Abstractions for Network Update

Mark Reitblatt
Cornell

Nate Foster
Cornell

Jennifer Rexford
Princeton

Cole Schlesinger
Princeton

David Walker
Princeton

ABSTRACT

Configuration changes are a common source of instability in networks, leading to outages, performance disruptions, and security vulnerabilities. Even when the initial and final configurations are correct, the update process itself often steps through intermediate configurations that exhibit incorrect behaviors. This paper introduces the notion of consistent network updates—updates that are guaranteed to preserve well-defined behaviors when transitioning between configurations. We identify two distinct consistency levels, per-packet and per-flow, and we present general mechanisms for implementing them in Software-Defined Networks using switch APIs like OpenFlow. We develop a formal model of OpenFlow networks, and prove that consistent updates preserve a large class of properties. We describe our prototype implementation, including several optimizations that reduce the overhead required to perform consistent updates. We present a verification tool that leverages consistent updates to significantly reduce the complexity of checking the correctness of network control software. Finally, we describe the results of some simple experiments demonstrating the effectiveness of these optimizations on example applications.

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

"Nothing endures but change."

—Heraclitus

SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.

Networks exist in a constant state of flux. Operators frequently modify routing tables, adjust link weights, and change access control lists to perform tasks from planned maintenance, to traffic engineering, to patching security vulnerabilities, to migrating virtual machines in a datacenter. But even when updates are planned well in advance, they are difficult to implement correctly, and can result in disruptions such as transient outages, lost server connections, unexpected security vulnerabilities, hiccups in VoIP calls, or the death of a player's favorite character in an online game.

To address these problems, researchers have proposed a number of extensions to protocols and operational practices that aim to prevent transient anomalies [8, 2, 9, 3, 5]. However, each of these solutions is limited to a specific protocol (*e.g.*, OSPF and BGP) and a specific set of properties (*e.g.*, freedom from loops and blackholes) and increases the complexity of the system considerably. Hence, in practice, network operators have little help when designing a new protocol or trying to ensure an additional property not covered by existing techniques. A list of example applications and their properties is summarized in Table 1.

We believe that, instead of relying on point solutions for network updates, the networking community needs foundational principles for designing solutions that are applicable to a wide range of protocols and properties. These solutions should come with two parts: (1) an abstract interface that offers strong, precise, and intuitive semantic guarantees, and (2) concrete mechanisms that faithfully implement the semantics specified in the abstract interface. Programmers can use the interface to build robust applications on top of a reliable foundation. The mechanisms, while possibly complex, would be implemented once by experts, tuned and optimized, and used over and over, much like register allocation or garbage collection in a high-level programming language.

Software-defined networks. The emergence of Software Defined Networks (SDN) presents a tremendous opportunity for developing general abstractions for managing network updates. In an SDN, a program running on a logically-centralized controller manages the network directly by configuring the packet-handling mechanisms in the underlying switches. For example, the OpenFlow API allows a controller to install rules that each specify a *pattern* that matches on bits in the packet header, *actions* performed on matching packets (such as drop, forward, or divert to the controller), a *priority* (to disambiguate between overlapping patterns), and *timeouts* (to allow the switch to remove stale rules) [10]. Hence, whereas today network operators have (at best) indirect control over the distributed implementations of routing, access control, and load balancing, SDN platforms like OpenFlow provide programmers with direct control over the processing of packets in the network.

However, despite the conceptual appeal of centralized control,

Example Application	Policy Change	Desired Property	Practical Implications
Stateless firewall	Changing access control list	No security holes	Admitting malicious traffic
Planned maintenance [1, 2, 3]	Shut down a node/link	No loops/blackholes	Packet/bandwidth loss
Traffic engineering [1, 3]	Changing a link weight	No loops/blackholes	Packet/bandwidth loss
VM migration [4]	Move server to new location	No loops/blackholes	Packet/bandwidth loss
IGP migration [5]	Adding route summarization	No loops/blackholes	Packet/bandwidth loss
Traffic monitoring	Changing traffic partitions	Consistent counts	Inaccurate measurements
Server load balancing [6, 7]	Changing load distribution	Connection affinity	Broken connections
NAT or stateful firewall	Adding/replacing equipment	Connection affinity	Outages, broken connections

Table 1: Example changes to network configuration, and the desired update properties.

an OpenFlow network is still a distributed system, with inevitable delays between the switches and the controller. To implement a transition from one configuration to another, programmers must issue a painstaking sequence of low-level install and uninstall commands that work rule by rule and switch by switch. Moreover, to ensure that the network behaves correctly during the transition, they must worry about the properties of every possible intermediate state during the update, and the effects on any packets already in flight through the network. This often results in a combinatorial explosion of possible behaviors—too many for a programmer to manage by hand, even in a small network. A recent study on testing OpenFlow applications shows that programmers often introduce subtle bugs when handling network updates [11].

Our approach. This paper describes a different alternative. Instead of requiring SDN programmers to implement configuration changes using today's low-level interfaces, our high-level, abstract operations allow the programmer to update the configuration of the entire network in one fell swoop. The libraries implementing these abstractions provide strong semantic guarantees about the observable effects of the global updates, and handle all of the details of transitioning between old and new configurations efficiently.

Our central abstraction is *per-packet consistency*, the guarantee that every packet traversing the network is processed by exactly one consistent global network configuration. When a network update occurs, this guarantee persists: each packet is processed either using the configuration in place prior to the update, or the configuration in place after the update, but never a mixture of the two. Note that this consistency abstraction is *more* powerful than an "atomic" update mechanism that simultaneously updates all switches in the network. Such a simultaneous update could easily catch many packets in flight in the middle of the network, and such packets may wind up traversing a mixture of configurations, causing them to be dropped or sent to the wrong destination. We also introduce *per-flow consistency*, a generalization of per-packet consistency that guarantees all packets in the same flow are processed with the same configuration. This stronger guarantee is needed in applications such as HTTP load balancers, which need to ensure that all packets in the same TCP connection reach the same server replica to avoid breaking connections.

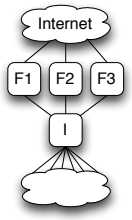
To support these abstractions, we develop several *update mechanisms* that use features commonly available on OpenFlow switches. Our most general mechanism, which enables transition between any two configurations, performs a two-phase update of the rules in the new configuration onto the switches. The other mechanisms are optimizations that achieve better performance under circumstances that arise often in practice. These optimizations transition to new configurations in less time, update fewer switches, or fewer rules.

To analyze our abstractions and mechanisms, we develop a simple, formal model that captures the essential features of OpenFlow networks. This model allows us to define a class of network properties, called *trace properties*, that characterize the paths individual packets take through the network. The model also allows us to prove a remarkable result: if *any* trace property P holds of a network configuration prior to a per-packet consistent update as well as after the update, then P also holds continuously throughout the update process. This illustrates the true power of our abstractions: programmers do not need to specify *which* trace properties our system must maintain during an update, because a per-packet consistent update preserves *all* of them! For example, if the old and new configurations are free from forwarding loops, then the network will be loop-free before, during, and after the update. In addition to the proof sketch included in this paper, this result has been formally verified in the Coq proof assistant [12].

An important and useful corollary of these observations is that it is possible to take any verification tool that checks trace properties of *static* network configurations and transform it into a tool that checks invariance of trace properties as the network configurations evolve *dynamically*—it suffices to check the static policies before and after the update. We illustrate the utility of this idea concretely by deploying the NuSMV model checker [13] to verify invariance of a variety of important trace properties that arise in practice. However, other tools, such as the recently-proposed header space analysis tool [14], also benefit from our approach.

