands are executed exactly once, leading to serious problems when commands are not idempotent.

Experiment 3: Consider a controller application that installs rules with overlapping patterns. The rule that a packet matches depends on the presence or absence of other higher-priority rules. As shown in Figure 2c, the switch starts with a forwarding table with two rules that both match on the source address and forward packets to host $h2$. Suppose host $h1$ has address 10.0.0.21, which matches the first rule. Now assume that the master sends a set of three commands to the switch to redirect traffic from the /16 subnet to $h3$. After these commands, the rule table becomes the following:

```
3  10.0.0.21/32  fwd(2)
2  10.0.0.0/16   fwd(3)
1  10.0.0.0/8    fwd(2)
```

If the master crashes before replicating the information about commands it already issued, the new master would repeat these commands. When that happens, the switch first removes the first rule in the new table. Before the switch executes the second command, traffic sent by $h1$ can match the rule for `10.0.0.0/16` and be forwarded erroneously to $h3$. If there is no controller failure and the set of commands are executed exactly once, $h3$ would never have received traffic from $h1$; thus, in the failure case, the correctness property is violated. The duration of this erratic behavior may be large owing to the slow rule-installation times on switches. Leaking traffic to an unexpected receiver $h3$ could lead to security or privacy problems.

Figure 3c shows the traffic received by $h2$ and $h3$ when sending traffic from $h1$ at a constant rate. When commands are repeated by the new master, $h3$ starts receiving packets from $h1$. No traffic leakage occurs under Ravana. While missing commands will obviously cause trouble in the network, from this experiment we see that command repetition can also lead to unexpected behaviors. As a result, a correct protocol must meet the third design goal:

Exactly-Once Execution of Commands: Any given series of commands are executed once and only once on the switches.

2.4 Handling Switch Failures

Unlike traditional client-server models where the server processes a client request and sends a reply to the same client, the event-processing cycle is more complex in the SDN context: when a switch sends an event, the controller may respond by issuing *multiple* commands to *other* switches. As a result, we need additional mechanisms when adapting replication protocols to build fault-tolerant control platforms.

Suppose that an event generated at a switch is received at the master controller. Existing fault-tolerant controller platforms take one of two possible approaches for replication. First, the master replicates the event to other replicas immediately, leaving the slave replicas unsure whether the event is completely processed by the master. In fact, when an old master fails, the new master may not know whether the commands triggered by past events have been executed on the switches. The second alternative is that the master might choose to replicate an event only after it is completely processed (*i.e.*, all commands for the event are executed on the switches). However, if the original switch *and* later the master fail while the master is processing the event, some of the commands triggered by the event may have been executed on several switches, but the new master would never see the original event (because of the failed switch) and would not know about the affected switches. The situation could be worse if the old master left these switches in some transitional state before failing. Therefore, it is necessary to take care of these cases if one were to ensure a consistent switch state under failures.

In conclusion, the examples show that a correct protocol should meet all the aforementioned design goals. We further summarize the desired properties and the corresponding mechanisms to achieve them in Table 2.

3 Ravana Protocol

Ravana Approach: Ravana makes two main contributions. First, Ravana has a novel two-phase replication protocol that extends replicated state machines to deal with switch state consistency. Each phase involves adding event-processing information to a replicated in-memory log (built using traditional RSM mechanisms like viewstamped replication [6]). The first stage ensures that every received event is reliably replicated, and the second stage conveys whether the event-processing transaction has completed. When the master fails, another replica can use this information to continue processing events where the old master left off. Since events from a switch can trigger commands to multiple other switches, separating the two stages (event reception and event completion) ensures that the failure of a switch along with the master does not corrupt the state on other switches.

Second, Ravana extends the existing control channel interface between controllers and switches (the OpenFlow protocol) with mechanisms that mitigate missing or repeated control messages during controller failures. In particular, (i) to ensure that messages are delivered *at least once* under failures, Ravana uses RPC-level acknowledgments and retransmission mechanisms and (ii) to guarantee *at most once* messages, Ravana associates messages with unique IDs, and performs receive-side filtering.

Thus, our protocol adopts well known distributed systems techniques as shown in Table 2 but combines them in a unique way to maintain consistency of both the controller and switch state under failures. To our knowledge, this enables Ravana to provide the first fault-tolerant SDN controller platform with concrete correctness properties. Also, our protocol employs novel optimizations to execute commands belonging to multiple events in parallel to decrease overhead, without compromising correctness. In addition, Ravana provides a transparent programming platform—unmodified control applications written for a single controller can be made automatically fault-tolerant without the programmer having to worry about replica failures, as discussed in Section 6.

Ravana has two main components—(i) a controller runtime for each controller replica and (ii) a switch runtime for each switch. These components together make sure that the SDN is fault-tolerant if at most f of the $2f+1$ controller replicas crash. This is a direct result of the fact that each phase of the controller replication protocol in turn uses Viewstamped Replication [6]. Note that we only handle crash-stop failures of controller replicas and do not focus on recovery of failed nodes. Similarly we assume that when a failed switch recovers, it starts afresh on a clean slate and is analogous to a new switch joining the network. In this section, we describe the steps for processing events in our protocol, and further discuss how the two runtime components function together to achieve our design goals.

