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?

In this paper, we show a few examples of unexpected behaviors (as observed by end-hosts and control applications) under existing mapping techniques. We also show that those incorrect behaviors occur despite the fact that the most pervasive and commonly-used 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.

We also demonstrate that, for any logical forwarding element, a Markov property of packet-handling actions of that element (i.e., having no dependence on the history of previous packets matching that element) is the necessary and sufficient condition for correctly distributing it under existing schemes.

Categories and Subject Descriptors
C.2.3 [Computer Systems Organization]: Network Operations—Network management

Keywords
Software Defined Networking, Virtualization, One big switch, many-to-many mapping, Consistency, Correctness, Markov property, Memoryless

1. INTRODUCTION

Virtualization refers to the act of decoupling the logical service from its physical realization [7] with some mapping between them. Accordingly, full network virtualization solutions strive not only to multiplex multiple virtual networks on a single physical network (many-to-one mapping of virtual to physical networks), but also to use multiple physical networking elements to implement a single virtual network (hereafter called one-to-many, for brevity) [7, 17, 19]. Many-to-one mapping of multiple logical abstractions to a single physical networking element 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), is done for providing basic support for function mobility [9, 23], for enabling a “scale-out” approach to network design, in which additional physical networking elements scale-out a single logical abstraction [7], for providing high-availability [7], and for overcoming 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 datacenter might be presented one logical “big switch” abstraction that may in reality span multiple physical hardware or software switches. 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 [7]. 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) [4], integrates software network functions 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.

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? While many-to-one mapping techniques have been the subject of extensive investigation to guarantee correctness (i.e., provide isolation), little is known about the fidelity of physical implementation of one-to-many mapping techniques (e.g., one big switch and its policies) among a distributed set of switches (one-to-many). In Section 2.3, we provide several examples of the incorrect behaviors caused by existing realizations of one-to-many mapping: a NAT that erroneously drops packets, or 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 while focusing on the hop-by-hop journey of a single packet (or flow) in the network [12, 13, 18, 20] is invaluable in preventing certain classes of incorrect network behaviors such as loops or black-holes, it is far from sufficient for today’s networks. Thus, there is a need for defining a new notion of correctness that takes the observations made by the controller and end-hosts into account. Towards this goal, we present the initial sketch of a probabilistic model of SDN that also considers the partial ordering between events and define the correctness using that model (Section 5.2). In tune with what applications expect from best-effort networks with occasional packet re-ordering, this model is 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 some definitions and results 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 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 interesting 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

In this section, we discuss the logical or virtual abstractions provided by virtualization solutions before discussing how those abstractions are mapped to physical forwarding elements.

2.1 Logical Abstractions

The first design challenge of network virtualization is the choice of the right abstractions. While there exists 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 network at higher-level interfaces [6, 11, 19], it is generally argued that the correct layer at which to virtualize is the full forwarding plane [7], and that existing SDN forwarding plane abstractions such as OpenFlow flow table entries are the right logical abstractions to present when virtualizing the network [3]. In this work, we focus on the solutions that provide forwarding abstractions similar to flow table entries. In this case, mapping a single logical abstraction to multiple physical forwarding elements essentially reduces to copying that abstraction (after topological modifications and adding tags for isolating virtual networks) to multiple physical flow tables. We leave the investigations and verification of higher level abstractions that are not directly and entirely supported by the underlying physical network for future work.

2.2 Flow-table Entry Abstractions
A flow table entry is generally composed of the following components [2]:

- **Match fields** to match against packets. These include packet headers, as well as the ingress ports and egress ports.
- **Priority** to determine the matching precedence of the flow entries.
- **Local state** that depends on the local state of the flow entry on the switch. This could contain: (a) **counters** that are updated when packets are matched, (b) **timeouts** that show the maximum amount of time (hard timeout) or idle time (soft timeout) before flow is expired by the switch. (c) **rate** that is updated when packets are matched. Rates are used to implement various simple QoS operations, such as rate-limiting.
- A set or an ordered list of **actions** that are executed when a packet matches the match field, within the confines of local state and given that the flow entry has the highest priority among all existing and matching entries. These actions could result in changes in the packet, dropping it, or forwarding it.

