Cross-Layer Troubleshooting for the Software-Defined Networking Stack

Colin Scott
UC Berkeley

Andreas Wundsam
ICSI

Scott Shenker
ICSI & UC Berkeley

ABSTRACT

The predominant technique for troubleshooting bugs in software-defined networks, log analysis, is tedious and error-prone. We argue that a more principled approach should be built around identifying inconsistencies between lower-level network configuration and higher-level policies dictated by control applications. Towards this end we present cross-layer correspondence checking, a mechanism to automatically detect and isolate policy-violations due to bugs in the SDN platform. We observe, however, that in distributed systems such as SDN transient policy-violations due to failures and delays are inevitable. Our simulation-based replay analysis framework augments correspondence checking by helping troubleshooters distinguish persistent from harmless ephemeral policy-violations. We show that these mechanisms in combination can be used to quickly identify the root cause of difficult errors, including isolation breaches, faulty failover logic, and consistency problems between replicated controllers.

1. INTRODUCTION

The SDN platform's $raison\ d'\hat{e}tre$ is to hide complexity from control applications. To this end, modern platforms perform replication, resource arbitration, failure recovery, and network virtualization on the control application's behalf.

While these measures are effective at simplifying control applications, they do not remove any complexity from the overall system. Rather, they merely move the complexity from control applications into the underlying SDN platform.

As in any software system, complexity increases the probability of bugs. When network operators encounter erratic behavior in their network they are currently forced to manually trace through multiple layers of abstraction: policy-specification, virtualization logic, distribution logic, and network devices. SDN system developers' hope is that the platform will stable, such that operators can safely assume correctness of lower layers. Nevertheless, the evolution of SDN is far too nascent to provide stability today.

Besides *ad-hoc* measurement tools, the predominant troubleshooting mechanism in SDN is log analysis: manually specifying log statements at relevant points throughout the system; collecting and aligning distributed log files; and analyzing the results *post-hoc* when a error is encountered in production. Log analysis is an important component of the troubleshooting process, but it is deficient as a stand-alone solution: logs events are enormous in number, impossible to aggregate into a single serial execution of the system, and often at the wrong level of granularity.

Recent work has contributed troubleshooting techniques focused on the highest (control application) and lowest (dataplane forwarding tables) layers of the SDN stack. NICE applies concolic execution and model checking to SDN control applications, thereby automating the testing process and catching bugs before they are deployed [5]. Anteater [17] and HSA [14] introduce mechanisms for checking static invariants in the dataplane. Nonetheless, no automated troubleshooting mechanism exists for problems that span multiple layers of the SDN stack, specifically those caused by bugs in the platform itself.

In this paper we present an algorithm for verifying correctness of the platform. We observe that verifying correctness of the SDN platform is equivalent to showing that high-level policies specified by control applications correspond with low-level network configuration. Our algorithm, which we term 'correspondence checking', leverages the structure of the SDN platform (graphs at every layer) to enumerate all policy-violations at any point in time, and localize their root cause to a particular component of the system.

Correspondence checking does not suffice on its own; like any distributed system, transient policy-violations due to failures and delays are common, especially at large scale. We present simulation-based replay analysis to augment the diagnostic information provided by correspondence checking. Simulation-based replay analysis allows troubleshooters to differentiate transient from persistent policy-violations by tracking the life cycle of problems both forward and backward in time.

We demonstrate that these mechanisms combined can be used to quickly identify the root cause of difficult errors, including isolation breaches, faulty failover logic, and consistency problems between replicated controllers. We have implemented prototypes of correspondence checking and simulation-based replay analysis, and made the code publicly available [2].

The rest of this paper is organized as follows. In §2, we present an overview of the SDN stack and its failure modes. In §3 we present correspondence checking and simulation-based replay analysis in detail. In §4 we present a use-case and a preliminary performance evaluation. Finally, in §5 we discuss related work, and in §6 we conclude.

2. SDN OVERVIEW

In this section we provide an overview of the SDN architecture and examples of errors observed in production software-defined networks.

2.1 SDN Architecture

Modern SDN controllers are constructed as three separate layers, depicted in Figure 1: the lowest layer maintains the state of each switch in the network; the virtualization layer abstracts the details of the physical network into a simple graph representation; and the control application layer specifies network policies in terms of the virtual graph. We describe the details of these components below.

