Towards Correct Network Virtualization

Soudeh Ghorbani and Brighten Godfrey
University of Illinois at Urbana-Champaign
{ghorban2, pbg}@illinois.edu

ABSTRACT

In SDN, the underlying infrastructure is usually abstracted for applications that can treat the network as a logical or virtual entity. Commonly, the "mappings" between virtual abstractions and their actual physical implementations are not one-to-one, e.g., a single "big switch" abstract object might be implemented using a distributed set of physical devices. A key question is, what abstractions could be mapped to multiple physical elements while faithfully preserving their native semantics? E.g., can an application developer always expect her abstract "big switch" to act exactly as a physical big switch, despite being implemented using multiple physical switches in reality?

We show that the answer to that question is "no" for existing virtual-to-physical mapping techniques: behavior can differ between the virtual "big switch" and the physical network, providing incorrect application-level behavior. We also show that that those incorrect behaviors occur despite the fact that the most pervasive correctness invariants, such as per-packet consistency, are preserved throughout. These examples demonstrate that for practical notions of correctness, new systems and a new analytical framework are needed. We take the first steps by defining *end-to-end correctness*, a correctness condition that focuses on applications only, and outline a research vision to obtain virtualization systems with correct virtual to physical mappings.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*

Keywords

Network Virtualization, One Big Switch, Correctness

1. INTRODUCTION

Virtualization refers to the act of decoupling the logical service from its physical realization [5] with some mapping between them. Accordingly, full network virtualization solutions strive not only to multiplex multiple virtual networks on a single physical network (hereafter called *many-to-one mapping*), but also to use multiple physical networking elements to implement a single virtual network (hereafter called *one-to-many mapping*) [5, 17, 19]. Many-to-one mapping is mainly a means to share resources [21], whereas one-to-many mapping (i.e., implementing a single abstraction using a distributed set of physical networking elements) provides basic support for function mobility [8,23], enables a "scale-out" approach to network design in which additional physical networking elements scale-out a single logical abstraction [5], provides high-availability [5], and overcomes lack of capacity of physical elements (e.g., when the capacity of one switch is insufficient for a full implementation of the logical abstraction). For example, each tenant in a public data center might be presented one logical "big switch" abstraction that may in reality span multiple physical hardware or software switches. We present more use-cases for one-to-many virtualization in Section 2.

The immediate question is, **could the one-to-many mapping lead to incorrect behavior in the network?** Could an application that a developer writes on top of its logical "one big switch" presented to her by the virtualization solutions perform unexpectedly due to the one-to-many mapping of the intended abstract functionalities to the physical switches? In Section 3, we provide **several examples of the incorrect behaviors caused by existing realizations of one-to-many mapping**: a NAT that erroneously drops packets, and a firewall that erroneously blocks hosts. We also note that those erroneous behaviors happen while the network is still meeting the most commonly-used correctness and consistency requirements defined by previous work, such as "per packet consistency" or "per flow consistency" [20] the whole time. Interestingly, **deploying the techniques that are specifically designed to preserve those consistency requirements could occasionally break the otherwise correct behavior** (Section 5.1).

In light of those observations, we argue that for practical notions of correctness, new systems and a new analytical framework are needed. While focusing on *trace properties* of a packet (or flow), *i.e.*, the invariants related to the hop-by-hop journey of a single packet (or flow) in the network [11,12,18,20], is invaluable in preventing certain classes of incorrect network behaviors such as loops, black-holes, or congestion, it is far from sufficient for today's networks. Thus, there is a need for defining a new notion of correctness that takes into account the sequence of observations made by the

HotSDN'14, August 22, 2014, Chicago, IL, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620754.

controller and end-hosts, with potential interdependencies. Towards this goal, we present the initial sketch of a probabilistic model of SDN that also considers the partial ordering between events and define correctness of virtualization using that model (Section 5.2). In tune with what applications expect from best-effort networks with occasional packet re-ordering, this model is general enough to be permissive of occasional packet drops, and allows some packet re-ordering.

