Tolerating SDN Controller Failures with SDN-SafeMode.

Balakrishnan Chandrasekaran and Theophilus Benson
Duke University

ABSTRACT

Today, the failure of an SDN controller application deprives the network of its control functions and renders it effectively inoperable. Given the increasing importance of networks, in cloud services to enterprise customers and end-users by web services such as online social networks, this unavailability is unacceptable.

In this paper, we make the argument that the SDN controller stack, should be re-designed to support better abstractions and provide greater control over applications for features such as better reliability. We propose a system, SDN-SafeMode, that embodies these principles by providing an isolation layer between applications. SDN-SafeMode proactively detects events that may trigger application crashes and ignores or corrects the offending events to ensure SDN applications do not crash. We present an initial implementation of SDN-SafeMode based on the FloodLight controller and show that SDN-SafeMode introduces minimal overheads. Furthermore, we show that the controller refactoring required to support SDN-SafeMode enables a large array of features currently absent.

1. INTRODUCTION

Despite Software Defined Network's, SDN, proven benefits, many companies remain hesitant in adopting it. Two of the foremost issues that hamper to SDN adoptions are reliability and fault tolerance []. **Asim says: I remember reading security is the biggest problem** . In a traditional SDN deployment, there are four main failure scenarios: controller server failures (hardware failures), controller crashes (bugs in the controller code), network device failure (switch, application server or link failure), and application crashes (bugs in the application code). While much work has gone into debugging the SDN ecosystem [2, 4, 5, 8], little work has gone into hardening SDN again these failures.

Most recent work focuses on overcoming hardware failures, namely controller server failures [6, 9] and network device failures [10, 20]. Unfortunately, software failures remain un-addressed. Furthermore, traditional operating system and application-server fault-tolerant techniques like reboot or replay [1, 23] can not simply be applied to the SDN control plane for three reasons.

First, both the network and the applications both contain state, rebooting [1] the application will eliminate this state leading to inconsistency issues. Second, unlike with in an OS, where-in a web-server can be rebooted without affecting other applications on the same OS, in an SDN environment, the SDN app is intermingled with the controller, so a reboot affects all other apps. Third, replaying events [23] to recover this state implicitly assumes that the bugs are non-deterministic and thus wouldn't be encountered during replay. However, we argue that given the event driven nature of SDN applications, bugs will most likely be deterministic. In fact a brief survey of controller and application forums confirms this. **Asim says: this is a very strong claim: all events, no threading. are you sure a single application does not have multiple threads in a controller library?**

In this work, we focus on application crashes because the controller code represents a common layer that is highly re-used and thus has a lower probability of bug, whereas the application code is more likely to be provided by third party entities with limited testing. A trend that is expected to become even more prevalent given the recent success of project FloodLight [19] and the development OpenDayLight project [17]. Furthermore, as stated earlier the lack of a clean separation between applications and controllers, introduces a fate-sharing relationship where-in an application crash induces a controller crash.

The principle position of this paper is that availability is the upmost concern – second only to perhaps security. To this end, we propose a re-architecture of the controller architecture that eliminates fate-sharing between applications through isolation and that relaxes the correctness of the polices implemented by an application with a *best-effort model*. We argue for a model where-in applications may not process all network events, to maintain controller availibility.

The challenge lies in determining when to compromise correctness, how much to comprise, and how to ensure safety while compromising correctness. Where-in safely is defined as: at most one copy of a message is sent from the application to the network that the application is aware of all messages sent to the network. Thus during an application rollback, the network must also rollback. To overcome these challenges, we leverage properties of SDN applications, namely that they are event driven. This allows us to isolate and detect crash-triggering events and in turn determine when to compromise. In addition given the nature of network events, we can allow the operator to choose between various levels of comprise. For example, if a link down event triggers an application crash. The operator can choose compromise 100% by ignoring the event, or 0% compromise by crashing the application, or something in between by transforming the port down event into a switch down event. Finally, for safety, we implement a network roll-back mechanism which leverages the insight that most SDN events are invertible. **Asim says: is this true for all events, then say all SDN events are invertible** For example, installing a rule is invertible by deleting the rule.

To this end, we propose SDN-SafeMode a system that embodies the described principles. SDN-SafeMode provides isolation layer between applications and introduces a fault tolerance layer that detects crash triggering events and overcomes them through event transformations. In SDN-SafeMode applications run unmodified, thus shielding application programmers from having to understand the functionality and recovery techniques provided by SDN-SafeMode. SDN-SafeMode provides a policy language for operators allowing them to specify the set of applications and network events whose accuracy and efficacy they are willing to compromise on. This enables operators to trade-off between control plane available and correctness of network policy. Thus allowing the network to persevere in the face of application crashes and provide slightly degraded functionality until a fix or patch can be deployed. As a proof-of-concept, we re-architected the FloodLight controller to support SDN-SafeMode. Using this prototype, we demonstrate that SDN-SafeMode imposes minimal overheads and requires no application modifications.