The lowest layer of the SDN stack maintains a graph data-structure known as the 'physical view', which has a one-to-one correspondence with the physical network. When a higher layer specifies a policy change on the physical view, synchronization logic generates configuration changes and sends it to the corresponding network devices. Conversely, when a state change occurs in the network, this component notifies upper layers of the change.

The virtualization layer facilitates a concise specification of intended network behavior by abstracting the physical view into a simplified graph. A common pattern is to represent an entire datacenter network as a single logical switch [6]. In this manner, operators can specify routing, access control, and QoS policies by configuring a single forwarding device; the platform then maps the configuration onto sequences of forwarding elements in the physical network.

Virtualization additionally facilitates multi-tenancy, and isolates control applications from the specifics of the underlying network. Ideally, each application is reduced to a state-less, side-effect free function [21]:

$$f(view) \rightarrow configuration$$

Production software-defined networks may consist of thousands of network devices. Modern SDN platforms replicate control across cluster(s) of servers for fault tolerance scalability. Onix [15], for example, partitions a graph of the network state across either an eventually-consistent DHT or a transactional database. In this manner control applications can make their own tradeoffs in choosing consistency models, degree of fault tolerance, and other properties.

SDN controllers can be further categorized according to their flow installation model: proactive or reactive. Proactive controllers pre-compute forwarding tables for the entire network, and only push down updates periodically to react to link failures, changes in traffic mix, *etc.*. In contrast, reactive controllers forward all new flows to control servers. After a control decision is made, a flow entry is installed in the ingress switch, and the packet is forwarded along.

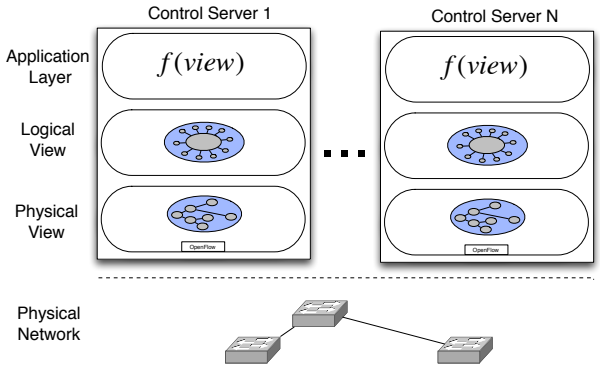

Figure 1: Depiction of the layered SDN stack.

Production SDN deployments are commonly proactive, primarily due to the large scale of datacenter networks and the current capabilities of forwarding hardware. We focus on proactive controllers for the remainder of this paper, although our troubleshooting mechanisms are also applicable to reactive applications.

2.2 Platform Failure Modes

Here we provide a few examples of platform failures. Note that we do not address failures in the highest (control application) and lowest (dataplane forwarding tables) levels of abstraction. Previous work exists for finding and troubleshooting faults in the application layer [5] and the forwarding plane [17].

As described in the previous section, modern SDN platforms differ from 'first-generation' controllers such as NOX [13] in two dimensions. First, they extend vertically by providing a virtualization layer on top of which control logic resides. Second, they extend horizontally by distributing state across multiple control servers. Errors arise from both of these architectural additions. Moreover, the sheer scale of production SDN deployments amplifies the frequency and severity of errors; consider that Microsoft reports 36M minor to fatal error events over one year across 8 datacenters, which implies 8.5 error events per minute per datacenter [12]. Errors due to virtualization, controller coordination, scale, and the confluence of these factors are the main focus of our work.

Virtualization. Virtualization errors often result from a mismapping between the logical graph and the physical network topology. Consider that an entire datacenter network (up to 10,000 switches) may be abstracted into a single logical switch. Especially in the presence of hardware failures, the mapping between the logical switch and the physical topol-

ogy is highly complex. In a multi-tenant environment, the mapping is often many-to-many [6]; each tenant specifies policies on their own logical switch, while the platform multiplexes each slice over the same physical network. If the logical address spaces are not disjoint, or a change in the physical network causes the paths to temporarily overlap, breaches of tenant isolation or failure to properly install configuration changes may occur,

Controller Coordination. Coordination between controllers entails the same classes of error conditions that plague general distributed systems; inconsistent reads and writes, race conditions over message arrivals, and unintended consequences of failover logic are common. As an example, suppose a controller fails, and all the switches under its purview are adopted by a new control server. If the new parent neglects to properly query the switches for their current state, or reads stale control information from the data store, it may inadvertently install conflicting flow entries in the network. Errors may also result from non-disjoint partitioning schemes between controllers, or incorrect delegation of control for different portions of the network. High churn in the network topology due to VM migration and hardware failures exacerbate these issues.

