Consistent Updates for Software-Defined Networks: Change You Can Believe In!

Mark Reitblatt
Cornell University

Nate Foster
Cornell University

Jennifer Rexford
Princeton University

David Walker
Princeton University

ABSTRACT

Configuration changes are a common source of instability in networks, leading to broken connectivity, forwarding loops, and access control violations. Even when the initial and final states of the network are correct, the update process often steps through intermediate states with incorrect behaviors. These problems have been recognized in the context of specific protocols, leading to a number of point solutions. However, a piecemeal attack on this fundamental problem, while pragmatic in the short term, is unlikely to lead to significant long-term progress.

Software-Defined Networking (SDN) provides an exciting opportunity to do better. Because SDN is a clean-slate platform, we can build general, reusable abstractions for network updates that come with strong semantic guarantees. We believe SDN desperately needs such abstractions to make programs simpler to design, more reliable, and easier to validate using automated tools. Moreover, we believe these abstractions should be provided by a runtime system, shielding the programmer from these concerns. We propose two simple, canonical, and effective update abstractions, and present implementation mechanisms. We also show how to integrate them with a network programming language, and discuss potential applications to program verification.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Distributed Systems—*Network Operating Systems*

General Terms

Design, Languages, Theory

Keywords

Consistency, planned change, software-defined networking, OpenFlow, network programming languages, Frenetic.

1. INTRODUCTION

What's the best thing a network operator can do for their network? Take a vacation. Sadly, this joke is all too true. Planned change accounts for a significant percentage of failures [1], leading to hiccups in VoIP calls, lost server connections, or the death of a player's favorite character in an online game. To address these problems, researchers have proposed extensions to routing protocols and operational practices that prevent transient anomalies during changes [2, 3, 4, 5, 6]. However useful, these solutions are limited to specific protocols (*e.g.*, BGP and OSPF) and properties (*e.g.*, loops and blackholes), and increase system complexity.

Recent trends toward Software Defined Networking (SDN) could easily exacerbate these problems by making it easy for programmers to make frequent changes to networks. However, the opposite is also true: SDN presents a tremendous opportunity to finally get network change right by empowering researchers to develop new, reusable, and robust abstractions for managing updates.

In SDN, a program running on a logically-centralized controller directly configures the packet-handling mechanisms in the underlying switches. For example, the OpenFlow API allows the controller to install rules that each specify a *pattern* that matches on bits in the packet header, *actions* (such as drop, forward, or send to the controller) performed on matching packets, a *priority* (to disambiguate between overlapping patterns), and *timeouts* (to allow the switch to remove stale rules) [7]. This is in contrast to today's networks, where operators have (at best) indirect control over the distributed implementations of routing, access control, load balancing, and so on. However, today's controller platforms [8, 9, 10, 11] provide a low-level interface for installing packet-processing rules, reading traffic counters, and discovering the topology. As a result, programmers must painstakingly express transitions from one configuration to another through a sequence of (un)install commands—rule by rule, switch by switch—and worry about the properties of every intermediate state in the process.

However, it does not have to be this way. Rather than asking users to program with low-level, single-switch updates that are difficult to reason about, SDN makes it possible, in principle, to design platforms that provide powerful *multi-switch* updates with strong semantic guarantees. Programmers who use these new operations will never have to devise (or think about) incremental, step-by-step transitions between intermediate configurations. The major difficulty, of course, is finding those abstract update operations that have both useful semantics and an efficient implementation.

In this paper, we propose several useful and implementable abstract update operations. Each of these operations identifies certain *sets of packets* and when updating from old policy to new policy across multiple switches, provides the guarantee that every packet in the set exclusively uses either the old policy or the new policy, not some combination of the two. This frees the programmer from thinking about the interaction between the old and new policies.

