Tolerating SDN Application Failures with LegoSDN

Balakrishnan Chandrasekaran, Theophilus Benson
Duke University
{balac,tbenson}@cs.duke.edu

ABSTRACT

Despite Software Defined Network's (SDN) proven benefits, there remains significant reluctance in adopting it. Among the issues that hamper SDN's adoption two stand out: *reliability* and *fault tolerance*. At the heart of these issues is a set of fate-sharing relationships: The first between the SDN-Apps and controllers, where-in the crash of the former induces a crash of the latter, and thereby affecting availability; and, the second between the SDN-App and the network, where-in a byzantine failure e.g., black-holes and network-loops, induces a failure in the network, and thereby affecting network availability. The principal position of this paper is that availability is of utmost concern – second only to security. To this end, we present a re-design of the controller architecture centering around a set of abstractions to eliminate these fate-sharing relationships, and make the controllers and network resilient to SDN-App failures. We illustrate how these abstractions can be used to improve the reliability of an SDN environment, thus eliminating one of the barriers to SDN's adoption.

Categories and Subject Descriptors

C.2 [**Computer Communication Networks**]: Network Architecture and Design; C.4 [**Performance of Systems**]: Reliability, availability and serviceability

Keywords

Software-Defined Networking; Fault Tolerance

1. INTRODUCTION

Software Defined Networking (SDN) has made great strides in the past few years, with active involvement from both academia and the industry. Inspite of SDN's benefits and the industry endorsements, however, there remains significant reluctance in SDN's adoption. Amongst the factors impeding the adoption of SDN, two stand out: *reliability* and *fault tolerance*. In an SDN deployment, there are four main failure scenarios: controller server failures (hardware failures), controller crashes (bugs in the controller code), network device failures (switch, end-host or link failures), and SDN application (SDN-App) crashes (bugs in the application code). While much focus has been on overcoming hardware and network device failures [22, 25, 21], and debugging SDN-Apps and the ecosystem [15, 16, 10, 22], little has been done to protect SDN-Apps against failures.

Fault-tolerant design has been studied extensively in different contexts, viz., operating systems and application-servers. Unfortunately, techniques like reboot [9] or replay [30] cannot be applied directly to the SDN control plane: certain fundamental assumptions of these techniques do not hold true in the SDN ecosystem. First, both the network and the SDN-Apps contain state, and rebooting [9] the SDN-App will result in loss of state and consequently, introduce inconsistency issues (the SDN-App's view of the network and the actual network state might no longer be the same). Second, the state of an SDN-App might be possibly interdependent on the state of other SDN-Apps. Reboots of an SDN-App, hence, can potentially affect this entire ecosystem. Third, naïvely replaying events [30] to recover this state implicitly assumes that the bugs are non-deterministic (or *transient*) and thus would not be encountered during replay. However, we argue that given the event-driven nature of SDN-Apps, bugs will most likely be deterministic.

In this work, we focus on SDN-App failures, fail-stop crashes and byzantine failures, and not on controller failures because the controller code represents a common layer that is highly reused and, thus, has a lesser likelihood of containing bugs. The principal position of this paper is that availability is of upmost concern – second only to security.

We argue that the availability of an SDN controller is reduced by the existence of two *fate-sharing* relationships in the SDN ecosystem: The first between the SDN-Apps and controllers, where-in the crash of the former induces a crash of the latter, and thereby affecting availability; and the

Generic Controller Stack	Floodlight Stack
Application	RouteFlow
Controller	FloodLight
Server Operating System	Ubuntu
Server Hardware	Dell Blade

Table 1: SDN Stack Illustration

second between the SDN-App and the network, where-in a byzantine failure within an SDN-App may lead to the violation of a network safety property, and thereby affecting network availability. The issue is symptomatic of the lack of proper abstractions between the controller and the SDN-Apps, and also between the SDN-Apps and the network. Our solution to this issue is a re-design of the controller architecture centering around a set of abstractions that enable two new features:

- **Isolating the SDN-Apps from the Controller**. Promote isolation and eliminate the two fate-sharing relationships.
- **Isolating the SDN-Apps from the Network**. Support the notion of network-wide transactions to manage the explicit and implicit relationships between various SDN-Apps running on a controller; transactions in conjunction with roll-backs help in guaranteeing consistency across SDN-Apps and undoing the impact of byzantine and fail-stop bugs.