3. APPROACH

In this section we present two mechanisms to facilitate the troubleshooting process. Correspondence checking allows troubleshooters to isolate the cause of policy-violations to a particular layer. Simulation-based replay analysis allows troubleshooters to isolate relevant events throughout the system execution. We provide the details of these techniques below.

3.1 Correspondence Checking

Platform correctness can be expressed as a simple invariant: the policy specified by the application layer should correspond to the configuration of the physical network. We check this invariant by applying the virtual packet algebra pioneered in headerspace analysis [14].

Formally, each layer of the SDN stack can be represented as a graph, $G = (V, E)$. Packets are series of bits, $h \in \{0,1\}^L = H$, where L is the maximum number of bits in the header. Upon receiving a packet, forwarding elements apply a transformation function, potentially modifying packets before forwarding them on[1]:

$$T : (H \times E) \rightarrow (H \times E_\emptyset)$$

We use 'Ψ' to denote the collection of all transfer functions present in the network at a particular point in time. In this model, network traversal is simply a composition of transformation functions. For example, if a header h enters the network through edge e, its state after k hops will be:

$$\Phi^k(h, e) = \Psi(\Psi(\dots \Psi(h, e) \dots))$$

The externally visible behavior of the network can be expressed as the transitive closure of Φ:

$$\Omega : (H \times E_{access}) \rightarrow (H \times E_\emptyset)$$
$$\Omega(h, e) = \Phi^\infty(h, e)$$

Here, E_{access} denotes access links adjacent to end-hosts.

In SDN, it should always be the case that:

$$\Omega^{view} \sim \Omega^{physical}$$

Informally, this means that any packet injected at an access link in $G^{virtual}$ should arrive at the same final location as the corresponding (encapsulated) packet injected at the corresponding access link in $G^{physical}$. Note that hosts are represented in all graphs, although there may not be a one-to-one mapping between the internal vertices of $G^{virtual}$ and $G^{physical}$.

To check correspondence in SDN, we begin by taking a causally consistent snapshot [7] of the physical network. The routing tables of forwarding elements can then be translated into transformation functions. Finally, we feed a symbolic packet x^L to each access link of the network.[2] The end result is a propagation graph representing all possible paths taken by a packet injected at the access link.

The leaves of the propagation graph represent Ω. We verify correspondence in SDN by generating propagation graphs for all SDN layers, and comparing the leaves. Any mismatch in leaves of the propagation graphs represent policy-violations between control applications and network configuration.

3.2 Simulation-based Replay Analysis

Correspondence checking only captures a snapshot of the network state. We have developed simulation-based replay analysis to allow enable troubleshooters to explore policy-violations over time. We base our replay on a consistent trace of low level failure and topology change events, as enabled, *e.g.*, by OFRewind [22]. The events from the trace are fed into a simulator, which allows the troubleshooter to invokes correspondence checking at any point in time. The simulator focuses on corner-case events, and models the failure modes in sufficient detail to reproduce the error, while allowing for complete control of the timing, ordering, and constuction of the events.

If users were to run simulation-based replay analysis on raw event logs, they would encounter a large number of failure events and transient policy-violations. We describe how users can apply simulation-based replay analysis to differentiate transient from persistent policy-violations below.

Identifying persistent policy-violations. When a policy-violation is detected, the simulators forks off a simulation

[1]Multicast forwarding can expressed by extending the range to sets of output tuples

[2]The rules for process wildcard bits x^n are defined in the HSA paper [14]

branch that investigates the future system behavior in a case where no further external events are played out. If the detected policy-violation is resolved in isolation within a customizable number of simulation time steps, it is considered a temporary problem. If it is not, this is a strong indication that the system has indeed reached a persistent policy-violation that needs to be investigated.