Contributions. This paper makes the following contributions:

- **Update abstractions:** We propose per-packet and per-flow consistency as canonical, general abstractions for specifying network updates (§2,§7).
- **Update mechanisms:** We describe OpenFlow-compatible implementation mechanisms and several optimizations tailored to common scenarios (§5,§8).
- **Theoretical model:** We develop a mathematical model that captures the essential behavior of SDNs, and we prove that the mechanisms correctly implement the abstractions (§3). We have formalized the model and proved the main theorems in the Coq proof assistant.
- **Verification tools:** We show how to exploit the power of our abstractions by building a tool for verifying properties of network control programs (§6).
- **Implementation:** We describe a prototype implementation on top of the OpenFlow/NOX platform (§8).
- **Experiments:** We present results from experiments run on small, but canonical applications that compare the total number of control messages and rule overhead needed to implement updates in each of these applications (§8).



Configuration I

	Type	Action
I	U, G	Forward F_1
	S	Forward F_2
	F	Forward F_3
F_1	*SSH*	Monitor
	$*$	Allow
F_2	$*$	Allow
F_3	$*$	Allow

→

Configuration II

	Type	Action
I	U	Forward F_1
	G	Forward F_2
	S, F	Forward F_3
F_1	*SSH*	Monitor
	$*$	Allow
F_2	*SSH*	Monitor
	$*$	Allow
F_3	$*$	Allow

Figure 1: Access control example.

2. EXAMPLE

To illustrate the challenges surrounding network updates, consider an example network with one ingress switch I and three "filtering" switches F_1, F_2, and F_3, each sitting between I and the rest of the Internet, as shown on the left side of Figure 1. Several classes of traffic are connected to I: untrustworthy packets from Unknown and Guest hosts, and trustworthy packets from Student and Faculty hosts. At all times, the network should enforce a security policy that denies SSH traffic from untrustworthy hosts, but allows all other traffic 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.

There are several ways to implement this policy, and depending on the traffic load, one may be better than another. Suppose that initially we configure the switches as shown in the leftmost table in Figure 1: switch I sends traffic from U and G hosts to F_1, from S hosts to F_2, and from F hosts to F_3. Switch F_1 monitors (and denies) SSH packets and allows all other packets to pass through, while F_2 and F_3 simply let all packets pass through.

Now, suppose the load shifts, and we need more resources to monitor the untrustworthy traffic. We might reconfigure the network as shown in the table on the right of Figure 1, where the task of monitoring traffic from untrustworthy hosts is divided between F_1 and F_2, and all traffic from trustworthy hosts is forwarded to F_3. Because we cannot update the network all at once, the individual switches need to be reconfigured one-by-one. However, if we are not careful, making incremental updates to the individual switches can lead to intermediate configurations that violate the intended security policy. For instance, if we start by updating F_2 to deny SSH traffic, we interfere with traffic sent by trustworthy hosts. If, on the other hand, we start by updating switch I to forward traffic according to the new configuration (sending U traffic to F_1, G traffic to F_2, and S and F traffic to F_3), then SSH packets from untrustworthy hosts will incorrectly be allowed to pass through the network. There is one valid transition plan:

1. Update I to forward S traffic to F_3, while continuing to forward U and G traffic to F_1 and F traffic to F_3.
2. Wait until in-flight packets have been processed by F_2.
3. Update F_2 to deny SSH packets.
4. Update I to forward G traffic to F_2, while continuing to forward U traffic to F_1 and S and F traffic to F_3.

But finding this ordering and verifying that it behaves correctly requires performing intricate reasoning about a sequence of intermediate configurations—something that is tedious and error-prone, even for this simple example. Even worse, in some examples it is impossible to find an ordering that implements the transition simply by adding one part of the new configuration at a time (*e.g.*, if we swap the roles of F_1 and F_3 while enforcing the intended security policy). In general, more powerful update mechanisms are needed.

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 of commands that implement an update should be factored out, optimized, and reused across many applications. This is the main achievement of this paper. To implement the update using our abstractions, the programmer would simply write:

```
per_packet_update(config2)
```

Here `config2` represents the new global network configuration. The per-packet update library analyzes the configuration and topology and selects a suitable mechanism to implement the update. Note that the programmer does not write any tricky code, does not need to consider how to synchronize switch update commands, and does not need to consider the packets in flight across the network. The `per_packet_update` library handles all of the low-level details, and even attempts to select a mechanism that minimizes the cost of implementing the update.

Further, suppose the programmer knows that the security policy holds initially but wants to be sure it is enforced continuously through the update process and also afterwards when `config2` is in force. In this case, the programmer can execute another command:

```
ok = verify(config2, topo, pol)
```

If the boolean `ok` is true, then the security policy represented by `pol` holds continuously. If not, the programmer has made a mistake and can work on debugging it. The policy `pol` itself is expressed in a common specification language called CTL and is verified with the help of a model checker. We supply a library of common network properties such as loop-freeness for use with our system and programmers can write their own custom properties.

To implement the update, the library could use the safe, switch-update ordering described above. However, in general, it is not always possible to find such an ordering. Nevertheless, one can always achieve a per-packet consistent update using a two-phase update supported by configuration versioning. Intuitively, this universal update mechanism works by stamping every incoming packet with a version number (*e.g.*, stored in a VLAN tag) and modifying every configuration so that it only processes packets with a set version number. To change from one configuration to the next, it first populates the switches in the middle of the network with new configurations guarded by the next version number. Once that is complete, it enables the new configurations by installing rules at the perimeter of the network that stamp packets with that next version number. Though this general mechanism is somewhat heavyweight, our libraries identify and apply lightweight optimizations.

This short example illustrates some of the challenges that arise when implementing a network update with strong semantic guarantees. However, it also shows that all of these complexities can be hidden from the programmer, leaving only the simplest of interfaces for global network update. We believe this simplicity will lead to a more reliable and secure network infrastructure. The following sections describe our approach in more detail.

3. THE NETWORK MODEL

This section presents a simple mathematical model of the essential features of SDNs. This model is defined by a relation that describes the fine-grained, step-by-step execution of a network. We write the relation using the notation $N \xrightarrow{us}{}^{\star} N'$, where N is the network at the beginning of an execution, N' is the network after some number of steps of execution, and us is a list of "observations" that are made during the execution.[1] Intuitively, an observation should be thought of as a message between the controller

[1]When a network takes a series of steps and there are no observa-

Bit	b	$::= 0 \mid 1$
Packet	pk	$::= [b_1, ..., b_k]$
Port	p	$::= 1 \mid \cdots \mid k \mid \mathit{Drop} \mid \mathit{World}$
Located Pkt	lp	$::= (p, pk)$
Trace	t	$::= [lp_1, ..., lp_n]$
Update	u	$\in \mathit{LocatedPkt} \rightharpoonup \mathit{LocatedPkt}\ \mathit{list}$
Switch Func.	S	$\in \mathit{LocatedPkt} \to \mathit{LocatedPkt}\ \mathit{list}$
Topology Func.	T	$\in \mathit{Port} \to \mathit{Port}$
Port Queue	Q	$\in \mathit{Port} \to (\mathit{Packet} \times \mathit{Trace})\ \mathit{list}$
Configuration	C	$::= (S, T)$
Network State	N	$::= (Q, C)$

(a)