### 2.3 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. The one-to-many mapping technique makes two categories of modifications to any logical flow table entry before installing it on multiple physical flow tables: (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 [7,9]. Those logical ports are translated to corresponding physical ports before the logical flow table entry is installed on physical flow tables. (b) **Tagging as a means of isolation**: Multiple logical flow table entries of different logical networks may be multiplexed over the same physical switch. To distinguish between them and provide isolation, a distinct tag (for each logical flow table entry) is usually added to the mapped physical flow entries [7,9]. The lookup operation for each packet would entail checking these tags, in addition to the match fields and local state.

Hence, while the one-to-many mapping makes some modifications in the “header fields” and “actions” of

---

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

In this section, we show a few illustrating examples in which the one-to-many mapping could break the semantics of the logical network and lead to incorrect behaviors before discussing the root-cause of those incorrect behaviors. In the first example, we check a stateful firewall application that, under the one-to-many mapping, blacklists the legitimate hosts. In the second example, we see how a NAT application drops the packets that are supposed to be delivered. Last example shows a load balancer that overloads some servers while leaving the rest of the servers underutilized.

**Example 1: Stateful 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, a low priority flow table entry on the logical switch that matches 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 another low priority flow table entry on the logical switch that
matches the server traffic and sends it to the controller. 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, \textit{L}, is in reality mapped to more than one physical switches, then the client-to-server traffic could traverse one physical switch, \textit{P}_1, and the resulting server-to-client traffic traverses a different physical switch \textit{P}_2. In this case, the response traffic may reach \textit{P}_2 before the control message from \textit{P}_1 reaches the controller; and the server-to-client “packet-in” event that \textit{P}_2 sends to the controller could reach the controller before the client-to-server “packet-in” event from \textit{P}_1. 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 \textit{L} was in reality mapped to one physical switch only.

We first noticed this incorrect behavior in an earlier work in which a logical rule need to be **temporarily** mapped to multiple physical rules for efficient migration [9]: 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 temporarily modifying all rules that both forward traffic and send a message to the controller 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**

As the second example, consider a Network Address Translation (NAT) application that hides an 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 period of inactivity, e.g., an active connection will be closed if the external host doesn’t send packet for a period of time, e.g., for about 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 tables 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 readdresses the outgoing packets on exit so they appear to originate from the single public IP address, and (b) a rule for the the reverse communications path, to readdress responses back to the originating IP addresses with a soft-timeout (e.g., 60 sec) such that the rule is flushed after a certain period unless new traffic refreshes the local timer of the switch.

Now, if this one single logical NAT is in reality mapped to more than one 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 60sec. 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.

**Example 3: Server Load Balancer**

Many large scale web applications (web search, social network content composition, and advertisement selection) have multi-layer partition/aggregate pattern workflow where requests from higher layers of the application are broken into pieces and farmed out to workers in lower layers [5]. E.g., High Level Aggregators (HLAs) could execute queries and contact some Mid-Level Aggregators (MLAs) to send them the “subqueries”. An MLA that receives a subquery will, in turn, distribute it to some hosts, known as workers. The MLA will then collect the partial results from the workers and the responses of these workers are aggregated to produce a result. For interactive, soft-real-time applications that use partition/aggregate pattern, latency is the key metric [5]. Therefore, worker nodes are assigned tight deadlines, and when a node misses its deadline, the computation continues without that response, lowering the quality of the result [5]. Consequently, it is important to distribute the load among the workers evenly and avoid overloading any worker.

Assume for simplicity that workers and subqueries are homogeneous and all MLAs have access to the same set of workers. In this scenario, for evenly distributing the requests from the MLAs between the workers, the network provider deploys a logical load balancer between the MLAs and the workers which sends each request from a MLA to a worker in a round robin fashion.