Checking related problems by fuzzing. Input traces can be *fuzzed*, i.e., randomly perturbed, to expose the system to similar error conditions, and confirm that a proposed solution is not just a point-fix.

Investigating pathological environment conditions. The simulator allows for investigation of pathological environment conditions difficult to achieve in a real world test bed (*e.g.*, correlated failure rates, extremely long delays etc.). This enables investigation of situations that have a high potential for triggering errors.

Interactive exploration. Troubleshooters can also interactively bisect the trace or modify specific events to further pinpoint the cause for a failure. This is useful as soon as a suspect event sequence has been identified.

3.3 Discussion

Correspondence checking and simulation-based replay analysis serve to isolate the platform layer and event sequence responsible for a given error. To identify the root cause of the failure in the code, they can be complemented by classical debugging techniques,*i.e.* log messages and source code debugging. These are much more effective when applied to investigate a specific event sequence. Once a potential fix has been developed, it can be validated by repeating the problematic replay. Fuzzing helps to validate whether there may be related error events that the patch may have left open.

4. EVALUATION

4.1 Use-Case

Here we present a hypothetical use-case from two perspectives for correspondence checking and simulation-based replay analysis.

Network Operator. An enterprise network operator pays a third-party SDN provider to virtualize their network and run it in the cloud. The network operator receives a complaint from an internal team that two servers cannot reach other, and verifies the reachablility problem with `ping`. Unsure whether the problem is in her ACL/routing rules or in the underlying SDN platform, the operator runs correspondence checking on a snapshot of the network state. She finds that there is an inconsistency between her policy and the network state, and proceeds to call up the third-party SDN provider to complain about the problem.

SDN Developer. A developer at the third-party SDN company receives the customer's trouble-ticket and begins to investigate the problem. The developer examines the system logs and sees a long list of link status events, control server reboots, VM migration events, and other diagnostic information. These events are numerous (the datacenter contains 8,000 switches and more than 100,000 hosts) and interleaved, so the developer is unsure what caused the problem. The developer feeds a snapshot of the network from before the trouble-ticket was issued into the simulator, and begins replaying the execution of the system. The developer runs correspondence checking and finds a substantial list of policy-violations, manifesting as loops, blackholes, and other problems. The developer excludes all external events unrelated to the two disconnected hosts. As the developer is stepping through the execution, he periodically runs correspondence checking and tracks the policy-violations over time. He finds that most violations resolve within a short time. However, he eventually encounters a blackhole that lasts a considerable time. The developer backs up the execution to the point when the blackhole begins, and observes a switch failure followed directly by a reboot of the switch's parent controller. A correspondence check between the intermediate layers of the SDN stack indicate that the problem is present in the physical view, but is not manifest in the virtualization layer. The developer adds log statements to the platform's failover logic, and re-runs the execution. The developer eventually verifies that the controller pushed a routing change to the failed switch's neighbors, but did not update the platform's representation of the network state. Upon closer examination, the developer finds that the new parent controller for that portion of the network assumed that the platform's representation of network state was up-to-date. As such, the partially installed flow entries remained in the neighboring switches, resulting in the blackhole. The developer fixes the platform's recovery code, adds this case to the platform's integration test suite, and pushes the change to production.

4.2 Overhead

Simulator Scalability. We have implemented a prototype simulator. In a single process, our prototype models hosts and OpenFlow switches as lightweight python objects that communicate with a SDN controller.

To gauge the scalability of modeling the network within a single-process, we generated a fat tree network of 48 pods (2,880 switches), and measured the time to send 1 PORT_STATUS message and process 5 FLOW_MOD messages per switch. The execution completed in 5.8 seconds, well within the bounds of reasonable interactive use.

Record and Replay Overhead. In contrast to general record-and-replay mechanisms, the amount of recorded state needed for high-fidelity replay is tractable. With proactive flow installation, updates are pushed to routing tables over a relatively long time scale; periodic FIB snapshots along with a log of link state events, control server downtime, and host mobility information suffice for our purposes. As a point of reference, the Cisco 7000 core switch model supports a maximum of 128K MAC entries and 128K ACL entries [1].