T-PROCESS

if p is any port (1)
and $Q(p) = [(pk_1, t_1), (pk_2, t_2), ..., (pk_j, t_j)]$ (2)
and $C = (S, T)$ (3)
and $S(p, pk_1) = [(p'_1, pk'_1), ..., (p'_k, pk'_k)]$ (4)
and $T(p'_i) = p''_i$, for i from 1 to k (5)
and $t'_1 = t_1 +\!\!+ [(p, pk_1)]$ (6)
and $Q'_0 = \mathit{override}(Q, p \mapsto [(pk_2, t_2), ..., (pk_j, t_j)])$ (7)
and $Q'_1 = \mathit{override}(Q'_0, p''_1 \mapsto Q(p''_1) +\!\!+ [(pk'_1, t'_1)])$
$\vdots$
and $Q'_k = \mathit{override}(Q'_{k-1}, p''_k \mapsto Q(p''_k) +\!\!+ [(pk'_k, t'_1)])$
then $(Q, C) \longrightarrow (Q'_k, C)$ (8)

T-UPDATE

if $S' = \mathit{override}(S, u)$ (9)
then $(Q, (S, T)) \xrightarrow{u} (Q, (S', T))$ (10)

(b)

Figure 2: The network model: (a) syntax and (b) semantics.

and the network. In this paper, we are interested in a single kind of message—a message u that directs a particular switch in the network to update its forwarding table with some new rules. The formal system could easily be augmented with other kinds of observations, such as topology changes or failures. For the sake of brevity, we elide these features in this paper.

The main purpose of the model is to compute the *traces*, or paths, that a packet takes through a network that is configured in a particular way. These traces in turn define the properties, be they access control or connectivity or others, that a network configuration satisfies. Our end goal is to use this model and the traces it generates to prove that, when we update a network, the properties satisfied by the initial and final configurations are preserved. The rest of this section will make these ideas precise.

Notation. We use standard notation for types. In particular, the type $T_1 \to T_2$ denotes the set of total functions that take arguments of type T_1 and produce results of type T_2, while $T_1 \rightharpoonup T_2$ denotes the set of partial functions from T_1 to T_2; the type $T_1 \times T_2$ denotes the set of pairs with components of type T_1 and T_2; and T *list* denotes the set of lists with elements of type T.

We also use standard notation to construct tuples: (x_1, x_2) is a pair of items x_1 and x_2. For lists, we use the notation $[x_1, ..., x_n]$ for the list of n elements x_1 through x_n, $[\,]$ for the empty list, and $xs_1 +\!\!+ xs_2$ for the concatenation of the two lists xs_1 and xs_2. Notice that if x is some sort of object, we will typically use xs as the variable for a list of such objects. For example, we use u to represent a single update and us to represent a list of updates.

Basic Structures. Figure 2(a) defines the syntax of the elements of the network model. A *packet* pk is a sequence of bits, where a *bit* b is either 0 or 1. A *port* p represents a location in the network where packets may be waiting to be processed. We distinguish two kinds of ports: ordinary ports numbered uniquely from 1 to k, which correspond to the physical input and output ports on switches, and two special ports, Drop and World. Intuitively, packets queued at the Drop port represent packets that have been dropped, while packets queued at the World port represent packets that have been forwarded beyond the confines of the network. Each ordinary port will be located on some switch in the network. However, we will leave the mapping from ports to switches unspecified, as it is not needed for our primary analyses.

Switch and Topology Functions. A network is a packet processor that forwards packets and optionally modifies the contents of those packets on each hop. Following Kazemian *et al.* [14], we model packet processing as the composition of two simpler behaviors: (1) forwarding a packet across a switch and (2) moving packets from one end of a link to the other end. The *switch function* S takes a located packet lp (a pair of a packet and a port) as input and returns a list of located packets as a result. In many applications, a switch function only produces a single located packet, but in applications such as multicast, it may produce several. To drop a packet, a switch function maps the packet to the special Drop port. The *topology function* T maps one port to another if the two ports are connected by a link in the network. Given a topology function T, we define an ordinary port p to be an *ingress port* if for all other ordinary ports p' we have $T(p') \neq p$. Similarly, we define an ordinary port p to be an *internal port* if it is not an ingress port.

To ensure that switch and topology functions are reasonable, we impose the following conditions:

(1) For all packets pk, $S(\mathit{Drop}, pk) = [(\mathit{Drop}, pk)]$ and $S(\mathit{World}, pk) = [(\mathit{World}, pk)]$;

(2) $T(\mathit{Drop}) = \mathit{Drop}$ and $T(\mathit{World}) = \mathit{World}$; and

(3) For all ports p and packets pk if $S(p, pk) = [(p_1, pk_1), ..., (p_k, pk_k)]$ we have $k \geq 1$.

Taken together, the first and second conditions state that once a packet is dropped or forwarded beyond the perimeter of the network, it must stay dropped or beyond the perimeter of the network and never return. If our network forwards a packet out to another network and that other network forwards the packet back to us, we treat the return packet as a "fresh" packet—*i.e.*, we do not explicitly model inter-domain forwarding. The third condition states that applying the forwarding function to a port and a packet must produce at least one packet. This third condition means that the network

tions (*i.e.,* no updates happen), we omit the list above the arrow, writing $N \longrightarrow^{\star} N'$ instead.

cannot drop a packet simply by not forwarding it anywhere. Dropping packets occurs by explicitly forwarding a single packet to the $Drop$ port. This feature makes it possible to state network properties that require packets either be dropped or not.

Configurations and Network States. A *trace* t is a list of located packets that keeps track of the hops that a packet takes as it traverses the network. A *port queue* Q is a total function from ports to lists of packet-trace pairs. These port queues record the packets waiting to be processed at each port in the network, along with the full processing history of that packet. Several of our definitions require modifying the state of a port queue. We do this by building a new function that overrides the old queue with a new mapping for one of its ports: $override(Q, p \mapsto l)$ produces a new port queue Q' that maps p to l and like Q otherwise.

$$override(Q, p \mapsto l) = Q'$$
$$\text{where } Q'(p') = \begin{cases} l & \text{if } p = p' \\ Q(p') & \text{otherwise} \end{cases}$$

A *configuration* C comprises a switch function S and a topology function T. A *network state* N is a pair (Q, C) containing a port queue Q and configuration C.

Transitions. The formal definition of the network semantics is given by the relations defined in Figure 2(b), which describe how the network transitions from one state to the next one. The system has two kinds of transitions: packet-processing transitions and update transitions. In a packet-processing transition, a packet is retrieved from the queue for some port, processed using the switch function S and topology function T, and the newly generated packets are enqueued onto the appropriate port queues. More formally, packet-processing transitions are defined by the T-PROCESS case in Figure 2(b). Lines 1-8 may be read roughly as follows:

(1) If p is any port,
(2) a list of packets is waiting on p,
(3) the configuration C is a pair of a switch function S and topology function T,
(4) the switch function S forwards the chosen packet to a single output port, or several ports in the case of multicast, and possibly modifies the packet
(5) the topology function T connects each output port to an input port,
(6) a new trace t'_1, which extends the old trace and records the current hop, is generated,
(7) a new queue Q'_k is generated by moving packets across links as specified in steps (4), (5) and (6),
(8) then (Q, C) can step to (Q'_k, C).