We elaborate on the root-causes of wrong behaviors that could occur under the one-to-many mapping (initially, qualitatively), in Section 4: by borrowing from the seminal work on ordering of events in distributed systems by Lamport [16], we elaborate on the distinction between *causality* and ordering of events in SDN, and show that while *causality* of sending and receiving packets is preserved under the one-to-many mapping (i.e., a packet will be causally generated by another packet in the physical network if and only if it is supposed to be causally generated by that same packet in the logical abstract network), the *ordering* of packets is not always preserved under the one-to-many mapping. In Section 5.3, we show that under the existing realizations of one-to-many mapping, abstractions that are *Markov* or *memoryless* (i.e., their actions do not depend on the history of previous matching packets or previous actions) could be correctly mapped, and that they are the only class of abstractions that could be correctly mapped under the existing schemes.

By demonstrating the unexpected behaviors that could result from a common virtualization technique (one-to-many mapping), and showing the need for a different notion of consistency and correctness, this work tries to open up an important line of research for (a) defining alternative correctness notions and (b) building provably-correct virtualization systems to realize the one-big switch idea.

2. BACKGROUND AND MOTIVATION

One-to-many virtualization implements a single virtual abstraction using multiple physical elements. Use-cases for one-to-many virtualization include building distributed virtual switches: Even though host-hypervisors provide virtual switches to control the network at one end-host, the dynamic nature of virtual environments such as VM migration, spinning up new VMs on other end-hosts that should be connected to the same virtual network used by VMs on different hosts, etc. requires these abstract virtual switches to be implemented using a distributed set of physical switches [5]. In this case, one logical switch is in reality implemented with a distributed set of virtual switches. As another example, Andromeda (Google's virtualized SDN) [3], integrates software network function virtualization (NFV), such as virtual firewall, rate limiting, etc. into the data-path, and deploys replication (one functionality mapped to multiple physical elements) in the data-plane as a technique to meet its performance and scalability needs. The physical replicas, in this case, collectively represent a logical network. Rule or flow table entries caching schemes, where topologically-mapped "rule caches" at multiple locations in the network concurrently handle traffic for enhancing performance [22], are another instance of implementing logical abstractions via a distributed set of physical networking elements. In this section, we discuss the common logical or virtual abstractions provided by virtualization solutions before discussing how some existing systems *map* those virtual abstractions to physical forwarding elements.

2.1 Choosing the Right Logical Abstractions

The first design challenge of network virtualization is the choice of the right abstractions. While there exist some network virtualization solutions that (partially) virtualize the network at the lower levels, e.g., with tunnels and tags, and some solutions that try to virtualize the network at higher-level interfaces [4, 10, 19], it is generally argued that a useful layer at which to virtualize is the **full forwarding plane** [5], and that existing SDN forwarding plane abstractions such as **flow table entries** are the right logical abstractions to present when virtualizing the network [2, 15]. In this work, we focus on the solutions that provide forwarding abstractions similar to flow table entries.

Upon receiving a packet, a switch executes a set or an ordered list of **actions** of the highest priority flow table entry that matches the packet. These actions could result in changes in the packet, dropping it, or forwarding it. The actions are determined by the header of the packet and its ingress port, as well as the *forwarding state* of the flow table entry: different components of the flow-table entry that affect the forwarding decision such as match fields, timers, and meter tables. From the programmability perspective, the forwarding state could be categorized based on the entities that compute and modify it:

- **Forwarding state computed and modified solely by controllers.** Examples include layer 1 (switch port) to layer 4 **match fields** of a flow-table entry.
- **Forwarding state computed or modified by switches** locally and without consulting the controllers. Examples include **random forwarding**, e.g., for supporting ECMP, and **hard timeouts**.
- **Forwarding state computed or modified by data plane packets** (as opposed to control and management packets) locally and without consulting the controllers. For example, **soft timeouts** measure idle time and are updated by matched packets, thus affecting when a flow table entry expires. Packet **counters** that are updated when packets are matched and could determine the forwarding decision in round-robin forwarding [1], and **flow meters** used for making the forwarding decision based on the rate of the flow matching the flow table entry [1] are other examples.