In the simplest case, the "set" of packets is just a single packet. In this case, each individual packet is guaranteed to be handled by either the old policy or the new policy, not some combination. In a more sophisticated case, the set consists of every packet in a flow. The latter is useful, for example, to ensure that all packets in a TCP connection reach the same back-end server. Given these kinds of consistency semantics, programmers can reduce the difficult problem of verifying program invariants across a series of incremental updates to the much simpler problem of verifying the property for just the initial and final network states.

In summary, we propose a general and rigorous solution to the problem of consistent multi-switch updates for SDN. More specifically, we make the following contributions:

- **Consistency classes:** We introduce two notions of consistency (per-packet and per-flow), which are useful across a wide range of applications.

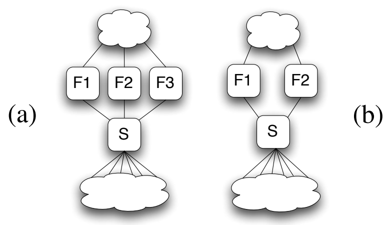
- **Consistency mechanisms:** We propose concrete mechanisms for ensuring consistency, drawing upon features already available in today's OpenFlow switches.

- **Language abstractions:** We show how to incorporate our abstractions into a programming language, making programs easier to write, understand, test, and verify.

After a brief example, we present our consistency semantics, and then show how to incorporate these abstractions into a network programming language. We focus on *planned change* in this paper, rather than unplanned events such as link failures, because changes planned in advance are by far the most common source of errors. Exploring abstractions for unplanned changes is part of our ongoing work.

2. CHANGE YOU CAN'T BELIEVE IN

To illustrate the challenges surrounding updates in conventional SDN, consider an example where the network consists of four switches: a load-balancing switch S and three "filtering" switches F_1, F_2, and F_3, each sitting between S and the upstream connection to the rest of the Internet. Figure 1(a) depicts the topology graphically. Two classes of traffic are connected to S: traffic arriving on ports 1-2



Configuration I			
	P	**T**	**Action**
S	$1, 2$		Fwd F_1
	$3, 4$		Fwd F_2
	$5, 6$		Fwd F_3
F_1		A, B	Monitor
			Allow
F_2			Allow
F_3			Allow

$\rightarrow$

Configuration II			
	P	**T**	**Action**
S	1		Fwd F_1
	2		Fwd F_2
	3-6		Fwd F_3
F_1		A, B	Monitor
			Allow
F_2		A, B	Monitor
			Allow
F_3			Allow

Figure 1: Access control example 1. Columns 'P' and 'T' match on in-port and packet type respectively.

comes from untrustworthy, unregistered guests while traffic arriving on ports 3-6 is from trustworthy, registered hosts. At all times, the network should treat packets of types A and B from untrusted hosts as potential threats (*e.g.*, A and B might capture packets destined for certain administrative servers), and monitor and possibly deny these packets based on an access control policy. On the other hand, all packets from hosts attached to ports 3-6 should be allowed to pass through the network unmodified. We assume that any of the filtering switches have the capability to perform the requisite monitoring, blocking, and forwarding.

The high-level policy can be implemented in several ways; depending on the load of the network, one may be better than another. Initially, we will assume the switches are configured as shown in the table on the left of Figure 1: the switch S sends traffic from ports 1 and 2 to F_1, from ports 3 and 4 to F_2 and from ports 5 and 6 to F_3. Switch F_1 monitors (and denies) A and B packets and allows all other packets to pass through while F_2 and F_3 simply let all traffic pass through.

Now, suppose the load shifts, and we need more resources to monitor traffic from untrusted hosts. We might then reconfigure as shown on the right side of Figure 1, where the task of monitoring port 1-2 traffic is split between F_1 and F_2 and port 3-6 traffic is pushed onto F_3. Because we cannot update the network in one atomic step, the individual switches must be reconfigured one-by-one. However, if we are not careful, incremental updates of individual switches can lead to transient configurations that do not implement the intended access control policy. For example, if we start by updating F_2 so that it denies A and B traffic, we interfere with traffic sent by trustworthy hosts. If, on the other hand, we start by updating switch S so that it forwards its traffic according to configuration II (sending port 1 traffic to F_1, port 2 traffic to F_2 and ports 3-6 traffic to F_3), we also encounter a transient error: A and B packets from untrustworthy hosts are allowed. There exists a valid transition plan:

