Isolating and Tolerating SDN Application Failures with LegoSDN

Balakrishnan Chandrasekaran
Duke University

Brendan Tschaen
Duke University

Theophilus Benson
Duke University

ABSTRACT

Despite software-defined networking's proven benefits, there remains a significant reluctance in adopting it. Among the issues that hamper SDN's adoption, two issues stand out: reliability and fault tolerance. At the heart of these issues is a set of fate-sharing relationships: the first between the SDN control applications and controllers, wherein the crash of the former induces a crash of the latter, thereby affecting the controller's availability; and, the second between the SDN-Apps and the network, wherein the failure of the former violates network safety, e.g., network-loops, or network availability, e.g., black holes.

In this paper, we argue for a redesign of the controller architecture centering around a set of abstractions to eliminate these fate-sharing relationships and thus improve the controller's availability. We present a prototype implementation of a framework, called LegoSDN, that embodies our abstractions, and we demonstrate the benefits of our abstractions by evaluating LegoSDN on an emulated network with five real SDN-Apps. Our evaluations show that LegoSDN can recover failed SDN-Apps $3\times$ faster than controller reboots while simultaneously preventing policy violations.

1. INTRODUCTION

Software-Defined Networking (SDN) decouples the data-plane from the control-plane and provides an open API for programmatic control over the network. A key appeal of SDN lies in its ability to enable innovative control applications (SDN-Apps). These SDN-Apps contain sophisticated code to interface with increasingly complicated controller code-bases and to interoperate with complex and, often, buggy switches [23]. The end-result of these complexities is that SDN-Apps are prone to a variety of bugs (e.g., timing bugs [6, 29], or null pointers). The SDN-Apps, furthermore, are likely to be provided with limited testing by third party entities—a trend that is expected to become more prevalent given the recent success of open-source controllers, e.g., OpenDaylight [27], and the emergence of SDN app stores, e.g., HP's SDN App Store [17].

Recent efforts to improve the reliability of SDN-Apps and availability of the SDN controller have been along three directions: first, diagnosing and pinpointing the root cause of failures [6, 35]; second, attacking the root cause of failures by providing better programming abstractions for developers [5, 29]; and third, developing better fault-recovery techniques through controller replication [20, 22, 44]. Of the three, only the last direction enables the SDN controller to recover from SDN-App failures in production networks. In case of controller replication, multiple replicas of the controller are deployed with each replica maintaining a state machine. Events are input to all the replicas in the same order, thus keeping all state machines identical to one another. One of the replicas is designated as the *leader* (or *master*), and when the master fails, a different replica can transparently take over and resume control of the SDN-Apps and the network. Controller replication, however, does not offer much help, when the crash of the SDN-App is caused by deterministic bugs.

There is a rich literature on the topic of fault tolerance, with studies addressing the issue in different contexts and particularly, distributed systems, operating systems and application servers. Direct application of these well-researched techniques to the problem of making controllers fault-tolerant to SDN-App failures, nevertheless, is not feasible. Techniques like reboot [4] or replay [41], for instance, cannot be applied directly to the SDN control plane; certain fundamental assumptions made by these techniques do not hold true in an SDN environment. Both the network and SDN-Apps, moreover, contain state, and rebooting [4] the SDN-App (after a crash) will eliminate this state and consequently, introduce inconsistency issues. Further, if the crash was due to deterministic bugs, replaying events [30, 33, 41] to restore the SDN-App's state will be stuck in a never-ending crash-and-reboot loop. In addition, attempts to tackle deterministic bugs [30, 33] tend to change the application's environment and may introduce erroneous output [33].

A key feature of SDNs is the clean, narrow and open application programming interface (API) that is used for com-

SOSR '16, March 14-15, 2016, Santa Clara, CA, USA
 ISBN 978-1-4503-4211-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2890955.2890965

munication between the data and control planes. This feature differentiates SDN's fault tolerance from traditional OS fault tolerance wherein little is known about the semantics of the messages that the application processes. Using the API's specification, we can extract the semantics of the API and understand the intent of each message or event. Further, we can exploit this knowledge to develop recovery mechanisms that can make modifications to the SDN-App code, environment, and/or its input without violating the semantics of the SDN-App code.

There are a number of challenges in safely recovering SDN-Apps from failures. First, network state is manipulated and shared by many SDN-Apps; a stateful SDN-App internally maintains a subset of the network state, which reflects its view of the network. Given this replicated state across stateful SDN-Apps, a key challenge lies in maintaining a consistent network state across the different SDN-Apps during failure recovery. Second, although, the semantics of the SDN control messages are well-specified and well-documented, no protocols exist to exploit this information for designing a better recovery mechanism. Determining how to leverage the semantic knowledge of the protocol in improving recovery mechanisms is also a hard problem. Third, the monolithic design of current SDN controllers implies that a failure of any one component renders the entire control plane unavailable [7]. There is a need, consequently, for a better isolation between the different components.

In this paper, we take a bold step towards developing a fault-tolerant controller architecture, called *LegoSDN*, that explicitly aims to safely recover SDN-Apps from both deterministic and non-deterministic failures. LegoSDN builds on the abstractions presented in our earlier position paper [7] to develop a heirarchy of event transformations, and to add support for transactions that cut across data and control planes. We present a prototype implementation that isolates the SDN-Apps from one another and from the controller by running each within a *sandbox*; failure of an SDN-App is restricted to the sandbox in which it is run. The interactions between the controller and SDN-Apps are carried out via remote procedure calls (RPC).

LegoSDN tackles the challenge of maintaining consistency across different SDN-Apps by employing fine-grained transactional semantics spanning both the data and control planes, and thereby enabling the controller to rollback changes made by a crashing SDN-App without impacting other (healthy) SDN-Apps. The transactional semantics, moreover, include conflict detection and resolution to maintain consistency between the SDN-Apps.

LegoSDN employs novel domain-specific transformations that modify crash-inducing events into *semantically equivalent*, but syntactically different, events to safely recover a crashed SDN-App. These transformed events are then replayed to the SDN-App restored from the crash. The transform-and-replay process continues until the SDN-App can process the transformed events successfully (without crashing). The system also provides a framework for searching through the space of equivalent events and automating this transform-and-replay process.

As a proof-of-concept, we re-architected the FloodLight controller to support LegoSDN. Using this prototype, we demonstrate that LegoSDN imposes minimal overheads and requires no modifications to the SDN-Apps (designed to run using FloodLight), thus shielding SDN-App developers from having to spend time learning the complexities of LegoSDN. Using a *synthetic* fault injector to crash-test SDN-Apps and using *mininet* [24] for emulating a network topology, we evaluated LegoSDN using five different SDN-Apps by exploring the time to recovery and by analyzing the implications of maintaining consistency.

Our contributions can be summarized as follows.

- **Cross-Layer Transaction Manager:** We propose a framework to provide transactional semantics across both control and data planes; this framework allows us to isolate failures and maintain consistency by rolling back changes made by failed SDN-Apps without requiring hardware changes to the switches (§4).
- **SDN Event Transformer:** We present a protocol for overcoming deterministic failures by extending traditional log-replay-based techniques to include domain-specific transformations that preserve the SDN-App semantics (intent and meaning) (§5).
- **Implementation and Evaluation:** We build a working prototype of a controller architecture that implements our primitives in FloodLight.

Roadmap. We begin by describing SDN-App failure scenarios and the design goals for a fault-tolerant controller in §2. We describe the architecture of LegoSDN in §3. We present our approach for supporting cross-layer transactions in §4 and for performing semantics-preserving transformations in §5. §6 presents our prototype implementation in §7. Related works are examined in §8 and open issues are discussed in §9. We present concluding remarks in §10.

