Transparent, Live Migration of a Software-Defined Network

Soudeh Ghorbani[⋆] Cole Schlesinger[†] Matthew Monaco[‡] Eric Keller[‡]
Matthew Caesar[⋆] Jennifer Rexford[†] David Walker[†]

[⋆]University of Illinois at Urbana-Champaign [†]Princeton University [‡]University of Colorado

Abstract

Increasingly, datacenters are virtualized and software-defined. Live virtual machine (VM) migration is becoming an indispensable management tool in such environments. However, VMs often have a tight coupling with the underlying network. Hence, cloud providers are beginning to offer tenants more control over their virtual networks. Seamless migration of all (or part) of a virtual network greatly simplifies management tasks like planned maintenance, optimizing resource usage, and cloud bursting. Our LIME architecture efficiently migrates an *ensemble*, a collection of virtual machines and virtual switches, for any arbitrary controller and end-host applications. To minimize performance disruptions, during the migration, LIME temporarily runs all or part of a virtual switch on multiple physical switches. Running a virtual switch on multiple physical switches must be done carefully to avoid compromising application correctness. To that end, LIME merges events, combines traffic statistics, and preserves consistency among multiple physical switches even across changes to the packet-handling rules. Using a formal model, we prove that migration under LIME is *transparent* to applications, *i.e.,* any execution of the controller and end-host applications during migration is a completely valid execution that could have taken place in a migration-free setting. Experiments with our prototype, built on the Floodlight controller, show that ensemble migration can be an efficient tool for network management.

Categories and Subject Descriptors C.2.3 [*Computer-Communication Networks*]: Network Operations

Keywords Virtualization, Migration, Correctness, Transparency, Consistency, Software-defined networks

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3252-1/14/11…$15.00.
http://dx.doi.org/10.1145/2670979.2670982

1. Introduction

Multi-tenant cloud environments are increasingly "software-defined" and virtualized, with a controller orchestrating the placement of VMs and the configuration of the virtual networks. In a software-defined network (SDN), the controller runs applications that install packet-processing rules in the switches, using an API like OpenFlow [48]. In a virtualized multi-tenant cloud, each tenant has its own VMs, a controller application, and a "virtual network", catered to the tenant's requirements, that are completely under its control [12, 19, 44]. The virtual networks are only as complicated as needed to express the tenants' desired policies: One tenant might prefer a single virtual "big-switch" for managing its traditional enterprise workload [19, 44], while a multi-tier virtual topology might suit the web-service workload of another tenant better [44].

VM migration is an invaluable management tool for cloud applications [8, 16, 17, 20, 55]. Frequently migrating VMs, both within and across data centers, gives network administrators the flexibility to consolidate servers, balance load, perform maintenance, prepare for disasters, optimize user performance, provide high fault-tolerance, and reduce bandwidth usage without disrupting the applications [16].

However, a VM rarely acts alone. Modern cloud applications consist of multiple VMs that have a tight coupling with the underlying network, and almost all network policies, *e.g.,* policy routes, ACLs, QoS, and isolation domains, depend on the topology of the virtual network and the relative positions of the VMs [19]. VM migration (without migrating the virtual networks), therefore, necessitates re-implementing the policies—a notoriously complex and error-prone task that often requires significant manual reconfiguration [19]. Moreover, changes in the *physical* locations of the VMs triggers additional, unexpected "events" for the virtual network's control plane to handle. *Therefore, when the VMs migrate, the virtual networks should move, too.*

In this paper, we propose that live migration of an *ensemble*, a set of virtual machines and virtual switches, should be a core infrastructure primitive in cloud networks. Moreover, we propose that this live migration should be *seamless*, so that arbitrary applications function correctly throughout transitions. Such an ability would be an invaluable management tool for cloud providers for:

Planned maintenance: Migrate all (or part) of a virtual network to perform maintenance on a physical switch without disrupting existing services.

Optimizing resource usage: Migrate a resource-hungry tenant to another "pod" during peak load, and consolidate onto fewer physical components otherwise.

Cloud bursting: Migrate all or part of an enterprise's application between a private data center and the public cloud as the traffic demands change.

Disaster recovery: Migrate a tenant's network to a new location before a natural disaster (e.g., a hurricane), or after a data center starts operating on backup power.

In each case, live ensemble migration would minimize performance disruptions and avoid the unwieldy, error-prone task of reconfiguring the new servers and switches[1].

Migrating the virtual network amounts to mapping the virtual topology to a new set of physical switches. On the surface, this seems like a simple matter of copying the packet-processing rules from the old physical switch to the new one. In practice, migrating an SDN is much more challenging. While migrating a VM involves a temporary "freeze" for the final copy of the run-time state, we cannot "freeze" the switches, even for a short period, without causing excessive packet loss for several seconds ([47, 52, 54] and Section 5.1.2). This excessive loss rate causes connection drops and is considered unacceptable for "live" migration [52]. To avoid that, our LIME (Live Migration of Ensembles) architecture "clones" the virtual network, so both instances of the network can carry traffic during the migration process.

However, *running multiple clones of a virtual network concurrently while projecting the view of a single network to the control applications could lead to incorrect application-level behaviors* [27]. A NAT that drops legitimate traffic, a firewall that erroneously blacklists legitimate hosts, and a load-balancer that overloads some servers while leaving the rest of the servers underutilized are some examples of the applications behaving unexpectedly because of cloning [27]. To project a correct view of a single network when running multiple clones simultaneously, we must combine events from multiple clones into a single event stream to the controller application to preserve the semantics of the applications that react to events sent by the switches. A greater challenge arises in updating the rules in the virtual switch during the migration process. Simply deferring all updates until the migration process completes would compromise performance and reliability by not reacting to unplanned

[1] More use-cases for virtual network migration, *e.g.,* to increase utilization, increase revenue, decrease cost, decrease energy consumption, and enhance performance are provided in [10, 18, 37].

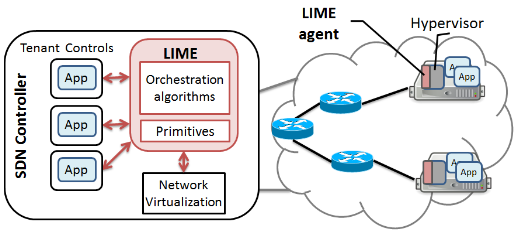


Figure 1: LIME architecture.

events like link failures. Yet, updating multiple clones at the same time is challenging, since unavoidable variations in delays mean that different instances of a virtual switch would commit the same rule at different times. The problem is exacerbated by the fact the tenants can run any applications, with arbitrary, unknown inter-packet dependencies. To serve as a general management tool, our migration solution must be correct for any possible workload.

To that end, we present a *network abstraction layer* that supports transparent migration of virtual networks (and VMs), so that *any* SDN application continues to work correctly. LIME runs on top of an SDN controller and below the (unmodified) controller applications, as shown in Figure 1. LIME provides each tenant with a virtual topology—identical to the topology provided by the network virtualization layer—and controller. It also augments the southbound SDN API, used by the network virtualization layer to implement the virtual networks on the physical network, with the migration primitives that enable seamless relocation of the virtual elements on the physical network. We ensure that this migration process is *transparent*—that any execution of the tenant's controller and end-host applications during migration is a completely valid execution that could have taken place in a migration-free setting. LIME does not require any modifications to the switches and applications. In designing, prototyping, and evaluating LIME, we make three research contributions:

Efficient cloning algorithm: To prevent disruptions during migration, multiple clones of a virtual switch may run simultaneously, while still forwarding and measuring traffic correctly, *even across changes to the rules* (§2).