As we will show, the above categorization of forwarding state, based on entities in charge of "programing" the forwarding state, plays a key role in determining whether or not the flow-table entry could be correctly virtualized under the existing mapping techniques.

2.2 One-to-Many Mapping

Starting with the simplest case, in this work, we focus on mapping a logical flow table entry abstraction to a distributed set of physical flow table entries where each physical flow table entry is individually capable of fully implementing the logical flow table entry. In the *one-to-many mapping* technique, before installing a flow table entry on multiple physical flow tables, the hypervisor typically performs some modifications to the actions and forwarding state of any logical flow table entry. Specifically, the actions and forwarding state computed by the controller and switches are usually subject to (a) **Topological modifications:** In the "match fields" and "actions", a logical flow table entry could contain a set of logical ports that need to be mapped to physical ports to account for the possible distinctions between the physical and logical topologies [5,8]. Those logical ports are translated to corresponding physical ports before the logical flow table entry is installed on physical flow tables. (b) **Address translation:** Ideally, each virtual network should operate in a virtual address space that is decoupled from the physical network address space and the virtual address spaces of other virtual networks [15]. In this scenario, the network hypervisors are in charge of address space translation. (c) **Rule-cloning:** One wildcard flow table entry is sometimes "cloned" to multiple entries in which all of the wildcarded fields are replaced by exact-match values and all other aspects of the original entry are inherited [6,15]. This mechanism is used in software switches for caching the wildcard entries residing in the userspace as exact-match entries into a flow table in the kernel to enhance performance, *e.g.*, in Nicira's NVP [15].

Hence, while the one-to-many mapping makes some modifications in the actions and forwarding state computed and modified by the controller and switches, other components of the logical and physical flow table entries, *i.e.*, those modified or computed by the packets, can remain identical. We will see in examples in the next section that this is the root cause of one subclass of incorrect behaviors.

Trivially, each single physical flow table entry is a faithful implementation of the logical abstraction in isolation. I.e., if the logical single flow table entry was in fact implemented via a single element only, that single flow table entry would act indistinguishably from the logical abstraction. In particular, if sending one packet by one application that matches the logical entry results in receiving a set of (potentially modified) packets by a set of applications (on end-hosts or the controller), then sending the same packet via any mapped physical flow table entry leads to receipts of the same set of packets by the same set of applications. Similarly, if all the mapped physical flow table entries could be serialized or consistently synchronized, their collective behavior would still be indistinguishable from the logical abstraction. In practice, however, due to performance and scalability requirements, attempting to serialize and synchronize those physical mapped flow table entries to provide an always-consistent logical view is not feasible [2,22]. Hence, after a logical abstraction is mapped to multiple physical flow table entries under the one-to-many mapping, those entries handle traffic autonomously. The main question of this paper is, what types of abstractions could be correctly mapped to multiple physical flow table entries?

3. ONE-TO-MANY MAPPING: WHAT COULD GO WRONG?

We show a few illustrative examples in which the common technique for realizing a one-to-many mapping can break the semantics of the logical network and lead to incorrect behavior — blacklisting legitimate hosts, or dropping packets that should be delivered[1]. We then discuss the root cause of these incorrect behaviors.

[1]Other examples of incorrect behavior, e.g., overutilizing servers are omitted due to space constraints and can be found in [7].

Example 1: Logical Firewall. Imagine that an enterprise network has a *logical* stateful firewall at the periphery of its network that permits an external server to talk to an internal client if and only if the client has sent a request to the server. This simple policy could be achieved by a stateful-firewall application running on the controller, and two low priority flow table entries on the logical switch: One entry that matches internal client traffic and sequentially performs the following two actions on it: (1) it sends the packet header to the controller to trigger installation of a rule permitting server traffic in the reverse direction to be sent to the client, and (2) it forwards the traffic to the server. And a second flow entry that that matches the server traffic and sends it to the controller to check if it is permissible.