3.1 Protocol Overview

To illustrate the operation of a protocol, we present an example of handling a specific event—a packet-in event. A packet arriving at a switch is processed in several steps, as shown in Figure 4. First, we discuss the handling of packets during normal execution without controller failures:

1. A switch receives a packet and after processing the packet, it may direct the packet to other switches.
2. If processing the packet triggers an event, the switch runtime buffers the event temporarily, and sends a copy to the master controller runtime.
3. The master runtime stores the event in a replicated in-memory log that imposes a total order on the logged events. The slave runtimes do not yet release the event to their application instances for processing.
4. After replicating the event into the log, the master acknowledges the switch. This implies that the buffered event has been reliably received by the controllers, so the switch can safely delete it.
5. The master feeds the replicated events in the log order to the controller application, where they get processed. The application updates the necessary internal state and responds with zero or more commands.
6. The master runtime sends these commands out to the corresponding switches, and waits to receive acknowledgments for the commands sent, before informing the replicas that the event is processed.
7. The switch runtimes buffer the received commands, and send acknowledgment messages back to the master controller. The switches apply the commands subsequently.
8. After all the commands are acknowledged, the master puts an *event-processed* message into the log.

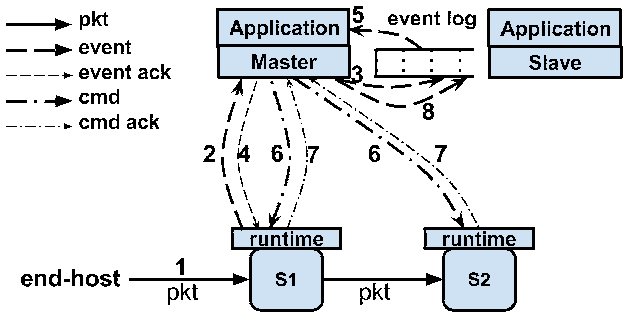


Figure 4: Steps for processing a packet in Ravana.

A slave runtime does not feed an event to its application instance until *after* the event-processed message is logged. The slave runtime delivers events to the application in order, waiting until each event in the log has a corresponding event-processed message before proceeding. The slave runtimes also filter the outgoing commands from their application instances, rather than actually sending these commands to switches; that is, the slaves merely simulate the processing of events to update the internal application state.

When the master controller fails, a standby slave controller will replace it following these steps:

1. A leader election component running on all the slaves elects one of them to be the new master.
2. The new master finishes processing any logged events that have their *event-processed* messages logged. These events are processed in slave mode to bring its application state up-to-date without sending any commands.
3. The new master sends *role request* messages to register with the switches in the role of the new master. All switches send a *role response* message as acknowledgment and then begin sending previously buffered events to the new master.
4. The new master starts to receive events from the switches, and processes events (including events logged by the old master without a corresponding *event processed* message), in master mode.

3.2 Protocol Insights

The Ravana protocol can be viewed as a combination of mechanisms that achieve the design goals set in the previous section. By exploring the full range of controller crash scenarios (cases (i) to (vii) in Figure 5), we describe the key insights behind the protocol mechanisms.

Exactly-Once Event Processing: A combination of temporary event buffering on the switches and explicit acknowledgment from the controller ensures *at-least* once delivery of events. When

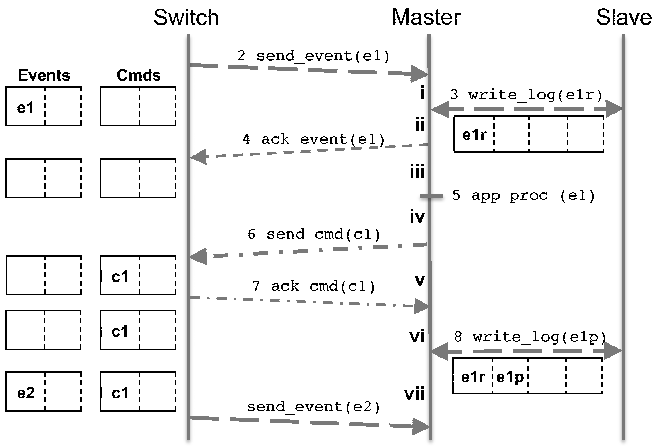


Figure 5: Sequence diagram of event processing in controllers: steps 2–8 are in accordance with in Figure 4.

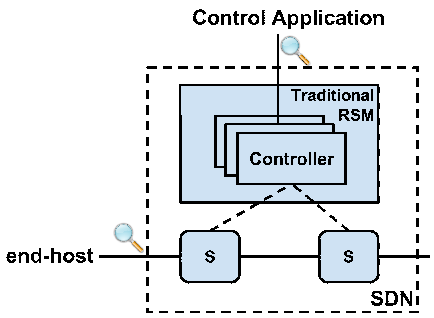


Figure 6: In SDN, control applications and end hosts both observe system evolution, while traditional replication techniques treat the switches (S) as observers.

sending an event $e1$ to the master, the switch runtime temporarily stores the event in a local event buffer (Note that this is different from the notion of buffering PacketIn payloads in OpenFlow switches). If the master crashes before replicating this event in the shared log (case (i) in Figure 5), the failover mechanism ensures that the switch runtime resends the buffered event to the new master. Thus the events are delivered at least once to all of the replicas. To suppress repeated events, the replicas keep track of the IDs of the logged events. If the master crashes after the event is replicated in the log but before sending an acknowledgment (case (ii)), the switch retransmits the event to the new master controller. The new controller's runtime recognizes the duplicate eventID and filters the event. Together, these two mechanisms ensure *exactly once* processing of events at all of the replicas.

Total Event Ordering: A shared log across the controller replicas (implemented using viewstamped replication) ensures that the events received at the master are replicated in a consistent (linearized) order. Even if the old master fails (cases (iii) and (iv)), the new master preserves that order and only adds new events to the log. In addition, the controller runtime ensures exact replication of control program state by propagating information about non-deterministic primitives like timers as special events in the replicated log.