2. BACKGROUND AND DESIGN

We begin this section with a review of the state of the SDN ecosystem (§2.1). We motivate our solution by describing SDN-App failure scenarios (§2.2), by discussing the implications of bugs in SDN-Apps (§2.3), and some insights behind our approach to ensuring safe recovery (§2.4). We conclude with a discussion of the key design goals of a fault-tolerant controller architecture (§2.5).

2.1 State of the SDN Ecosystem

In Table 1, we present a small list of FloodLight SDN-Apps, describe their purpose, and indicate whether they are third-party SDN-Apps or SDN-Apps developed by the developers of the controller. This table reinforces the notion that the SDN ecosystem embodies an *à la carte* system, wherein different portions of the stack are developed by different entities. We expect, furthermore, the diversity at each layer to only increase as SDN grows in popularity. In fact, movements such as the OpenDayLight Consortium [27] and

Application	*Developer*	*Purpose*
RouteFlow [34]	Third-Party	Routing
FortNox	Third-Party	Security
FlowScale [2]	Third-Party	Traffic Engineering
CloudNaas	Third-Party	Cloud Provisioning
SNAC	Third-Party	Enterprise Provisioning
BigTap [1]	FloodLight	Security
Stratos [13]	Third-party	Cloud Provisioning
VTN	Third-party	Cloud Provisioning

Table 1: *Survey of popular SDN-Apps*

the SDN-hackathons hosted by SDN Hub [36] are already promoting such diversity.

Unfortunately, many SDN-Apps lack public support forums or a public-facing bug trackers. Consequently, we are unable to quantify the impact of bugs. Luckily, recent work [6, 29, 35] on debugging SDN-Apps have uncovered a number of interesting bugs in the SDN-Apps for FloodLight, Pox, and Nox controllers. A preliminary analysis of the OpenDayLight bug-repository shows that bugs are present even in the SDN-Apps that come bundled, by default, with the controller. While novel abstractions [5] are being developed to minimize the number of bugs, extensive studies on software engineering practices indicate that bugs are prevalent in most applications and most bugs in production quality code do not even have fixes [43]. In an SDN network, bugs in an SDN-App can bring down the entire SDN stack [7, 38].

2.2 SDN-App Failure Scenarios

SDN-Apps are designed to be event-driven[1] with the implementation comprising event handlers for different network events, e.g., link activation (*link-up*) or switch failures (*switch-down*). Anecdotal evidences [6, 29], unsurprisingly, suggest that most of the errors occur in these event-handlers. We focus, hence, on these event handlers.

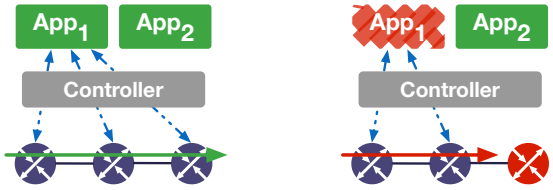
In this paper, we focus broadly on two types of failures. The first of these is *fail-stop* where the SDN-App unexpectedly terminates due to invalid memory accesses, e.g., null pointer dereferences, or due to evaluation of an erroneous expression, e.g., division by zero. The second is *invariant-violation* in which the SDN-App installs a set of OpenFlow rules that violate a network invariant, e.g., creating a loop or creating a black hole.

Although techniques exist [20, 21, 38] to detect and potentially isolate these failures, these techniques cannot safely recover the SDN-App; in particular, there are very few tools for recovering SDN-Apps from deterministic failures.

2.3 Implications of SDN-App Failures

Network policies generated by SDN-Apps often affect multiple devices, requiring many network actions. While these policies are atomic, the mechanisms for enforcing them are not. For instance, in Figure 1a, App_1 modifies three switches on the network to setup a path for a flow. App_1, however, may fail midway (Figure 1b) during the route setup operation resulting in only a *partial update* of the network; in this

[1]Traditional control-plane applications, e.g., OSPF, Spanning-Tree, are similarly event-driven.



(a) *Normal scenario* (b) *Failure scenario*

Figure 1: *Path setup under different scenarios.*

case, not all rules required for the path setup were issued to the switches before the SDN-App crashed. Such partial updates to the network introduce inconsistency, i.e., an incomplete path (Figure 1b), and should be removed in a careful and consistent manner.

SDN-App failures have implications on both the control and data planes. Recovery efforts can be simplified by treating the network, SDN controller and all the SDN-Apps as one holistic system and creating a *snapshot* (or *checkpoint*) of this holistic system [45]. Rolling back to a prior snapshot safely removes all actions effected on the network and restores the SDN-Apps and controller to the last known (healthy) state. But adding support to generate and restore snapshots of state across the network requires addition of new primitives to the switches. A more feasible alternative is to log all network events and replay them to the controller (and SDN-Apps) in the event of a crash [42]. But this approach is extremely time consuming. The overheads of these recovery mechanisms renders them impractical in a production network.

We argue that failure-recovery efforts should ensure that the network state remains consistent with the state of the different SDN-Apps.

2.4 Challenges in Crash Recovery

Anecdotal evidences suggest that the most common failures are due to a combination of unexpected timing and concurrency issues [29]. Regardless, these errors are often deterministic in nature and simply replaying the events in the same order will be insufficient. For example, the concurrency bugs tackled in OF.CPP [29] are deterministic and require more advanced techniques than simple reboot [4] or log-replay [42]. Many of these bugs require some transformations of the input messages; these can be simple transformations, e.g., reordering, to solve the concurrency related errors [6, 29], or more complex transformations involving change or deletion of messages to tackle other issues.

A key challenge in transforming messages is in ensuring that the transformed events maintain *protocol equivalence* with the original input messages. More concretely, the transformations performed should retain the semantic intent of the original messages while adhering to the protocol specification. Fortunately, OpenFlow events are well-documented and their semantics are clearly defined in the OpenFlow specification.

We can extract the intent and meaning of each message using the OpenFlow specification, and use this information to create a set list of potential transformations for each event

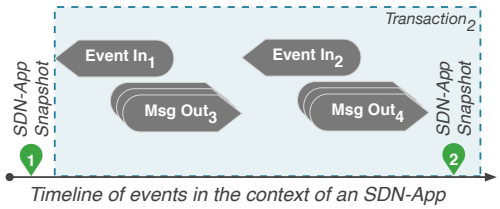


Figure 2: *Timeline of transactions delimited by checkpoints.*

or message. This effort represents a one-time cost that can be shared across various controllers and SDN-Apps.

2.5 Design Goals

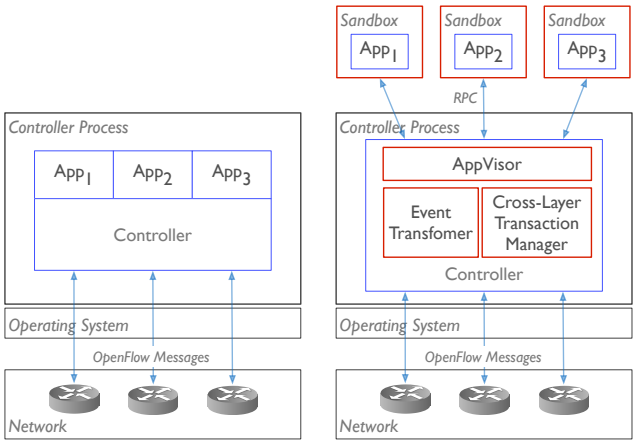
We can summarize our observations as follows: many failures in SDN-Apps are deterministic in nature, and require recovery mechanisms that are more sophisticated compared to checkpoint-restore or reboot (and replay); the semantics of SDN events (or control messages) are well-known and can be used to create a list of event-specific transformations to provide for safe recovery of SDN-Apps, particularly in case of deterministic faults; safe recovery entails maintaining consistency across both control and data planes, and we need a set of novel techniques that are faster and safer than existing approaches. Based on these observations, we identify the following design goals for a fault-tolerant SDN controller architecture.