The re-design allows us to safely run SDN-Apps with a *best-effort model* that overcomes SDN-App failures by detecting failure triggering events, and ignoring or transforming these events. Ignoring or transforming events, however, compromises the SDN-Apps' *correctness* (ability to completely implement its policies). The challenge lies in (1) detecting faults, *i.e.*, determining when to compromise *correctness*, (2) in determining how to overcome the faults, *i.e.*, how much to compromise, and (3) in ensuring safety while performing fault recovery (compromising *correctness*).

Our proposal calls for a re-design of the controller to support three key abstractions:

1. An isolation layer between SDN-Apps.
2. A transaction layer that bundles operations issued by the controller or SDN-Apps, with implicit or explicit dependencies, into one *atomic* update that facilitates roll-back or recovery operations without sacrificing consistency.
3. A network roll-back mechanism that accurately and efficiently undoes transformations to network state.

In this paper, we present LegoSDN a straw-man that embodies the described abstractions by providing *AppVisor* – an isolation layer between SDN-Apps, and *NetLog* – a network-wide transaction system that supports atomic updates and efficient roll backs. We exemplify the strength of these abstractions by using them to build *Crash-Pad* – a fault tolerance layer that detects crash triggering events and overcomes them through event transformations. LegoSDN *does not require any modifications* to the SDN controller[1] or the SDN-Apps.

2. MOTIVATION AND RELATED WORKS

In this section, we review the properties of a canonical SDN environment, focusing on the salient interactions between applications and controllers. We follow up with an overview of the fault model within SDN environments coupled with a discussion of related works and relevant techniques from other domains, most notably, operating systems and distributed systems.

2.1 Properties of the SDN Ecosystem

Software-Defined Networking separates the network control logic from the switches and centralizes it; therefore, failure of the controller or controller services significantly impacts the network. Table 1 illustrates the layers in a canonical SDN control platform. In platforms such as NoX [31] and FloodLight [7], failures of any component in the stack renders the control plane unavailable. Perhaps surprisingly, in a FloodLight stack, Table 1, an un-handled exception (fault) in one SDN-App will result in the failure of other SDN-Apps and the controller itself. Similarly, a segmentation fault in a NoX SDN-App takes down both the controller and SDN-App. The crashes of SDN-Apps have a disastrous effect on the entire platform because of a lack of isolation, both between the SDN-Apps and the controller, and between the SDN-Apps themselves.

While certain companies, such as BigSwitch [1], aim to provide the entire SDN stack as a single vertical, monolithic bundle, most of the popular controller platforms are released as open-source software precisely to decouple the stack and to promote the creation a rich ecosystem of third-party SDN-Apps. For example, FloodLight [7], a Java controller platform released by BigSwitch, boasts a diverse spectrum of SDN-Apps ranging from performance enhancing [27, 4, 13], to security enforcement [13]. In Table 2, we present a small list of FloodLight SDN-Apps, their purpose, and indicate whether they are developed by a third-party or developed (in-house) by the developers of the controller.

Table 2 reinforces the notion that the SDN ecosystem embodies an *à la carte* system, where-in different portions of the stack are developed by different entities. Furthermore, we expect the diversity at each layer to only increase as SDN grows in popularity. In fact, movements such as the OpenDayLight Consortium [6] and the SDN-hackathons hosted by SDN-Hub are already promoting this diversity.

Given the infancy of these SDN-Apps, unfortunately, many lack a public-facing support forum or bug-tracker. Luckily, we were able to find a public bug-tracker for the

[1]The current LegoSDN prototype is designed to work with FloodLight.