The load balancer could be implemented with a flow
table entry with round-robin forwarding (e.g., the first subquery to port \( p_1 \), the second to port \( p_2 \), ...) \(^3\). While this simple forwarding scheme provides near optimal load balancing if it is actually implemented with one single switch between the MLAs and the workers [8], it could severely overload a subset of workers and leave the rest of them underutilized if the logical load balancer is mapped to multiple physical switches: each MLA could be connected to a physical switch, e.g., an OVS [1], and the load balancer is implemented in a distributed fashion, e.g., by using the one-to-many mapping to map the logical flow table entry of the logical load balancer to each of those OVSes. Now, each time a HLA receives a query, it sends some subqueries to a few MLAs. Those MLAs receive the subqueries at roughly the same time and send them to the same set of workers (because the counters used for implementing the round-robin forwarding would show the same value across the OVSes), overloading those workers and leaving the rest of the workers ideal. Worth noting that the problem above could be largely circumvented by using random forwarding rather than round-robin forwarding.

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

- Some of the operations in those examples have dependencies on the sequence of packets proceeding them (e.g., timers or counters 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, etc. [13–15,18,18,20]) 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 each of these issues in turn.

4. ORDERING OF SENDING AND RECEIVING PACKETS IN SDN

In the previous section, we showed some examples of incorrect behaviors resulted 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 reply triggered by a request only after it has received the request. 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. Applications’ logics, 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 before examining the factors that determine the orderings in SDN and the reasons that they could break under the one-to-many mapping.

4.1 Happened-before Relationship

In absence of synchronized physical clocks and with a communication medium that could re-order the messages, 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 need 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 \( b \). Causality, the relation between an event (the cause) and a second event (the effect), where the second event is understood as a consequence of the first, and happened before relations are not equivalent: 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 firewall example, for instance, the event of receiving the reply by the controller (\( e_2 \)) is not causally generated by the event of receiving the request by the controller (\( e_1 \)). Yet, \( e_1 \rightarrow e_2 \). In SDN, the ordering of two sending and/or receiving events might be known (e.g., event \( a \) might known be known to happen before event \( b \) 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 time-
out, the controller could receive a second packet—in from a flow after the first one (i.e., the first packet-in happened before the second packet-in) 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. In addition to causality and local state, existence of channels that provide in-order message delivery, like the control channel in OpenFlow, could determine the ordering of events. The firewall example shows one instance of the influence of the ordered channel on the ordering of events.

5. MODELING SDN AND DEFINING CORRECTNESS

The first challenge in providing correct one-to-many mappings is the 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 the examples in the previous section. We further elaborate on the insufficiency of existing notions by an illustrating example (without the one-to-many mapping) in which deploying the techniques used for meeting the existing notions breaks the otherwise correct behavior. Motivated by these observations, we then introduce our preliminary model of SDN and a new notion of correctness, end-to-end correctness. Finally, we deploy this model and notion to show that the Markov property of actions of a flow table entry is the sufficient and necessary condition for the one-to-many mapping of that entry to be correct.

5.1 Per-packet Consistency

Existing notions mostly focus on the journey of a single packet (or flow) to make sure the packet (or flow) is handled by one and only one policy [20]. Per-packet (or flow) consistency, as the most general and inclusive (compared to correctness requirements focusing on subsets of trace-properties of a packet such as loop or blackhole [12, 18]), for instance, guarantees that every packet (or flow) traversing the network is processed by exactly one consistent 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 the reordering of packets (or flows) and violations of the dependencies between different packets (or flows). In the firewall example, for instance, the policy generated by the firewall depends not only on the “packet-in”s that the controller application receives, but also on their ordering, e.g., it will permit the communication if the internal host’s request is received before the reply from the external host and will otherwise block the communication. In the NAT example, evicting a flow table entry depends not just on a single packet but a trace of packets. Similarly, in the load-balancer example, forwarding decision for a packet depends on the history of packets matching the same flow table entry. In non of those examples, one single packet (or flow) is handled by a mixture of policies. Hence, per-packet (or flow) consistency is preserved.