- **Isolation:** The impact of failures should be limited to the failing SDN-App. The costs and overhead of failure recovery should be limited to the failed SDN-App or, in the worst case, the control plane. Isolation of SDN-Apps promotes high availability and simplifies failure recovery.
- **Safe Recovery:** A failed SDN-App should be recovered in a manner that ensures consistent state across the data and control planes. Ensuring consistency prevents the recovery efforts from violating network policies.
- **Minimal SDN-App Developer Effort:** Failure recovery techniques have a steep learning curve and hence, developers should not be required to make any code changes to take advantage of a new fault-tolerant architecture. Ensuring that failure recovery is transparent to the SDN-Apps removes the barriers to adoption.
- **Fast Recovery:** Although the control plane is not required to respond at the granularity of microseconds or milliseconds, several seconds of downtime [42] can result in violations of service level agreements (SLAs)[2]. Thus, recovery should happen in a timely manner.

3. LegoSDN

LegoSDN is a fault-tolerant controller framework that embodies the design goals presented in the last section. The architecture of a LegoSDN controller (Figure 3b) represents a significant departure from traditional controllers (Figure 3a) in the following three ways.

[2]SLAs can be violated, for instance, by failing to create, within a given time, the required low-latency or high-throughput network paths.



(a) *Current architecture* (b) *Proposed architecture*

Figure 3: *Comparison of controller architectures*

Sandboxing. In LegoSDN each SDN-App runs as a separate process in a *sandbox* with the controller also running (without any SDN-Apps) in its own sandbox. LegoSDN has a component, called *AppVisor*, that handles the exchange of SDN events and control messages between the controller and the SDN-Apps' sandboxes using remote procedure calls (RPC). Isolating an SDN-App in a dedicated sandbox allows LegoSDN to detect fail-stop failures.

Transaction Support. For each event processed by an SDN-App, LegoSDN keeps track of messages that the SDN-App generated in response to that event. These pairs of input events and output messages along with a snapshot of the state of the SDN-App is defined as a *cross-layer transaction* (named so, since the transaction cuts across the control and data planes; refer to §4), and shown in Figure 2. A transaction is considered *committed* with the creation of a successful *snapshot* or checkpoint (Figure 4a) of the concerned SDN-App. In the event of a failure (Figure 4c), cross-layer transactions allow LegoSDN both to revert the SDN-App using the snapshot from the previously committed transaction and to *undo* the impact of the output messages, if any, of the failed SDN-App on the network—in essence reverting the changes to the network state. Cross-layer transactions offer an *all-or-none* semantics for the actions of an SDN-App on the network; in other words, the SDN-App either successfully processes a set of events and effects necessary changes to the network in response to the events, or it does not process the events at all.

Replay. The (cross-layer) transaction manager maintains a set of *replay buffers* (Figure 4b), one for every SDN-App, that hold all input events of the transaction that is currently in progress. The replay buffer of an SDN-App is cleared when the transaction associated with the SDN-App is committed. After restoring an SDN-App from a crash using a prior snapshot, the contents of the SDN-App's replay buffer are used to bring the SDN-App to the state prior to the crash. The size of the replay buffers depends on the size of the transaction and determines the time and complexity of the replay efforts.

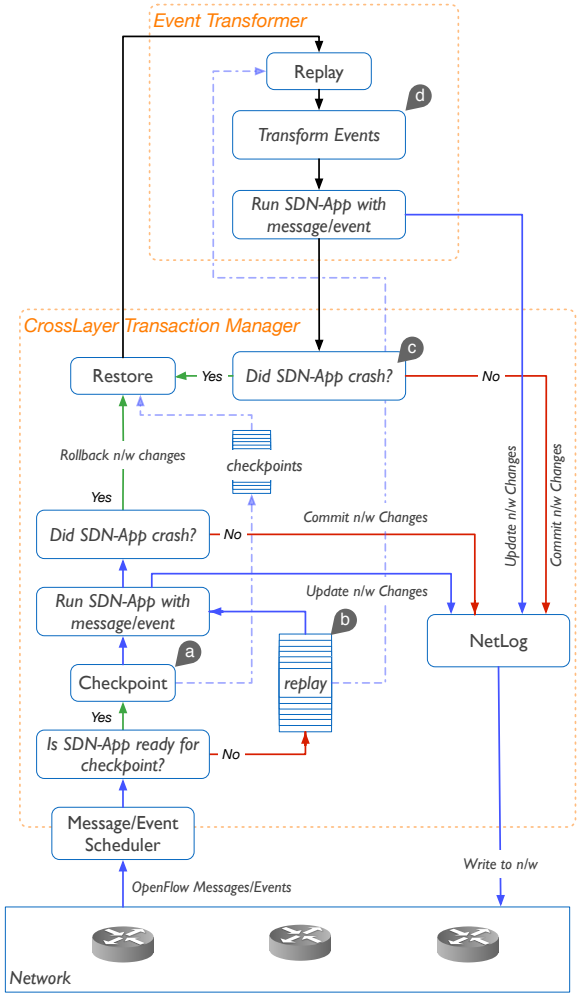


Figure 4: *Control and data flow schematic in LegoSDN*

Safe Recovery. LegoSDN includes a component, called *Event Transformer*, that assists in recovering from deterministic faults. After restoring an SDN-App, rather than replaying the exact messages available in the replay buffer, the event transformer can transform an event (Figure 4d) to a different (but equivalent) event that the SDN-App can, perhaps, process without encountering another fault. The event transformer cycles through a set of predefined transformations for each event until it finds the tranformation that allows the recovered SDN-App to make progress.

4. CROSS-LAYER TRANSACTIONS

We define a cross-layer transaction as the state generated and/or modified in the control and data plane as a result of an SDN-App processing one or more events. More concretely, for a set of input events $\{e_1, e_2, \ldots, e_N\}$ processed by an SDN-App, a cross-layer transaction is the set of variables $\{v_1, v_2, \ldots, v_n\}$ that changed within the SDN-App and the set of output messages $\{o_1, o_2, \ldots, o_n\}$ sent to the network. The variables make up the control-plane portion of the transaction whereas the output messages to the network comprise the data-plane portion of the transaction. The events

Control Message	*Inverse Control Message*
FlowMod (add flow)	FlowMod (remove flow)
Change port status	Change port status
PacketOut	**No known inverse**

Table 2: *State-changing control messages and their inverses.*

$\{e_1, e_2, \ldots, e_N\}$ are the messages stored in the *replay buffer* until the transaction containing them is committed. Tracking cross-layer transactions allows LegoSDN to perform a successful rollback of the SDN ecosystem to a consistent state after a failure. In addition, these transactions ensure that the implications of the failed SDN-App on the shared network state are not exposed to the other (healthy) SDN-Apps.

Cross-layer transactions require support for the following primitives:

- **Cross-Layer Snapshots:** Before an SDN-App begins a transaction, we have to generate snapshots of the state of both the SDN-App and the network. Checkpointing an SDN-App is relatively easy [8] compared to checkpointing network state since data plane devices currently lack appropriate primitives to support the operation.