In an update transition, the switch forwarding function is updated with new behavior. We represent an *update* u as a partial function from located packets to lists of located packets (*i.e.*, an update is just a "part" of a global (distributed) switch function). To apply an update to a switch function, we overwrite the function using all of the mappings contained in the update. More formally, $override(S, u)$ produces a new function S' that behaves like u on located packets in the domain[2] of u, and like S otherwise.

$$override(S, u) = S'$$
$$\text{where } S'(p, pk) = \begin{cases} u(p, pk) & \text{if } (p, pk) \in \mathsf{dom}(u) \\ S(p, pk) & \text{otherwise} \end{cases}$$

[2]Domain of an update is the set of located packets it's defined upon.

Update transitions are defined formally by the T-UPDATE case in Figure 2(b). Lines 9-10 may be read as follows: if S' is obtained by applying update u to a switch in the network then network state $(Q, (S, T))$ can step to new network state $(Q, (S', T))$.

Network Semantics. The overall semantics of a network in our model is defined by allowing the system to take an arbitrary number of steps starting from an *initial state* in which the queues of all internal ports as well as $World$ and $Drop$ are empty, and the queues of external ports are filled with pairs of packets and the empty trace. The reflexive and transitive closure of the single-step transition relation $N \xrightarrow{us}{}^{\star} N'$ is defined in the usual way, where the sequence of updates recorded in the label above the arrow is obtained by concatenating all of the updates in the underlying transitions in order.[3] A network *generates* a trace t if and only if there exists an initial state Q such that $(Q, C) \longrightarrow^{\star} (Q', C)$ and t appears in Q'. Note that no updates may occur when generating a trace.

Properties. In general, there are myriad properties a network might satisfy—*e.g.*, access control, connectivity, in-order delivery, quality of service, fault tolerance, to name a few. In this paper, we will primarily be interested in *trace properties*, which are prefix-closed sets of traces. Trace properties characterize the paths (and the state of the packet at each hop) that an individual packet is allowed to take through the network. Many network properties, including access control, connectivity, routing correctness, loop-freedom, correct VLAN tagging, and waypointing can be expressed using trace properties. For example, loop-freedom can be specified using a set that contain all traces except those in which some ordinary port p appears twice. In contrast, timing properties and relations between multiple packets including quality of service, congestion control, in-order delivery, or flow affinity are not trace properties.

We say that a port queue Q satisfies a trace property P if all of the traces that appear in Q also appear in the set P. Similarly, we say that a network configuration C satisfies a trace property P if for all *initial* port queues Q and all (update-free) executions $(Q, C) \longrightarrow^{\star} (Q', C)$, it is the case that Q' satisfies P.

4. PER-PACKET ABSTRACTION

One reason that network updates are difficult to get right is that they are a form of concurrent programming. Concurrent programming is hard because programmers must consider the interleaving of every operation in every thread and this leads to a combinatorial explosion of possible outcomes—too many outcomes for most programmers to manage. Likewise, when performing a network update, a programmer must consider the interleaving of switch update operations with every kind of packet that might be traversing their network. Again, the number of possibilities explodes.

Per-packet consistent updates reduce the number of scenarios a programmer must consider to just two: for every packet, it is as if the packet flows through the network *completely before* the update occurs, or *completely after* the update occurs.

One might be tempted to think of per-packet consistent updates as "atomic updates", but they are actually better than that. An atomic update would cause packets in flight to be processed partly according to the configuration in place prior to the update, and partly according to the configuration in place after the update. To understand what happens to those packets (*e.g.*, whether they get dropped), a programmer would have to reason about every possible

[3]The semantics of the network is defined from the perspective of an omniscient observer, so there is an order in which the steps occur.

trace formed by concatenating a prefix generated by the original configuration with a suffix generated by the new configuration.

Intuitively, per-packet consistency states that for a given packet, the traces generated during an update come from the old configurations, or the new configuration, but not a mixture of the two. In the formal definition of per-packet consistency, we introduce an equivalence relation $\sim$ on packets. We extend this equivalence relation to traces by considering two traces to be equivalent if the packets they contain are equivalent according to the $\sim$ relation (similarly, we extend $\sim$ to properties in the obvious way). We then require that all traces generated *during* the update be equivalent to a trace generated by either the initial or final configuration. For the rest of the paper, when we say that $\sim$ is an equivalence relation on traces, we assume that it has been constructed like this. This specification gives implementations of updates flexibility by allowing some minor, irrelevant differences to appear in traces (where $\sim$ defines the notion of irrelevance precisely). For example, we can define a "version" equivalence relation that relates packets pk and pk' which differ only in the value of their version tags. This relation will allow us to state that changes to version tags performed by the implementation mechanism for per-packet update are irrelevant. In other words, a per-packet mechanism may perform internal bookkeeping by stamping version tags without violating our technical requirements on the correctness of the mechanism. The precise definition of per-packet consistency is as follows.

Definition 1 (Per-packet $\sim$-consistent update) *Let $\sim$ be a trace-equivalence relation. An update sequence us is a per-packet $\sim$-consistent update from C_1 to C_2 if and only if, for all*

- *initial states Q,*
- *executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*
- *and traces t in Q',*

there exists

- *an initial state Q_i,*
- *and either an execution $(Q_i, C_1) \longrightarrow^{\star} (Q'', C_1)$ or an execution $(Q_i, C_2) \longrightarrow^{\star} (Q'', C_2)$,*

such that Q'' contains t', for some trace t' with $t' \sim t$.

From an implementer's perspective, the operational definition of per-packet consistency given above provides a specification that he or she must meet. However, from a programmer's perspective, there is another, more useful side to per-packet consistency: *per-packet consistent updates preserve every trace property.*

Definition 2 ($\sim$-property preservation) *Let C_1 and C_2 be configurations and $\sim$ be a trace-equivalence relation. A sequence us is a $\sim$-property-preserving update from C_1 and C_2 if and only if, for all*

- *initial states Q,*
- *executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*
- *and properties P that are satisfied by C_1 and C_2 and do not distinguish traces related by $\sim$,*

we have that Q' satisfies P.

Universal $\sim$-property preservation gives programmers a strong principle they can use to reason about their programs. If programmers check that a trace property such as loop-freedom or access control holds of the network configurations before and after an update, they are guaranteed it holds of every trace generated throughout the update process, even though the series of observations us may contain many discrete update steps. Our main theorem states that per-packet consistent updates preserve all properties:

Theorem 1 *For all trace-equivalence relations $\sim$, if us is a per-packet $\sim$-consistent update of C_1 to C_2 then us is a $\sim$-property-preserving update of C_1 to C_2.*

The proof of the theorem is a relatively straightforward application of our definitions. From a practical perspective, this theorem allows a programmer to get great mileage out of per-packet consistent updates. In particular, since per-packet consistent updates preserve *all* trace properties, the programmers do not have to tell the system *which* specific properties must be invariant in their applications.

From a theoretical perspective, it is also interesting that the converse of the above theorem holds. This gives us a sort of completeness result: if programmers want an update that preserves all properties, they need not search for it outside of the space of per-packet consistent updates—any universal trace-property preserving update is a per-packet consistent update.

Theorem 2 *For all trace-equivalence relations $\sim$, if us is a $\sim$-property-preserving update of C_1 to C_2 then us is a per-packet $\sim$-consistent update of C_1 to C_2.*

The proof of this theorem proceeds by observing that since us preserves all $\sim$-properties, it certainly preserves the following $\sim$-property P_{or}:

$$\{t \mid \text{there exists an initial } Q \text{ and a trace } t' \\ \text{and } ((Q, C_1) \longrightarrow^{\star} (Q', C_1) \text{ or } (Q, C_2) \longrightarrow^{\star} (Q', C_2)), \\ \text{and } t \sim t', \\ \text{and } t' \in Q'\}$$

By the definition of P_{or}, the update us generates no traces that cannot be generated either by the initial configuration C_1 or by the final configuration C_2. Hence, by definition, us is per-packet consistent.