Configuration I			
	P	**T**	**Action**
S	$1, 2$ $3{-}6$		`Fwd` F_1 `Fwd` F_2
F_1		A, B	Monitor Allow
F_2			Allow

$\longrightarrow$

Configuration II			
	P	**T**	**Action**
S	$1, 2$ $3{-}6$		`Fwd` F_2 `Fwd` F_1
F_1			Allow
F_2		A, B	Monitor Allow

Figure 2: Access control example 2.

1. Update S to forward 1-2 traffic to F_1 and 3-6 to F_3.
2. Wait until in-flight packets have been processed by F_2.
3. Update F_2 to monitor/deny A and B packets.
4. Update S to forward 1 to F_1, 2 to F_2 and 3-6 to F_3.

But finding this ordering requires performing fairly intricate reasoning about the effective forwarding policies during a sequence of intermediate configurations—something that is tedious and easy to get wrong, even for this simple example. Moreover, the space of potential intermediate configurations is potentially very large and many intermediate configurations violate important properties (*e.g.*, implementing the access control policy) that hold in both the old and new configurations. We believe that any energy the programmer devotes to navigating this space would be better spent in other ways. The tedious job of finding a safe sequence update ordering should be factored out, optimized, implemented in an automatic tool, and shared across applications.

Interestingly, in certain situations it is impossible to find an ordering that implements the transition. Consider a variation on the first example, shown in Figure 1(b), where there are only two filtering switches and we want to swap the roles of F_1 and F_2 while maintaining the access control invariant. Surprisingly, this is impossible unless S itself performs the monitoring for port 1 and 2 traffic during the transition, which may not be possible if S lacks the required monitoring capability. Thus, a mechanism more powerful than careful ordering is needed to provide a fully general solution.

3. CONSISTENCY ABSTRACTIONS

Problems with network updates arise because packets see an inconsistent "view" of the network and are processed according to a mixture of old and new configurations. While developers can sometimes code around these problems by writing programs that update switches in a particular order, writing such programs is difficult. This section presents a different approach: two core abstractions, each embodying a different notion of consistency, that provide a natural guarantee about how packets will be processed. For each abstraction, we present a motivating application, characterize its formal properties, and discuss implementation mechanisms.

3.1 Per-packet Consistency

The most fundamental consistent update abstraction is *per-packet consistency*, which guarantees that each packet flowing through the network will be processed according to a single network configuration—either the old configuration prior to the update, or the new one after the update, but not a mixture of the two. In the example in Figure 1, per-packet consistency rules out situations in which a packet is processed by the new configuration on ingress switch S and the old configuration on the filtering switch F_2, thereby circumventing the access control policy.

To implement per-packet consistency, we propose a simple mechanism that stamps packets with their configuration version at ingress switches and tests for the version number in all other rules. This can be implemented in OpenFlow using a header field to encode version numbers (*e.g.*, VLAN tags or MPLS labels). To update to a new configuration, the controller first pre-processes the rules in the new configuration, augmenting the pattern of each rule to match the new version number in the header. Next, it installs these rules on all of the switches, leaving the rules for the old configuration (whose rules match the previous version number) in place. At this point, every switch can process packets with either the old or new policy, depending on the version number on the packet.

The controller then starts updating the ingress switches, replacing their old rules with new rules that stamp incoming packets with the new version number. Because the ingress switches cannot all be updated atomically, packets entering the network are processed with a mixture of the old and new policies for a time, but *any individual packet* is handled by just one policy throughout the network. Finally, once all packets following the "old" policy have left the network, the controller deletes the old configuration rules from all switches, completing the update.[1] Figure 3 shows the intermediate configurations generated by this approach. Note that in going from one configuration to the next, the individual switches can be updated in any order.