Exactly-Once Command Execution: The switches explicitly acknowledge the commands to ensure *at-least* once delivery. This way the controller runtime does not mistakenly log the event-processed message (thinking the command was received by the switch), when it is still sitting in the controller runtime's network stack (case (iv)). Similarly, if the command is indeed received by the switch but the master crashes before writing the event-processed message into the log (cases (v) and (vi)), the new master processes the event $e1$ and sends the command $c1$ again to the switch. At this time, the switch runtime filters repeated commands by looking up the local command buffer. This ensures *at-most* once execution of commands. Together these mechanisms ensure *exactly-once* execution of commands.

Consistency Under Joint Switch and Controller Failure: The Ravana protocol relies on switches retransmitting events and acknowledging commands. Therefore, the protocol must be aware of switch failure to ensure that faulty switches do not break the Ravana protocol. If there is no controller failure, the master controller treats a switch failure the same way a single controller system would treat such a failure – it relays the network port status updates to the controller application which will route the traffic around the failed switch. Note that when a switch fails, the controller does not fail the entire transaction. Since this is a plausible scenario in the fault-free case, the runtime completes the transaction by executing commands on the set of available switches. Specifically, the controller runtime has timeout mechanisms that ensure a transaction is not stuck because of commands not being acknowledged by a failed switch. However, the Ravana protocol needs to carefully handle the case where a switch failure occurs along with a controller failure because it relies on the switch to retransmit lost events under controller failures.

Suppose the master and a switch fail sometime *after* the master receives the event from that switch but *before* the transaction completes. Ravana must ensure that the new master sees the event, so the new master can update its internal application state and issue any remaining commands to the rest of the switches. However, in this case, since the failed switch is no longer available to retransmit the event, unless the old master reliably logged the event *before* issuing any commands, the new master could not take over correctly. This is the reason why the Ravana protocol involves *two* stages of replication. The first stage captures the fact that event e is received by the master. The second stage captures the fact that the master has completely processed e, which is important to know during failures to ensure the exactly-once semantics. Thus the event-transaction dependencies across switches, a property unique to SDN, leads to this two-stage replication protocol.

4 Correctness

While the protocol described in the previous section intuitively gives us necessary guarantees for processing of events and execution of commands during controller failures, it is not clear if they are sufficient to ensure the abstraction of a logically centralized controller. This is also the question that recent work in this space has left unanswered. This led to a lot of subtle bugs in their approaches that have erroneous effect on the network state as illustrated in section 2.

Thus, we strongly believe it is important to concretely define what it means to have a logically centralized controller and then analyze whether the proposed solution does indeed guarantee such an abstraction. Ideally, a fault-tolerant SDN should behave the same way as a fault-free SDN from the viewpoint of all the users of the system.

Observational indistinguishability in SDN: We believe the correctness of a fault-tolerant SDN relies on the users—the end-host and controller applications—seeing a system that always behaves like there is a single, reliable controller, as shown in Figure 6. This is what it means to be a logically centralized controller. Of course, controller failures could affect performance, in the form of additional delays, packet drops, or the timing and ordering of future events. But, these kinds of variations can occur even in a fault-free setting. Instead, our goal is that the fault-tolerant system evolves in a way that *could have happened* in a fault-free execution, using observational indistinguishability [7], a common paradigm for comparing behavior of computer programs:

Definition of observational indistinguishability: If the trace of observations made by users in the fault-tolerant system is a possible trace in the fault-free system, then the fault-tolerant system is observationally indistinguishable from a fault-free system.

An observation describes the interaction between an application and an SDN component. Typically, SDN exposes *two* kinds of observations to its users: (i) end hosts observe requests and responses (and use them to evolve their own application state) and (ii) control applications observe events from switches (and use them to adapt the system to obey a high-level service policy, such as load-balancing requests over multiple switches). For example, as illustrated in section 2, under controller failures, while controllers fail to observe network failure events, end-hosts observe a drop in packet throughput compared to what is expected or they observe packets not intended for them.

Commands decide observational indistinguishability: The observations of *both* kinds of users (Figure 6) are preserved in a fault-tolerant SDN if the series of *commands* executed on the switches are executed just as they could have been executed in the fault-free system. The reason is that the commands from a controller can (i) modify the switch (packet processing) logic and (ii) query the switch state. Thus the commands executed on a switch determine not only what responses an end host receives, but also what events the control application sees. Hence, we can achieve observational indistinguishability by ensuring "command trace indistinguishability". This leads to the following correctness criteria for a fault-tolerant protocol:

Safety: For any given series of switch events, the resulting series of commands executed on the switches in the fault-tolerant system *could have been* executed in the fault-free system.

Liveness: Every event sent by a switch is eventually processed by the controller application, and every resulting command sent from the controller application is eventually executed on its corresponding switch.

Transactional and exactly-once event cycle: To ensure the above safety and liveness properties of observational indistinguishability, we need to guarantee that the controller replicas output a series of commands "indistinguishable" from that of a fault-free controller for any given set of input events. Hence, we must ensure that the same input is processed by all the replicas and that no input is missing because of failures. Also, the replicas should process all input events in the same order, and the commands issued should be neither missing nor repeated in the event of replica failure.

In other words, Ravana provides transactional semantics to the entire "control loop" of (i) event delivery, (ii) event ordering, (iii) event processing, and (iv) command execution. (If the command execution results in more events, the subsequent event-processing cycles are considered separate transactions.) In addition, we ensure that any given transaction happens exactly once—it is not aborted or rolled back under controller failures. That is, once an event is sent by a switch, the entire event-processing cycle is executed till completion, and the transaction affects the network state exactly once. Therefore, our protocol that is designed around the goals listed in Table 2 will ensure observational indistinguishability between an ideal fault-free controller and a logically centralized but physically replicated controller. While we provide an informal argument for correctness, modeling the Ravana protocol using a formal specification tool and proving formally that the protocol is indeed sufficient to guarantee the safety and liveness properties is out of scope for this paper and is considered part of future work.