- **Synchronized Rollback:** In the event of a failure of an SDN-App, we have to rollback the current (uncommitted) transaction, which implies undoing changes effected by the SDN-App on the network. Rollback of changes to the network, unsurprisingly, is quite challenging. While orthogonal state management tools [22, 39] facilitate tracking of changes to the network, none provide adequate primitives to support rollback.

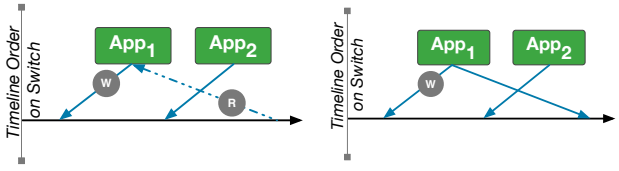
- **Data Plane Consistency:** Since network state may be shared among SDN-Apps, a naïve rollback of network state, when an SDN-App fails, may lead to inconsistency issues in other (healthy) SDN-Apps. Hence, we need support for primitives to detect conflicts and systematically resolve them.

4.1 Snapshots

In LegoSDN, the controller can easily checkpoint an SDN-App before triggering one of its event handlers with an input event or message. Creating checkpoints of the data plane, as mentioned before, is hard since it requires modifications to the switch hardware [10, 45]. We argue, however, for the addition of a state management layer to the controller. The state management layer, hence referred to as *NetLog*, models the data plane (or network) as a transactional system that supports grouping of multiple actions (entries to be installed in flow tables of switches) into one atomic operation (or update). NetLog provides support for atomic updates, and leverages existing work [32] to support consistent updates.

4.2 Rollbacks

Rollback of changes made to the control plane is a relatively simple and straightforward process, and it can be done using existing checkpoint-and-restore tools, e.g., CRIU [8].



(a) *Write-Read Conflict* (b) *Write-Write Conflict.*

Figure 5: *Conflicts that may arise during failure recovery.*

Undoing changes made to the data plane, however, is complicated since the transactions are maintained by a state-management layer in LegoSDN rather than in the switches. Control messages must be generated to undo all changes associated with the failed transaction. The rollback operation also results in the loss of soft-state stored in the switch hardware. For instance, while we can undo a flow-delete event by adding the rule back to the appropriate switch, the soft-state associated with the flow entry, e.g., timeout and flow counters, cannot be restored.

To tackle these problems, NetLog leverages the insight that control messages which modify network state are *invertible*: for every state altering control message, A, there exists another control message, B, that rolls back A's state change. Table 2 lists a representative sample of state-altering control messages and their inverses. The key observation is that a message A and it's inverse B are of the same type but with different payloads. For instance, the inverse of a *FlowMod* message adding a flow entry is also a *FlowMod* but with a different flag that indicates deletion of the entry.

NetLog also tracks and maintains the soft-state in different switches. Therefore, should NetLog need to restore a flow-table entry, it can add the entry with the appropriate timeout value. For counters, it stores the old counter values in a dedicated counter-cache and updates, in flight, the counter value in messages, e.g., statistics reply, (to be delivered to SDN-Apps) with appropriate values from its counter-cache.

4.3 Maintaining Consistency

The switches in an SDN network are a shared substrate; different SDN-Apps can access and manipulate the same flow-table entries on a switch. During a rollback, hence, changes to the network state may affect other healthy SDN-Apps. For example, in Figure 5a, a flow-table entry created by a failing SDN-App (App_2) may be read by a healthy SDN-App (App_1), and removing this flow-table entry during the rollback of the transaction associated with App_2 may impact the healthy SDN-App App_1; we refer to this issue as a *read-write* conflict (Figure 5a). Orthogonally, in Figure 5b, the flow-table entry created by the failing SDN-App App_2 may be modified, at a later time by the healthy SDN-App App_1; this issue is referred to as a *write-write* conflict.

These conflicts can be avoided by rolling back all conflicting SDN-Apps, but due to the intricate in-network dependencies the strategy can result in a cascade of rollbacks, leading to a rollback of all SDN-Apps. This strategy violates several of our design goals.

Original	*Transformation Type*	*Transformed*
link-down	Escalate	switch-down
flow-removed	Escalate	link-down
switch-up	Toggle	switch-down, switch-up

Table 3: *Sample of transformation rules.*

Write-Read Conflicts: We utilize special control messages which allows the data plane to inform an SDN-App that the network state has changed. For example, the "flow removed" message can be used to inform an SDN-App that a table entry is removed from the network. When rolling back a flow-table entry, NetLog sends the inverse message to the network to undo the changes and, in parallel, sends the "flow removed" message to all SDN-Apps with a *write-read* conflict on this entry. These messages allow the different SDN-Apps to react appropriately to the change in the network state. Ideally, the SDN-Apps will act in a manner that resolves all inconsistencies.

Write-Write Conflicts: The changes made by the failing SDN-App have been overwritten by another SDN-App. LegoSDN neither sends an inverse message to the network nor does it try to send messages to the healthy SDN-Apps. Essentially, LegoSDN supports "last writer wins" semantics.

5. EVENT TRANSFORMATIONS

Upon restoring the SDN-App and the data-plane to a consistent snapshot, the SDN-App must be fed the set of inputs that it had received between the start of the transactions and the event that caused the SDN-App to crash. The events must be replayed to ensure that the SDN-App is aware of all changes to the network; e.g., arrival of a new flow or a link failure. In the best case, the replay buffer will contain one event, and in the worst-case, it will contain N events – where N is the number of events in a cross-layer transaction.

Traditional checkpoint-replay techniques simply replay these events in the same order with no modifications. Unfortunately, such a strict replay of the events only works for non-deterministic failures. In case of deterministic failures, strict replay will result in reproducing the failure. If LegoSDN relies on such naïve replay mechanisms then the SDN-App will be stuck in an endless recover-replay loop.

To overcome deterministic faults, the event transformer determines the smallest set of events in the replay buffer that causes the SDN-App to crash and changes these events to ensure that the SDN-App recovers without failures. To do this, the event transformer includes a predefined set of rules that specify how to modify a given event (to an SDN-App) and under which condition, such transformations are valid.

These rules are created based an extensive study of the OpenFlow specification and switch behavior. By studying the specification, we are able to determine the meaning and intent of each event. More importantly, we are able to develop systematic methods for transforming an event M into a set of events, $T = \{M_1, M_2 .... M_N\}$, such that T conveys the intent of the original event, M, according to the protocol specification. In determining valid transformations to create new events, our goal is not necessarily to explore the

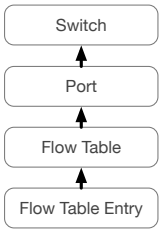


Figure 6: *Hierarchy of Event Transformations.*

exact same code-path or ensure that the SDN-App will produce identical output events. Our goal is to rather find a set of events, T, that allow the SDN-App to process the intent of the original event and to respond to this intent, without crashing again. Currently, LegoSDN supports the following three transformations, with Table 3 listing a few samples.

- **Toggle:** Most OpenFlow events express a state change in a network element, e.g., *link-up* expresses a state change in a port. Many events, moreover, have two states–up/down, on/off, or installed/un-installed. A *toggle* transformation changes an event M into two events $\{M', M\}$, where M' is the inverse of the state represented by M. For example, a *link-down* is changed to $\{$*link-up*, *link-down*$\}$.

- **Escalate:** There is a natural hierarchy amongst network elements, e.g., a flow table is a member of a switch, a link is a member of a switch. This hierarchy can be leveraged in designing event transformations (Figure 6). To this end, an *escalate* transformation changes an event M into M', where M' is the equivalent state transition in the parent element. For example, a *link-down* becomes a *switch-down*.