5. PER-PACKET MECHANISMS

Depending on the network topology and the specifics of the configurations involved, there may be several ways to implement a per-packet consistent update. However, all of the techniques we have discovered so far, no matter how complicated, can be reduced to two fundamental building blocks: the one-touch update and the unobservable update. For example, our two-phase update mechanism uses unobservable updates to install the new configuration before it is used, and then "unlocks" the new policy by performing a one-touch update on the ingress ports.

One-touch updates. A one-touch update is an update with the property that no packet can follow a path through the network that reaches an updated (or to-be-updated) part of the switch rule space more than once.

Definition 3 (One-touch Update) *Let $C_1 = (S, T)$ be the original network configuration, $us = [u_1, ..., u_k]$ an update sequence, and $C_2 = (S[u_1, ..., u_k], T)$ the new configuration, such that the domains of each update u_1 to u_k are mutually disjoint. If, for all*

- *initial states Q,*
- *and executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*

there does not exist a trace t in Q' such that

- *t contains distinct trace elements (p_1, pk_1) and (p_2, pk_2),*
- *and (p_1, pk_1) and (p_2, pk_2) both appear in the domain of update functions $[u_1, ..., u_k]$,*

then us is a one-touch update from C_1 to C_2.

Theorem 3 *If us is a one-touch update then us is a $\sim$-per-packet consistent update for any $\sim$.*

The proof proceeds by considering the possible traces t generated by an execution $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$. There are two cases: (1) There is no element of t that appears in the domain of an update function in us, or (2) some element lp of t appears in the domain of an update function in us. In case (1), t can also be generated by an execution with no update observations: $(Q, C_1) \longrightarrow^{\star} (Q'', C_1)$, and the definition of per-packet consistency vacuously holds. In case (2), there are two subcases:

(i) lp appears in the trace prior to the update taking place and so t is also generated by $(Q, C_1) \longrightarrow^{\star} (Q'', C_1)$.

(ii) lp appears in the trace after the update has taken place and so t is also generated by $(Q, C_2) \longrightarrow^{\star} (Q'', C_2)$.

The one-touch update mechanism has a few immediate, more specific applications:

- *Loop-free switch updates:* If a switch is not part of a topological loop (either before or after the update), then updating all the ports on that switch is an instance of a one-touch update and is per-packet consistent.
- *Ingress port updates:* An ingress port interfaces exclusively with the external world, so it can not be a part of an internal topological loop and is never on the same trace as any other ingress port. Consequently, any update to ingress ports is a one-touch update and is per-packet consistent. Such updates can be used to change the admission control policy for the network, either by adding or excluding flows.

When one-touch updates are combined with unobservable updates, there are many more possibilities.

Unobservable updates. An unobservable update is an update that does not change the set of traces generated by a network.

Definition 4 (Unobservable Update) *Let $C_1 = (S, T)$ be the original network configuration, $us = [u_1, ..., u_k]$ an update sequence, and $C_2 = (S[u_1, ..., u_k], T)$ the new configuration. If, for all*

- *initial states Q,*
- *executions $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$,*
- *and traces t in Q',*

there exists

- *an initial state Q_i,*
- *and an execution $(Q_i, C_1) \longrightarrow^{\star} (Q'', C_1)$,*

such that the trace t is in Q'', then us is an unobservable update from C_1 to C_2.

Theorem 4 *If us is an unobservable update then us is a per-packet consistent update.*

The proof proceeds by observing that every trace generated during the unobservable update $(Q, C_1) \xrightarrow{us}{}^{\star} (Q', C_2)$ also appears in the traces generated by C_1.

On their own, unobservable updates are useless as they do not change the semantics of packet forwarding. However, they may be combined with other per-packet consistent updates to great effect using the following theorem.

Theorem 5 (Composition) *If us_1 is an unobservable update from C_1 to C_2 and us_2 is a per-packet consistent update from C_2 to C_3 then $us_1 +\!\!+ us_2$ is a per-packet consistent update from C_1 to C_3.*

A simple use of composition arises when one wants to achieve a per-packet consistent update that extends a policy with a completely new path.

- *Path extension:* Consider an initial configuration C_1. Suppose $[u_1, u_2, ..., u_k]$ updates ports $p_1, p_2, ..., p_k$ respectively to lay down a new path through the network with u_1 updating the ingress port. Suppose also that the ports updated by $us = [u_2, ..., u_k]$ are unreachable in network configuration C_1. Hence, us is an unobservable update. Since $[u_1]$ updates an ingress port, it is a one-touch update and also per-packet consistent. By the composition principle, $us +\!\!+ [u_1]$ is a per-packet consistent update.

Notice that the path update is achieved by first laying down rules on switches 2 to k and then, when that is complete, laying down the rules on switch 1. A well-known (but still common!) bug occurs when programmers attempt to install new forwarding paths but lay down the elements of the path in wrong order [11]. Typically, there is a race condition in which packets traverse the first link and reach the switch 2 before the program has had time to lay down the rules on links 2 to k. Then when packets reach switch 2, it does not yet know how to handle them, and a default rule sends the packets to the controller. The controller often becomes confused as it begins to see additional packets that should have already been dealt with by laying down the new rules. The underlying cause of this bug is explained with our model—the programmer intended a per-packet consistent update of the policy with a new path, but failed to implement per-packet consistency correctly. All such bugs are eradicated from network programs if programmers use per-packet consistent updates and never use their own ad hoc update mechanisms.

Two-phase update. So far, all of our update mechanisms have applied to special cases in which the topology, existing configuration, and/or updates have specific properties. Fortunately, provided there are a few bits in packets that are irrelevant to the network properties a programmer wishes to enforce, and can be used for bookkeeping purposes, we can define a mechanism that handles arbitrary updates using a two-phase update protocol.

Intuitively, the two-phase update works by first installing the new configuration on internal ports, but only enabling the new configuration for packets containing the correct version number. It then updates the ingress ports one-by-one to stamp packets with the new version number. Notice that the updates in the first phase are all unobservable, since before the update, the ingress ports do not stamp packets with the new version number. Hence, since updating ingress ports is per-packet consistent, by the composition principle, the two-phase update is also per-packet consistent.

To define the two-phase update formally, we need a few additional definitions. Let a version-property be a trace property that does not distinguish traces based on the value of version tags. A configuration C is a version-n configuration if $C = (S, T)$ and S modifies packets processed by any ingress port p_{in} so that after passing through p_{in}, the packet's version bit is n. We assume that the S function does not otherwise modify the version bit of the packet. Two configurations C and C' coincide internally on version-n packets whenever $C = (S, T)$ and $C' = (S', T')$ and for all internal ports p, and for all packets pk with version bit set to n, we have that $S(p, pk) = S'(p, pk)$. Finally, an update u is a refinement of S, if for all located packets lp in the domain of u, we have that $u(lp) = S(lp)$.

Definition 5 (Two-phase Update) *Let $C_1 = (S, T)$ be a version-1 configuration and $C_2 = (S', T)$ be a version-2 configuration. Assume that C_1 and C_2 coincide internally on version-1 packets. Let $us = [u_1^i, ..., u_m^i, u_1^e, ..., u_n^e]$ be an update sequence such that*

- *$S' = override(S, us)$,*
- *each u_j^i and u_k^e is a refinement of S',*
- *p is internal, for each (p, pk) in the domain of u_j^i,*
- *and p is an ingress, for each (p, pk) in the domain of u_k^e.*

Then us is a two-phase update from C_1 to C_2.