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. Then present an overview of the fault model within SDN environments coupled with a discussion of related works and relevant techniques from other domains; most notable 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. Figure 1 illustrates the layers in a canonical SDN control platform. We note that in platforms such as NoX [24] and FloodLight [18], failures of any component in the stack renders the control plane unavailable. Perhaps surprisingly, in a FloodLight deployment, Figure 2, an unhandled exception in one application will result in the failure of other applications and the controller itself. Similarly, a segment-fault in a NoX controller application takes down both the controller and application. The applications have a disastrous effect on the entire platform because of a lack of isolation. Unsurprisingly, the failure of either the server, O.S, or controller platform will also render the control plane unavailable.

Figure 1: Generic Controller Stack

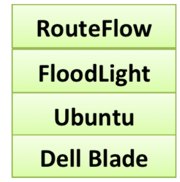

Figure 2: FloodLight Stack

While certain companies, such as BigSwitch [12], aim to provide this entire stack 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 applications. For example, FloodLight [18], a Java controller platform released by BigSwitch, boasts a large array of applications ranging from performance enhancing [**?**, **?**, 22], to security enforcement [**?**, **?**]. In Table 1, we present a small list of FloodLight applications, their purpose, and indicate whether they are third-party applications or applications developed by the developers of the controller.

This table reinforces the notion that the SDN ecosystem embodies an a 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 [17] and the SDN-hackathons hosted by SDN-Hub are already promoting this diversity.

Theo says: there's a point missing above — need to discuss bugs in opensource or third party application

Unfortunately, given the novelty of these applications, many lack public support forums or a public facing bug tracker. Luckily, we were able to find a public bug-track for FlowScale [16]. 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 remains unclosed. While these numbers do not generalize to other SDN applications, we note that extensive studies on software engineering practice show that bugs are prevalent in most application and even worst most bugs in production quality code do not have fixes [].

Theo says: do we need more apps in this table? can we find more?

2.2 Fault Tolerance in SDN Networks and Operating Systems

Application	Developer	Purpose
RouteFlow [22]	Third-Party	Routing
FlowScale [15]	Third-Party	Traffic Engineering
BigTap [13]	BigSwitch	Security
Stratos [?]	Third-party	Cloud Provisioning

Table 1: Survey of Popular SDN Applications

SDN Fault Tolerance: Fault tolerance within the SDN environment has been a subject of research for quite some time; however, research has focused on shielding application developers from writing application code that needs to reacts to switch failures [10, 20], link failures, or application server failures [25]. Unfortunately, little has been done to shield the network and the controller from application failure. It is precisely this that SDN-SafeMode focuses on.

Recent approaches [6, 9] to tackle controller failure, focus on recovering from hardware failures: applying Paxos [11] to ensure robustness to controller server failures (hardware failures). Unfortunately, little has been done to detect or recover from software failures. Most specifically failures induced by bugs within the controller platform or the controller applications. In this paper, we argue these problems are equally as important as hardware failures (server failures) and provide a solution that tackles application failures. We leave controller platform failures for future work.

We note that high-level controllers restrict and control the application syntax thus potentially eliminating errors. However, current commercial deployments are dominated by modular or low-level controllers due to their low adoption barrier [19].

Bugs in SDN Code: Recent work [4, 8] to detect bugs in SDN applications focus on detecting bugs that generate output that violates certain invariants; essentially focusing on bugs that impact the safety or correctness of the network. Instead, we attack an orthogonal problem, that of detecting bugs that result in controller failures; essentially focusing on bugs that impact the availability of the controller.

Operating System Fault Tolerance: Our approach builds on several key operating system principles, namely; isolation [1], failure oblivious computation [21], and shadowing [7, 23]. However, these approaches either assume that bugs are non-deterministic and thus can be fixed by a reboot [1, 7, 23] or that safety and completeness can be compromised in the face of availability [21]. Similarly, we compromise completeness for availability but focus on ensuring safety in all situations and ensuring that network operators have control over the degree of completeness that can be compromised for availability.

More concretely, we utilize SDN's programmatic control to roll-back and undo the network events created by failure inducing input thus ensuring safety. In addition, we provide operators with an interface for specifying the degrees of correctness they are willing to tolerate on a per-application basis.