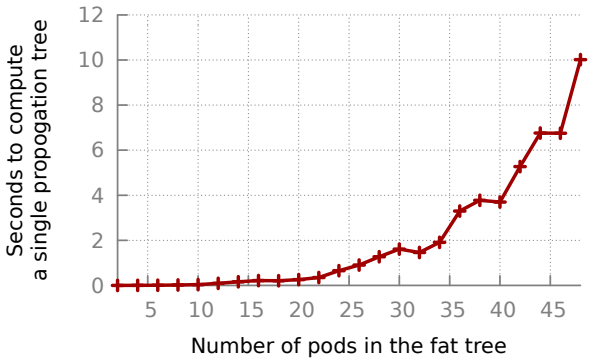

Figure 2: Serial runtime of correspondence checking on PORTLAND fat tree networks. Each datapoint consists of $x^3/4$ hosts and $5x^2/4$ switches (*e.g.* 48 pods means 27,468 hosts attached to 2,880 switches)

Assuming 36 bytes per flow entry, (larger than the OpenFlow 13-tuple), each FIB will contain a maximum of 9216 bytes, uncompressed. A datacenter of 100,000 hosts includes roughly 8,000 switches [4]. Therefore a snapshot of the FIBs of the entire network takes up roughly 74 MB. The VL2 paper reports 36M network error events over one year over 8 datacenters, which implies 8.5 error events per minute per datacenter [12]. Suppose we took a snapshot of the FIBs in the network every second. Then we would need to store roughly 4GB, uncompressed, per minute, a relatively small growth rate for datacenter logs. This information, in addition to a log of host mobility events (*e.g.* VM migrations) will suffice for our purposes. Note that this is a conservative overestimate.

Correspondence Checking Runtime. Computing the propagation graph for correspondence checking is equivalent to enumerating all possible paths in the network, which scales with the diameter of the network and the number of routing entries per switch. The propagation graph for each host can be computed in parallel however, so the computation is bottlenecked by the serial runtime of computing a single host's propagation graph.

We show the serial runtime of correspondence checking in Figure 2. For this analysis we generated fat tree topologies between 2 and 48 pods wide, with pre-installed PORTLAND [18] routing tables in each switch. Each data point is the minimum of three runs on a single Intel Xeon 2.80GHz core. Note that the number of PORTLAND routing entries per switch scales with the number of pods in the fat-tree. We excluded the time to convert flow tables to HSA transfer functions, since transfer functions can be maintained offline.

As the figure depicts, even for large networks (27,648 hosts) the serial runtime of correspondence checking is reasonable for interactive use. The number of serial tasks to be executed is the number of hosts in the network squared, disregarding ECMP load balancing.

4.3 Replay fidelity

If our simulated model of the network is not sufficiently complex, it may not be able to reproduce error conditions observed in production. Conversely, resolving bugs observed in a simulated environment may not ensure correct behavior of the production network. In future work, we hope to verify the fidelity of simulation-based replay analysis by collaborating with industry partners to reproduce bugs observed in practice. In addition, we plan to gather error logs by deploying our own applications in Google's datacenter networking research cluster [11].

Finally, note that our correspondence checking algorithm can not verify time-dependent policies such as "No link should be congested more than 1% of the time", or "No server should receive more than 500MB/s of external traffic". In future work we will extend our correspondence checking algorithm to account for this class of policies.

5. RELATED WORK

This work extends a growing literature on troubleshooting tools for Software-Defined Networks.

The work most closely related to ours is NICE [5]. NICE combines concolic execution and model checking to automate the process of testing NOX applications, thereby catching bugs before they are deployed. Our approaches complement each other in several ways. Simulation-based replay analysis places a high-burden on human intuition; conceptualizing a 10,000 node network is difficult, and it's feasible that users will not be able to reproduce the errors they have in mind. Systematic exploration of failure orderings is potentially of great use for finding corner-case errors. In complement to NICE, correspondence checking helps developers isolate the specific component of the SDN platform responsible for an error, without needing to specify invariants. NICE may also be applied directly to the code-base of the SDN platform, but in the case that only a subset of all possible code-paths in the SDN platform can be model-checked due to state-space explosion, our mechanisms allows users to troubleshoot errors *post-hoc* after they are observed in production.