SDN-App FlowScale [5]. Upon examination of this bug-tracker, we discovered that 16% of the reported bugs resulted in catastrophic exceptions. Furthermore, careful analysis showed that even now, at the submission of this paper, one of these bugs still remains open. While these numbers do not generalize to other SDN-Apps, we note that extensive studies on software engineering practices show that bugs are prevalent in most applications and, even worse, most bugs in production quality code do not have fixes at the time they are encountered [34].

Application	Developer	Purpose
RouteFlow [27]	Third-Party	Routing
FlowScale [4]	Third-Party	Traffic Engineering
BigTap [2]	BigSwitch	Security
Stratos [13]	Third-party	Cloud Provisioning

Table 2: Survey of Popular SDN Applications

2.2 Fault Tolerance in SDN Networks and OS

SDN Fault Tolerance: Fault tolerance within the SDN environment has been a subject of research for quite some time. Recent efforts [21, 17] in the context of handling controller failures focus on recovery from hardware failures, applying *Paxos* [23] to ensure robustness to controller server failures. Unfortunately, such techniques do not protect the controller against deterministics bugs. Researchers have also focused on shielding developers from writing application code to handle switch failures [25, 22], link failures, or application server failures [33]. But, surprisingly little has been done to shield the network and the controller from application failure, and it is precisely this shortcoming that LegoSDN addresses. Most specifically, LegoSDN offers a solution to handle failures induced by bugs within the SDN-Apps. We plan on handling failures arising out of bugs in controller code in future work.

Bugs in SDN Code: Research efforts in the past few years [20, 15], have addressed the problem of detecting bugs that violate network consistency, defined through a set of network invariants. In an effort more relevant to debugging of SDN-Apps, STS [28] offers a solution to determine the minimal set of events required to trigger a bug. These efforts can prevent bugs in an SDN-App from installing a faulty rule [20, 15], or help in reproducing the bug to triage the issue [28], but restoring an application once it has crashed is, for the most part, out of their scope. While we leverage these efforts, we attack an orthogonal problem, that of overcoming crashes of SDN-Apps that lead to controller crashes or violation of network invariants. Our approach allows the network to maintain availability in the face of SDN-App failures.

OS Fault Tolerance: LegoSDN builds on several key operating system techniques, namely, *isolation* [9], *speculative fixes* [26], *changes to the operating environment* [24], and *checkpoint replay* [18, 29]. These techniques, nevertheless, assume that bugs are non-deterministic and, thus, can be fixed by a reboot [9, 18, 29], or that safety and *correctness* can be compromised to improve availability [26, 24]. While LegoSDN compromises *correctness* for availability, unlike the previous techniques [9, 18, 29], it does not, by default, assume non-determinism. Unlike previous approaches [26, 24], our system provides network operators with control over the extent to which *correctness* can be compromised to guarantee availability. Futhemore, unlike previous approaches we are able to roll-back the SDN-App's output (e.g. the rules installed) by utilizing SDN's programmatic control and we are able to modify the failure inducing input (event or message) by leveraging domain specific knowledge.

3. RE-THINKING SDN CONTROLLERS

The controller's inability to tolerate crashes of SDN-Apps is symptomatic of a larger endemic issue in the design of the controller's architecture. In this section, we present LegoSDN— a system that allows the controller to run in spite of SDN-App crashes. LegoSDN transparently[2] modifies the application-controller interface via two components: *AppVisor* (§3.1) and *NetLog* (§3.2). AppVisor is a regular SDN-App running within the controller, while NetLog is a standalone program that provides support for rolling back changes to the network's state. To offer a glimpse of the new capabilities that LegoSDN provides, we discuss the design of Crash-Pad (§3.3), a system that provides failure detection and recovery support. Crash-Pad serves to demonstrate the benefits of a controller re-design supporting our abstractions.

3.1 Isolation and Modularity: AppVisor

The SDN controller should be designed and architected like any platform that allows third-party code to execute, with SDN-Apps running as isolated modules with clearly defined fault and resource allocation boundaries. Further, we make the fundamental assumption that SDN-Apps may become momentarily unavailable due to failures; bug-free or crash-free SDN-Apps should be treated exactly as what they are — an *exception*, and not the norm.