- **Reorder:** When there is more than one event in the replay buffer, a *reorder* transformation can reorder the events prior to replay. When reordering events in the replay buffer, however, LegoSDN maintains the following invariant: no reordering of messages from the same switch.

Restoring an SDN-App with event transformers: After restoring a crashed SDN-App, there are two scenarios to consider during replay: the first, the last event handled is the cause of the crash and transforming this last event, hence, will allow the SDN-App to recover successfully; the second, the failure is the culmination of one or more earlier events (essentially, not the last event). In this work, we focus on the former and leave discussions on how to extend LegoSDN to address the latter scenario in §9.

When the last event in the replay buffer is the cause of the failure, LegoSDN replays all but the last event and transforms the last event prior to replaying it. If replay fails, LegoSDN retries with a different transformation. This transform-and-replay process is repeated until any of the following conditions is met: all transformations are exhausted; (recovery) timer expired; the SDN-App successfully recovered from the failure.

Ordering of Transformations: To guide the exploration through the space of transformations, the event transformer ranks the transformations in terms of potential impact on the network and favors less disruptive over more disruptive transformations. The event transformer, further, prefers transformations that result in a smaller set of messages (i.e., smaller value for $|T|$).

Bounding Replay Time with a Recovery Timer: The time spent in event-replay and transformations depends on the number of transformations tried and replay attempts both of which are bounded by the desired level of reactivity. We bound the recovery time, in our case, based on the time taken for controller restarts. In practice, we limit the size of the transaction buffer to the average number of messages that LegoSDN can successfully replay and transform in the specified amount of time. While this practice provides low recovery times, there may be room for further optimizations.

Exposing Transformations to Network Operators: The act of transforming events compromises an SDN-App's ability to implement network policies completely. Unfortunately, for security related SDN-Apps, network operators may be unwilling to compromise on network policies. To account for this, we provide a simple configuration file through which operators can specify, on a per SDN-App basis, the set of events, if any, that can be transformed and the set of valid transformations to consider.

Ultimately, LegoSDN's goal is to make the SDN-Apps and not the network operators oblivious to failures. Thus, while during recovery, LegoSDN can generate a problem ticket from the captured stack-traces (or core dump), controller logs and the offending event. This ticket can help network operators to understand and triage the bug.

Limitations: Currently, LegoSDN supports a predefined set of transformations. The space of potential transformations, however, is larger than that currently explored. We envision that a domain-specific language will be provided to allow developers or network operators to define new transformations.

6. PROTOTYPE

We developed a prototype implementation of LegoSDN[3] to illustrate the utility of our architecture. We realized our prototype by modifying the Floodlight controller to include a transaction manager and an event transformer. We also made appropriate changes to support isolation and sand-boxing of the SDN-Apps from the controller and from each other. Although LegoSDN is designed to work with FloodLight, the architecture and abstractions can be easily ported to other modular SDN controllers, such as, *OpenDayLight* [27] and ONOS [3]. Next, we discuss the highlights of our prototype.

Sand-boxing SDN-Apps: To sandbox and isolate each SDN-App, we run each SDN-App in a different JVM. We eliminate the need to modify or rewrite the SDN-Apps by instrumenting the sandbox (JVM) and adding java classes to implement the controller interface. These classes convert all

[3]Source code of the prototype implementation and documentation is available at http://legosdn.cs.duke.edu

local function calls from the SDN-App to the controller into remote procedure calls over UDP.

At the controller, we create a proxy SDN-App that processes these remote procedure calls and converts them back into function calls to the appropriate methods in the controller. To ensure that the SDN-Apps get all their subscribed events, the proxy SDN-App registers itself with the controller, on behalf of the SDN-Apps, to receive those events.

Transaction Manager: The transaction manager is implemented within the controller as part of the proxy code that controls interactions between the SDN-App and the controller. To create the control plane snapshots, we capture a checkpoint of the SDN-App's JVM with CRIU [8]. [4]

Event-Transformer: A key property of this component is to perform consistent transformations for each SDN-App. Floodlight, like most other controllers, caches topology information within the controller and provides access to this cache through a number of controller modules. In implementing the event-transformer, we had to ensure that access to this cached information returned consistent transformed events regardless of the controller modules used to access the cache. Since all calls between the SDN-App and the controller happen via our proxy, we were able to consistently enforce transformations by implementing the event-transformer within the proxy.

7. EVALUATION

In this section, we evaluate LegoSDN using a number of realistic SDN-Apps and compare LegoSDN's performance with the state-of-the-art approaches, e.g., controller restarts. We quantify the time spent in various phases of recovery and highlight opportunities for further improvement. We conclude by showcasing the benefits of supporting cross-layer transactions and the capabilities of event transformations.

7.1 Experimental Setup

In our experiments, we used five different SDN-Apps: *Hub*, *Learning Switch (L.Switch)*, *Load Balancer (Load-Bal.)*, *Route Manager (Rt.Flow)*, and *Stateful Firewall*. These SDN-Apps cover both the *proactive* and *reactive* development paradigms, with L.Switch, Hub, and Stateful Firewall being reactive and the remaining being proactive SDN-Apps. The SDN-Apps were written for the vanilla FloodLight controller and needed no modifications to run on LegoSDN.

The SDN-Apps Hub and Learning Switch came bundled with FloodLight [11]. Hub simply floods a packet received on a switch port over all the remaining ports on the switch. Learning Switch examines the packet to learn the MAC-address-port mapping, flooding the packet only when a mapping is unknown. While Hub installs no rules on any switch, Learning Switch installs rules based on the mappings it learns. The Route Manager SDN-App computes the shortest path distance between all pairs of hosts and pre-installs, on each switch, the route to all reachable hosts on the network. The Load Balancer is a specialized version of the Router Manager that attempts to evenly balance the number of flows across different links to a given destination.

The Stateful Firewall SDN-App intercepts TCP packets between the trusted and untrusted parts of the network and only allows the packets if the connection was initiated from the trusted part of the network; it drops any attempt to open a connection from the untrusted part to the trusted part of the network. For the purpose of demonstration, we designed the Stateful Firewall SDN-App to not install any rule; hence, actual routing of the packet (once it is deemed as allowed by Stateful Firewall) is done using Hub in our experiments.

We used mininet [24] to emulate a network with the desired topology and *Python* scripts to emulate the traffic between hosts as required for different experiments. The experiments were carried out on a small testbed consisting of two Linux servers running Ubuntu 14.04 LTS and connected by a 1 Gbps link with a latency of 10 ms. Each machine had 12 cores with 16 GB of memory. All the experiments were conducted with the controller and isolated SDN-Apps running on one server, and with mininet and the scripts running on the other.

