Tolerating SDN Application Failures with LegoSDN.

Balakrishnan Chandrasekaran
Duke University
balac@cs.duke.edu

Theophilus Benson
Duke University
tbenson@cs.duke.edu

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

Despite Software Defined Network's (SDN) proven benefits, there remains significant reluctance in adopting it. Amongst the issues that hamper SDN's adoption 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, application server or link failure), and SDN application (SDN-App) crashes (bugs in the application code). While much focus has been on overcoming hardware and network device failures [7–9] and debugging SDN-Apps and/or the ecosystem [2,4–6], 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 [1] or replay [10] cannot be applied direcly to the SDN control plane: certain fundamental assumptions do not hold true in the SDN ecosystem and we cite here three such issues. First, both the network and the SDN-Apps contain state, and rebooting [1] the SDN-App will eliminate this state and consequently, introduce inconsistency issues. 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, replaying events [10] 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 application crashes because the controller code represents a common layer that is highly re-used and thus has a lower probability of bugs. Further, application code is more likely to be provided by third party entities with limited testing – a trend that is expected to become more prevalent given the recent success of open-source controllers, e.g, HP's SDN App Store.

The principle position of this paper is that availability is of upmost concern – second only to security. There exists a *fate-sharing* relationship between the SDN-Apps and controllers, where-in the crash of the former induces a crash of the latter, and thereby, affecting availability. The issue is symptomatic of the lack of proper abstractions between the controller and the SDN-Apps. Our solution to this issue is a radical re-design of the controller architecture centering around a set of abstractions that enable two key features:

- Promote isolation and eliminate the fate-sharing relationship between the entities of an SDN ecosystem.
- Support the notion of network-wide transactions to manage the explicit and implicit relationships between various SDN-Apps running on a controller; transactions help in guaranteeing consistency across SDN-Apps.

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 the events. Ignoring or transforming events, however, compromises the SDN-Apps' ability to completely implement its policies; thus, compromises *completeness*. The challenge lies in detecting faults (determining when to compromise *completeness*), in determining how to overcome the faults (how much to compromise), and in ensuring safety while performing fault recovery (compromising *completeness*).

Our proposal calls for a re-design of the controller to support two key abstractions: (1) isolate SDN-Apps running on a controller from one another and from the controller itself, and (2) bundle operations issued by the controller or SDN-Apps with implicit or explicit dependencies into one atomic transaction that facilitates rollback or recovery operations without sacrificing consistency. We present LegoSDN that demonstrates the power and utility of these abstractions.

2. RE-THINKING SDN CONTROLLERS

We argue that the controller's inability to tolerate bugs in applications is symptomatic of a larger endemic issue the design of a controller's architecture. We present LegoSDN, a system that transparently modifies the application-controller interface via two components: AppVisor (§2.1) and NetLog (§2.2). To offer a glimpse of the new capabilities that LegoSDN provides, we discuss the design of Crash-Pad (§2.3), a system that provides failure detection and recovery support. LegoSDN's components are regular SDN-Apps running within the controller and serve to demonstrate the benefits of a controller re-design supporting our abstractions.

HotSDN'14, August 22, 2014, Chicago, Illinois, USA.
ACM 978-1-4503-2989-7/14/08.
http://dx.doi.org/10.1145/2620728.2620781.

2.1 Isolation and Modularity: AppVisor

An SDN controller should be designed and architected like any platform that allows third-party code to execute, in isolated modules with clearly defined fault and resource allocation boundaries. Further, we make the fundamental assumption that applications may become momentarily unavailable due to failures. The assumption is in fact a requirement to catalyse the adoption of the SDN ecosystem; it allows developers to rapidly protytpe and deploy applications.

We recommend that the communication protocol between the controller and SDN-Apps include both a time-out and retry mechanism. Aside from improving availability, we argue that this isolation opens up a wide range of novel use cases: per-application resource limits, application migration, multi-version software testing, and allows for certain seamless controller upgrades. We note that this isolation introduces additional latency into the control-loop but argue that such additional latency is acceptable as introducing the controller into the critical-path already slows down the network by a factor of four [3].

We designed the AppVisor by building on well studied and widely used isolation techniques used in Operating Systems. The AppVisor separates the address space of SDN-Apps from each other and, more importantly, from the controller by running them in different processes (or JVMs). **In the context of fault-tolerance** this isolation ensures that crashes of any SDN-App do not affect other SDN-Apps or the controller.

2.2 Network Transactions: NetLog

Network policies are atomic and often span multiple devices thus requiring many network actions. Controllers, however, treat these actions (or events) independently. We propose that these events be treated together as an atomic operation; failure of either an SDN-App or action generated by one should trigger a network-wide rollback on all related events. The controller should expose the network as a transactional system that supports the grouping of multiple events into an atomic operation (or update).

NetLog leverages the insight that each control messages that modify 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 time-out 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.

2.3 Surviving amidst failures: Crash-Pad

Consider the two trends: (1) 80% of bugs in production quality software do not have fixes at the time they are encountered [11], and (2) bugs in SDN applications will like be deterministic. Leveraging these trends, we aim to build Crash-Pad, a system that overcomes application failures by detecting failure-triggering events and ignoring or transforming these events. Crash-Pad builds on the fault containment provided by AppVisor and the atomic updates provided by NetLog.

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

When to compromise completeness? SDN applications are largely event driven and in most situations, the cause of a failure is just the last event processed by an application before failure. We clone an application (or take a snapshot of its state) prior to its processing of an event and hence, should failure occur, we can revert to the state prior to the failure. However, replay of the offending event most likely will cause the application to crash. Therefore, we apply a *completeness*-compromising transformation on the offending event prior to the replay.

How much of completeness to compromise? We envision Crash-Pad to provide a simple interface through which operators can specify policies (*completeness*-compromising transformations) that dictate how to compromise *completeness* when a crash is encountered. In the initial straw-man, we aim to make available two basic policies: (1) *Absolute Compromise* ignores the event (sacrificing *completeness*) which causes the crash and makes SDN-Apps *failure oblivious*, and (2) *No Compromise* allows the SDN-App to crash, thus sacrificing availability to ensure *completeness*.

How to specify the availability-completeness trade-off? For security applications, network operators may be unwilling to compromise on the *completeness* of certain events. To account for this, we envision 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 crashes or compromises? Our goal is to make the SDN-Apps and not the SDN-App developers oblivious to failures, thus, when subverting a failure, Crash-Pad will generate a problem ticket from the captured stack-traces (or core dump), controller logs and the offending event. The ticket can help developers to triage the SDN-App's bug.

3. REFERENCES

[1] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — A Technique for Cheap Recovery. In *OSDI 2004*.

[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test Openflow Applications. In *NSDI 2012*.

[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 *SIGCOMM 2011*.

[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 *NSDI 2014*.

[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 *HotSDN 2013*.

[6] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *HotSDN 2012*.

[7] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI 2010*.

[8] M. Kuźniar, P. Perešíni, N. Vasić, M. Canini, and D. Kostić. Automatic Failure Recovery for Software-defined Networks. In *HotSDN 2013*.

[9] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *HotSDN 2013*.

[10] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. *ACM Trans. Comput. Syst.*, 24, Nov. 2006.

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