We argued that a per-packet consistent update would enforce the access control policy in our example, but, in general, how can developers know if their invariants will be preserved? Formally, per-packet consistent updates preserve all path properties. A *path property* captures behaviors that can be expressed in terms of packets p and the list l of links the packet traverses as it is forwarded through the network. Many useful properties can be expressed as path properties including basic connectivity, loop-freeness, and security properties such as "all packets from host h must be dropped" or "all Web traffic must waypoint via middlebox m." When using a per-packet consistent update, the programmer is guaranteed that if a path property P is true of all valid packet-path pairs in both the initial and final configurations, it will be true of all paths taken by any packet at run time before,

[1]The controller can safely delete the old rules after some maximum transmission delay (*i.e.*, the sum of propagation and queuing delay, maximized over all paths) has elapsed. Since our mechanisms never introduce loops, we do not need to account for increases in delay due to transient forwarding loops. In practice, the controller can be quite conservative in estimating the delays and simply wait for several seconds (or even minutes) before removing the old rules.

Configuration I				
	P	**T**	**V**	**Action**
S	$1, 2$			SetV 1, Fwd F_1
	$3, 4$			SetV 1, Fwd F_2
	$5, 6$			SetV 1, Fwd F_3
F_1		A, B		Monitor
				Allow
F_2				Allow
F_3				Allow

→

Configuration II				
	P	**T**	**V**	**Action**
S	$1, 2$			SetV 1, Fwd F_1
	$3, 4$			SetV 1, Fwd F_2
	$5, 6$			SetV 1, Fwd F_3
F_1		A, B	1	Monitor
			1	Allow
		A, B	2	Monitor
			2	Allow
F_2			1	Allow
		A, B	2	Monitor
			2	Allow
F_3			1	Allow
			2	Allow

→

Configuration III				
	P	**T**	**V**	**Action**
S	$1, 2$			SetV 2, Fwd F_1
	$3, 4$			SetV 2, Fwd F_2
	$5, 6$			SetV 2, Fwd F_3
F_1		A, B	1	Monitor
			1	Allow
		A, B	2	Monitor
			2	Allow
F_2			1	Allow
		A, B	2	Monitor
			2	Allow
F_3			1	Allow
			2	Allow

→

Configuration IV				
	P	**T**	**V**	**Action**
S	$1, 2$			SetV 2, Fwd F_1
	$3, 4$			SetV 2, Fwd F_2
	$5, 6$			SetV 2, Fwd F_3
F_1		A, B	2	Monitor
			2	Allow
F_2		A, B	2	Monitor
			2	Allow
F_3			2	Allow

Figure 3: Configurations for per-packet consistent update example. The V column matches on version number.

during, and after the update. In other words, per-packet consistent update *preserves all path properties*.

In addition to studying per-packet consistency from the perspective of property-preservation, one can consider its *responsiveness*—how fast the network converges to the new configuration. We consider two aspects of responsiveness: initial responsiveness (how long before *some traffic* is handled by the new configuration), and complete responsiveness (how long before *all traffic* is handled by the new configuration). For the implementation just described, initial responsiveness is proportional to the number of internal switches plus the cost of installing a new configuration on them. Complete responsiveness is proportional to the total number of switches and the transmission delay of the network.

3.2 Per-flow Consistency

Per-packet consistency, while simple and powerful, is not always enough. For example, consider a network in which a single switch S balances load between two back-end servers A and B. Initially, S directs traffic from hosts whose IP addresses end with 0 to A and 1 to B. Now suppose that some time later, we bring two additional servers C and D online, and re-balance the load using the last two bits of the IP address, directing traffic from hosts whose addresses end in 00 to A, 01 to B, 10 to C, and 11 to D.