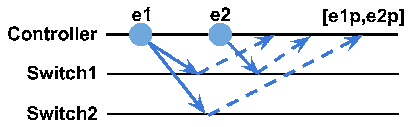


Figure 7: Optimizing performance by processing multiple transactions in parallel. The controller processes events $e1$ and $e2$, and the command for $e2$ is acknowledged before both the commands for $e1$ are acknowledged.

5 Performance Optimizations

In this section, we discuss several approaches that can optimize the performance of the protocol while retaining its strong correctness guarantees.

Parallel logging of events: Ravana protocol enforces a consistent ordering of all events among the controller replicas. This is easy if the master were to replicate the events one after the other sequentially but this approach is too slow when logging tens of thousands of events. Hence, the Ravana runtime first imposes a total order on the switch events by giving them monotonically increasing log IDs and then does parallel logging of events where multiple threads write switch events to the log in parallel. After an event is reliably logged, the master runtime feeds the event to its application instance, but it still follows the total order. The slaves infer the total order from the log IDs assigned to the replicated events by the master.

Processing multiple transactions in parallel: In Ravana, one way to maintain consistency between controller and switch state is to send the commands for each event transaction one after the other (and waiting for switches' acknowledgments) before replicating the event processed message to the replicas. Since this approach can be too slow, we can optimize the performance by pipelining multiple commands in parallel without waiting for the ACKs. The runtime also interleaves commands generated from *multiple* independent event transactions. An internal data structure maps the outstanding commands to events and traces the progress of processing events. Figure 7 shows an example of sending commands for two events in parallel. In this example, the controller runtime sends the commands resulting from processing $e2$ while the commands from processing $e1$ are still outstanding.

Sending commands in parallel does not break the ordering of event processing. For example, the commands from the controller to any given individual switch (the commands for $e1$) are ordered by the reliable control-plane channel (e.g., via TCP). Thus at a given switch, the sequence of commands received from the controller must be consistent with the order of events processed by the controller. For multiple transactions in parallel, the runtime buffers

the completed events till the events earlier in the total order are also completed. For example, even though the commands for $e2$ are acknowledged first, the runtime waits till all the commands for $e1$ are acknowledged and then replicates the event processed messages for both $e1$ and $e2$ in that order. Despite this optimization, since the event processed messages are written in log order, we make sure that the slaves also process them in the same order.

Clearing switch buffers: The switch runtime maintains both an event buffer (`EBuf`) and a command buffer (`CBuf`). We add *buffer clear* messages that help garbage collect these buffers. As soon as the event is durably replicated in the distributed log, the master controller sends an `EBuf_CLEAR` message to confirm that the event is persistent. However, a `CBuf_CLEAR` is sent only when its corresponding event is done processing. An event processed message is logged only when all processing is done in the current protocol, so a slave controller gets to know that all the commands associated with the event are received by switches, and it should never send the commands out again when it becomes a master. As a result, when an event is logged, the controller sends an event acknowledgment, and at the same time piggybacks both `EBuf_CLEAR` and `CBuf_CLEAR`.

6 Implementation of Ravana

Implementing Ravana in SDN involves changing three important components: (i) instead of controller applications grappling with controller failures, a *controller runtime* handles them transparently, (ii) a *switch runtime* replays events under controller failures and filters repeated commands, and (iii) a modified *control channel* supports additional message types for event-processing transactions.

6.1 Controller Runtime: Failover, Replication

Each replica has a runtime component which handles the controller failure logic transparent to the application. The same application program runs on all of the replicas. Our prototype controller runtime uses the Ryu [8] message-parsing library to transform the raw messages on the wire into corresponding OpenFlow messages.

Leader election: The controllers elect one of them as master using a leader election component written using ZooKeeper [9], a synchronization service that exposes an atomic broadcast protocol. Much like in Google's use of Chubby [10], Ravana leader election involves the replicas contending for a ZooKeeper lock; whoever successfully gains the lock becomes the master. Master failure is detected using the ZooKeeper failure-detection service which relies on counting missed heartbeat messages. A new master is elected by having the current slaves retry gaining the master lock.

Event logging: The master saves each event in ZooKeeper's distributed in-memory log. Slaves monitor the log by registering a trigger for it. When a new event is propagated to a slave's log, the trigger is activated so that the slave can read the newly arrived event locally.

Event batching: Even though its in-memory design makes the distributed log efficient, latency during event replication can still degrade throughput under high load. In particular, the master's write call returns only after it is propagated to more than half of all replicas. To reduce this overhead, we batch multiple messages into an ordered group and write the grouped event as a whole to the log. On the other side, a slave unpacks the grouped events and processes them individually and in order.

6.2 Switch Runtime: Event/Command Buffers

We implement our switch runtime by modifying the Open vSwitch (version 1.10) [11], which is the most widely used software OpenFlow switch. We implement the event and command buffers as additional data structures in the OVS connection manager. If a master fails, the connection manager sends events buffered in `EBuf` to the new master as soon as it registers its new role. The command buffer `CBuf` is used by the switch processing loop to check whether a command received (uniquely identified by its transaction ID) has already been executed. These transaction IDs are remembered till they can be safely garbage collected by the corresponding `CBuf_CLEAR` message from the controller.