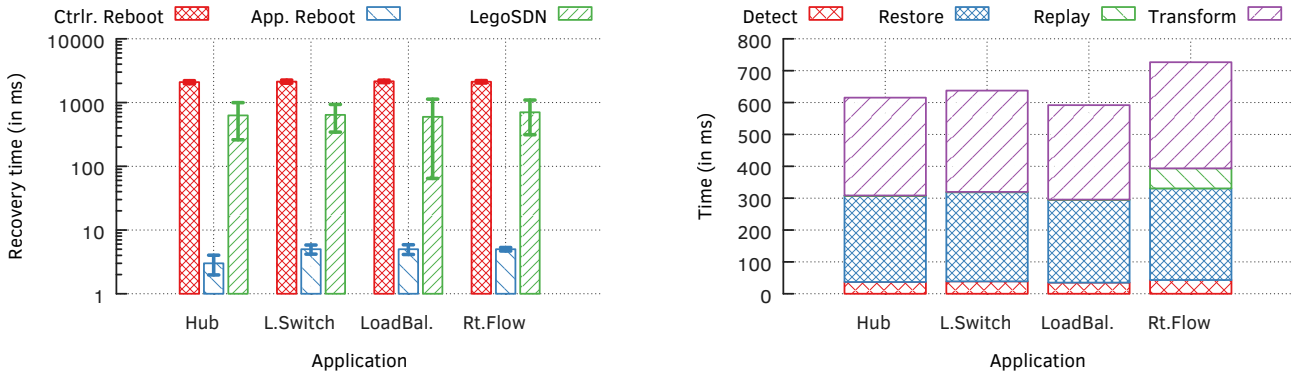
To crash SDN-Apps when using LegoSDN, we developed a *fault-injector shim* that can be configured to generate a *RuntimeException* in response to a specific input. When launching the SDN-Apps, we can pass configuration parameters (via a *properties* file) to crash one or more SDN-Apps either deterministically on a specific event, e.g., a port-down event, or randomly on any input (to emulate non-deterministic faults). When using the SDN-Apps with the traditional FloodLight controller, we use a carefully re-written faulty version of the SDN-Apps that injected fault in a manner similar to that of the fault-injector shim. This fault injection approach allows us to emulate most crash inducing errors except memory corruptions.

7.2 Recovery Time

To compare LegoSDN against other recovery mechanisms, we crashed different SDN-Apps, each for 60 times, by generating faults, and measured the recovery time of each SDN-App under different recovery mechanisms. We define the recovery time of an SDN-App as the time between when an SDN-App (or SDN stack) crashed and when the SDN-App was ready to process the next input. The SDN-Apps were crash tested separately with each test comprising only one SDN-App.

Figure 7a plots the median recovery time of the SDN-Apps under three different recovery mechanisms: LegoSDN, *controller reboot (Ctrlr. Reboot)*, and *application reboot (App. Reboot)*. By LegoSDN, we refer to the use of our prototype implementation with a transaction size of 16 input messages or events. Controller reboot implies letting the SDN-App (and the SDN controller stack) crash and restarting the SDN stack again. Application reboot, or naïve restarts, refer to restarting of the SDN-App after a crash and

[4]CRIU cannot checkpoint processes with active network connections and this posed a key challenge for implementing the RPC calls between the SDN-App and the controller. This motivated our choice of UDP as the communication channel for RPCs.



(a) *Median recovery time of different techniques.*

(b) *Median time spent in different recovery functions.*

Figure 7: *Comparison of different recovery techniques and analysis of time spent in different recovery functions.*

retrying the message again. Application reboot, hence, implicitly assumes that some mechanism exists to run the SDN-App in a separate OS process address space.

Unsurprisingly, controller reboots are infeasible, since the time to recover is in the order of seconds, and the network will be unavailable during this period. Crash of any one SDN-App also translates to crash of the entire SDN stack, as all applications are bundled with the controller and run as a single OS process. Application reboots are at least two orders of magnitude faster than the other two recovery mechanisms, but are ineffective against deterministic bugs (we elaborate on this point in § 7.3). Controller reboots, in contrast, can recover from deterministic faults since the crash inducing message is dropped or ignored; in fact, messages that arrive at the controller while recovery is in progress are also dropped.

LegoSDN is slower compared to application reboots, but it is more than 3× faster than controller reboots. Similar to application reboots, the controller is not affected and there is no loss of network control during recovery in LegoSDN. Unlike application reboots, however, LegoSDN can recover from both non-deterministic and deterministic faults (with support from the event transformer component). The error bars on top of the bar plots in Figure 7a represent one standard deviation around the median recovery time. The standard deviations of recovery times are much larger for LegoSDN compared to the rest, since some of the recovery functions that LegoSDN performs, e.g., reading the checkpointed image of an SDN-App from disk, can easily introduce a lot of variance in recovery time.

The median time spent in the different recovery functions is shown in Figure 7b. Majority of the recovery time is spent in *restore* followed by *transform*. Restore refers to resurrecting the SDN-App from an earlier checkpoint and transform implies converting of the crash inducing message to a different but equivalent message. LegoSDN uses CRIU for checkpoint and restore operations, and the RPC calls between the Java-based LegoSDN controller and the C-based CRIU service introduce some overhead. LegoSDN can benefit either from performance improvements to CRIU or from a better interface to the CRIU service. The design of LegoSDN also makes it easier to switch to other alternatives should they prove to be faster compared to CRIU. Time spent in *replay* seems only to be significant in the case of Route Manager SDN-App, since a replay of past inputs causes the SDN-App to recompute shortest paths which takes a considerable amount of time; replay time is, hence, dependent on the implementation of the SDN-App.

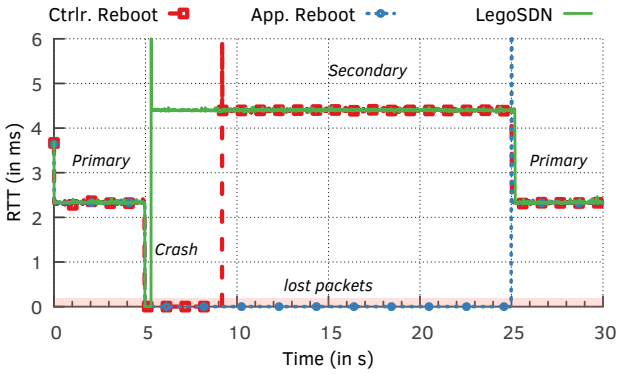
7.3 Event Transformations

While naïve reboot and replay techniques can offer very low recovery times, they do not work against deterministic faults. LegoSDN, on the other hand, can safely recover an SDN-App from deterministic faults using event transformations. To demonstrate the benefits of event transformations, we modified the Route Manager SDN-App such that the SDN-App will unconditionally fail on a *port-down* event. We used mininet to emulate a network with 8 switches connected in a *ring* topology: $s_1 - s_2 - s_3 - s_4 - s_5 - s_6 - s_8 - s_7 - s_1$, with each switch s_x having a host h_x attached to it. The experiment starts a UDP flow from h_2 to h_8, and while the flow is in progress brings *down* the link $s_1 - s_7$, and after a period of 20 s brings it back *up*.

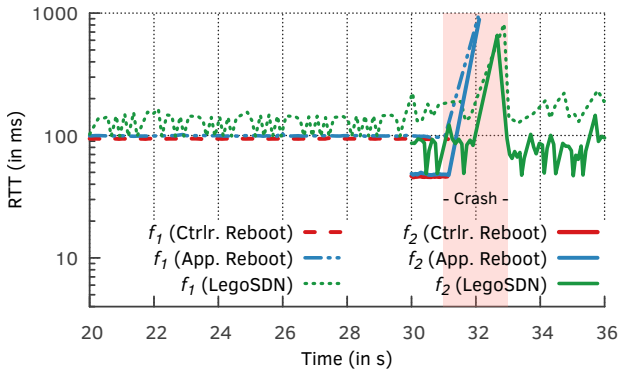
Initially, the shortest path p_1 from h_2 to h_8 is over the link $s_1 - s_7$: $h_2 - s_2 - s_1 - s_7 - s_8 - h_8$. When the link $s_1 - s_7$ is down, however, the shortest path changes to $p_2 : h_2 - s_2 - s_3 - s_4 - s_5 - s_6 - s_8 - h_8$. The paths p_1 and p_2 are referred to as *primary* and *secondary* paths, respectively, as shown in Figure 8a. We assigned a 2 ms latency to the link $s_1 - s_7$ and 4 ms latency to the link $s_3 - s_4$ to use the RTTs observed by the UDP flow in identifying the path of the flow. The RTTs observed by the UDP flow during the duration of the experiment are recorded separately for each recovery mechanism and shown in Figure 8a.