Theorem 6 *If us is a two-phase update then us is per-packet consistent.*

The proof simply observes that $us_1 = [u_1^i, ..., u_m^i]$ is an unobservable update, and $us_2 = [u_1^e, ..., u_n^e]$ is a one-touch update (and therefore per-packet consistent). Hence, by composition, the two-phase update $us_1 +\!\!+ us_2$ is per-packet consistent.

Optimized mechanisms. Ideally, update mechanisms should satisfy *update proportionality*, where the cost of installing a new configuration should be proportional to the size of the configuration change. A perfectly proportional update would (un)install just the "delta" between the two configurations. The full two-phase update mechanism that installs the full new policy and then uninstalls the old policy lacks update proportionality. In this section, we describe optimizations that substantially reduce overhead.

Pure extensions and retractions are one important case of updates where a per-packet mechanism achieves perfect proportionality. A pure extension is an update that adds new paths to the current configuration that cannot be reached in the old configuration—*e.g.*, adding a forwarding path for a new host that comes online. Such updates do not require a complete two-phase update, as only the new forwarding rules need to be installed—first at the internal ports and then at the ingresses. The rules are installed using the current version number. A *pure retraction* is the dual of a pure extension in which some paths are removed from the configuration. Again, the paths being removed must be unreachable in the new configuration. Pure retractions can be implemented by updating the ingresses, pausing to wait until packets in flight drain out of the network, and then updating the internal ports.

If paths are not only added or removed but are modified then more powerful optimizations are available. Per-packet consistency requires that the active paths in the network come from either of the configurations. The subset mechanism works by identifying the paths that have been added, removed or changed and then updating the rules along the entire path to use a new version. This optimization is always applicable, but in the degenerate case it devolves into a network-wide two-phase update.

6. CHECKING PROPERTY INVARIANCE

As per-packet consistent updates preserve all trace properties, programmers can turn any trace property checker that verifies individual, static network configurations into a verification engine that verifies the *invariance* of trace properties as configurations evolve over time. In this section, we demonstrate this idea concretely using a model checker, but it applies quite broadly; any kind of static network analysis can also benefit from our abstractions.

Model Checking Trace Properties. Intuitively, trace properties describe the paths that an individual packet is allowed to travel through the network, as well as the packet state at each point in the path. Temporal logic is a natural fit for the specification of such properties.

We use Computation Tree Logic (CTL) [15], a branching time temporal logic, to specify the legal paths that a packet may take through the network. Simple formulae describe attributes of packets in the network. For example, the formula

```
port = 3 & src_ip = 10.0.0.1
```

describes a packet with a source IP $10.0.0.1$ at port 3. As a second example, recall that our network model contains two special ports, $Drop$ and $World$. A dropped packet is one that arrives at the $Drop$ port; packets forwarded through our network correctly arrive at the $World$ port.[4] Hence the formula `port = DROP` simply states that the packet in question has been dropped.

While simple formulae describe a packet at one place in the network, the following quantified formulae describe a packet's path through the network.

- `AX` ϕ or `EX` ϕ. The formula ϕ holds at the next position on all paths (`AX`) from the current position, or at the next position along at least one path (`EX`).
- `AF` ϕ or `EF` ϕ. On all paths (`AF`) or on some path (`EF`) from the current position, ϕ holds on *some* future position.
- `AG` ϕ or `EG` ϕ. On all paths (`AG`) or on some path (`EG`) from the current position, ϕ holds on *all* future positions.

With these operators, we can specify many interesting trace properties. For example, the 'no black holes' property (incoming traffic must not be dropped) can be specified as `AG (port != DROP)`. Read aloud, this formula says, "On all paths, and at all future positions on those paths, the current port is never $Drop$."

Below, we state other useful trace properties as CTL formulae.

- **No loops:** every packet will eventually be dropped or leave the network. We represent this with the formula

  ```
  AF (port = DROP | port = WORLD).
  ```

  That is, on all paths through the network, a packet either reaches the outside world or is dropped.

- **Egress:** all packets with a header field `h = 1` reach the external world:

  ```
  h = 1 -> AF (port = WORLD).
  ```

  That is, all paths for packets with `h = 1` eventually lead to the $World$ port.

- **Waypointing:**[5] all packets with a header field `h = 1` reach switch `s4`.

  ```
  h = 1 -> AF (switch = s4).
  ```

  That is, all paths for packets with `h = 1` eventually pass through switch `s4`.

- **Blacklisting:** no packets with header `h = 0` traverse link (`s2`, `s3`).

[4] Recall that in our model, no packet is ever silently dropped—packets are only explicitly dropped by virtue of being sent to the $Drop$ port.

[5] Though the model from the previous section did not mention switches, our implementation contains a notion of a set of ports belonging to a switch.

```
(h = 0) ->
  AG (switch = s2 -> ~(EX switch = s3))
```

That is, if a packet header matches `h = 0`, then on all paths where the packet reaches switch `s2`, it will not reach switch `s3` on the next hop—i.e. it cannot traverse the edge between `s2` and `s3`, if such an edge exists.

Blacklisting, which stipulates that a packet must not reach an edge or switch, is roughly the negation of waypointing, which stipulates that a packet must pass through an edge or switch. Thus, both switches and edges can be waypointed or blacklisted. Moreover, both properties generalize to sets of edges and switches by adding a clause for each element to the conclusion of the implication. For example, the waypointing property where packets with headers `h = 0` must pass through switches `s1` and `s2` is:

```
h = 0 -> (AF switch = s1 & AF switch = s2).
```

Deploying Trace Verification. Static analyses play a significant role in the traditional software development cycle, because they offer an opportunity to verify program properties before the final software deliverable ships. Software-defined networking provides a similar opportunity for network programmers: network updates are programs and can be analyzed before being deployed to update the network. As in the traditional software development cycle, the judicious use of static analyses in network programming can pinpoint bugs before deployment, reducing time spent diagnosing performance problems and patching security vulnerabilities. We envision verification procedures, like the trace verification described above, being deployed as part of the following scenarios:

- **Planned change.** Before deploying a planned network update, programmers verify that the new configuration maintains expected security and reliability properties.
- **Debugging.** Trace verification can serve as an "assert" statement, allowing the programmer to quickly check properties of the configuration under development.
- **Background verification.** The programmer may also verify a network configuration after deployment, reducing the delay in making updates while still catching bad configurations in a reasonably timely manner.

We implemented a library of canonical formulae based on the properties presented above, which we used to verify the example benchmarks we report on in this paper. On small examples—with tens of switches and hundreds of rules—the NuSMV tool reports results in under a second. Larger examples—such as guaranteeing no loops for shortest-path routing over Waxman graphs [16] with hundreds of switches and hundreds of thousands of rules—take over an hour.We leave it as an open question for the model checking community to determine how far verification of network trace properties can scale.

7. PER-FLOW CONSISTENCY

Per-packet consistency, while simple and powerful, is not always enough. Some applications require a stream of related packets to be handled consistently. For example, a server load-balancer needs all packets from the same TCP connection to reach the same server replica. In this section, we introduce the per-flow consistency abstraction, and discuss mechanisms for per-flow consistent updates.