Correctness proof: We develop a new network model and use that to prove that switch cloning works correctly for *any* controller and end-host applications, under *any* possible unspoken dependencies between different packets (§3). Reasoning about the correctness of concurrent distributed systems is notoriously challenging [9], and existing work on preserving correctness and consistency in networks fall short of detecting the incorrect application-level behaviors resulting from running multiple copies of a network concurrently [27]. To the best of our knowledge, our work is the first work that formally defines migration correctness and proposes an analytical framework to reason about it.

Performance evaluation: Experiments with our LIME prototype on a 13-switch SDN testbed ("sliced" to emulate a 45-switch fat-tree topology), with micro-benchmarks on Mininet [35], and "in the wild" experiments on Emulab [2] and FutureGrid [3] demonstrate that switch cloning enables network migration with minimal performance disruptions and controller overhead (§4, §5).

2. Transparent and Efficient Migration

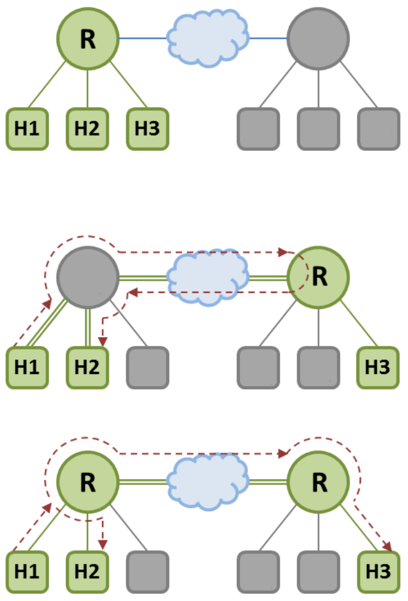
LIME must ensure correctness and efficiency when migrating all or part of a virtual switch from one physical switch to another[2]. The first challenge for designing a correct migration system is providing a correctness definition. In this section, we define a novel and intuitive notion of correctness for migration, *transparency*, before presenting a straw-man *move* primitive that ensures transparency at the expense of efficiency. Then, we show how switch *cloning* improves efficiency by allowing multiple copies of all or part of a virtual switch to run at the same time. Finally, we show how to update the rules in multiple clones without compromising transparency.

2.1 Transparency During Migration

Arbitrary tenant applications must continue to operate correctly despite the migration of virtual machines and switches. That is, applications should not experience any "important" differences between a network undergoing migration and a migration-free network. But what is an "important difference"? To speak precisely about transparency, we must consider the *observations* that the VMs and the controller application can make of the system. As in any network, these components can only make limited observations about the underlying network: (a) VMs observe packets they receive, (b) the controller can observe events (e.g., topology changes and packets the switch directs to the controller), and (c) the controller may query traffic statistics from the switches. Note that the VMs and the controller application cannot, for example, determine the precise location of every packet in the network.

On the other hand, there are some differences the tenant may detect during a migration: (a) latency or throughput of data traffic may change, (b) packets may be lost or delivered out of order, and (c) control messages between the controller and the switches may experience variable delays. We consider these to be acceptable deviations from a migration-free setting—networks are traditionally "best effort" and may suffer from packet loss, reordering, or fluctuations in latency or throughput, even in the absence of migration. As such, we consider a migration to be *logically unobservable* if, for all observations generated during a migration, there exists a migration-free execution that could make the same observations. Using a formal model, Section 3 proves that our migration primitives ensure this property.

[2] If a virtual switch is implemented using multiple physical switches, it might be desirable to migrate only a part of it, *e.g.*, the part residing on one physical switch.



(A) The switch R, along with VMs on hosts H1, H2, and H3, make up an ensemble.

(B) *Moving* R forces H1 and H2 to tunnel to R's new location, even when they could communicate locally.

(C) *Cloning* R only sends necessary traffic through the tunnel.

Figure 2: A simple ensemble migration.

To illustrate the concept of transparency, consider a straw-man *move* primitive for migrating a virtual switch from one physical switch to another. To "move" a virtual switch, LIME first installs a copy of each rule into the new physical switch, before any packets can start reaching the new switch. Then, LIME "freezes" the old switch by first installing a high-priority "drop" rule (that drops all traffic entering the physical switch), and then querying the counters of the old rules to record the traffic statistics. (Later, LIME can combine these statistics with the values of counters in the new physical switch.) To direct traffic to the new physical switch, LIME would move the virtual links. After copying the rules of R from the physical switch on the left in Figure 2(A) to the one on the right, LIME creates tunnels from hosts $H1$ and $H2$ to it, bringing us to the state depicted in Figure 2(B).

Informally, the *move* primitive achieves transparency by ensuring that the new physical switch handles packets and generates events the same way as the old physical switch, with at most one switch operating at a time. To prevent inconsistencies in packet forwarding and monitoring, LIME has the old switch *drop* packets— acceptable in a best-effort network, and preferable to violating transparency. However, the *move* primitive is not efficient. First, a "frozen" switch is unresponsive, leading to a high rate of packet loss. Second, the tunnels to the new switch increase latency. In fact, neighboring components may send traffic over *two* tunnels—to the new location and back—even though they are physically close, as with $H1$ and $H2$ in Figure 2(B).

2.2 Running Multiple Clones of Switches

To reduce packet loss, LIME allows multiple running copies ("clones") of all or part of a virtual switch to coexist during the migration process. This allows traffic to traverse either instance of the switch without packet loss. Hence, as shown in Figure 2(C), cloning R to the physical switches on right and left allows $H1$ and $H2$ to communicate locally, and only traffic destined for (or originating from) $H3$ needs to traverse the tunnel. Better yet, as $H1$ and $H2$ migrate to their new locations, they can immediately gain the benefit of communicating locally with $H3$. Eventually, when the migration of VMs completes, the old instance of the switch is isolated and unused. At this point, LIME simply applies a *delete* to clear the state of a given switch instance in its old physical location, freeing its resources.

LIME ensures that the tenant's controller application sees a single virtual switch, despite the presence of multiple clones. LIME achieves transparency by *merging* the events from the clones into a single event stream for the virtual switch. The merging process must consider all possible events an SDN switch sends to the controller:

- **Packet-in events:** A packet-in event directs a packet to the controller. Since packets can be reordered in a best-effort network, the LIME controller can simply merge the ordered packet-in events coming from multiple physical switches.
- **Traffic statistics:** Requests for traffic statistics are relayed to each instance of the switch, and the results combined before the replies are relayed to the applications. For the existing traffic statistics of OpenFlow, such as byte counters, packet counters counters, and meters, this operation is as straight-forward as summing them into a single value for each rule.
- **Link failures:** If a link fails at either switch, LIME reports a link failure to the tenant application, making the (virtual) link unavailable at both physical switches.
- **Rule timeouts:** A switch can automatically delete rules after a hard timeout (a fixed time interval) or a soft timeout (a fixed period with no packet arrivals) expires. We cannot ensure that two physical switches delete rules at the same time. Instead, LIME does not use rule timeouts during the migration process, and instead *emulates* the timeouts[3].

The merging process cannot preserve the *relative order* of events sent by different physical switches, due to variable delays in delivering control messages. Fortunately, even a *single* physical switch does not ensure the ordering of events—and the OpenFlow specification does *not* require switches to create control messages in the order events occur. This is natural, given that switches are themselves small-scale distributed systems with multiple line cards, a switching fabric, *etc*. For example, packets arriving on different line cards on the same physical switch could easily trigger packet-in events in the opposite order. As such, we need *not* preserve the relative ordering of events at clones of a virtual switch.