The AppVisor builds on well-studied isolation techniques used in Operating Systems. AppVisor's objective is to separate the address space of the SDN-Apps from each other, and more importantly, from that of the controller, by running them in different processes. The address space separation enables containment of SDN-App crashes to the processes (or containers) in which they are running in. Aside from improving availability, this design opens itself to a wide range of novel use cases: per-application resource limits, application migration, multi-version software testing, and allows for certain seamless controller upgrades.

Not surprisingly, this design calls for a simple communication protocol between the controller and the isolated SDN-Apps. We note that serialization and de-serialization of messages, and the communication protocol overhead introduce

[2]Neither the controller nor the SDN-App require any code change.

additional latency into the control-loop (between the switch and the SDN-Apps). The additional latency, however, is acceptable as introducing the controller into the critical-path (of flow setup or packet processing) already slows down the network by a factor of four [11].

In the context of fault-tolerance AppVisor ensures, beyond any doubt, that failures in any SDN-App do not affect other SDN-Apps, or the controller.

3.2 Network Transactions: NetLog

Network policies often span multiple devices, and hence, mechanisms to implement a policy may comprise many network actions (e.g. OpenFlow messages). Controllers treat these as independent actions. We propose that the actions associated with any policy must be executed in an atomic fashion, with *all-or-nothing* semantics. Failure of either an SDN-App or an action generated by an SDN-App should trigger a network-wide roll-back of all related actions of the concerned policy.

NetLog leverages the insight that each control message that modifies network state is *invertible*: for every state altering control message, A, there exists another control message, B, that undoes A's state change. Not surprisingly, undoing a state change is imperfect as certain network state is lost. For instance, while it is possible to undo a flow delete event, by adding the flow back to the network, the flow timeout and flow counters cannot be restored. Consequently, NetLog, stores and maintains the timeout and counter information of a flow table entry before deleting it. Therefore, should NetLog need to restore a flow table entry, it adds it with the appropriate time-out information. For counters, it stores the old counter values in a counter-cache and updates the counter value in messages (viz., statistics reply) to the correct one based on values from its counter-cache.

In the context of fault-tolerance NetLog ensures that the network-wide state remains consistent regardless of failures.

3.3 Surviving amidst failures: Crash-Pad

Consider the following three trends: (1) 80% of bugs in production quality software do not have fixes at the time they are encountered [34], (2) bugs have been found in three of the dominant controllers [28] (not all controllers were written in the same programming language), and (3) bugs in SDN-Apps are mostly deterministic. We envision Crash-Pad to leverage these trends and overcome SDN-App failures by detecting failure-inducing events, and ignoring or transforming these events. To achieve this, Crash-Pad exploits the fault isolation (or containment) provided by AppVisor, and the support for atomic updates provided by NetLog.

The act of ignoring or transforming events compromises an SDN-App's ability to completely implement its policies (*correctness*), and through the design of Crash-Pad, we aim to explore and provide insight into certain key design questions in this context.

How to detect a bug? (When to compromise correctness?) SDN-Apps are largely event-driven and in most situations, the cause of an SDN-App's failure is simply the last event processed by the SDN-App before failure. We classify failures as follows:

- *Fail-stop* failures: the SDN-App crashes and it can be detected using techniques described in Section §4.1.
- *Byzantine* failures: the output of the SDN-App violates network invariants, which can be detected using policy checkers [20].

Crash-Pad takes a *snapshot* of the state of the SDN-App prior to its processing of an event and should a failure occur, it can easily revert to this snapshot. Replay of the offending event, however, will most likely cause the SDN-App to fail. Therefore, Crash-Pad either ignores or transforms the event, referred to as a *correctness*-compromising operation on the offending event, prior to the replay.

How to overcome a bug? (How much correctness to compromise?) Crash-Pad can provide a simple interface through which operators can specify policies (*correctness*-compromising transformations) that dictate how to compromise *correctness* when a crash is encountered. In this initial straw-man, we aim to expose three basic policies:

- *Absolute Compromise* ignores the offending event (sacrificing *correctness*) and makes SDN-Apps *failure oblivious*
- *No Compromise* allows the SDN-App to crash, thus sacrificing availability to ensure *correctness*.
- *Equivalence Compromise* transforms the event into an equivalent one, e.g. a switch down event can be transformed into a series of link down events. Alternatively, a link down event may be transformed into a switch down event. This transformation exploits the domain knowledge that certain events are *super-sets* of other events and vice versa.