6.3 Control Channel Interface: Transactions

Changes to OpenFlow: We modified the OpenFlow 1.3 controller-switch interface to enable the two parties to exchange additional Ravana-specific metadata: `EVENT_ACK`, `CMD_ACK`, `EBuf_CLEAR`, and `CBuf_CLEAR`. The ACK messages acknowledge the receipt of events and commands, while CLEAR help reduce the memory footprint of the two switch buffers by periodically cleaning them. As in OpenFlow, all messages carry a transaction ID to specify the event or command to which it should be applied.

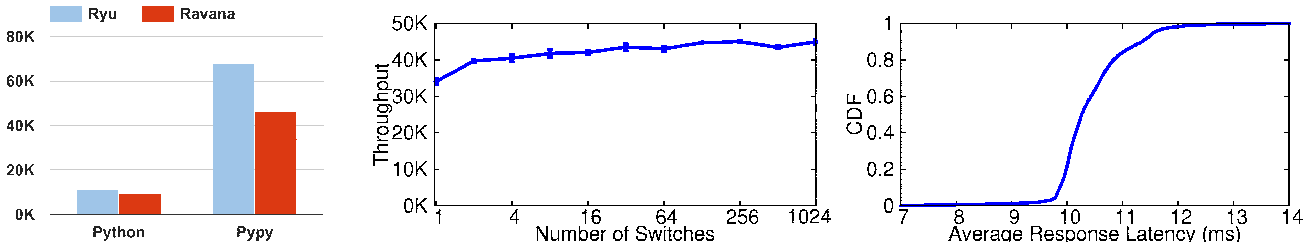
Unique transaction IDs: The controller runtime associates every command with a unique transaction ID (XID). The XIDs are monotonically increasing and identical across all replicas, so that duplicate commands can be identified. This arises from the controllers' deterministic ordered operations and does not require an additional agreement protocol. In addition, the switch also needs to ensure that unique XIDs are assigned to events sent to the controller. We modified Open vSwitch to increment the XID field whenever a new event is sent to the controller. Thus, we use 32-bit unique XIDs (with wrap around) for both events and commands.

6.4 Transparent Programming Abstraction

Ravana provides a fault-tolerant controller runtime that is completely transparent to control applications. The Ravana runtime intercepts all switch events destined to the Ryu application, enforces a total order on them, stores them in a distributed in-memory log, and only then delivers them to the application. The application updates the controller internal state, and generates one or more commands for each event. Ravana also intercepts the outgoing commands — it keeps track of the set of commands generated for each event in order to trace the progress of processing each event. After that, the commands are delivered to the corresponding switches. Since Ravana does all this from inside Ryu, existing single-threaded Ryu applications can directly run on Ravana without modifying a single line of code.

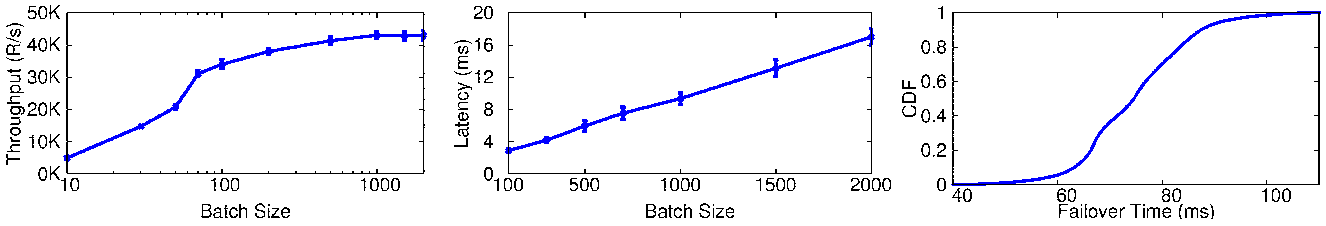
To demonstrate the transparency of programming abstraction, we have tested a variety of Ryu applications [12]: a MAC learning switch, a simple traffic monitor, a MAC table management app, a link aggregation (LAG) app, and a spanning tree app. These applications are written using the Ryu API, and they run on our fault-tolerant control platform without any changes.

Currently we expect programmers to write controller applications that are single-threaded and deterministic, similar to most replicated state machine systems available today. An application can introduce nondeterminism by using timers and random numbers. Our prototype supports timers and random numbers through a standard library interface. The master runtime treats function calls through this interface as special events and persists the event meta-



(a) Ravana Throughput Overhead (b) Throughput with Different Number of Switches (c) CDF for cbench Latency

Figure 8: Ravana Event-Processing Throughput and Latency



(a) Event-Processing Throughput with Batching (b) Event-Processing Latency with Batching (c) CDF for Ravana Failover Time

Figure 9: Variance of Ravana Throughput, Latency and Failover Time

data (timer begin/end, random seeds, etc.) into the log. The slave runtimes extract this information from the log so their application instances execute the same way as the master's. State-machine replication with multi-threaded programming has been studied [13], and supporting it in Ravana is future work.

7 Performance Evaluation

To understand Ravana's performance, we evaluate our prototype to answer the following questions:

- What is the overhead of Ravana's fault-tolerant runtime on event-processing throughput?
- What is the effect of the various optimizations on Ravana's event-processing throughput and latency?
- Can Ravana respond quickly to controller failure?
- What are the throughput and latency trade-offs for various correctness guarantees?

We run experiments on three machines connected by 1Gbps links. Each machine has 12GB memory and an Intel Xeon 2.4GHz CPU. We use ZooKeeper 3.4.6 for event logging and leader election. We use the Ryu 3.8 controller platform as our non-fault-tolerant baseline.