[3] See our Tech Report [29] for further details on implementing this.

2.3 Updating Multiple Switch Clones

The LIME controller may need to install or uninstall rules during the migration process. Buffering every update until the migration completes can lead to an unacceptable performance disruption. Due to the distributed nature of switches, LIME cannot ensure that both clone instances are updated at exactly the same time. However, applying updates at different times can violate application dependencies. We give an illustrative example below.

Running Multiple Clones: What Could Go Wrong? As an example of the incorrect behaviors caused by running multiple clones of a switch, consider a campus network that has a peripheral gateway – all the traffic from external hosts to the internal hosts and vice versa go through this gateway. Assume, further, that in addition to this gateway, the campus network deploys a number of OpenFlow switches for implementing its policies. Specifically, it deploys a switch to act as a firewall with the following simple policies: (a) drop all traffic from internal to external hosts, and (b) black list any external host that sends traffic to any internal host, *e.g.,* when the controller receives a packet-in event from the firewall with an external host's address in the source address field, the controller black-lists that external host by installing the rules on the gateway for dropping the packets sent from that external host. The campus network also deploys another OpenFlow switch to perform destination based forwarding.

Now, imagine that initially the network operators want to have a policy, *policy 1*, to block any web-communication between internal and external hosts, blacklist any external host that tries to send web-traffic to any internal host, and permit any non-web communication between the internal and external hosts. This simple policy can be implemented by installing a single rule on the gateway to send all the web traffic to the firewall, and all other kinds of traffic to the forwarding switch. Assume that at some point, the policy is updated to *policy 2* which simply permits any web and non-web communication between the internal and external hosts. This policy update could be simply executed by updating a single rule on the gateway to send web-traffic to the forwarding switch, instead of the firewall.

In this scenario, if an external host receives web traffic from an internal host, it effectively *learns* that the policy update has been done, and it can send back web traffic to the internal host without getting black-listed.

However, if the gateway is *cloned* for migration, and the operators update the policy (to send web traffic to the forwarding switch instead of the firewall) while having two clones of the gateway, then one clone of the gateway could be updated while the other clone still has its old policy (*i.e.,* it still needs to be updated). In that case, the web traffic from the internal hosts to the external hosts could go through the updated clone of the gateway (and hence go through the forwarding switch and be delivered to the external host), but the web traffic from the external hosts to the internal hosts could go through the yet-to-be-updated clone of the gateway, and be forwarded to the firewall, which implies that the external host will be black-listed if it sends web-traffic to the internal hosts, even after it receives web traffic from internal hosts and learns that it's allowed to send them web-traffic. This breaks the transparency, as something (in this example, "blacklisting") that would not happen in a migration-free execution could happen while migrating.

Note that despite the visibly-incorrect behavior in the example above, some of the most common notions of correctness such as "per-packet consistency" [51] are preserved throughout: Each packet is handled by one configuration only. In fact, we have shown in a recent work that existing update mechanisms such as "consistent updates" [51] cannot prevent the incorrect application-level behavior (such as firewalls erroneously blacklisting legitimate hosts) caused by cloning [27]. Therefore, those mechanisms are insufficient for correct migration. LIME ensures transparency of migration by updating rules using one of two mechanisms:

- **Temporarily drop affected traffic:** This two-step approach provides transparency at the cost of some packet loss. Given a new rule from the controller, LIME first installs a rule on each clone instance with the same pattern but with an action of "drop" to discard the packets. Once LIME is notified that both switch instances have installed the new rule (e.g., through the use of a barrier), the second phase installs the original rule with the action specified by the application. This ensures that any dependent packet is either dropped or processed by the updated rule. Note that the packet loss only affects traffic matching the pattern in the updated rule, and only during the first phase of the update.
- **Temporarily detour affected traffic:** This mechanism provides transparency at the cost of increased latency. Given a rule from the controller, LIME picks one instance of the cloned switch, establishes tunnels to it, and routes all traffic through it (which can therefore be atomically updated). With the other clone instance isolated, LIME can safely update both instances. Once LIME receives notification that the updates have been applied, traffic can be restored to its original routes.

Either approach ensures transparency, with different tradeoffs between packet loss and packet latency.

3. Proving Transparency

In the previous section, we qualitatively defined transparency and argued how LIME migrates transparently. To be a reliable network management tool for critical infrastructures such as multi-tenant cloud environments, however, LIME's correctness needs to be analyzed more rigorously. Toward this goal, in this section, we develop an abstract network model to formalize the concept of "transparency" and use it to prove that LIME migrations are transparent, *i.e.,* every set of observations that the controller application and end-host virtual machines can make during a migration could also have been observed in a migration-free setting. Due to space constraints, the full technical development, proof of correctness, and supporting lemmas are presented in a separate technical report [29].

Packets:		
Bit	b	$::= 0 \mid 1$
Packet	pk	$::= [b_1, ..., b_n]$
Port	p	$::= 1 \mid ... \mid k$
Located Pkt (LP)	lp	$::= (p, pk)$
Observations:		
Query ID	id_q	
Host ID	id_h	
Observation	o	$::= id_q \mid (id_h, pk)$
Network State:		
Switch	S	$\in LP \rightharpoonup (LP\ Set \times ID_q\ Set)$
Topology	T	$\in (Port \times Port)\ Set$
Hosts	H	$\in Port \rightharpoonup (ID_h \times (LP \rightharpoonup LP\ Set))$
Packet Queue	Q	$\in Port \to Packet\ List$
Configuration	C	$::= (S, T, H)$
Updates:		
Switch Update	u_s	$\in LP \rightharpoonup LP\ Set \times ID_q\ Set$
Host Update	u_h	$\in Port \rightharpoonup ID_h \times (LP \rightharpoonup LP\ Set)$
Update	u	$::= (\mathsf{LIME}, u_s) \mid (\mathsf{LIME}, u_h)$
		$\mid (\mathsf{Controller}, u_s)$
Network	N	$::= (Q, C, [u_1, ..., u_n])$

Figure 3: Network elements.

3.1 The Network Model

We use a simple mathematical model to describe the fine-grained, step by step execution of a network, embodied in the relation $N \xrightarrow{os}{}^{\star} N'$, which states that a network N may take some number of steps during execution, resulting in the network N'. Our model extends the network model presented in [51] to account for end host and switch migration. But where the model in [51] produces sets of packet traces, the primary output of this relation is a multiset[4] os containing the observations that the controller and end hosts can make as the network transitions from N to N'. The traces of [51] essentially model an omniscient observer that can see the location of every packet at all times—useful for reasoning about per-packet consistency, but too strong an assumption for reasoning about transparency: an omniscient observer can easily distinguish between a network undergoing migration and a migration-free network.

[4] A multiset counts *how many* of each observation is made.

Basic Network Elements. Figure 3 defines the syntax of the network elements in our model. We take packets to be uninterpreted lists of bits, and ports bear unique names drawn from the set of integers. A *located packet* pairs a packet with a port, modeling its location in the network. An observation o comprises either an opaque query identifier id_q, representing information sent to the controller, or a pair (id_h, pk) of a packet pk arriving at a host named id_h.

Network State. As in [51], we model switch behavior as a switch function S from located packets to sets of located packets. We augment S to produce the set of observations made as the switch processes the packet—these observations represent both traffic statistics and packets sent directly to the controller. A relation T describes the topology: if $(p, p') \in T$, then a link connects ports p and p'. We use this to model both physical links and virtual tunnels. Hosts are modeled as a pair (id_h, f), where id_h is the name of the host and f is a function that models the behavior of the host: given a located packet lp, then $f(lp)$ is the (possibly empty) set of packets that id_h may produce after receiving the packet lp. A host map H connects ports to hosts. Finally, a queue Q maps each port in the network to a list of packets. Each list represents the packets to be processed at that port. We say that Q is an *initial* queue if every non-empty list is associated with a port connected to a host.