How to specify the availability-correctness trade-off? For security applications, network operators may be unwilling to compromise on the *correctness* of certain SDN-Apps, depending on the nature of the event. To account for this, Crash-Pad can support a simple policy language that allows operators to specify, on a per application basis, the set of events, if any, that they are willing to compromise on.

How to alert operators of failures or compromises? Our research agenda is to make the SDN-Apps and not the SDN-App developers oblivious to failures. When subverting a failure, Crash-Pad will generate a *problem ticket* from the captured stack-traces generated by the SDN-App, controller logs and the offending event. The problem ticket can help developers to triage the SDN-App's bug.

3.4 Enabling novel techniques

Besides providing an environment where SDN operators can deploy SDN-Apps without fear of crashing the controller or other SDN-Apps, LegoSDN addresses the following use-cases that are not achievable in today's SDN environments:

Enabling Software and Data Diversity in SDNs: A popular software engineering technique is to have multiple teams develop identical versions of the same application. The idea being that most teams will implement the functionality correctly; the correct output for any given input can be chosen using a majority vote on the outputs from the different versions. LegoSDN can be used to distribute events to the different versions of the same SDN-App, and compare the outputs.

Per Application Resource Limits: Currently, there is no way to limit the resource consumption of individual SDN-Apps. Consequently, a rogue SDN-App can consume all of the server's resources. With the isolation provided by LegoSDN, however, an operator can define resource limits for each SDN-App, thus limiting the impact of misbehaving applications.

Controller Upgrades: Upgrades to the controller codebase must be followed by a controller reboot. Such events also cause the SDN-App to unnecessarily reboot and lose state. Some SDN-Apps may incorrectly recreate this state [32]. Furthermore, this state recreation process can result in network outages lasting as long as 10 seconds [32]. The isolation provided by LegoSDN shields the SDN-Apps from such controller reboots. Although, designing SDN-Apps to be *stateless* can also alleviate this problem, currently, several SDN-Apps are stateful.

Atomic Network Updates: Katta et al. [19] present a mechanism to support consistent network updates; they support *all-or-nothing* semantics in the context of network updates made by SDN-Apps. LegoSDN's support for transactions supports similar semantics, and does not require any support from SDN-App developers. When an application crashes after installing a few rules, it is not clear whether the few rules issued were part of a larger set (in which case the transaction is incomplete), or not. LegoSDN can easily detect such ambiguities and roll back only when required.

4. IMPLEMENTATION

Motivated by its lush support ecosystem [3], we decided to build LegoSDN to run on the FloodLight stack and demonstrate the utility of our abstractions. The architectural changes discussed and the abstractions demonstrated are easily generalizable to other controllers such as NOX [14], OpenDayLight [6] and Beacon [12].

4.1 Prototype

We present an overview of the proposed LegoSDN architecture and compare it with existing FloodLight architecture in Figure 1. Although FloodLight's monolithic architecture is simpler to manage and visualize, the fate-sharing relationships clearly manifest themselves in the schematic. By running all components within a single process, the monolithic architecture makes it infeasible to provide any fault isolation; failure of any one component, implies failure of the entire stack.