Intuitively, we want to process packets from new TCP connections according to the new configuration. However, to avoid disrupting applications, it is important that all packets in existing flows should go to the same server, where a *flow* is a sequence of packets with related header fields not separated by more than n seconds, and the particular value of n depends upon the specific protocol and application. For example, the switch should send packets from a host whose address ends in "10" to A, and not to C as the new policy would dictate, if the packets belong to an existing TCP connection. Simply processing individual packets with a single configuration will not guarantee the desired behavior.

Per-flow consistency guarantees that all packets in the same flow will be handled by the same version of the policy. Formally, the per-flow abstraction preserves all path properties, just like per-packet consistency. In addition, it preserves properties that can be expressed in terms of the paths that *sets* of packets belonging to the same flow take through the network. Such properties can be used to capture richer notions, including in-order delivery of packets in the same flow.

Implementing per-flow consistency is significantly more complicated than per-packet consistency because the system must keep track of the active flows and be able to identify packets that belong to such flows. A simple mechanism for implementing per-flow consistency can be obtained by combining versioning with rule timeouts. Similar ideas have been explored in the context of an OpenFlow load balancer [12]. The idea is to pre-install the new configuration on the internal switches, leaving the old version in place, as in per-packet consistency. Then, on ingress switches, the controller sets timeouts on the rules for the old configuration and installs the new configuration at low priority. When all flows matching a given rule die, the rule expires and the rules for the new configuration take effect. This scheme works as long as the packets in a given flow enter the network at a single ingress point. Dealing with flows involving multiple ingresses is subtle, and beyond the scope of this paper.

Note that in situations where multiple flows are processed using the same rule, the rule may be artificially kept alive even though the individual flows have died. Thus, it may be necessary to refine the old configuration rules to cover a smaller piece of the flow space, to ensure that rules expire in a timely fashion. Note that the refinement scheme has a strong effect on the responsivity of this mechanism—if the refined rules are coarse, they may never die! On the other hand, "finer" rules require more rules on the switch, a potentially scarce commodity. Managing the rules and dynamically refining them over time can be a complex bookkeeping task, especially if the network undergoes a *subsequent* policy transition before the previous one completes. However, this task can be implemented and optimized once in a generic run-time system, and leveraged over and over again by network programmers in different applications.

An alternative mechanism for implementing per-flow consistency is to exploit the wildcard *clone* feature of the DevoFlow extension of OpenFlow [13]. When processing a packet with a wildcard *clone* rule, a DevoFlow switch creates a new "microflow" rule that matches the header fields of the packet exactly, and then uses the new rule to process the

```
void main() {
  eventStream change;
  install(perpacket,C1);
  ... run ...
  wait(change);
  install(perpacket,C2);
}
```

(a)

```
property P1(p,l) =
  (inport(hd(l)):[1,2]) &
  (p in [A,B])
  ==> last(l) == deny

property P2(p,l) =
  (inport(hd(l):[1,2]) &
  not(p:[A,B])
  ==> last(l) == allow

property P3(p,l) =
  (inport(hd(l):[3..6])
  ==> last(l) == allow
```

(b)

```
config C1 =
 { S=[(inport:[1,2]) -> fwd(F1),
      (inport:[3,4]) -> fwd(F2),
      (inport:[5,6]) -> fwd(F3)]
   F1=[(packet:[A,B]) -> deny,
       true -> allow],
   F2=[true -> allow],
   F3=[true -> allow] };

config C2 =
 { S=[(inport:[1]) -> fwd(F1),
      (inport:[2]) -> fwd(F2),
      (inport:[3..6]) -> fwd(F3)],
   F1=[(packet:[A,B]) -> deny,
       true -> allow],
   F2=[(packet:[A,B]) -> deny,
       true -> allow],
   F3=[true -> allow] };
```

(c)

Figure 4: Program, properties, and configurations.

packet. In effect, clone rules cause the switch to maintain a concrete representation of all active flows in its flow table. This enables a simple implementation mechanism: first, use clone rules whenever installing configurations; second, to update from old to new, simply replace all old clone rules with new clone rules for the new configuration. Existing flows will continue to be handled by the exact-match rules generated by the old clone rules, and new flows will be handled by the new clone rules, which will themselves immediately begin spawning new microflow rules. Unlike the first scheme, this mechanism does not require complicated bookkeeping on the controller, although it does require a more complex switch.