Network Updates. As the network runs, the controller may emit updates to the switch behavior. A switch update u_s is a partial function that describes new switch behavior, and $(\mathsf{Controller}, u_s)$ is a switch update sent from the controller. During a migration, LIME will also update switch behavior and the locations of end hosts. A host update u_h is a partial function that describes new host locations, and (LIME, u_h) is a host update sent from the LIME platform. We define *override* to update one partial function with another:

$$\begin{array}{rcl} override(S, u_s) & = & S' \\ \text{where } S'(lp) & = & \begin{cases} u_s(lp) & \text{if } lp \in \mathsf{dom}(u_s) \\ S(lp) & \text{otherwise} \end{cases} \end{array}$$

A network N is made up of a queue, a configuration, and a list of updates to be applied to the network.

Transitions. Transitions fall into two categories: processing a packet, and updating the network state. For example, the rule [Switch] in Figure 4 describes a packet being processed at a switch. Lines 1–7 roughly state:

(1) If p is a port,
(2) and Q' is a queue resulting from removing pk from the head of $Q(p)$,
(3) and C is made up of S, T, and H,
(4) and applying the switch function S to the packet pk located at port p results in a set of (possibly modified) packets lps at new ports, along with the observations ids_q,

$\boxed{\text{SWITCH}}$

$$\begin{array}{ll} \text{if } p \text{ is a port} & (1) \\ \text{and } (pk, Q') = dequeue(Q, p) & (2) \\ \text{and } C = (S, T, H) & (3) \\ \text{and } S((p, pk)) = (lps, ids_q) & (4) \\ \text{and } lps' = \{(p'_i, pk_i) \mid (p_i, pk_i) \in lps \wedge (p_i, p'_i) \in T\} & (5) \\ \text{and } Q'' = enqueue(Q', lps') & (6) \\ \text{then } (Q, C, us) \xrightarrow{ids_q} (Q'', C, us) & (7) \end{array}$$

$\boxed{\text{SWITCH UPDATE}}$

$$\begin{array}{ll} \text{if } us = (\_, u_s) :: us' \text{ is a list of updates} & (8) \\ \text{and } C = (S, T, H) & (9) \\ \text{and } C' = (override(S, u_s), T, H) & (10) \\ \text{then } (Q, C, us) \longrightarrow (Q, C', us') & (11) \end{array}$$

Figure 4: The network model.

(5) and forwarding each packet in lps across the topology yields a set of packets lps' at new locations,
(6) and Q'' is the result of adding the packets in lps' to the queue Q',
(7) then a network (Q, C, us) steps to (Q'', C, us), producing observations ids_q.

The rule [Switch Update] (Figure 4) updates the switch behavior. Lines 8–11 state:

(1) If u_s is at the head of the update list us,
(2) and C is made up of S, T, and H,
(3) and C' is the configuration resulting from overriding the switch function S with u_s,
(4) then (Q, C, us) steps to (Q, C', us') without producing any observations.

The rules for processing packets at hosts and updating the host map are similar [29].

We write $N \xrightarrow{os}{}^{\star} N'$ as the reflexive, transitive closure of the step relation defined in Figure 4, where os is the union of observations occurring at each step.

3.2 Formal Results

Armed with the network model, we can define precisely the behavior that makes a migration transparent. We begin by defining the observations that the controller and end hosts can make of an ensemble that is not undergoing migration.

Definition 1 (Migration-free Execution). *An execution*

$$(Q, N, us) \xrightarrow{os}{}^{\star} (Q', N', us')$$

is a migration-free execution if Q is an initial queue and all updates in us are of the form $(\mathsf{Controller}, u_s)$.

In other words, a migration-free execution simply runs an ensemble in its current location, possibly updating the switch behavior with rules from the controller, and os captures every observation that the controller and end hosts can make.

Definition 2 (Observationally Indistinguishable)**.** *For all pairs of update sequences, us_1 is observationally indistinguishable to us_2 if for all initial queues Q and executions*

$$(Q, C, us_1) \xrightarrow{os_1}{}^{\star} (Q', C', us_1')$$

there exists an execution

$$(Q, C, us_2) \xrightarrow{os_2}{}^{\star} (Q'', C'', us_2')$$

and $os_1 \subseteq os_2$.

Our definition of a transparent migration follows from the notion of *observation equivalence*, a common means of comparing the behavior of two programs [49]. Intuitively, a sequence of updates us_1 is observationally indistinguishable to another sequence us_2 if every observation produced by an execution applying us_1 can also be made in some execution applying us_2. Hence, a transparent migration is one that is observationally indistinguishable from a migration-free execution.

A migration can be described by two pieces of information: the new locations of each switch and port, and the sequence of updates that enacts the migration to the new locations. We use a partial function M from ports to ports to describe the new locations: $M(p) = p'$ if p is being migrated to p'. And we observe that **clone** (and **move**) can be represented as a sequence of switch and host updates.

During a migration, the LIME framework instruments switch updates from the controller. An update sequence us from LIME that both effects a migration and contains instrumented updates from the controller can be split into two lists: one that effects the migration, and another that contains instrumented updates from the controller. We write $us_C \sqsubset us$ to mean that us_C is the original sequence of controller updates that have been instrumented and interspersed with migration updates.

Theorem 1 (Migrations are Transparent)**.** *For all port maps M and update sequences us that define a migration, let us_C be the list of uninstrumented updates from the controller such that $us_C \sqsubset us$. The following conditions hold:*

- *us_C induces a migration-free execution, and*
- *the sequence us is observationally indistinguishable from us_C.*

The proof proceeds by induction on the structure of the network execution induced by us and hinges on preoserving two invariants, roughly stating:

1. Every VM is either located at its new or old location, as defined by M, and
2. Every rule that was in the switch configuration at the start of the migration is either installed in its original location, new location, or both and perhaps modified to forward packets into a tunnel (**clone** and **move** primitives), or an identical rule is installed that drops or detours all packets and emits no observations (**update** algorithms).

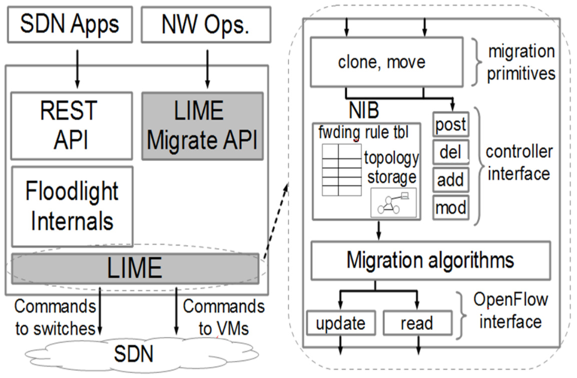


Figure 5: Prototype Implementation.

Together, these invariants ensure that after every step of execution, every packet is either processed in the original network, or in the new location with an equivalent rule, or is dropped before emitting any observations.

4. Implementation

To evaluate the performance of our system in practice, we built a prototype implementation using the Floodlight controller platform [24]. An overview of our design is shown in Figure 5. Floodlight runs a series of *modules* (*e.g.,* user applications), and each module is supplied with common mechanisms to control and query an OpenFlow network [6]. Our design is naturally implemented as a layer (which is itself a module) residing between other modules and the controller platform. LIME exposes much of the same interface as the controller platform—modules that wish to be LIME clients simply use the LIME interface instead. The LIME prototype instruments updates and queries received from client modules, and notifies clients of underlying changes to the topology.