3. DESIGN

In this section, we describe the three key design principles that SDN-SafeMode embodies.

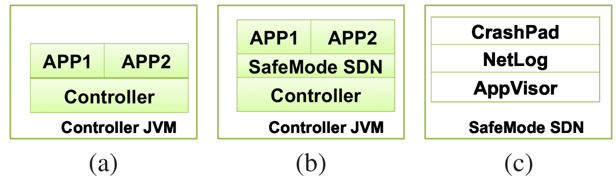

Figure 3: (a) Overview of the floodlight controller. (b) Overview of a floodlight controller modified to support SDN-SafeMode. (c) Architecture of SDN-SafeMode.

SDN-SafeMode provdies exports programming the application-controller interface with *Appvisor*. We demonstrate its usefulness by building *Crashpad* and *Netlog*, that provide failure detection and recovery support.

3.1 AppVisor: Isolating Applications

To ensure the availability of the control plane, we must isolate applications from each other. Thus ensuring that un-handled-exceptions or segment-faults within one application, do not affect other applications or the controller. Operating systems deal with a similar issue by modularizing and separating the address space of server-application from one-another. Similarly, we separate the address space from SDN-application from each other and more importantly from the controller. This separation, isolates the applications from each other and similarly isolates the controller from them.

Unlike, with Operating Systems we make the fundamental assumption that applications may become momentarily unavailable due to failures. As such, we require that communication between controller-application include both a timeout/retry mechanism. Apart from improving availability, this separation opens up a wide range of novel use cases: per-application resource limits, application migration, multiversion software testing, and allows for certain seamless controller upgrades. We elaborate on these use-cases in section 3.4.

We note that this separation and subsequent interposition, may introduce additional latency into the control-loop but we argue that such additional latency is acceptable as controllers introducing the controller into critical path slows down the network by an factor four [3] and thus controllers are not involved performance sensitive operations.

3.2 CrashPad: Detecting and Isolating Failures

SDN applications are event driven and process a single network event at a time. We assume these network events to be independent of one-another that simplifies the fault tolerance expectations. Faults occur in network due by events either from the network or from services running on the controller. Thus in many situations detecting the cause of failure can be a simple as determining the last event processed by an application before failure. **Asim says: t** he event driven confuses me

CrashPad needs to maintian an uncorrupted state to ensure the application can be rolled back upon failure. To do so, Crashpad clones application logic and feeds it the network events prior to applying them to original applications. If a failure occurs, Crashpad reverts to the clone and applies a range of correction techniques to the *offending event* before sending to the clone. **Asim says: I am confused what is clone and what is original and what is reveretd. please clarify** Concretely, the clone is an in-memory check-point of the application and thus the process of reverting the clone is essentially rolling back to the last check-point.

As such, the first step to recovery is ensuring that potential recovery does not violate operator preferences then applying applying recovery actions. In our initial straw-man, we propose two recovery actions: The first, a naive and lightweight approach, is to ignore the network event which causes the crash. The second, a heavyweight and invasive approach, is to apply symbolic execution techniques to determine the which changes to the internal state will allow the application to process the event without crashing. We limit changes to internal state to changes in the topology which can be accomplished by sending the application switch or port drop events. We note that while symbolic execution techniques are time consuming, they can be run apriori and cached for future analysis.

In addition to simply dropping/ignoring the event, we can apply transformations to the event. For example, if we detect that a port-down event is the *offending event*, we may transform it to a switch down event. Thus essentially enlarging the scope of the event with the hope that the application will run bug free given the transformed event.

Simultaneously with roll-backing, we generate a problem ticket from the captured stack-traces or core dump, controller logs and the *offending event*. This ticket can in-turn be submitted to the developer forum for future analysis.

We note that dropping or modifying the event does not itself solve the problem but instead creates a temporarily band-aide until a solution can be created. We find this to be acceptable as 80% of bugs in production quality software do not have fixes at the time they are encountered []. Furthermore, we acknowledge that for certain SDN application, such as security applications, network operators may be unwilling to ignore certain events. To account for this, we provide 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 ignore.

3.3 NetLog: Ensuring Safety

To ensure the safety of the network, in addition to tracking and detecting failure causing input, the system must provide a mechanism that allows recovery. Failure to do this could result in network black-holes, cycles, and violation other network invariants. Thus, SDN-SafeMode includes a rollback module, NetLog, which tracks the network events that an application emits in response to a input event from the controller.