Per-flow abstraction. To see the need for per-flow consistency, consider a network where a single switch S load-balances between two back-end servers A and B. Initially, S directs traffic from IP addresses starting with 0 (*i.e.*, source addresses in 0.0.0.0/1) to A and 1 (*i.e.*, source addresses in 128.0.0.0/1) to B. At some time later, we bring two additional servers C and D online, and re-balance the load using a two-bit prefix, directing traffic from addresses starting with 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, all packets in existing flows must go to the same server, where a *flow* is a sequence of packets with related header fields, entering the network at the same port, and not separated by more than n seconds. The particular value of n depends upon the protocol and application. For example, the switch should send packets from a host whose address starts with "11" to B, and not to D as the new configuration would dictate, if the packets belong to an ongoing TCP connection. Simply processing individual packets with a single configuration does not guarantee the desired behavior.

Per-flow consistency guarantees that all packets in the same flow are handled by the same version of the configuration. Formally, the per-flow abstraction preserves all path properties, as well as all properties that can be expressed in terms of the paths traversed by *sets* of packets belonging to the same flow.

Per-flow mechanisms. Implementing per-flow consistent updates is much more complicated than per-packet consistency because the system must identify packets that belong to active flows. Below, we discuss three different mechanisms. Our system implements the first of the three; the latter two, while promising, depend upon technology that is not yet available in OpenFlow.

Switch rules with timeouts: A simple mechanism can be obtained by combining versioning with rule timeouts, similar to the approach in [7]. 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 soft timeouts on the rules for the old configuration and installs the new configuration at lower priority. When all flows matching a given rule finish, the rule automatically expires and the rules for the new configuration take effect. When multiple flows match the same rule, the rule may be artificially kept alive even though the "old" flows have all completed. If the rules are too coarse, then they may never die! To ensure rules expire in a timely fashion, the controller can refine the old rules to cover a progressively smaller portion of the flow space. However, "finer" rules require more rules, 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* configuration change before the previous one completes. However, this task can be implemented and optimized once in a run-time system, and leveraged over and over again in different applications.

Wildcard cloning: An alternative mechanism exploits the wildcard *clone* feature of the DevoFlow extension of OpenFlow [17]. When processing a packet with a *clone* rule, a DevoFlow switch creates a new "microflow" rule that matches the packet header fields exactly. In effect, clone rules cause the switch to maintain a concrete representation of each active flow. This enables a simple update mechanism: first, use clone rules whenever installing configurations; second, to update from old to new, simply replace all old clone rules with the new configuration. Existing flows will continue to be handled by the exact-match rules previously generated by the old clone rules, and new flows will be handled by the new clone rules, which themselves immediately spawn new microflow rules.

While this mechanism does not require complicated bookkeeping on the controller, it does require a more complex switch.

End-host feedback: The third alternative exploits information readily available on the end hosts, such as servers in a data center. With a small extension, these servers could provide a list of active sockets (identified by the "five tuple" of IP addresses, TCP/UDP ports, and protocol) to the controller. As part of performing an update, the controller would query the local hosts and install high-priority microflow rules that direct each active flow to the assigned server replica. These rules could "timeout" after a period of inactivity, allowing future traffic to "fall through" to the new configuration. Alternatively, the controller could install "permanent" microflow rules, and explicitly remove them when the socket no long exists on the host, obviating the need for any assumptions about the minimum interval time between packets of the same connection.

8. IMPLEMENTATION AND EVALUATION

We have built a system called Kinetic that implements the update abstractions introduced in this paper, and evaluated its performance on small but canonical example applications. This section summarizes the key features of Kinetic and presents experimental results that quantify the cost of implementing network updates in terms of the number of rules added and deleted on each switch.

Implementation overview. Kinetic is a run-time system that sits on top of the NOX OpenFlow controller [18]. The system comprises several Python classes for representing network configurations and topologies, and a library of update mechanisms. The interface to these mechanisms is through the `per_packet_update` and `per_flow_update` functions. These functions take a new configuration and a network topology, and implement a transition to the new configuration while providing the desired consistency level. Both functions are currently based on the two-phase update mechanism, with the `per_flow_update` function using timeouts to track active flows. In addition to this basic mechanism, we have implemented a number of optimized mechanisms that can be applied under certain conditions—*e.g.*, when the update only affects a fraction of the network or network traffic. The runtime automatically analyzes the new configuration and topology and applies these optimizations when possible to reduce the cost of the update.

As described in Section 5, the two-phase update mechanism uses versioning to isolate the old configuration and traffic from the updated configuration. Because Kinetic runs on top of OpenFlow 1.0, we currently use the VLAN field to carry version tags (other options, like MPLS labels, are available in newer versions of OpenFlow). Our algorithms analyze the network topology to determine the ingress and internal ports and perform a two-phase update.

Experiments. To evaluate the performance of Kinetic, we developed a suite of experiments using the Mininet [19] environment. Because Mininet does not offer performance fidelity or resource isolation between the simulated switches and the controller, we did not measure the time needed to implement an update. However, as a proxy for elapsed time, we counted the total number of install OpenFlow messages needed to implement each update, as well as the number of extra rules (beyond the size of either the old or new configurations) installed on a switch.

To evaluate per-packet consistency, we have implemented two canonical network applications: routing and multicast. The routing application computes the shortest paths between each host in the topology and updates routes as hosts come online or go offline and switches are brought up and taken down for maintenance. The multicast application divides the hosts evenly into two multicast groups and implements IP multicast along a spanning tree that connects all of the hosts in a group. To evaluate the effects of our optimizations, we ran both applications on three different topologies each containing 192 hosts and 48 switches in each of three different scenarios. The topologies were chosen to represent realistic and proposed network topologies found in datacenters (fattree, small-world), enterprises (fattree) and a random topology (waxman). The three scenarios can be divided up into:

1. Dynamic hosts and static routes
2. Static hosts and dynamic routes
3. Dynamic hosts and dynamic routes

In each scenario, we moved between 3 different configurations, changing the network in a well-prescribed manner. In the dynamic host scenario, we randomly selected between $10\% - 20\%$ of the hosts and added or removed them from the network. In the dynamic routes scenario, we randomly selected 20% of the routes in the network, and forced them to re-route as if one of the switches in the route had been removed. For the multicast example, we changed one of the multicast groups each time. Static means that we did not change the host or routes.

To evaluate per-flow updates, we developed a load-balancing application that divides traffic between two server replicas, using a hash computed from the client's IP address. The update for this experiment involved bringing several new server replicas online and re-balancing the load among all of the servers.

Results and analysis. Table 2 compares the performance of the subset optimization to a full two-phase update. Extension updates are not shown: whenever an extension update is applicable, our subset mechanism performs the same update with the same overhead. The two-phase update has high overhead in all scenarios.

We subject each application to a series of topology changes—adding and dropping hosts and links—reflecting common network events that force the deployment of new network configurations. We measure the number of OpenFlow operations required for the deployment of the new configuration, as well as the overhead of installing extra rules to ensure per-packet consistency. The overhead is the ratio of the number of extra rules installed during the per-packet update of a switch divided by the (maximum) number of rules in the old or new configuration. For example, if the old and new configurations both had 100 rules and during the update the switch had 120 rules installed, that would be a 20% overhead. The *Overhead* column in Table 2 presents the maximum overhead of all switches in the network. Two-phase update requires approximately 100% overhead, because it leaves the old configuration on the switch as it installs the new one. Because both configurations may not be precisely the same size, it is not always exactly 100%. In some cases, the new configuration may be much smaller than the old (for example, when routes are diverted away from a switch) and the overhead is much lower than 100%.