To better illustrate that current consistency definitions are insufficient for practical uses, we give a simple example where the one-to-many mapping is not deployed and show that the mechanism deployed for preserving per-packet consistency, i.e., tagging packets using the policy ID and including the tags in the lookup operations, 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 port IDs of applications) into different classes, and then process traffic from different applications in different ways. A simple way to implement this policy is to deploy different logical units (that could be implemented with OpenFlow switches) 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 “censoring” box. The censoring box, in turn, sends packets from host $A$ or $B$ to the censor. Let’s call this policy $P1$. Now, assume that the tenant decides to switch from policy $P1$ described above 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 data center provider uses “consistent updates” mechanism for updating the policies, the update mentioned above will no longer be an atomic single step update, and 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 traffic with 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 the $x$ is no longer censored and replies back using application $x$. Its reply, alas, will be delivered to the censor 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 ensure a correctness model to capture the observations that end-host and controller applications can make. Hence, we borrow the notion of observational equivalence from applications’ perspective from LIME [9] (which uses it to reason about the correctness of a migration system that provides a restricted API), and we replace its model with a probabilistic model on a partially ordered set of events. This new model could be more 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 [10]), and of re-ordering of some packets (since a network that does not provide in-order delivery could change the order of some packets). This model could also capture in-order delivery (which is useful for reasoning about the control channel). In this work we assume that the network is best-effort and could have arbitrary delays.

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 input events, $E$, probability of observing $E$, by any arbitrary application, while having $L$ in the network is equal to the probability of observing $E$ while having $P$, i.e., $Pr_L[E] = Pr_P[E]$.

5.3 Memoryless Packet-handling Operations

In Section 4, we informally showed that what changes under the one-to-many mapping is the 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 a prior action on the same packet (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 for the correctness of the one-to-many mapping. We try to formalize this intuition in this section.

A flow table entry could be thought of as a function that accepts a sequence of matching packets as input, and produces partially ordered sets of (potentially modified and re-written) packets to be sent out of the switch as output. For each incoming packet, the set of output packets could be empty (if the packet is dropped), have one member (if the packet is sent to one output port only), or have more than one member (e.g., to multicast). In the latter case, even though the switch might perform the send operations in a particular order, or even if the flow table entry specifies the send events as an ordered-list to be performed with a pre-specified ordering, those orderings are not always observable to end-points and the controller. A send event by a switch only becomes observable after that packet is received by an end-point or by the controller, and, in general case, even if the flow table entry sends a packet to host $A$ before sending it to host $B$, $A$ might still receive the packet after $B$ does. Hence, in absence of in-ordered channels, no ordering between send events is observable. More formally, the output of a flow table entry is a partially ordered set of send events, where any pair of send events are ordered only if their ordering is observable, i.e., at least one of them is sent over an ordered channel.

We assume that a flow table entry only sends packets after receiving packets (i.e., it does not spontaneously generate packets). Hence, a send event always depends on at least one receive event (Lemma 1). An event could only depend on the events that precede it (not the events that could happen after it or in parallel with it). Hence, a send event of a flow table entry could only depend on the send events that happen before it. Given that the ordering of two send events could only be known if at least one of them uses an ordered channel, it follows that dependencies between send events of a flow table entry is only possible if at
least one of the events is sent over an ordered channel. In other words, no dependencies between send events of a flow table entry is observable in absence of an ordered channel (Lemma B).

Lemma C: Ordering of two send events of a flow table entry is observable, i.e., $s_1 \rightarrow s_2$, if and only if $s_1$ (the first send event) is sending over an ordered channel. I.e., either directly or with transitivity, $r_1 \rightarrow r_2$, where $r_1$ and $r_2$ are receive events of $s_1$ and $s_2$, respectively. Because having an ordered channel is the necessary condition for preserving ordering of send events of a flow table entry (Lemma C) and mapped physical networks do not have an ordered channel under one-to-many mapping, the ordering of send events will not be preserved under one-to-many mapping. I.e., there is no ordering between $r_1$ and $r_2$.

Definition of Markov property of an flow table entry: a flow table entry is said to have a Markov or memoryless property if any future event (send event) performed by it depends only upon the present event (send or receive event), not on the sequence of events that preceded the present event.

Theorem 1: Markov property of send events of a flow table entry is the necessary condition for end-to-end correctness of the one-to-many mapping of that entry.