4. THE ABSTRACTIONS AT WORK

In this section, we show how to incorporate these simple consistency abstractions into a network programming language, and we discuss the benefits that result.

Recall the example from Section 2 with four switches, S, F_1, F_2, and F_3, and two configurations, which we will call C1 and C2. Suppose that the program is organized as a simple state machine that transitions from one configuration to the other after receiving a signal on the event stream change. The code for this program is given in Figure 4 (a). To install the configuration C on the switches in the network, it uses the command install(con, C), where con describes the consistency guarantee we want (*e.g.*, perpacket consistency, as explained in Section 3.1).

The configurations themselves are defined in Figure 4(c); they use a simple notation inspired by Frenetic [14]. Each configuration lists a series of *policy clauses* for each switch. A policy clause consists of a predicate, which identifies a set of packets using constraints on the ingress port or headers, combined with an action. Clauses are matched in order, and if a packet arrives on a switch and does not match any of the clauses, the packet is dropped.

The program in Figure 4(a) is so simple, it barely deserves comment—indeed, the point of the example is to demonstrate how easy it is to write programs using well-designed abstractions. However, it is worth noting that the program has a clear meaning that can be understood independent of the specifics of C1 and C2: it is a program that installs a first configuration, waits for a signal and then installs a second configuration. A language with update operations supports a powerful kind of modularity where configurations can be defined (and modified) separately from the "glue" used to string configurations together. Without such operations, the programmer would have to issue a sequence of low-level commands to (un)install rules that transition the network from one configuration to the next. Even worse, these commands would be specific to C1 and C2 and any change to either configuration would likely invalidate the program.

Now suppose the programmer wants to determine that the network, running this program, always satisfies a critical invariant that is expressed as a path property. What must she do? She must establish that the intended path property holds in each independent configuration. To illustrate, consider the path property consisting of the conjunction of properties P1, P2, and P3 in Figure 4(b), which formalizes the access control policy. How difficult is it to verify that this holds of C1 and C2? For this artificial example, one could check by hand, but if the example were larger one would need a tool such as a model checker. In either case, the use of update operations is essential—it transforms a difficult (and perhaps infeasible) verification problem into one that can be easily automated.

In summary, operations for updating networks are essential abstractions for SDN programming. They turn hard-to-write programs into simple ones and enable formal verification of key safety invariants.

5. RELATED WORK

The problem of avoiding undesired transient behavior during planned change has been well studied in the domain of distributed routing protocols. Most prior work focused on protocol-specific approaches of adjusting link metrics to minimize disruptions [2, 3, 5]. The recent work by Vanbever *et. al* [6] also handles more significant intradomain routing changes, such as switching to a different routing protocol. Unlike our approach, these methods can only preserve basic properties such as loop-freedom and connectivity. In addition, these approaches are tied to distributed routing protocols, rather than the logically-centralized world of SDN.

The recent work on Consensus Routing [4] has similar goals to our research in that it seeks to eliminate transient errors such as disconnectivity or black holes that arise during BGP updates. In particular, Consensus Routing's "stable mode" is similar to our per-packet consistency, though computed in a distributed fashion for BGP routes. On the other hand, Consensus Routing only applies to a single protocol

(BGP), whereas our work may benefit any protocol or application developed in our linguistic framework. Along the same lines, the BGP-LP mechanism described by Katabi *et. al* [15] is essentially per-packet consistency for BGP routing.

6. FUTURE WORK

This paper outlines two key abstractions, per-packet consistency and per-flow consistency, and sketches implementation techniques for both of them. Still, the work described here is only a first step in this space. We have yet to implement our proposal and evaluate its performance in practice. Indeed, we anticipate the need for a number of optimizations such as those that limit the set of rules that need to be sent to switches to only those rules that changed between policies. In addition, while per-packet consistency and per-flow consistency appear to be core abstractions with excellent semantic properties, there may be other notions of consistency that either perform better (but remain sufficiently strong to provide benefits beyond the standard eventual consistency) or provide even richer guarantees.