The first routing scenario, where hosts are added or removed, demonstrates the potential of our optimizations. When a new host comes online, the application computes routes between it and every other online host. Because the rules for the new routes do not affect traffic between existing hosts, they can be installed without modifying or reinstalling the existing rules. Similarly, when a host goes offline, only the installed rules routing traffic to or from that host need to be uninstalled. This leads to update costs proportional to the number of rules that changed between configurations, as opposed to a full two-phase update, where the cost is proportional to the size of the entire new configuration.

Application	Toplogy	Update	2PC		Subset		
			Ops	*Max Overhead*	*Ops*	*Ops %*	*Max Overhead*
Routing	Fat Tree	Hosts	239830	92%	119003	50%	20%
		Routes	266234	100%	123929	47%	10%
		Both	239830	92%	142379	59%	20%
	Waxman	Hosts	273514	88%	136230	49%	66%
		Routes	299300	90%	116038	39%	9%
		Both	267434	91%	143503	54%	66%
	Small World	Hosts	320758	80%	158792	50%	30%
		Routes	326884	85%	134734	41%	23%
		Both	314670	90%	180121	57%	41%
Multicast	Fat Tree	Hosts	1043	100%	885	85%	100%
		Routes	1170	100%	634	54%	57%
		Both	1043	100%	949	91%	100%
	Waxman	Hosts	1037	100%	813	78%	100%
		Routes	1132	85%	421	37%	50%
		Both	1005	100%	821	82%	100%
	Small World	Hosts	1133	100%	1133	100%	100%
		Routes	1114	90%	537	48%	66%
		Both	1008	100%	1008	100%	100%

Experimental results comparing two-phase update (2PC) with our subset optimization (Subset). We add or remove hosts and change routes to trigger configuration updates. The *Ops* column measures the number of OpenFlow install operations used in each situation. The Subset portion of the table also has an additional column (Ops %) that tabulates (Subset Ops / 2PC Ops). *Overhead* measures the extra rules concurrently installed on a switch by our update mechanisms. We pessimistically present the maximum of the overheads for all switches in the network – there may be many switches in the network that never suffer that maximum overhead.

Table 2: Experimental results.

Our optimizations yield fewer improvements for the multicast example, due to the nature of the example: when the spanning tree changes, almost all paths change, triggering an expensive update.

We have not applied our optimizations to the per-flow mechanism, therefore we do not include an optimization evaluation of the load balancing application.

9. RELATED WORK

This paper builds on our earlier workshop paper [20], which did not include a formal model or proofs, the optimized per-packet update mechanisms, formal verification of network configurations, or an implementation and evaluation.

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 [8, 2, 3]. The recent work by Vanbever *et. al* [5] 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.

Consensus Routing [9] seeks to eliminate transient errors, such as disconnectivity, that arise during BGP updates. In particular, Consensus Routing's "stable mode" is similar to our per-packet consistency, though computed in a distributed manner 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 framework. The BGP-LP mechanism from [21] is essentially per-packet consistency for BGP.

The dynamic software update problem is related to network update consistency. The problem of upgrading software in a general distributed system is addressed in [22]. The scope of that work differs in that the nodes being updated are general purpose computers, not switches, running general software.

The related problem of maintaining safety while updating firewall configurations has been addressed by Zhang *et. al* [23]. That work formalized a definition of safety similar in spirit to per-packet consistency, but limited to a single device.

In the past, a number of tools have been developed to analyze static network configurations [24]. Our main contribution is not to show that it is possible to analyze static network configurations (or that we can do so using a model checker). Rather, it is the fact that *any* such static analysis technique (such as header space analysis [14]) can be combined with our per-packet consistent update mechanisms to guarantee that trace properties are preserved as configurations evolve over time.

The model developed by Kazemian *et. al* [14] was the starting point for our own model. Since their model only spoke of a single, static configuration, we extended the network semantics to include updates so we could model a network changing dynamically over time. In addition, while their model was used to help *describe* their algorithms, ours was used to help us state and prove various correctness properties of our system.

10. CONCLUSIONS AND FUTURE WORK

Reasoning about concurrency is notoriously difficult, and network software is no exception. To make fundamental progress, the networking field needs simple, general, and reusable abstractions for changing the configuration of the network. Our per-packet and per-flow consistency abstractions allow programmers to focus their attention on the state of the network before and after a configuration change, without worrying about the transition in between. The update abstractions are powerful, in that the programmer does not need to identify the properties that should hold during the transition, since *any* property common to both configurations holds for

any packet traversing the network during the update. This enables lightweight verification techniques that simply verify the properties of the old and new configurations. In addition, our abstractions are practical, in that efficient and correct update mechanisms exist and are implementable using today's OpenFlow switches.

In our ongoing work, we are exploring new mechanisms that make network updates faster and cheaper, by limiting the number of rules or the number of switches affected. In this investigation, our theoretical model is a great asset, enabling us to prove that our proposed optimizations are correct. We also plan to extend our formal model to capture the per-flow consistent update abstraction, and prove the correctness of the per-flow update mechanisms. In addition, we will make our update library available to the community, to enable future OpenFlow applications to leverage these update abstractions. Finally, while per-packet consistency and per-flow consistency are core abstractions with excellent semantic properties, we want to explore other notions of consistency that either perform better (but remain sufficiently strong to provide benefits beyond eventual consistency) or provide even richer guarantees.

Acknowledgments. The authors wish to thank Hussam Abu-Libdeh, Robert Escriva, Mike Freedman, Tim Griffin, Mike Hicks, Eric Keller, Srinivas Narayana, Alan Shieh, the anonymous reviewers, and our shepherd Ramana Kompella for many helpful suggestions. Our work is supported in part by ONR grant N00014-09-1-0770 and NSF grants 1111698, 1111520, 1016937 and 0964409.

11. REFERENCES

[1] P. Francois and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Trans. on Networking*, Dec 2007.

[2] 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.

[3] S. Raza, Y. Zhu, and C.-N. Chuah, "Graceful network state migrations," *IEEE/ACM Trans. on Networking*, vol. 19, Aug 2011.

[4] D. Erickson *et al.*, "A demonstration of virtual machine mobility in an OpenFlow network," Aug 2008. Demo at *ACM SIGCOMM*.

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

[6] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing web traffic using OpenFlow," Aug 2009. Demo at *ACM SIGCOMM*.

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

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

[9] 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.

[10] 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.

[11] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *NSDI*, Apr 2012.

[12] Y. Bertot and P. Casteran, "Interactive theorem proving and program development: Coq'Art the calculus of inductive constructions," in *EATCS Texts in Theoretical Computer Science*, Springer-Verlag, 2004.

[13] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," pp. 359–364, Springer, 2002.

[14] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, Apr 2012.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 244–263, 1986.

[16] B. M. Waxman, "Broadband switching," ch. Routing of Multipoint Connections, pp. 347–352, IEEE Computer Society Press, 1991.

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

[18] 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.

[19] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *HotNets*, Oct 2010.

[20] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!," in *HotNets*, Nov 2011.

[21] 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.

[22] S. Ajmani, B. Liskov, and L. Shrira, "Modular software upgrades for distributed systems," in *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, (Berlin, Heidelberg), pp. 452–476, Springer-Verlag, 2006.

[23] C. C. Zhang, M. Winslett, and C. A. Gunter, "On the safety and efficiency of firewall policy deployment," in *IEEE Symp. on Security and Privacy*, 2007.

[24] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *NSDI*, May 2005.