7.1 Measuring Throughput and Latency

We first compare the throughput (in terms of flow responses per second) achieved by the vanilla Ryu controller and the Ravana prototype we implemented on top of Ryu, in order to characterize Ravana's overhead. Measurements are done using the `cbench` [14] performance test suite: the test program spawns a number of processes that act as OpenFlow switches. In `cbench`'s throughput mode, the processes send PacketIn events to the controller as fast as possible. Upon receiving a PacketIn event from a switch, the controller sends a command with a forwarding decision for this packet. The controller application is designed to be simple enough to give responses without much computation, so that the experiment can effectively benchmark the Ravana protocol stack.

Figure 8a shows the event-processing throughput of the vanilla Ryu controller and our prototype in a fault-free execution. We used both the standard Python interpreter and PyPy (version 2.2.1), a fast Just-in-Time interpreter for Python. We enable batching with a buffer of 1000 events and 0.1s buffer time limit. Using standard Python, the Ryu controller achieves a throughput of 11.0K responses per second (rps), while the Ravana controller achieves 9.2K, with an overhead of 16.4%. With PyPy, the event-processing throughput of Ryu and Ravana are 67.6K rps and 46.4K rps, respectively, with an overhead of 31.4%. This overhead includes the time of serializing and propagating all the events among the three controller replicas in a failure-free execution. We consider this as a reasonable overhead given the correctness guarantee and replication mechanisms Ravana added.

To evaluate the runtime's scalability with increasing number of switch connections, we ran `cbench` in throughput mode with a large number of simulated switch connections. A series of throughput measurements are shown in Figure 8b. Since the simulated switches send events at the highest possible rate, as the number of switches become reasonably large, the controller processing rate saturates but does not go down. The event-processing throughput remains high even when we connect over one thousand simulated switches. The result shows that the controller runtime can manage a large number of parallel switch connections efficiently.

Figure 8c shows the latency CDF of our system when tested with `cbench`. In this experiment, we run `cbench` and the master controller on the same machine, in order to benchmark the Ravana event-processing time without introducing extra network latency. The latency distribution is drawn using the average latency calcu-

lated by `cbench` over 100 runs. The figure shows that most of the events can be processed within 12ms.

7.2 Sensitivity Analysis for Event Batching

The Ravana controller runtime batches events to reduce the overhead for writing several events in the ZooKeeper event log. Network operators need to tune the *batching size* parameter to achieve the best performance. A batch of events are flushed to the replicated log either when the batch reaches the size limit or when no event arrives within a certain time limit.

Figure 9a shows the effect of batching sizes on event processing throughput measured with `cbench`. As batching size increases, throughput increases due to reduction in the number of RPC calls needed to replicate events. However, when batching size increases beyond a certain number, the throughput saturates because the performance is bounded by other system components (marshalling and unmarshalling OpenFlow messages, event processing functions, etc.)

While increasing batching size can improve throughput under high demand, it also increases event response latency. Figure 9b shows the effect of varying batch sizes on the latency overhead. The average event processing latency increases almost linearly with the batching size, due to the time spent in filling the batch before it is written to the log.

The experiment results shown in Figure 9a and 9b allow network operators to better understand how to set an appropriate batching size parameter based on different requirements. If the application needs to process a large number of events and can tolerant relatively high latency, then a large batch size is helpful; if the events need to be instantly processed and the number of events is not a big concern, then a small batching size will be more appropriate.

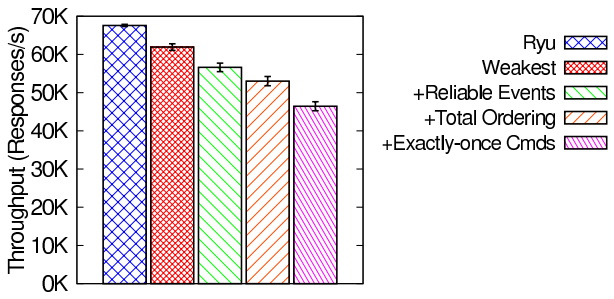
7.3 Measuring Failover Time

When the master controller crashes, it takes some time for the new controller to take over. To evaluate the efficiency of Ravana controller failover mechanism, we conducted a series of tests to measure the failover time distribution, as shown in Figure 9c. In this experiment, a software switch connects two hosts which continuously exchange packets that are processed by the controller in the middle. We bring down the master. The end hosts measure the time for which no traffic is received during the failover period. The result shows that the average failover time is 75ms, with a standard deviation of 9ms. This includes around 40ms to detect failure and elect a new leader (with the help of ZooKeeper), around 25ms to catch up with the old master (can be reduced further with optimistic processing of the event log at the slave) and around 10ms to register the new role on the switch. The short failover time ensures that the network events generated during this period will not be delayed for a long time before getting processed by the new master.

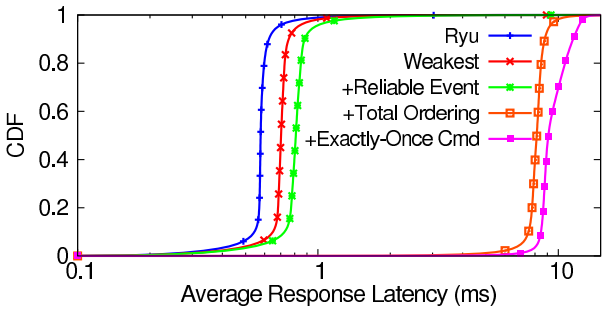
7.4 Consistency Levels: Overhead

Our design goals ensure strict correctness in terms of observational indistinguishability for general SDN policies. However, as show in Figure 10, some guarantees are costlier to ensure (in terms of through/latency) than the others. In particular, we looked at each of the three design goals that adds overhead to the system compared to the weakest guarantee.