Also, server-to-client and client-to-server "packet-in"s are required to be sent over the same connection between the switch and the controller, *e.g.*, the main TCP connection.[2] When the firewall application receives server traffic, (a) if the server traffic is preceded by the corresponding client's request, the firewall application installs a high priority flow table entry on the switch to forward the server-to-client traffic, (b) if the server traffic is not preceded by the client's request, then the firewall application blacklists the server by installing a high priority flow table entry that drops the packets from the server.

Now, if that logical switch, L, is in reality mapped to more than one physical switch, then the client-to-server traffic could traverse one physical switch, P_1, and the resulting server-to-client traffic traverses a different physical switch P_2. In this case, the response traffic and the server-to-client "packet-in" may reach P_2 and the controller before the control message from P_1 reaches and is processed by the controller. Hence, the controller proceeds to install a rule to block all traffic for that flow—an undesirable outcome and something that would not happen if the single logical switch L were implemented as a single physical switch.

We first noticed this incorrect behavior in an earlier work [8] in which, for efficient network migration, a logical rule needed to be *temporarily* mapped to multiple physical rules: the initial physical source of the rule and its final physical destination. Given the temporary nature of the one-to-many mapping in that setting, this issue was resolved by identifying all "unsafe" rules – those that both forward traffic and send a message to the controller – and temporarily modifying them to instead only send to the controller. This solution inherently enforces a message ordering that respects the dependencies between events and solves this particular issue. If the one-to-many mapping is not temporary, as in implementing a virtual big switch with multiple physical switches, however, such interventions are less satisfactory.

Example 2: Network Address Translation. Consider a Network Address Translation (NAT) application that hides the private network IP address space of an enterprise behind a single public IP address. The policy that is intended to be implemented via the NAT is to allow communication between internal and external hosts when the conversation originates from the masqueraded internal hosts. External hosts that are allowed to communicate with the internal hosts will lose the permission to do so after a pe-

[2]Note that the main control channel in OpenFlow 1.4 provides in-order and reliable delivery, *i.e.*, a client's request is guaranteed to reach the controller before the reply that it triggers.

riod of inactivity, e.g., an active connection will be closed if the external host doesn't send a packet for (say) 60 seconds.

This policy could be implemented via a NAT controller application and a single logical switch at the edge of the enterprise network. The application stores a stateful translation table to map the internal hidden addresses into the single public IP address, and upon receiving a "packet-in" for an outgoing flow, it installs two rules on the switch: (a) a rule to rewrite the port and source address of outgoing packets so they appear to originate from the single public IP address, and (b) a rule for the reverse communication path, to rewrite the port and IP address of responses back to the originating IP addresses with a soft timeout such that the rule is flushed after 60 sec unless new traffic refreshes the local timer of the switch.

Now, if this one single logical NAT is in reality mapped to multiple physical switches, an external host's responses to a request could hit different physical switches. Hence, the time-series of packets hitting each physical switch could be different from the case where all traffic goes through a single switch. E.g., while no two packets of a flow would be spaced by more than 60 *sec* if they hit the same switch, the gap between two consecutive packets hitting one of the many physical switches could be larger than 60*sec*. This might, in turn, trigger the timeouts in some physical switches (which will in turn cause the external host to lose its connection to the internal host); something that will not occur if all traffic goes through a single physical switch.

Despite the differences among the examples above, they share some commonalities:

- Some of the operations in those examples have dependencies on the sequence of packets proceeding them (e.g., timers updated by previous packets).
- Despite the visibly-incorrect behavior, some of the most pervasive notions of correctness in networking (such as loop-freedom, absence of blackholes, per-packet consistency, etc. [11–14, 18, 20]) are met throughout.
- There are alternative ways of implementing the exact same policy (sometimes with subtle differences from those implementations described above) for which the one-to-many mapping would not lead to incorrect behaviors.

We discuss these issues in Sections 4 and 5.