Our recovery solution leverages the insight that each control messages that modify network state is invertible: more precisely, for every state altering control message, A, there exists another control message, B, that undoes A's state change. Unfortunately undoing a state change is imperfect as certain network state is lost. For example, while it is possible to undo a flow delete event, by adding the flow back to the network, the flow time-out and the flow counters are lost. 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. More specifically, flow counters cannot be inserted into the network. For counters, NetLog stores the old counter values in a counter-cache and when a statistics message from the switch to the application is detected, NetLog updates the counter value in the message with the sum of the value within the message and the value in its counter-cache.

Together these three modules allow us to isolate applications, explicitly eliminating faith-sharing, and to detect and eliminate error inducing events, effectively improving the availability of individual applications.

3.4 Enabling novel techniques

The SDN-SafeMode enables 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 independent teams develop identical versions of the same application. The idea being that most teams will implement the functionality correctly. Thus if all versions are run, the most popular out is the correct output. The App-Cloner can be used to distribute events to the different versions and to compare output from the different versions.

Per Application Resource Limits: Currently, there is no way to limit the resource consumption of individual controller applications. Consequently, a rouge application can consume all of the server's resources. However with the isolation provided by SDN-SafeMode, a network can place limits on the resource consumption of each application, at the process level, further limiting the impact of rouge applications.

Controller upgrades: Upgrades to the controller codebase must be followed by a controller reboot. Currently, due to the lack of separation, such events eliminate application state and requires the application to recreate this state. Sadly, some applications may incorrectly recreate this state [**?**]. Furthermore, the process of state re-creation can result in network outages lasting as long as 10 seconds [**?**].

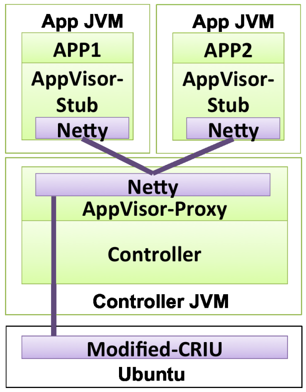

Figure 4: Prototype Implementation of SDN-SafeMode

Fortunately the application isolation layer proved by SDN-SafeMode ensures that a controller reboot does not impact the application or its state.

4. PROTOTYPE AND EVALUATION

Motivated by its lush support ecosystem, its frequent use as the controller platform of choice by SDN application developers, and the fact that FloodLight project boasts the world's largest SDN ecosystem [19] we decided to modify the FloodLight controller platform to support SDN-SafeMode. We note that these changes are easily generalizable to other modular controllers such as NOX and OpenDayLight and Beacon.

Furthermore, to illustrate the overhead both in terms of performance and in terms of developer effort, we run a number of *unmodified* FloodLight applications on our re-architected controller.

4.1 Prototype

In Figures 4 and 3 (a), we illustrate the architecture of a FloodLight controller before (3 (a)) and after (4) our modifications. We note that the key components of our prototype are: Controller Stub,App Loader, CheckPoint and Reload, App Proxy. The Controller Stub, App Loader and App Proxy together allow us to isolate applications and together make up the AppVisor. The error detection and recovery code lies within the App Proxy, which uses CRIU [14] checkpointing for cloning. An application crash is detected through the failure of the connection between the Controller Stub and the App Proxy.

Our current prototype lacks a NetLog. Instead we buffer messages from the Application to the network, in the App Proxy, and only release them when the Controller Stub sends a special message indicating that the application has completed event processing. We confess that while this approach is feasibility it is not ideal for two reasons: 1.) the buffering of messages could result in Application deadlock. and 2.) buffering will result in a flood of message between the controller and the switches. Prior work [3] has shown that batching messages to a switch can overwhelm the switch CPU and deteriorate the network performance. Thus in future implementations we intend to replace the buffer with an appropriate implementation of NetLog.

Next, we describe the implementation of each component:

- App Loader: This starts the FloodLight applications, each in a separate JVM with the Controller Stub. To minimize the size of the JVM, the App Loader loads only the minimal set of Java classes into the JVM.

- CRIU: Currently, we leverage CRIU which uses */proc/* to capture process information and store it to file. We've altered CRIU to store the process information in memory. Thus speeding up the action of cloning an application, AKA backing-up, and of reverting to a clone, AKA rolling-back, in the event of failure.

- Controller Stub: Currently, this implements key controller interfaces and intercepts calls from the application, serializes the calls, and transfer them to the App Proxy within the controller JVM. In the reverse, it accepts controller events from the App Proxy and deserializes them and call the appropriate application functions. For example, traditionally for a packet-in event FloodLight calls an applications *receive* method. However, now the FloodLight calls the App Proxy's *receive* method, the App Proxy serializes and sends it over to the Controller Stub which then deserializes and in turn call's the application's *receive* method.