LIME exposes two additional interfaces. A *migration interface* allows a network operator to configure ensembles, specify migration destinations, define custom migration algorithms, and enact ensemble migrations. An *endpoint interface* connects the LIME module to end hosts running VMs. A small daemon program on each end host relays LIME host migration commands to the KVM virtual machine platform [4], which performs the VM migration.

A key challenge in designing the prototype lay in managing the underlying network state to mask migration from client modules. LIME maintains a lightweight network information base (NIB), similar to what the Onix platform provides [43]. But unlike Onix, LIME maintains the state each client module *would* see, were the network not undergoing migration. LIME updates the NIB at each step in the migration, as well as when it receives queries and updates from clients.

Our prototype implements a simple tunneling algorithm to set up and tear down virtual links in the network. Traffic destined to enter a tunnel is tagged with a unique VLAN value unused by either the client module or other tunnels. Using the underlying topology graph provided by Floodlight, rules are installed along the shortest path to the tunnel destination; each rule is predicated on the VLAN tag, thus matching only traffic passing through the tunnel.

Finally, the LIME prototype implements the migration primitives described in Section 2. Using the primitives, we construct the following two end-to-end orchestration algorithms, and Section 5 reports on our experience using these algorithms in practice, providing empirical evidence of their tradeoffs:

Clone-based Migration: A *clone-based migration* will *clone* the switches and *move* the hosts[5]; it proceeds as follows. First, *clone* each switch. For each switch instance, select a virtual link to the closest neighbor; for cloned instances of switches connected to hosts, this will be a virtual link to the original host location. Then, for each host, *move* the host to its new location and update the links its neighbors use to communicate with it. Finally, delete the original clone instances.

This scheme minimizes back-haul traffic: for every path through the migrating network between every pair of hosts, traffic will traverse at most one virtual link between the new and old locations. It also minimizes packet loss: the only loss occurs during the brief time each VM is frozen during its move or when the rules are updated. The disadvantage lies in the additional overhead needed to maintain a consistent view of the network state. Note that from the two update primitives described in Section 2, LIME's prototype currently does not support the *detouring* mechanism because it requires some capabilities of the newer SDN protocols, such as *bundle messages* or *group tables* in OpenFlow 1.4 for atomically updating multiple rules or having a unified set of actions for multiple rules, that are not yet supported in the Floodlight controller.

Move-based Migration: Despite the advantages of clone-based migration, the additional overhead may be undesirable in some scenarios, such as an ensemble that includes a controller under heavy load. Extending this scenario, let us also say that we intend to migrate this ensemble across a wide-area network, where sending traffic between the new and old locations is expensive. However, the service deployed to this ensemble happens to be implemented robustly, such that it can tolerate some packet loss between VMs and total packet loss for a reasonably short period of time. Conveniently, the *move* and *clone* primitives are flexible enough to create an algorithm tailored for such a situation.

We call this algorithm a *move migration*, and it proceeds as follows. First, *move* each switch to its new location, but, as part of the move, leave the hosts connected to the (now empty) original locations—dropping any traffic. Next, *move* each host, reestablishing its connection at the new location. As a result, the ensemble will initially experience a brief window of downtime, followed by a period of time where only hosts in their new locations can communicate with each other, minimizing inter-location traffic.

[5] LIME does not currently support *VM* cloning as a migration technique.

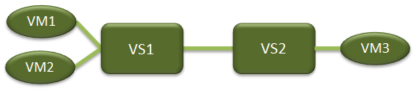


Figure 6: Topology for testbed experiment.

5. LIME Performance Evaluation

Modern datacenters serve many interactive applications such as search or social networking that require very low latency and loss rate. Even small increases in packet loss and delay can dramatically degrade user-perceived performance [13]. Hence, for LIME to be an "efficient" network management tool, we must keep increased packet loss and delay to a minimum, and ensure migration completes quickly. Moreover, migration should not introduce significant overhead on either the control or data planes.

In this section, we quantify performance and overhead through experiments with our prototype. In summary, (a) we evaluate transient behavior during migration, in terms of packet loss and latency, during intra- and inter- datacenter migration. We also compare the transient behavior of networks when using our "move-based" and "clone-based" orchestration algorithms (§5.1). (b) We show that LIME has negligible control and data plane overhead (§5.2). (c) We show that LIME scales by studying migration time as a function of the size and structure of the network topology. We also show that the limited migration performance and overhead penalties persist for a very short period of time with LIME, even in large networks (§5.3). (d) We close this section with a case study that demonstrates the benefits of migration for reducing congestion in datacenters (§5.4).

5.1 Performance During Migration

Running experiments on the Emulab [2] and FutureGrid [3] testbeds, we show that LIME introduces minimal packet loss and latency during intra- and inter-datacenter migration (§5.1.1). Moreover, using MiniNet experiments, we show the efficiency of *clone* for avoiding packet loss during migration, compared to our straw-man *move* primitive and a naive approach for migrating without LIME (§5.1.2).

5.1.1 LIME Running "in the Wild"

Performance during migration depends on the bandwidth and latency between the components, and the number and size of VMs. To demonstrate our system and highlight its

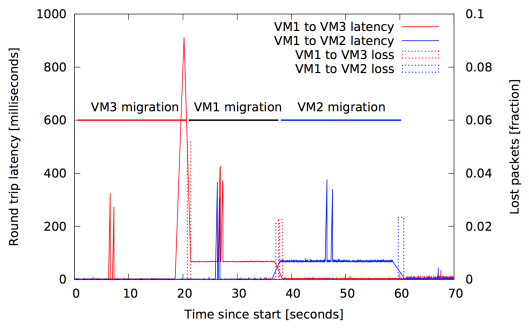


Figure 7: Time-series plot of the loss and latency observed during an ensemble migration from Emulab to FutureGrid. The horizontal lines each represent the time for which a VM is migrating.

performance properties, we conduct a live ensemble migration within and between two testbeds.

Testbed setup: Our experiments use the Emulab [2] and FutureGrid [3] testbeds. We run an ensemble consisting of three virtual hosts (each with a 1GB memory footprint and a pre-copied disk image) and two virtual switches, as shown in Figure 6. Within Emulab, we run KVM on a 2.4 GHz 64-bit Quad Core Xeon E5530 Nehalem processor, and connect the switches directly. Within FutureGrid, we have to run KVM nested within a FutureGrid VM and interconnect FutureGrid VMs with GRE tunnels [36] to emulate direct links. We use VM migration and switch cloning to migrate the entire ensemble from Emulab to FutureGrid (inter-datacenter migration with a round-trip time of roughly 62ms and measured bandwidth of 120$Mbps$), as well as exclusively within Emulab (intra-datacenter migration). During the migration, VM1 pings VM2 and VM3 every 5ms.