4. ORDERING OF SENDING AND RECEIVING PACKETS IN SDNS

In the previous section, we showed some examples of incorrect behaviors resulting from the one-to-many mapping. In those examples, under the one-to-many mapping, the controller or end-host applications could observe orderings between events that are different from the orderings that they expect to observe in the logical network. In the firewall example, for instance, with the logical view of a single switch, the controller expects to receive a request before it receives the corresponding reply. These examples demonstrate that even in the typical current networks that are not assumed to provide in-order delivery, some orderings are assumed to be always preserved between certain events and *observable* to applications. Application logic, consequently, could depend on those orderings (e.g., allowing the communication if the reply is received after the request, but disallowing it otherwise). We borrow some definitions and insight from the seminal work of Lamport on ordering of the events in a message passing distributed system [16] before examining the factors that determine the orderings in SDN and the reasons that they could break under the one-to-many mapping.

***Happened before* relationship.** In the absence of synchronized physical clocks, what orderings between send and receive events are observable by processes in a distributed message passing system that could re-order the messages? In search for an answer to this question, the pioneering work of Lamport defines the *happened before* relationship ($\rightarrow$) on the set of events in a message passing system to be the smallest relation satisfying the following three conditions [16]: (a) If a and b are events in the same process, and a comes before b, then a *happens before* b, i.e., $a \rightarrow b$. (b) If a is the sending of a located packet by one process and b is the receipt a of the located packet by another process, then a *happens before* b. (c) If a *happens before* b, and b *happens before* c, then a *happens before* c. The $happened\ before$ is a partial ordering, i.e., not every pair of events can be related by it.

Happened before* is not equivalent to *causality. Despite the fact that an alternative way of viewing the definition of *happened before* is to say that $a \rightarrow b$ means that it is **possible** for event a to causally affect event b [16], it should be noted that it is not required for a to causally affect b. While it is true that $a \rightarrow b$ if an event b is causally generated by event a, the reverse is not always true. I.e., $a \rightarrow b$ might be an observable ordering between a and b even without any causal relationship between them. In the correct realization of the firewall example, for instance, the event of receiving the reply at the controller (e_2) is not causally generated by the event of receiving the request at the controller (e_1). Yet, $e_1 \rightarrow e_2$ is required for the correct operation of the firewall application. In SDN, the ordering of two sending and/or receiving events might be known (e.g., $a \rightarrow b$) **without** any causality relationship between them (e.g., neither a causes b, nor b causes a). E.g., in a network that reactively installs flow rules with timeout, the controller could receive a second *packet-in* from a flow after the first one because the flow entry matching the flow has been removed after a certain period of time and not because the second packet-in was caused by the first one. Hence, the **local state** of the switch, which is affected by the sequence of the previous packets that matched the flow table entry, could affect the ordering of events. In addition to causality and local state, existence of channels that provide in-order message delivery, like the control channel in OpenFlow 1.4, could determine the ordering of events. The firewall example shows one instance of the influence of the ordered channel on the ordering of events.

To summarize, event dependencies are key to correctness. In Section 3, we provided several examples of application logic that depends on the orderings of events, *i.e.*, an event depending on some events that *happened before* it, and demonstrated how the one-to-many mapping could violate those orderings, and consequently the correctness of applications. In Section 5, we sketch a new analytical system and definition of correctness to argue that the logical abstractions that have no dependency on the events that happen before them, and only that class of abstractions, could be correctly implemented under the one-to-many mapping techniques.

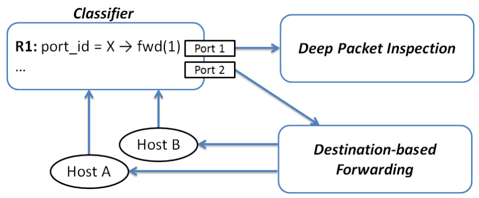

Figure 1: *"Consistent updates" could lead to unexpected application behavior.*

5. MODELING SDN AND DEFINING CORRECTNESS

The first challenge in providing correct mappings is finding a practical notion of correctness. In this section, we first show the most pervasive notions of correctness cannot capture the incorrect behavior demonstrated in §3. We further elaborate on the insufficiency of existing notions by an illustrative example (without the one-to-many mapping) in which deploying the techniques used for meeting per-packet consistency *breaks* the otherwise correct behavior.