The weakest consistency included in this study is the same as what existing fault-tolerant controller systems provide — the master immediately processes events once they arrive, and replicates them lazily in the background. Naturally this avoids the overhead



(a) Throughput Overhead for Correctness Guarantees



(b) Latency Overheads for Correctness Guarantees

Figure 10: Throughput and Latency Overheads with Varied Levels of Correctness Guarantees

of all the three design goals we aim for and hence has only a small throughput overhead of 8.4%.

The second consistency level is enabled by guaranteeing exactly-once processing of switch events received at the controller. This involves making sure that the master synchronously logs the events and explicitly sends event ACKs to corresponding switches. This has an additional throughput overhead of 7.8%.

The third consistency level ensures total event ordering in addition to exactly-once events. This makes sure that the order in which the events are written to the log is the same as the order in which the master processes them, and hence involves mechanisms to strictly synchronize the two. Ensuring this consistency incurs an additional overhead of 5.3%.

The fourth and strongest consistency level ensures exactly-once execution of controller commands. It requires the switch runtime to explicitly ACK each command from the controller and to filter repeated commands. This adds an additional overhead of 9.7% on the controller throughput. Thus adding all these mechanisms ensures the strongest collection of correctness guarantees possible under Ravana and the cumulative overhead is 31%. While some of the overheads shown above can be reduced further with implementation specific optimizations like cumulative and piggybacked message ACKs, we believe the overheads related to replicating the log and maintaining total ordering are unavoidable to guarantee protocol correctness.

Latency. Figure 10b shows the CDF for the average time it takes for the controller to send a command in response to a switch event. Our study reveals that the main contributing factor to the latency overhead is the synchronization mechanism that ties event logging to event processing. This means that the master has to wait till a switch event is properly replicated and only then processes the

event. This is why all the consistency levels that do not involve this guarantee have a latency of around 0.8ms on average but those that involve the total event ordering guarantee have a latency of 11ms on average.

Relaxed Consistency. The Ravana protocol described in this paper is oblivious to the nature of control application state or the various types of control messages processed by the application. This is what led to the design of a truly transparent runtime that works with unmodified control applications. However, given the breakdown in terms of throughput and latency overheads for various correctness guarantees, it is natural to ask if there are control applications that can benefit from relaxed consistency requirements.

For example, a Valiant load balancing application that processes flow requests (PacketIn events) from switches and assigns paths to flows randomly is essentially a stateless application. So the constraint on total event ordering can be relaxed entirely for this application. But if this application is run in conjunction with a module that also reacts to topology changes (PortStatus events), then it makes sense to enable to constraint just for the topology events and disable it for the rest of the event types. This way, both the throughput and latency of the application can be improved significantly.

A complete study of which applications benefit from relaxing which correctness constraints and how to enable programmatic control of runtime knobs is out of scope for this paper. However, from a preliminary analysis, many applications seem to benefit from either completely disabling certain correctness mechanisms or only partially disabling them for certain kinds of OpenFlow messages.

8 Related Work

Distributed SDN control with consistent reliable storage: The Onix [3] distributed controller partitions application and network state across multiple controllers using distributed storage. Switch state is stored in a strongly consistent Network Information Base (NIB). Controllers subscribe to switch events in the NIB and the NIB publishes new events to subscribed controllers independently and in an eventually consistent manner. This could violate the total event ordering correctness constraint. Since the paper is underspecified on some details, it is not clear how Onix handles simultaneous occurrence of controller crashes and network events (like link/switch failures) that can affect the commands sent to other switches. In addition, programming control applications is difficult since the applications have to be conscious of the controller fault-tolerance logic. Onix does however handle continued distributed control under network partitioning for both scalability and performance, while Ravana is concerned only with reliability. ONOS [4] is an experimental controller platform that provides a distributed, but logically centralized, global network view; scale-out; and fault tolerance by using a consistent store for replicating application state. However, owing to its similarities to Onix, it also suffers from reliability guarantees as Onix does.

Distributed SDN control with state machine replication: HyperFlow [15] is an SDN controller where network events are replicated to the controller replicas using a publish-subscribe messaging paradigm among the controllers. The controller application publishes and receives events on subscribed channels to other controllers and builds its local state solely from the network events. In this sense, the approach to building application state is similar to Ravana but the application model is non-transparent because the application bears the burden of replicating events. In addition, HyperFlow also does not deal with the correctness properties related to the switch state.

Distributed SDN with weaker ordering requirements: Early work on software-defined BGP route control [16, 17] allowed distributed controllers to make routing decisions for an Autonomous System. These works do not ensure a total ordering on events from different switches, and instead rely on the fact that the final outcome of the BGP decision process does not depend on the relative ordering of messages from different switches. This assumption does not hold for arbitrary applications.

Traditional fault-tolerance techniques: A well-known protocol for replicating state machines in client-server models for reliable service is Viewstamped Replication (VSR) [6]. VSR is not directly applicable in the context of SDN, where switch state is as important as the controller state. In particular, this leads to missing events or duplicate commands under controller failures, which can lead to incorrect switch state. Similarly, Paxos [18] and Raft [19] are distributed consensus protocols that can be used to reach a consensus on input processed by the replicas but they do not address the effects on state external to the replicas. Fault-tolerant journaling file systems [20] and database systems [21] assume that the commands are idempotent and that replicas can replay the log after failure to complete transactions. However, the commands executed on switches are not idempotent. The authors of [22] discuss strategies for ensuring exactly-once semantics in replicated messaging systems. These strategies are similar to our mechanisms for exactly-once event semantics but they cannot be adopted directly to handle cases where failure of a switch can effect the dataplane state on other switches.