Experimental results: Figure 7 shows a time-series plot of the loss and latency of two flows (from VM1 to VM2 and VM3, respectively). Migrating between Emulab and FutureGrid takes 65.5 seconds; migrating the same topology within Emulab takes 21.3 seconds. For each VM migration, we notice a spike in latency at around the midpoint of the VM migration—we speculate that this is due to extra CPU load on the VM and contention for bandwidth between the ensemble and the VM migration. At the end of the migration, we see a short spike in packet loss—due exclusively to the "freeze" period of the VM migration. Migrating a VM affects the latency for communicating with other VMs in the ensemble. After migrating, VM3's communication with VM1 must traverse the wide area, leading to higher latency. Fortunately, the "clone" mechanism ensures that two VMs at the same site can communicate locally through the local instance of the switch. So, once VM1 migrates, the two VMs communicate solely within Emulab, with much lower latency (0.25 ms), even while VM2 completes its own migration process. It should be noted that the limited loss observed

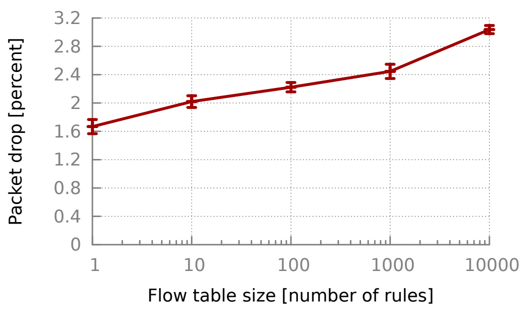


Figure 8: Percentage of packets lost during "move" migration (migration with "clone" is lossless).

is caused by VM (and not network) migration, and network migration with the *clone* algorithm is lossless (as shown in the following subsection using MiniNet experiments).

5.1.2 Transient Performance with Clone vs. Move

Migration without using LIME can lead to high packet loss. Under the *move* orchestration algorithm of LIME, applications experience a period of packet loss after stopping the old switch(es). Under the *clone* orchestration algorithm, there is no packet loss due to switch migration.

To quantify this effect, we experiment with Mininet HiFi [35] running on Intel Core i7-2600K machines with 16GB memory. The default Floodlight applications (routing, device discovery, etc.) run throughout the experiment. We use a simple topology with two hosts connected to one virtual switch. We migrate this network using three approaches: (a) a naive *freeze and copy* approach, where a simple script retrieves the network state, freezes the source network, and copies the rules to the destination, (b) LIME's *copy and freeze* algorithm (*move*), and (c) LIME's *clone* algorithm. We vary the number of rules in the switches to see how the size of the state affects performance, and send packets at a rate of 10,000 packets per second. We then measure packet losses in a 1-second time window, where the migration command is issued at the beginning of the time interval.

The loss rate of *freeze and copy* is 100% during the first second, and we continue having packet drops for 5 seconds. Figure 8 shows that the loss rate with "move" is also relatively high, e.g., for networks with around 10,000 rules per switch, packet loss during a "move" is 3%. The packet loss percentage drops to 0% after the first second. The loss rate is 0% with *clone* (not shown), and we observe no performance degradation in communications. In summary, we observed that migrating switches with *clone* is *live*, *i.e.,* end-to-end communication does not degrade during the migration.

5.2 Migration Overhead

Despite the performance benefits, LIME does cause higher processing overhead at the controller, and longer forwarding

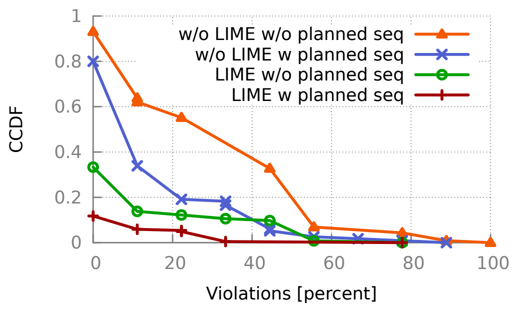


Figure 9: Bandwidth violations.

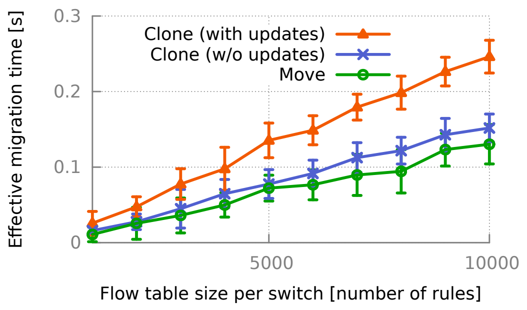


Figure 10: Migration latency as flow-table size grows.

paths due to tunneling. In this section, we show these overheads are small and manageable.

5.2.1 Control-Plane Overhead

To measure processing overhead at the controller, we use the `cbench` OpenFlow controller benchmarking tool [1], configured to emulate 30 switches sending packet-in events from 1 million source MACs per switch. As a baseline, we compare the number of packet-in events a LIME-augmented controller can handle compared to an unmodified Floodlight controller. The LIME prototype processed around 3 M events per second as compared to 3.2 M for the unaugmented Floodlight controller—an overhead of roughly 5.4%.

5.2.2 Data-Plane Overhead

By keeping traffic entirely in the old or the new physical network, LIME's *clone* algorithm reduces the bandwidth overhead compared to iteratively using *move*. To verify this, we simulate the same virtual and physical topologies, bandwidth requirements, and constraints used in prior work [14, 34, 59], and place $30-90\%$ load on the network by initially allocating virtual networks using an existing allocation algorithm [34]. We then select random virtual networks from the allocated virtual networks to migrate, randomly determine the destination of migration, and migrate them with the *clone* algorithm.

When the network is moderately or highly loaded, migration with LIME violates the bandwidth constraints much less often than migration without LIME (*i.e.,* sequentially moving switches). Moreover, our prior work shows that bandwidth cost of migration could be reduced by planning the sequence of migrating the VMs (as opposed to migrating them with some random ordering) [26]. Using the same sequence-planning algorithm to determine which nodes to move along with LIME offers further reductions in the bandwidth violations.

As an example, Figure 9 shows the complementary CDF of number of violations when 100 randomly picked virtual networks are migrated to locations determined by the SecondNet algorithm [34]. Virtual networks have 10 switches in form of trees (branching factor$=3$) each connected to one VM, and with links with 10Mbps bandwidth requirement. They are placed on top of a 200-node physical tree network where each substrate node has capacity for hosting 2 VMs, and substrate links have bandwidth 500Mbps. The network is initially loaded with around 70% of the maximum possible load.

On average, migrating with LIME and with and without planning the sequence of migrating VMs resulted in violations for migrating 2.7% and 9% of the time, respectively. While migrating without LIME, with and without using the sequence planning algorithm caused violations 17.8% and 35.2% of times, respectively.

In summary, migration with *clone* significantly reduces data-plane overhead, compared with *move*.

5.3 Migration Duration

To be a useful management tool, migration must complete quickly. In this section, we show that the duration of migration process with LIME is negligible even for large topologies for both hardware and software switches.

Testbed setup: We deploy the Ocean Cluster for Experimental Architectures in Networks, OCEAN [5], which is an SDN-capable network testbed composed of 13 Pica8 Pronto 3290 switches each with 48 ports. In order to run the experiments on a network with a larger scale and a more realistic topology, we *emulate* a fat-tree topology built out of 45 6-port switches by "slicing" the switches of OCEAN. That is, we deploy a simple home-grown virtualization layer, placed between LIME and the switches, that maps the data plane IDs and ports of the physical switches to the virtual switch IDs and ports of the virtual 6-port switches (more details can be found in [29]). LIME runs on a VM, with a 4,000-MHz CPU and 2 GB RAM, on a server in the same testbed. On the 45-switch fat-tree network, we then populate an aggregate switch and two edge switches connected to it and use the primitives of LIME (*clone*, *move*, and *update*) to migrate them to random subgraphs with similar topologies.