5.1 Per-packet Consistency Is Not Enough

Existing notions of network correctness mostly focus on the journey of a single packet or flow. Some notions involve specific policies like absence of loops and black holes [11,18], which often arise due to network dynamics. A more inclusive notion of correctness is *per-packet (or per-flow) consistency*, which guarantees that every packet (or flow) traversing the network is processed by exactly one global network configuration and never by a mix of two (or more) configurations [20].

While this requirement prevents some anomalies such as loops and black-holes, it falls short of detecting incorrect behaviors from end-points' perspective, because it cannot capture violations of the dependencies between different packets (or flows), such as ordering between them. In the firewall example, for instance, the policy generated by the firewall depends not only on the packets that the controller application receives, but also on their orderings, e.g., it permits the communication if the internal host's request is received before the reply from the external host and will otherwise block the communication. In these examples, each packet and flow is never handled by a mixture of policies. Hence, per-packet (or flow) consistency is preserved, despite the visible incorrect behavior.

To better illustrate that current consistency definitions are insufficient for some practical uses, we give a simple example (with no one-to-many mapping) showing that the mechanism usually deployed for preserving per-packet consistency, i.e., tagging packets and rules using the policy ID and including the tags in the lookup operations [11, 20], leads to incorrect behavior. Assume that a data center tenant intends to classify her traffic based on the applications that generated it (e.g., using the port IDs of applications), and then process traffic from different classes differently. A simple way to implement this policy is to deploy different logical units for processing traffic from each class of applications, and to have another logical unit as a "classifier" which is responsible for forwarding each packet to the unit associated with the application that generated the packet. In Figure 1, for example, the tenant sends the packets generated by a certain application x, with $port\_id = X$, to a deep packet inspection box (DPI) which inspects packets and drops them if they are being sent from hosts A and B. Let's call this policy $P1$. Now, assume that the tenant decides to switch from policy $P1$ to policy $P2$ that does simple forwarding for traffic from application x. The only affected module by this policy update is the "classifier" that should simply change its R1 rule to forward to port 2 instead of port 1: $(port\_id = X) \rightarrow fwd(2)$. At any point during the update process, if A (or B) receives an application-x packet from B (or A), it can logically infer that the policy update has been done and could reply by sending application x packets.

However, if the network uses the "consistent updates" mechanism for updating the policies, the update mentioned above will not longer be an atomic single step update, because the flows using rule $R1$ on the classifier need to be updated one by one. Without loss of generality, assume that the flow from A to B is updated, but other flows (including the one from B to A) are not still updated. In this case, the classifier will have 2 rules corresponding to $R1$ on the classifier in Figure 1 (not shown): an old rule to match traffic using old tags (old policy traffic) and the new rule matching traffic with new tags (new policy traffic). Now, host A sends application-x traffic with the new tag, which will be forwarded to B (new policy). B receives the packet, and since it sees an application-x packet from A, it concludes that x is no longer blocked and replies back using application x. Its reply to A, however, will be delivered to DPI and not A (since it has the old tag), something that would not happen if "consistent updates" was not being used for updating the policy. The underlying problem in this case is that "consistent updates" breaks the atomicity that would otherwise exist.

5.2 End-to-end Correctness

The previous examples demonstrate that rather than focusing solely on the trace properties of a single packet or flow, it is essential to for a correctness model to capture the observations that end-host and controller applications can make. Towards this goal, we sketch a probabilistic model on a partially ordered set of events that the controller and end-hosts could observe. This new model could be permissive of occasional packet-loss (as *best-effort* networks are), of random forwarding (e.g., to permit ECMP or other random forwarding capabilities that are invaluable for traffic engineering [9]), and of re-ordering of some packets. A one-to-many mapping of a logical flow table entry L to a set of physical flow table entries P is said to be an *end-to-end correct* mapping, *iff* for any partially ordered (defined by the *happened before* relation) set of events, E, probability of observing E, by any arbitrary application, while having L in the network is *similar* to the probability of observing E while having P, i.e., $Pr_L[E] \approx Pr_P[E]$. For the purposes of correctness, *similarity* should distinguish between events that happen sometimes, always, and never; but, probabilities 0.2 and 0.3, for example, can be considered similar in order to allow for differences in packet loss or timing that do not affect correctness.