The link $s_1 - s_7$ is brought down around the 5 s mark in Figure 8a generating two *port-down* events, one for the port on each switch on either end of the link; this causes the Route Manager application to crash. The link $s_1 - s_7$ is brought back up around the 25 s mark generating *port-up* events which the SDN-App can process without fail, as is evidenced by all three lines switching from *Secondary* to *Primary* path, in the figure, around the 25 s mark.

In case of controller reboot, the recovery takes approximately 4 s, but on recovery, since the entire stack was re-



(a) *Recovering from deterministic faults via event transformations.*



(b) *Recovery of TCP flows f_1 and f_2 after a crash.*

Figure 8: *Safe SDN-App recovery from deterministic and non-deterministic faults.*

booted, the SDN-App receives updates describing the recent network topology. Route Manager, naturally, recomputes the shortest paths and correctly routes the UDP flow from $h_2 - h_8$ over the path p_2. Not surprisingly, application reboot cannot recover from the crash until the time when the link $s_1 - s_7$ is brought back up; if the replay is skipped (i.e., the crash inducing message is dropped), the SDN-App recovers but remains oblivious to the change in the network topology until the next change at the 25 s mark. LegoSDN, compared to the other two mechanisms, recovers quickly (within 250 ms) from the crash and immediately re-routes the flows over the path p_2, regardless of the SDN-App's inability to process the *port-down* event.

To recover from the deterministic fault, the event transformer transforms the *port-down* events to *switch-down* events, effectively removing switches s_1 and s_7 from the topology. The event transformer, hence, presents an altered view of the network to the SDN-App to help it recover from the fault. When the link $s_1 - s_7$ comes back up, it first activates the switches s_1 and s_7 before delivering the events related to the new state of the link. Simply mapping messages from one to another without maintaining a *consistent per-app view* of the network, will fail to re-route the flow back to p_1 at the 25 s mark.

7.4 Application State Recovery

Controller and application reboots are also ineffective in recovering stateful SDN-Apps from non-deterministic failures. Using the same network topology as described in the previous section (§ 7.2), but with all links set to have a latency of 1 ms, we run two SDN-Apps–Stateful Firewall and Hub–with the three recovery mechanisms, and use two TCP flows from the trusted part to the untrusted part of the network. The flows repeatedly open a TCP connection, send 10 packets and close the connection. We carefully induce a fault after the TCP connection establishment phase of one of the flows. The fault causes only Stateful Firewall to crash, and leaves Hub intact; the experiment was setup, however, not to allow Hub to flood without Stateful Firewall.

We measure the RTT observed from the two TCP flows–f_1 and f_2–and plot a portion of the time series of RTTs in Figure 8b. The flow f_2 (from h_2 to h_3) starts 30 s after the start of flow f_1 (from h_1 to h_4). Stateful Firewall was configured to treat hosts h_1 and h_2 as trusted, and hosts h_3 and h_4 as untrusted. The RTTs are high since each packet is intercepted by Stateful Firewall and routed to its destination by Hub (which floods the packet). The fluctuations in RTTs of the flows when the SDN-Apps are deployed using LegoSDN are because of additional functions, e.g., checkpointing, performed by LegoSDN. These functions, however, are critical as the TCP flows continue after the crash only when using LegoSDN. Without checkpointing of the SDN-Apps' states and restoring them after the crash, Stateful Firewall will lose the connection establishment information and hence, will drop the TCP packets. The sender and the receiver, however, are also unaware that they have to re-establish the connection; the sender retransmits assuming packets are getting lost. The loss of state when recovering Stateful Firewall using either controller or application reboot prevents the TCP flows from making any progress after the crash (indicated by the shaded region) as shown in Figure 8b.

7.5 Cross-layer Transaction Support

We use mininet to emulate a network with 6 switches connected in a *ring*-like topology: $s_1 - s_2 - s_3 - s_4 - s_6 - s_5 - s_2$, with switch s_1 outside the ring. Switches s_1 through s_4 each have one host attached at port-1. We used a slightly modified version of the route manager application (introduced earlier in this section) that installs rules proactively to route flows between hosts h_2 through h_4 along the path $s_2 - s_3 - s_4$. Switch s_5 is setup to forward all packets to s_6 which in turn is setup to forward to s_4. Initially, the path through s_5 and s_6 remains unused and no rules are installed on Switch s_1.

Suppose that the path through s_5 and s_6 contains a middlebox that inspects the traffic to check for malicious flows. After approximately 5 s, a new flow is started from h_1 to h_3. Switch s_1 sends the first packet to the controller which in turn contacts the SDN-app to setup the route for the flow. The SDN-app does not trust the flow originating from h_1 and intends to route the flow over the alternate path $s_1 - s_2 - s_5 - s_6 - s_4 - s_3$ (response from h_3 to h_1 flows over the path $s_3 - s_2 - s_1$). To this end the SDN-app installs rules on switches s_4, s_3, s_2, s_1, s_6, in that order. The SDN-

app, however, is programmed to fail in the middle of this route setup—after setting up s_1 but before setting up s_6. When NetLog is enabled changes made to the network are reverted after a crash and, hence, the untrusted flow makes no progress. Disabling NetLog, on the other hand, causes the untrusted flow to proceed (over the path $s_1 - s_2 - s_3$), resulting in policy violations (since the flow is not being inspected by the middlebox on the path through $s_5 - s_6$).

8. RELATED WORKS

SDN Fault Tolerance: The most closely related works [9, 18, 20, 22] focus on recovering from controller failures, typically, by applying Paxos [28]; there is not much focus on handling SDN-App failures or deterministics failures, and these faults can cripple the framework. Specifically, Ravana [20] employs a similar notion of transactions, but, unlike LegoSDN, Ravana requires switch-side mechanisms and extensions to the OpenFlow interface to guarantee correctness.

Rosemary [38], like LegoSDN, adds isolation to SDN-App. LegoSDN, however, improves on Rosemary by maintaining transparency and by including a better fault tolerance model—one that can recover from a richer set of failures.

Bugs in SDN Code: Recent work to debug SDNs focus on detecting bugs [15, 21] or debugging bugs in SDN-Apps [16, 26, 35] and SDN switches [23, 25]. Building on these existing approaches, we attack an orthogonal problem, that of overcoming bugs which result in controller failures or violation of network invariants. Our approach allows the network to guarantee availability even after bugs trigger SDN-App failures.

Operating System Fault Tolerance: Our approach builds on several key operating system principles: isolation [4], failure oblivious computation [33], and checkpoint-replay [19, 40]. These approaches, however, assume that bugs are non-deterministic and thus can be fixed by a reboot [4, 19, 40]. To tackle deterministic bugs, LegoSDN extends these approaches [4, 19, 40] by employing domain specific knowledge to transform failure inducing events into safe events. Unlike in failure-oblivious computing [33] where transformations inject random data, in LegoSDN transformations maintain semantic equivalence and thus ensure safety.

Other SDN controllers: While recent SDN controllers, e.g., ONOS [3], OpenDayLight [27], have made significant progress towards addressing controller availability, tolerating deterministic SDN-App failures is not an objective of their designs. LegoSDN can be easily extended to such platforms; in case of ONOS, for instance, LegoSDN's NetLog can be incorporated into ONOS's global network view.