As a **baseline**, we also implement and compare performance with the *freeze and copy* approach described in §5.1.2. Moreover, for testing the sensitivity of migration duration to parameters such as using software *vs.* hardware switches, the topologies and scales of the underlying physical networks, and the topologies and scales of the virtual networks, we also experiment with virtual networks of various scales and topologies placed on top of networks with different scales and topologies built of both hardware and software switches, including various pairings of VL2, BCube, Fat-Tree, scale free, and tree topologies of varying sizes, and measure the migration time as well as the duration of updating the rules under LIME.

Experimental results: As shown in Figure 10, the migration duration stays under a second across a range of flow-table sizes, as computed over ten runs for each data point. Migration time increases roughly linearly with the number of rules that require special handling (*e.g.,* the rules that require cloning or translation), and *move* takes slightly less time than *clone* (since it is simpler). It also shows that migrating the network while using our update mechanism (explained in §2 and §4) still takes under a second despite the added complexity and the extra steps taken by the update mechanism.

The baseline approach has substantially worse migration times. For example, when migrating a 3-switch topology, where each switch has $10,000$ rules, migration with the naive approach takes over 20 seconds while the migration time for the exact same topology with LIME is under 0.3 seconds. Even worse, the naive approach causes significant performance degradation (see §5.1.2).

Our experiments with various topologies show that migration time depends on the number of rules requiring handling by LIME but is otherwise independent of the topology [29]. Our experiments with OVS software switches [7] (instead of hardware Pica8 switches) show similar results and trends in terms of the migration time [29].

In summary, migration duration is negligible with LIME's *clone* and *move* (with *move* slightly faster than *clone*), and is a function of the network size (*i.e.,* the number of rules), and is independent of the topology.

5.4 Case Study: Reducing Congestion

One of the benefits of migration is the flexibility it provides to balance traffic to reduce congestion. Congestion is prevalent in datacenters even when spare capacity is available elsewhere [31], e.g., while servers under a ToR can communicate with full line rate, inter-ToR communication usually traverses congested links, degrading application performance [15]. Even when the initial allocation of virtual networks is optimal, dynamics of the datacenters (incremental expansion and release of resources, failures, arrivals and departures of tenants, etc.) cause it to become inefficient over time. Hence, periodically re-optimizing the placement of tenants' virtual networks can reduce congestion on inter-cluster links [15, 30].

Our experiments show that inter-cluster congestion in common datacenter topologies, such as fat-tree, BCube [33], and DCell [32], could be significantly reduced with migration. For example, Figure 11 shows the results of an experiment where virtual networks of random sizes (random graphs with $40-60$ nodes, where each pair of nodes are connected with probability 1% with a link speed of $10Mbps$) are sequentially placed on a $27,648$-server fat-tree datacenter network in which all links between servers and ToR switches are $1Gbps$ while all other links are $10Gbps$. Initial allocations are done with an existing allocation algorithm [34] and we then applied a simple heuristic for migrating virtual networks (or parts of them) to reduce inter-ToR traffic in favor of increasing intra-ToR traffic [28][6]. As more virtual networks are allocated, the benefit of migration to reduce the load on inter-cluster links becomes more significant.

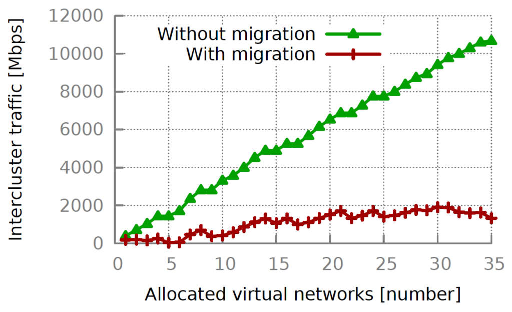


Figure 11: Reducing inter-cluster traffic by migration.

6. Related Work

LIME is a general technique for seamlessly migrating a virtual network running arbitrary control applications. As such, LIME relates to, and generalizes, previous work on migration mechanisms. LIME also relates to previous work on ensuring consistent network state. The current paper significantly expands on our workshop paper [42], with improved algorithms, a precise definition of correctness, a formal model and correctness proofs, and an implementation and evaluation.

6.1 Migration Mechanisms

Migrating and cloning VMs: Virtual machine migration is an old idea. Recent work shows how to make VM migration faster or efficiently migrate a collection of related VMs [11, 21, 23, 40, 53, 57, 58]. Remus [22] also supports *cloning* of VMs. Our work brings the benefits of migration to virtual *networks* supporting arbitrary end-host and controller applications.

Migrating at the network edge: Previous work shows how to migrate functionality running at the network *edge*—inside the VM or in the soft switch on the physical server. VIOLIN [39] runs virtual networks as overlays on hosts; the paper mentions that VMs running switches and routers could migrate, but does not present new migration techniques. VirtualWire [56] allows a VM to migrate from one

[6] See our Tech Report [28] for further discussion and analysis.

cloud provider to another, and use a tunnel to reattach the VM's virtual NIC to the network. Nicira's Network Virtualization Platform migrates a tenant's VM along with the relevant configuration of the soft switch on the server, so the VM can reattach to the overlay at its new location [44]. These approaches work for *software* switches and routers exclusively. VIOLIN and VirtualWire, for instance, implement virtual network components in VMs. In contrast, LIME migrates an entire virtual topology to a different set of hardware or software switches. Since "core" switches carry traffic for many VMs, LIME's switch migration must be *seamless*. This leads us to run multiple instances of a migrating virtual switch, and handle the resulting consistency challenges. Furthermore, migrating the VMs and the software switches at the edge could cause downtime for a few seconds [47, 52, 56] which is considered unacceptable for "live" migration since the active sessions are likely to drop [47, 52]. To avoid such downtime, similar to LIME, XenFlow and VMWare vNetwork Distributed Switch run multiple copies of the dataplane concurrently. Unlike LIME, however, they do not handle the possible resulting correctness violation [27].

Migrating in-network state: Other work shows how to migrate state inside the network [25, 41, 46, 54]. VROOM [54] can migrate a router running a BGP or OSPF control plane, not arbitrary control software; VROOM also requires modifying the network elements, whereas LIME works with unmodified OpenFlow switches and controller applications. Other work presents techniques for migrating *part* of the state of a component like a router [41]. These works have a different goal—explicitly *changing* the topology to direct traffic to a new location—and do not support arbitrary control applications and topologies.

LIME could be used for migrating the **middleboxes** that are implemented using only standard SDN switches and controller applications and operate correctly in best-effort networks. For the middleboxes that maintain and act on more sophisticated state or have different network requirements (such as requiring in-order delivery), the recent work on redistributing packet-processing across middleboxes [25, 50] could be used to migrate them. It should be noted, however, that these approaches require application source-code modifications, rely on some topological assumptions, and preserve consistency of state by synchronizing [50] or serialization [25] that comes at a significant performance cost.

XenFlow presents a hybrid virtualization system, based on Xen and OpenFlow switches, that migrates Xen VMs and the virtual network while providing isolation and QoS [47]. When migrating a VM, XenFlow runs the source and destination data-planes concurrently to avoid packet-loss. However, they ignore the correctness issues that could occur due to running multiple clones simultaneously [27]. Other recent work [46] presents orchestration algorithms for virtual network migration, but not specific migration mechanisms; LIME could adopt these orchestration algorithms.