7. CONCLUSIONS

In a recent blog post [16], Koponen and Casado describe three kinds of controller platforms for SDNs: (i) *application-specific* controllers, built with a single purpose in mind, (ii) *concrete* controllers that expose switch-level capabilities and communication protocols directly, and (iii) *abstract* controllers that provide high-level abstractions for specifying network behavior independent of the hardware. This third category of abstract controllers presents the most significant challenges for programming language and networking researchers. Here, the goal is to identify abstractions that make it easier to define more flexible, more reliable, more reusable, and more secure network applications.

Already, important progress has been made in this broad area. ONIX [10] provides the abstraction that the network is an eventually-consistent graph including both physical and logical elements. NetCore [17] provides the abstraction that the network forwarding policy may be defined using an arbitrary function from packets (located at switches) to actions, and compiles such functions to efficient switch-level rules. Frenetic [18] provides abstractions for *reading* network state that can be freely composed with forwarding policies.

This paper proposes consistent *writes*, the dual of Frenetic's reads, as another key abstraction in the SDN arsenal. With consistent writes, one can reason definitively about the invariant semantic properties enjoyed by network policies. There is no possibility of transient failures or security loopholes when programs switch from one policy to the next. In other words, we finally have change we can believe in.

Acknowledgments. We wish to thank Hussam Abu-Libdeh, Robert Escriva, Mike Freedman, Mike Hicks, Eric Keller, Srinivas Narayana, Alan Shieh, and the anonymous reviewers for many helpful suggestions. Our work is supported in part by ONR grant N00014-09-1-0770 and NSF grants CNS-1111698 and CNS-1111520. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the NSF or ONR.

8. REFERENCES

[1] A. Markopoulou, G. Iannaconne, S. Bhattacharrya, C. N. Chuah, and C. Diot, "Characterization of failures in an IP backbone," in *IEEE INFOCOM*, 2004.

[2] P. Franois, M. Shand, and O. Bonaventure, "Disruption-free topology reconfiguration in OSPF networks," in *IEEE INFOCOM*, May 2007.

[3] P. Francois, P.-A. Coste, B. Decraene, and O. Bonaventure, "Avoiding disruptions during maintenance operations on BGP sessions," *IEEE Trans. on Network and Service Management*, Dec 2007.

[4] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani, "Consensus routing: The Internet as a distributed system," in *NSDI*, Apr 2008.

[5] S. Raza, Y. Zhu, and C.-N. Chuah, "Graceful network operations," in *IEEE INFOCOM*, Apr 2009.

[6] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide IGP migration," in *SIGCOMM*, Aug 2011.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[8] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *SIGCOMM CCR*, vol. 38, no. 3, 2008.

[9] "Beacon: A Java-based OpenFlow control platform.," Nov 2010. See `http://www.beaconcontroller.net`.

[10] 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*, Oct 2010.

[11] A. Voellmy and P. Hudak, "Nettle: Functional reactive programming of OpenFlow networks," in *PADL*, Jan 2011.

[12] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *Hot-ICE*, Mar 2011.

[13] J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. Curtis, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *SIGCOMM*, Aug 2011.

[14] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ICFP*, Sep 2011.

[15] D. Katabi, N. Kushman, and J. Wrocklawski, "A Consistency Management Layer for Inter-Domain Routing," Tech. Rep. MIT-CSAIL-TR-2006-006, Cambridge, MA, Jan 2006.

[16] T. Koponen and M. Casado, "What might an SDN controller API look like?." `http://tinyurl.com/3nxnwgm`, Aug 2011.

[17] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programs," in *POPL*, Jan 2012.

[18] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ICFP*, Sep 2011.