Focusing on the physical network, Anteater [17] and HSA [14] are alternative approaches to statically checking invariants in the configuration of switches and routers. Both take take as input a snapshot of the FIB of each network device. To check invariants, Anteater generates a set of constraint functions and feeds them through a SAT solver, while HSA defines an algebra for virtual packets and their transformation through the network. Nonetheless, these techniques only detects bugs that manifest entirely within the physical network; they do not detect errors that manifest across the layers of the SDN stack. Furthermore, as we discuss in §3, static checking only detects errors at a particular point in time; it cannot detect errors that might arise under a different configuration or with different inputs to the same network controller.

Also focusing on the physical network, OFRewind [22]

develops record and replay techniques for the control plane of OpenFlow networks. Unlike simulation-based replay analysis OFRewind focuses specifically on OpenFlow interactions, while we focus on more course-grained replay of failures and topology changes. Running replay within a simulator also allows us to manually modify the execution of the system, rather than playing a static recording. Finally, we add correspondence checking to automate the process of isolating root causes.

Another line of work aims to prevent bugs from being introduced in the first place. Frenetic [10] presents a language-based approach to building robust SDN applications. By providing a specialized programming model, Frenetic helps developers avoid writing common classes of bugs, such 'composition errors' where installed flow entries override each other. Reitblatt et al. [19] develop a technique for ensuring consistent routing updates, guaranteeing that all switches in the network either route a given packet under the new configuration or under the old configuration, but not both. These abstractions are valuable for preventing common, difficult errors in platform logic.

Several other network simulators exist for testing SDN controllers. Mininet is a platform for emulating OpenFlow switches and hosts within a single VM [16]. The ns-series of network simulators provides a general framework for testing new protocols, topologies, and traffic mixes [3]. We found that these existing simulators did not provide sufficient support for the corner-cases situations which are the focus of our work, such as failures and VM migration.

Many of our ideas originate from the literature on troubleshooting general distributed systems. WiDS checker introduced the notion of recording production executions to be later replayed and verified in a controlled simulation. Pip [20] defines a DSL and collection of annotation tools to reason about causal paths throughout the execution of the distributed system. Finally, end-to-end tracing frameworks such as X-Trace [9] and Pinpoint [8] provide a framework for tracing requests throughout a distributed system in order to infer correctness errors between layers and across components. Our work solves a more constrained problem; we leverage the structure of the SDN stack to enable a simple notion of platform correctness. In addition, these systems assume that invariants should hold at all times; we observe that in an eventually-consistent system such as SDN, transient policy-violations are inevitable. We built simulation-based replay analysis to help troubleshooters differentiate ephemeral from persistent errors.

6. CONCLUSION

Our paper seeks to improve the process of troubleshooting errors in the SDN platform. We focus on large, production networks, where controllers are distributed across multiple servers and the platform provides a virtualization abstraction to one or more tenants. We present two mechanisms, cross-layer correspondence checking and simulation-based replay analysis, designed to help troubleshooters automatically identify and isolate inconsistencies between high-level policy and low-level configuration. Our intent throughout this work is to convince the troubleshooting community to think outside of the NOX!

7. REFERENCES

[1] Cisco 7000 series datasheet. http://tinyurl.com/77jorsq.
[2] Code repository. https://github.com/noxrepo/sdn-debugger.
[3] The ns-3 network simulator. http://www.nsnam.org/.
[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. SIGCOMM '08. ACM.
[5] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. NSDI '12.
[6] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. PRESTO '10.
[7] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst., 1985.
[8] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. Intl. Conf. on Dependable Systems and Networks, 2002.
[9] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: a pervasive network tracing framework. NSDI'07.
[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. ICFP '11.
[11] Google Inc. Datacenter networking research cluster.
[12] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. SIGCOMM '09.
[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. CCR, 2008.
[14] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. NSDI '12.
[15] 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.
[16] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. Hotnets '10.
[17] H. Mai et al. Debugging the data plane with Anteater. SIGCOMM '11.
[18] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PORRTLAND: a scalable fault-tolerant layer 2 data center network fabric. SIGCOMM '09.
[19] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! HotNets '11.
[20] P. Reynolds, C. Killian, J. L. Winer, J. C. Mogul, M. A. Shah, and A. Vadhat. Pip: Detecting the unexpected in distributed systems. NSDI '06.
[21] S. Shenker. The future of networking, and the past of protocols. Keynote, Open Networking Summit 2011.
[22] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. ATC '11.