6.2 Consistency of Network State

Consistent updates to a network policy: Recent work on "consistent updates" [51] shows how to change the rules in SDN switches while ensuring that a packet in flight experiences a single network policy. Informally, a consistent update focuses on the "network", whereas LIME focuses on "applications" running on end hosts and controllers. LIME must satisfy *application-specific* dependencies between *multiple* packets (possibly from different flows) while managing multiple, dynamically-changing copies of the same switch. Our theoretical model relates to the model in the consistent-updates paper, but with major extensions to capture the *observations* that end-host and controller applications can make, rather than simply the *trace properties* a single packet experiences. The consistent-updates work cannot solve the switch-migration problem because the consistency property is different, the underlying model does not include hosts and controllers, and the technique does not handle replicated state. The two works prove different network properties, using similar proof techniques.

More recently, several papers propose ways to update a network policy without causing transient congestion [26, 38, 45]. This line of work is orthogonal to LIME, since these solutions do not provide a framework for migrating virtual networks (*e.g.,* to jointly migrate network and VM state). However, their algorithms could help generate an efficient *plan* for migrating switches or the steps for migrating the tunneled traffic. We plan to explore this direction as part of our future work.

7. Conclusions

Live VM migration is a staple in the management of data centers and enterprises. When migrating VMs, their underlying network should migrate, too. LIME supports transparent migration of virtual machines and switches. Our efficient *cloning* technique minimizes performance disruptions by allowing multiple physical switches to act as a single virtual switch during the migration, while forwarding and measuring traffic correctly, even across updates to the rules. Using our formal model, we prove that *clone* works correctly for any controller and end-host applications, under any (unknown) dependencies between different packets. Experiments with our prototype also demonstrate that *clone* is efficient, enabling ensemble migration to be a general tool for network management.

Acknowledgments

We would like to thank Brighten Godfrey, Kyle Jao, Dushyant Arora, Diego Perez-Botero, the anonymous reviewers, and our shepherd, James Mickens, for their insightful feedback and their help with the experiments. We gratefully acknowledge the support from DARPA through grant MRC-007692-001, ONR grant N00014-12-1-0757, and NSF grants TC-1111520 and CNS-1320389.

References

[1] cbench OpenFlow controller benchmark. See http://www.openflow.org/wk/index.php/Oflops.

[2] Emulab - Network Emulation Testbed. http://www.emulab.net/.

[3] FutureGrid. https://portal.futuregrid.org/.

[4] Kernel Based Virtual Machine (KVM). http://www.linux-kvm.org/.

[5] Ocean cluster for experimental architectures in networks (ocean). http://ocean.cs.illinois.edu/.

[6] OpenFlow. http://www.openflow.org.

[7] Open vswitch. openvswitch.org/.

[8] NTT, in collaboration with Nicira Networks, succeeds in remote datacenter live migration, August 2011. http://www.ntt.co.jp/news2011/1108e/110802a.html.

[9] Microsoft Research Faculty Summit Keynote: L. Lamport, 2014. URL https://www.youtube.com/watch?v=n4gOZrUwWmc.

[10] G. S. Akula and A. Potluri. Heuristics for migration with consolidation of ensembles of virtual machines. In *COMSNETS*, 2014.

[11] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu. VMFlock: Virtual machine co-migration for the cloud. In *High Performance Distributed Computing*, 2011.

[12] A. Al-Shabibi, M. D. Leenheer, M. Gerola, A. Koshibe, E. Salvadori, G. Parulkar, and B. Snow. OpenVirteX: Make Your Virtual SDNs Programmable. In *HotSDN*, 2014.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, C. Faster, and D. Maltz. DCTCP: Efficient packet transport for the commoditized data center. In *SIGCOMM*, 2010.

[14] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM*, 2011.

[15] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

[16] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[17] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schioberg. Live wide-area migration of virtual machines including local persistent state. In *Virtual Execution Environments*, pages 169–179, 2007.

[18] N. F. Butt, M. Chowdhury, and R. Boutaba. *Topology-awareness and Reoptimization Mechanism for Virtual Network Embedding*. Springer, 2010.

[19] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *PRESTO*, 2010.

[20] Cisco and VMWare. Virtual machine mobility with VMware VMotion and Cisco data center interconnect technologies, 2009.

[21] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[22] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[23] U. Deshpande, X. Wang, and K. Gopalan. Live gang migration of virtual machines. In *HDPC*, 2011.

[24] floodlight. Floodlight OpenFlow Controller. http://floodlight.openflowhub.org/.

[25] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.

[26] S. Ghorbani and M. Caesar. Walk the line: Consistent network updates with bandwidth guarantees. In *HotSDN*, 2012.

[27] S. Ghorbani and B. Godfrey. Towards Correct Network Virtualization. In *HotSDN*, 2014.

[28] S. Ghorbani, M. Overholt, and M. Caesar. Virtual Data Centers. Technical report, CS UIUC, 2013. www.cs.illinois.edu/~ghorban2/papers/vdc.

[29] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker. Transparent, Live Migration of a Software-Defined Network. Technical report, CS UIUC, 2013. www.cs.illinois.edu/~ghorban2/papers/lime.

[30] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, 2011.

[31] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[32] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM Computer Communication Review*, volume 38, 2008.

[33] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[34] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. CoNEXT, 2010.

[35] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.

[36] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE), RFC 1701, 1994.

[37] F. Hao, T. Lakshman, S. Mukherjee, and H. Song. Enhancing dynamic cloud-based services using network virtualization. In *ACM Workshop on Virtualized infrastructure systems and architectures*, 2009.

[38] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[39] X. Jiang and D. Xu. Violin: Virtual internetworking on overlay infrastructure. In *Parallel and Distributed Processing and Applications*, 2005.

[40] A. Kangarlou, P. Eugster, and D. Xu. VNsnap: Taking snapshots of virtual networked infrastructures in the cloud. In *IEEE Transactions on Services Computing (TSC)*, October 2011.

[41] E. Keller, J. Rexford, and J. van der Merwe. Seamless BGP Migration with Router Grafting. In *NSDI*, 2010.

[42] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford. Live migration of an entire network (and its hosts). In *HotNets*, 2012.

[43] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[44] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[45] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. In *SIGCOMM*, 2013.

[46] S. Lo, M. Ammar, and E. Zegura. Design and analysis of schedules for virtual network migration. In *IFIP Networking*, 2013.

[47] D. M. F. Mattos and O. C. M. B. Duarte. XenFlow: Seamless Migration Primitive and Quality of Service for Virtual Networks. Technical report, COPPE/UFRJ, 2014. `http://www.gta.ufrj.br/ftp/gta/TechReports/MaDu14.pdf`.

[48] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computer Communications Review*, 38(2), 2008.

[49] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Springer, 1980.

[50] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *NSDI*, 2013.

[51] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[52] VMWare. VMware vNetwork Distributed Switch: Migration and Configuration. `http://www.vmware.com/files/pdf/vsphere-vnetwork-ds-migration-configuration-wp.pdf`.

[53] VMWare. vsphere. `http://www.vmware.com/products/datacenter-virtualization/vsphere/vmotion.html`.

[54] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: Live router migration as a network-management primitive. In *SIGCOMM*, 2008.

[55] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-blanket: Virtualize once, run everywhere. In *EuroSys*, 2012.

[56] D. Williams, H. Jamjoom, Z. Jiang, and H. Weatherspoon. VirtualWires for Live Migrating Virtual Networks across Clouds. Technical Report RC25378, IBM, 2013.

[57] T. Wood and J. van der Merwe. CloudNet: A platform for optimized WAN migration of virtual machines. In *International Conference on Virtual Execution Environments*, 2011.

[58] K. Ye, X. Jiang, R. Ma, and F. Yan. VC-Migration: Live migration of virtual clusters in the cloud. In *Grid Computing*, 2012.

[59] Y. Zhu and M. H. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *INFOCOM*, 2006.