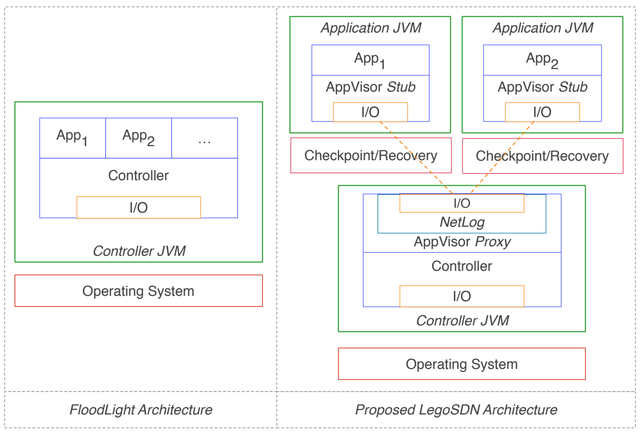

Figure 1: Schematic of LegoSDN Prototype

In LegoSDN, there are two parts to the AppVisor: AppVisor *Proxy* and AppVisor *Stub*. The former runs as an SDN-App in the controller, while the latter runs as a stand-alone application hosting an SDN-App, as shown in Figure 1. The proxy and stub communicate with each other using UDP. The proxy dispatches the messages it receives from the controller to the stub, which in turn delivers it to the SDN-App. The message processing order in LegoSDN is, for all purposes, identical to that in the FloodLight architecture.

The stub is a stand-alone Java application that launches an SDN-App. Once started the stub connects to the proxy and registers the SDN-App, and its subscriptions (what message types it intends to process) with the proxy. The proxy in turn registers itself for these message types with the controller and maintains the per-application subscriptions in a table.

The stub is a light-weight wrapper around the actual SDN-App and converts all calls from the SDN-App to the controller to messages which are then delivered to the proxy. The proxy processes these messages, calls the appropriate methods in the controller and returns the responses. In other words, the stub and proxy implement a simple RPC-like mechanism. The proxy uses communication failures with the stub to detect that the SDN-App has crashed. To further help the proxy in detecting crashes quickly, the stub also sends periodic *heart beat* messages.

For checkpointing and recovery, we use *Checkpoint and Restore In Userspace (CRIU)* [3]. The proxy creates a checkpoint of an SDN-App process (or JVM) prior to dispatching every message. In a normal scenario where the SDN-App processes the message and returns the responses, the proxy simply ignores the checkpoint created. In the event of crash, however, the proxy restores the SDN-App to the checkpoint (last known state prior to the processing of the message). Assuming the last message delivered to an SDN-App is the reason behind the crash, the AppVisor records the message

[3]FloodLight project boasts the world's largest SDN ecosystem [8]

along with the stack traces and other diagnostic information gathered during the crash, to help in triaging bugs.

The current prototype is based on the design described earlier, in Section §3, except that pertaining to NetLog. In the place of NetLog, our prototype contains a simple buffer that delays the actions generated by an application from being installed on the switch, until it is confirmed that the processing completed without failure. We note that this is not practical in a real-world environment and are working on a better implementation of NetLog that can guarantee performance and consistency.

As far as the applications are considered, we were able to successfully move applications like the *LearningSwitch*, *Hub*, *Flooder*, bundled with Floodlight to run within the stub with very minimal changes; we had to comment out use of services, viz., counter-store, the support for which is still a work-in-progress.

5. DISCUSSION & ON-GOING WORK

As part of ongoing efforts, we plan to extend LegoSDN to address the following challenges.

Handling failures that span multiple transactions: Currently, LegoSDN can easily overcome failure induced by the most recently processed event. If the failure is induced as a cumulation of events, we plan on extending LegoSDN to read a history of snapshots (or checkpoints of the SDN-App) and use techniques like *STS* [28] to detect the exact set of events that induced the crash. STS allows us to determine which checkpoint to roll back the application to.

Dealing with concurrency: SDN-Apps, being event-driven, can handle multiple events in parallel if they from multiple switches. Fortunately, these events are often handled by different threads and thus we can pin-point which event causes the thread to crash. Furthermore, we can correlate the output of this thread to the input.

Tackling non-deterministic bugs: The approaches presented in Section §3.3 can be easily modified to overcome non-deterministic bugs. For instance, LegoSDN can spawn a *clone* of an SDN-App, and let it run in parallel to the actual SDN-App. LegoSDN can feed both the SDN-App and its clone the same set of events, but only process the responses from the SDN-App and ignore those from its clone. This allows for an easy switch-over operation to the clone, when the primary fails. Since the bug is assumed to be non-deterministic, the clone is unlikely to be affected.