5.3 Memoryless Packet-handling Operations

In Section 4, we informally showed that the one-to-many mapping could change the observable ordering of events: if packet processing depends on the history of previous packets (e.g., forwarding decision for a packet depends on other packets that have changed the counter or the timer) or if packet processing depends on prior actions (e.g., packet is forwarded to the host only after it is sent to the controller), then the logical and physical networks could behave differently. This happens because each mapped physical flow table entry that handles the traffic could have a different local history which is different from the global history of the logical flow table entry. Intuitively, this demonstrates the significance of the memoryless property of packet-handling (i.e., each *action* not depending on the history of previous received packets or actions) for the correctness of the one-to-many mapping. We introduce the key lemmas and theorems related to the memoryless property below. Formal definitions and a sketch of the proofs are omitted due to space constraints and can be found in [7].

Lemma A: No dependency between "send actions" of a flow table entry is observable in the absence of ordered channels.

Lemma B: Ordering of two send actions of a flow table entry is observable if and only if the first send event is sending a packet over an ordered channel.

Theorem: A Markov (memoryless) property of send events of a flow table entry is necessary and sufficient for end-to-end correctness of the one-to-many mapping of that entry.

6. DISCUSSION AND CONCLUSION

Our results open up an important research direction: How do we automatically enable one-to-many virtualization while guaranteeing correctness? Our preliminary results in this paper provide a starting point, showing that a certain class of (Markov) behaviors can be mapped easily. However, many useful applications lie outside that space. Research directions include: (a) developing more advanced mapping techniques, e.g., techniques that detect the flow table entries that could trigger incorrect behavior and find alternative correct implementation for them, or (b) restricting the API of SDN to provide safe-to-map abstractions only (i.e., abstractions that possess a Markov property). For example, flow table entries with coarse-grained soft-timeouts (that are non-Markov) could be mimicked by using "permanent" Markov flow table entries and periodically polling the traffic counters to see if any traffic has matched the entries since the previous poll. These solutions, however, could be heavyweight, decrease efficiency, add to the delay, and increase the control overhead (e.g., to involve the controller in some traffic handling tasks that could be purely done by the forwarding plane). While there is purportedly a trade-off between performance and correctness for achieving correct mapping, it is not obvious if that trade-off is fundamental. We believe that searching for simple, general solutions with carefully measured trade-offs warrants further and deeper investigations.

Acknowledgment: We would like to thank Jennifer Rexford, Cole Schlesinger, and David Walker for their insightful comments and discussions. We gratefully acknowledge the support from NSA through Science of Security grant 2014-03124.

7. REFERENCES

[1] OpenFlow Switch Specification, version 1.4.0. Technical report, Open Networking Foundation, 2013.

[2] ONS 2014 Keynote: A. Greenberg, Microsoft Azure, 2014.

[3] ONS 2014 Keynote: A. Vahdat, Google, 2014.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking Enterprise Network Control. *CCR*, 2009.

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

[6] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. SIGCOMM, 2011.

[7] S. Ghorbani and B. Godfrey. Towards Correct Network Virtualization. Technical report, CS UIUC, 2014. `http://goo.gl/ks9Fp9`.

[8] 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. `http://goo.gl/yzVgGN`.

[9] 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.

[10] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *CCS*, 2000.

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

[12] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *HotSDN*, 2013.

[13] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking For Networks. In *NSDI*, 2012.

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

[15] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[17] M Ciosi et al. Network functions virtualization. Technical report, ETSI, 2013. `http://goo.gl/Q84Bxi`.

[18] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software-Defined Networks. In *HotNets*, 2013.

[19] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software defined networks. In *NSDI*, 2013.

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

[21] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *OSDI*, 2010.

[22] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *SIGCOMM*, 2011.

[23] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *CCR*, 2008.