TCP fault-tolerance: Another approach is to provide fault tolerance within the network stack using TCP failover techniques [23–25]. These techniques have a huge overhead because they involve reliable logging of each packet or low-level TCP segment information, in both directions. In our approach, much fewer (application-level) events are replicated to the slaves.

VM fault-tolerance: Remus [26] and Kemari [27] are techniques that provide fault-tolerant virtualization environments by using live VM migration to maintain availability. These techniques synchronize all in-memory and on-disk state across the VM replicas. The domain-agnostic checkpointing can lead to correctness issues for high-performance controllers. Thus, they impose significant overhead because of the large amount of state being synchronized.

Observational indistinguishability: Lime [28] uses a similar notion of observational indistinguishability, in the context of live switch migration (where multiple switches emulate a single virtual switch) as opposed to multiple controllers.

Statesman [29] takes the approach of allowing incorrect switch state when a master fails. Once the new master comes up, it reads the current switch state and incrementally migrates it to a target switch state determined by the controller application. LegoSDN [30] focuses on application-level fault-tolerance caused by application software bugs, as opposed to complete controller crash failures. Akella et. al. [31] tackle the problem of network availability when the control channel is in-band whereas our approach assumes a separate out-of-band control channel. The approach is also heavy-handed where every element in the network including the switches is involved in a distributed snapshot protocol. Beehive [32] describes a programming abstraction that makes writing distributed control applications for SDN easier. However, while the focus in Beehive is on controller scalability, they do not discuss consistent handling of the switch state.

9 Conclusion

Ravana is a distributed protocol for reliable control of software-defined networks. In our future research, we plan to create a formal model of our protocol and use verification tools to prove its correctness. We also want to extend Ravana to support multi-threaded control applications, richer failure models (such as Byzantine failures), and more scalable deployments where each controller manages a smaller subset of switches.

Acknowledgments.

We wish to thank the SOSR reviewers for their feedback. We would also like to thank Nick Feamster, Wyatt Lloyd and Siddhartha Sen who gave valuable comments on previous drafts of this paper. We thank Joshua Reich who contributed to initial discussions on this work. This work was supported in part by the NSF grant TS-1111520; the NSF Award CSR-0953197 (CAREER); the ONR award N00014-12-1-0757; and an Intel grant.

10 References

[1] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *SIGCOMM*, Aug. 2007.

[2] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences," in *SIGCOMM*, Aug. 2014.

[3] 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*, Oct. 2010.

[4] 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 *HotSDN*, Aug. 2014.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM CCR*, Apr. 2008.

[6] B. M. Oki and B. H. Liskov, "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," in *PODC*, Aug. 1988.

[7] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.

[8] "Ryu software-defined networking framework." See `http://osrg.github.io/ryu/`, 2014.

[9] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX ATC*, June 2010.

[10] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems," in *OSDI*, Nov. 2006.

[11] "The rise of soft switching." See `http://networkheresy.com/category/open-vswitch/`, 2011.

[12] "Ryubook 1.0 documentation." See `http://osrg.github.io/ryu-book/en/html/`, 2014.

[13] J. G. Slember and P. Narasimhan, "Static Analysis Meets Distributed Fault-tolerance: Enabling State-machine Replication with Nondeterminism," in *HotDep*, Nov. 2006.

[14] "Cbench - scalable cluster benchmarking." See `http://sourceforge.net/projects/cbench/`, 2014.

[15] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *INM/WREN*, Apr. 2010.

[16] M. Caesar, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and Implementation of a Routing Control Platform," in *NSDI*, May 2005.

[17] P. Verkaik, D. Pei, T. Scholl, A. Shaikh, A. Snoeren, and J. van der Merwe, "Wresting Control from BGP: Scalable Fine-grained Route Control," in *USENIX ATC*, June 2007.

[18] L. Lamport, "The Part-time Parliament," *ACM Trans. Comput. Syst.*, May 1998.

[19] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX ATC*, June 2014.

[20] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and Evolution of Journaling File Systems," in *USENIX ATC*, Apr. 2005.

[21] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging," *ACM Trans. Database Syst.*, Mar. 1992.

[22] Y. Huang and H. Garcia-Molina, "Exactly-once Semantics in a Replicated Messaging System," in *ICDE*, Apr. 2001.

[23] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud, "Practical and Low-overhead Masking of Failures of TCP-based Servers," *ACM Trans. Comput. Syst.*, May 2009.

[24] R. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith, "Transparent TCP Connection Failover," in *DSN*, June 2003.

[25] M. Marwah and S. Mishra, "TCP Server Fault Tolerance Using Connection Migration to a Backup Server," in *DSN*, June 2003.

[26] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *NSDI*, Apr. 2008.

[27] "Kemari." See `http://wiki.qemu.org/Features/FaultTolerance`, 2014.

[28] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker, "Transparent, Live Migration of a Software-Defined Network," in *SOCC*, Nov. 2014.

[29] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin, "A Network-state Management Service," in *SIGCOMM*, Aug. 2014.

[30] B. Chandrasekaran and T. Benson, "Tolerating SDN Application Failures with LegoSDN," in *HotNets*, Aug. 2014.

[31] A. Akella and A. Krishnamurthy, "A Highly Available Software Defined Fabric," in *HotNets*, Aug. 2014.

[32] S. H. Yeganeh and Y. Ganjali, "Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking," in *HotNets*, Aug. 2014.