Avoiding violations of network invariants: Ignoring events such as switch-down, to overcome an SDN-App-crash, can result in black-holes. We argue that, in general, sacrificing the availability a few flows dependent on a switch is better than sacrificing availability of all flows dependent on the network. When unacceptable, a host of policy checkers [20] can be used to ensure that the network maintains a set of "No-Compromise" invariants. If any of these "No-Compromise" invariants are indeed affected, then the network shuts down.

Surviving deterministics controller failures: By providing fault isolation, our re-design also ensures that an application can persist despite a controller crashing. In this situation, however, the application cannot continue to function since communication with the network only happens through the controller. We, however, believe some of the techniques embodied in the design of Crash-Pad can be used to harden the controller itself against failures.

Minimizing checkpointing overheads: Crash-Pad creates a checkpoint after every event, and this can be prohibitively expensive. Thus, we plan to explore a combination of checkpointing and event replay. More concretely, rather than checkpointing after every event, we can checkpoint after every few events. When we do roll back to the last checkpoint, we can replay all events since that checkpoint.

6. CONCLUSION

Today, SDN applications are bundled and executed with the controller code as a single monolithic process; crash of any one SDN-App brings down the entire controller. Furthermore, an SDN-App crash may result in an inconsistent network, as the controller is unable to roll back network changes made by the SDN-App. We argue that these problems are symptomatic of a lack of proper abstractions between the controller and the SDN-Apps, and also between the SDN-Apps and the network

In this paper, we propose a set of abstractions for improving controller availability: AppVisor (fault isolation) and NetLog (network transactions). We demonstrate the efficacy of our abstractions by building a system, LegoSDN, that retrofits an existing controller platform to support these abstractions without any changes to either the controller or SDN-Apps. Our system allows SDN operators to readily deploy new SDN-Apps in their networks without fear of crashing the controller, and this is key to enable a thriving SDN ecosystem. We believe this system represents the first step towards a controller framework that epitomizes availability as a first-class citizen.

7. REFERENCES

[1] Big Switch Networks, Inc. http://goo.gl/sr2Vs.
[2] Big Tap Monitoring Fabric. http://goo.gl/UHDqjT.
[3] Checkpoint/Restore In Userspace (CRIU). http://goo.gl/OMb5K.
[4] FlowScale. http://goo.gl/WewH1U.
[5] FlowScale Bug Tracker. http://goo.gl/4ChWa4.
[6] OpenDaylight: A linux foundation collaborative project. http://goo.gl/1uobC.
[7] Project Floodlight. http://goo.gl/aV1E40.
[8] Project Floodlight Grows to the World's Largest SDN Ecosystem. http://goo.gl/xTslJ1.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — A Technique for Cheap Recovery. OSDI'04.

[10] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test Openflow Applications. NSDI'12.

[11] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. SIGCOMM '11.

[12] D. Erickson. The beacon openflow controller. HotSDN '13.

[13] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-Class Entities. (TR1771), 06/2012 2012.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3), 2008.

[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 '14.

[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 '13.

[17] 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 '13.

[18] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained Fault Tolerance Using Device Checkpoints. ASPLOS '13.

[19] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. HotSDN '13.

[20] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. HotSDN '12.

[21] 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'10.

[22] M. Kuźniar, P. Perešíni, N. Vasić, M. Canini, and D. Kostić. Automatic Failure Recovery for Software-defined Networks. HotSDN '13.

[23] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.

[24] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. SOSP '05, 2005.

[25] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. HotSDN '13.

[26] 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'04.

[27] 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 '12.

[28] 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 '14, 2014.

[29] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4), Nov. 2006.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, Feb. 2005.

[31] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. HotNets '09.

[32] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. HotSwap: Correct and Efficient Controller Upgrades for Software-defined Networks. HotSDN '13.

[33] D. Williams and H. Jamjoom. Cementing High Availability in Openflow with RuleBricks. HotSDN '13.

[34] A. P. Wood. Software Reliability from the Customer View. *Computer*, 36(8):37–42, Aug. 2003.