On the other hand, approaches like Frenetic [12] and Pyretic [31] focus on providing a better programming platform for developing SDN-Apps, thus potentially eliminating bugs. The objectives of such programming platforms is to allow developers to write modular applications with ease and such objectives are orthogonal to that of LegoSDN. These controllers, nevertheless, neither address isolation of SDN-App crashes nor offer support for checkpoint and recovery of SDN-Apps. LegoSDN can, however, provide support for some of the high-level objectives of these programming platforms, e.g., avoiding writing of redundant or conflicting rules, by extending the AppVisor component.

9. DISCUSSION

Next, we discuss several limitations of LegoSDN as well as a few interesting approaches for extending LegoSDN to address other types of failures.

Controller Failures: LegoSDN isolates the SDN-Apps from the controller and runs them in separate processes. The approach also implies that despite the controller crashing, an SDN-App can continue unaffected (although the SDN-App will be unable to interact with the network).

Failures caused by Multiple Events: Although LegoSDN, currently, focuses on failures induced by the last event processed, the system can be extended by incorporating STS [35] to address failures caused by an arbitrary set of events. We envision that upon detecting a failure, LegoSDN can run STS on the failed SDN-App to determine the minimal set of input events, S, associated with the failure. S is then placed in the replay buffer, and LegoSDN rolls back the SDN-App to the last checkpoint before the first message in S. The transform-and-replay process continues from this point onwards exactly as described before. We suspect, however, that a new set of transformations may be required to improve the speed of replay.

Network Function Virtualization (NFV) While the approaches in §4 focus on network switches, we believe they can be extended to include network functions. We can treat network functions like we treat switches and expand NetLog to include network-function state by incorporating existing techniques [14] for managing network function states. Alternatively, we could leverage existing approaches for checkpointing middleboxes [37].

Limitations of LegoSDN's Rollback: LegoSDN's roll back only impacts network elements and not end hosts. Hence, packets sent to end hosts cannot be modified as part of the rollback, and this leaves the end-host stack in an inconsistent state. Fortunately, unlike in the SDN switches and controller, multiple layers of failure-recovery are built into the end-host stack and the stack will adjust to the modified network. For example, TCP contains mechanisms to deal with out of order packets and duplicate packets.

Maintain network invariants: In case LegoSDN cannot successfully recover from a crash (i.e., all transformations still result in a crash), LegoSDN drops the input event. The SDN-App at this point can continue or be halted depending on the SDN-App's configuration (specified at startup). Dropping an input (message or event) can lead to a violation of network invariants, i.e., dropping a *switch-down* can result in *black holes*. We argue that, in general, compromising the availability of a few flows (as a result of violating network invariants) is *acceptable* compared to sacrificing control of the entire network. On the other hand, if network invariants cannot ever be violated, a host of policy checkers [21] can be used to check network invariants; in case the guarantee

fails to hold, LegoSDN reverts to using controller reboots.

10. CONCLUSION

Currently, SDN-Apps are used over controllers in an ad-hoc fashion, without any isolation or abstractions to control properties for all SDN-Apps. LegoSDN demonstrates how one can retrofit this functionality in the existing SDN stack without any modifications to the controller or SDN-Apps. We use LegoSDN to provide fault tolerance and address the need for a safe and fault-tolerant controller by detecting SDN-App failures in real time and modifying network events to eliminate the crash. We implemented a prototype of LegoSDN based on the popular FloodLight controller platform and show that the performance overhead of fault tolerance is reasonable.

Acknowledgments

This work was partially supported by the NSF award CSR #1409426.

11. REFERENCES

[1] Big Tap Monitoring Fabric. http://goo.gl/UHDqjT.
[2] FlowScale. http://goo.gl/WewH1U.
[3] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. HotSDN, 2014.
[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — A Technique for Cheap Recovery. OSDI, 2004.
[5] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software Transactional Networking: Concurrent and Consistent Policy Composition. HotSDN, 2013.
[6] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test Openflow Applications. NSDI, 2012.
[7] B. Chandrasekaran and T. Benson. Tolerating SDN Application Failures with LegoSDN. HotNets, 2014.
[8] Checkpoint/Restore In Userspace (CRIU). http://goo.gl/OMb5K.
[9] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. SOSR, 2015.
[10] A. Dwaraki, S. Seetharaman, S. Natarajan, and T. Wolf. GitFlow: Flow Revision Management for Software-defined Networks. SOSR, 2015.
[11] Project Floodlight. http://goo.gl/aV1E40.
[12] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: A High-level Language for OpenFlow Networks. PRESTO, 2010.
[13] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-Class Entities. Technical Report TR1771, University of Wisconsin-Madison, June 2012.
[14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. SIGCOMM, 2014.
[15] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. NSDI, 2014.
[16] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, K. Zarifis, and P. Kazemian. Leveraging SDN Layering to Systematically Troubleshoot Networks. HotSDN, 2013.
[17] HP's SDN App Store. https://goo.gl/2vtdHH.
[18] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. SIGCOMM, 2013.
[19] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. ASPLOS, 2013.
[20] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller Fault-tolerance in Software-defined Networking. SOSR, 2015.
[21] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. NSDI, 2013.
[22] 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. OSDI, 2010.
[23] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for Openflow Switch Interoperability Testing. CoNEXT, 2012.
[24] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. HotNets, 2010.
[25] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev. SDNRacer: Detecting Concurrency Violations in Software-defined Networks. SOSR, 2015.
[26] T. Nelson, D. Yu, Y. Li, R. Fonseca, and S. Krishnamurthi. Simon: Scriptable Interactive Monitoring for SDNs. SOSR, 2015.
[27] OpenDaylight: A linux foundation collaborative project. http://goo.gl/1uobC.
[28] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2), 1980.
[29] P. Perešíni, M. Kuzniar, N. Vasić, M. Canini, and D. Kostiū. Of. cpp: Consistent packet processing for openflow. 2013.
[30] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. SOSP, 2005.
[31] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN Programming with Pyretic. *USENIX ;login*, 38(5), Oct. 2013.
[32] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. SIGCOMM, 2012.
[33] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing Server Availability and Security Through Failure-oblivious Computing. OSDI, 2004.
[34] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk. Revisiting Routing Control Platforms with the Eyes and Muscles of Software-defined Networking. HotSDN, 2012.
[35] C. Scott, A. Wundsam, B. Raghavan, Z. Liu, S. Whitlock, A. El-Hassany, A. Or, J. Lai, E. Huang, H. B. Acharya, K. Zarifis, and S. Shenker. Troubleshooting SDN Control Software with Minimal Causal Sequences. SIGCOMM, 2014.
[36] SDN Hub. http://sdnhub.org/.
[37] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. SIGCOMM, 2015.
[38] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A Robust, Secure, and High-performance Network Operating System. CCS, 2014.
[39] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A network-state management service. SIGCOMM, 2014.
[40] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. *ACM Trans. Comput. Syst.*, 24(4), Nov. 2006.
[41] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Trans. Comput. Syst.*, 23(1), 2005.
[42] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. HotSwap: Correct and Efficient Controller Upgrades for Software-defined Networks. HotSDN, 2013.
[43] A. P. Wood. Software Reliability from the Customer View. *Computer*, 2003.
[44] S. H. Yeganeh and Y. Ganjali. Beehive: Simple Distributed Programming in Software-Defined Networks. SoSR, 2016.
[45] Y. Zhang, N. Beheshti, and R. Manghirmalani. NetRevert: Rollback Recovery in SDN. HotSDN, 2014.