  In addition, the Controller Stub alerts the App Proxy when event-processing is complete: we identify the boundary of event processing as the end/return of the application's event-handler. For example, when the application's *receive* method returns.

- App Proxy: As the heart of our prototype, this implements the set of interfaces that the application implements and it similarly registers for the events that the application registers for (this information is passed to the App Proxy by the Controller Stub.) Similar to the Controller Stub, the proxy uses Netty for communication and it serializes/de-serializes messages sent over Netty. Unsurprisingly, the App Proxy performs the reverse of the Controller Stub.

  Prior to sending an event, from either the network or a service, to the Controller Stub, the App Proxy uses the CheckPoint and Reload to clone the application's JVM. Should the App Proxy detect a failure, via a Controller Stub disconnect, it rolls back to the last checkpoint. At most, one clone needs to be maintained per-application and this clone can be deleted when the Controller Stub indicates successful event process.

4.2 Evaluation

xx xxx
(a) (b)

Figure 5: Comparison between a vanilla FloodLight instance and one modified to support SDN-SafeMode.

We implemented and tested 3 unmodified FloodLight Applications; a simple MacTracker, a hub, and a switch. Next, we examine the overheads placed by our technique.

We examine overheads in terms of delay placed on the control loop. To do this, we measure the time between receiving a network event and when an output is generated by the controller. For the MacTracker, this is the time between Packet-In events and print out statements. For the other 2 applications, this is the time between Packet-In events and Flow-Mod events. In Figure 5 (b), we present the results for the most complicated application, the switch, however, these results generalize to the other applications. We observe that SDN-SafeMode introduces at most a 100 μ second latency.

Our results, show that highly availability is achievable with minimal costs. Given that involving the controller introduces at least a 10ms latency to packet processing, we argue that the additional 100μ second latency, or 1% inflation in latency, is both acceptable and ultimately inconsequential. We believe that these costs can be further reduced when our approach is streamlined into the controller's core design.

5. DISCUSSION

How does SDN-SafeMode overcome non-deterministic bugs?

Can using SDN-SafeMode result in network loops or blackholes?

What happens if the controller crashes?

6. CONCLUSION

SDN-SafeMode address the need for a dependable and highly-available controller applications by detecting application crashes in real time and modifying network events to eliminate the crash. In essence, the modifications made to the network events impact the correctness of the policies being implemented. Thus, SDN-SafeMode presents a novel point in the design space of SDN systems where-in trade-offs are made between correctness of the network and availability of the controller applications. We implemented a prototype of SDN-SafeMode based on the popular FloodLight controller platform and explored the performance implications and overheads of SDN-SafeMode. Ultimately, we believe SDN-SafeMode is the first step towards a bug-crash-free controller framework that epitomizes availability as a first-class citizen.

7. REFERENCES

[1] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[3] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[4] 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. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 71–85, Seattle, WA, 2014. USENIX.

[5] 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. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 37–42, New York, NY, USA, 2013. ACM.

[6] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[7] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 473–484, New York, NY, USA, 2013. ACM.

[8] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 49–54, New York, NY, USA, 2012. ACM.

[9] 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.

[10] M. Kuźniar, P. Perešíni, N. Vasić, M. Canini, and D. Kostić. Automatic failure recovery for software-defined networks. HotSDN '13, pages 159–160, New York, NY, USA, 2013. ACM.

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

[12] Big Switch Networks, Inc. `http://www.bigswitch.com`.

[13] Big Tap Monitoring Fabric. `http://bigswitch.com/products/big-tap-monitoring-fabric`.

[14] CRIU. `http://criu.org/Main_Page`.

[15] FlowScale. `http://www.openflowhub.org/display/FlowScale/FlowScale+Home`.

[16] FlowScale Issues. `http://bugs.openflowhub.org/`.

[17] OpenDaylight: A linux foundation collaborative project. `http://www.opendaylight.com`.

[18] Project Floodlight. `http://www.projectfloodlight.org/`.

[19] Project Floodlight Grows to the World's Largest SDN Ecosystem. `http://www.bigswitch.com/press-releases/2013/03/26/project-floodlight-grows-to-the-worlds-largest-sdn-ecosystem-globa`

[20] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. HotSDN '13, New York, NY, USA. ACM.

[21] 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. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[22] 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. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 13–18, New York, NY, USA, 2012. ACM.

[23] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, Nov. 2006.

[24] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying nox to the datacenter. In L. Subramanian, W. E. Leland, and R. Mahajan, editors, *HotNets*. ACM SIGCOMM, 2009.

[25] D. Williams and H. Jamjoom. Cementing high availability in openflow with rulebricks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 139–144, New York, NY, USA, 2013. ACM.