Proof: Assume, by contradiction, that there is non-Markov flow table entry, $f$, which could be correctly mapped under one-to-many mapping, i.e., $f$ has a send event, $s$, which depends on the present and past events, $e_1, e_2, e_3, ...$. Non-Markov property of $f$ implies that $Pr[s|e_1, e_2, e_3, ...] \neq Pr[s|e_1]$. (a) If there exists a send event $e_i$ among $e_1, e_2, e_3, ...$, then there is a dependency between two send events, $s$ and $e_i$ under the one-to-many mapping. This contradicts with Lemma C. (b) If all $e_1, e_2, e_3, ...$ are receive events, then while they will all be received by the logical flow table entry, under the one-to-many mapping, any partitioning of those receive events between the physical mapped flow entries is permissible. Assume that a physical flow entry only receives the present event, $e_1$. Its send event probability, therefore, will be $Pr[s|e_1]$ which, given the non-Markov property of $f$ is unequal to $Pr[s|e_1, e_2, e_3, ...]$. Now, let’s call corresponding send events (by end-hosts) of the receive events ($e_1, e_2, e_3, ...$) $E_1, E_2, E_3, ...$ (i.e., $e_1$ is the event of receiving a packet by $f$, and $E_1$ is the event of sending that packet by an end-host), and assume send event $s$ by the switch results in receive event $S$ in a host. For simplicity, assume that we have no packet drops. In this case, the probability of $R$ in the logical network is: $Pr[R] = Pr[E_1, E_2, ...] \times Pr[s|e_1, e_2, ...]$, but the probability of $R$ under the one-to-many mapping is: $Pr[R] = Pr[E_1, E_2, ...] \times Pr[s|e_1]$. Hence, $Pr[R] \neq Pr[R]$ which contradicts with the assumption that the correctness is preserved under the one-to-many mapping of $f$.

Theorem 2: Markov property of send events of a flow table entry is the sufficient condition for end-to-end correctness of the one-to-many mapping of that entry.

Proof: Intuitively, if all network operations are memoryless, i.e., if no send events depends on the history of previous events (receive of other events or other send events), Memoryless property of an entry implies that for each send event $s$ and all the events that preceded it $e_1, e_2, ...$ (where $e_1$ is the current event), $Pr[s|e_1, e_2, ...] = Pr[s|e_1]$, i.e., $s$ depends on $e_1$ only. From Lemma A, we know that a send event depends on a receive event, and from Lamport definition of events, we know that the receipt of a single message does not coincide with the sending or receipt of any other message. Hence, $e_1$ is the event of receiving one single packet. Therefore, $s$ is caused by $e_1$, and the one-to-many mapping preserves causality.

6. DISCUSSION AND CONCLUSION

While certain classes of abstractions cannot be mapped with the simple existing one-to-many technique, there are at least two straight forwarded (yet, possibly suboptimal) solutions around this limitation: (a) developing more advanced mapping techniques, e.g., to detect the minimum set of dependencies that could between distributed set of flow table entries that need to be synchronized and kept consistent for the mapping to handle traffic correctly. For instance, Lemma C shows that ordering of actions of a rule only violate correctness if there exists an ordered list of two actions where the first one is sending over an ordered channel. To fix that issue, a more advanced mapping technique could scan the flow table entries for instances of such actions and apply solutions like serializing the physical switches for those actions, (b) alternatively, the developers could implement their policy that do not violate the correctness, i.e., with flow table entries that possess memoryless property. For the previous example (chain of actions, where the first action is sending over an ordered channel), for instance, rather than having the switch to autonomously performs an ordered set of actions, it is possible to install a flow entry that with the first action (forwarding to the ordered control channel) only, and delegate the task of installing a rule to perform the other action to the controller. Similar fixes could be applied for other examples such as querying traffic statistics (instead of relying on soft-timers).

There is, however, a cost: those fixes and solutions are heavy-weight, decrease efficiency, add to the delay, and increases the control overhead (e.g., to involve the controller in the decisions that switch could locally make). While there is an obvious trade-off between performance and correctness for simple solutions, it is not clear if that trade-off is fundamental. We believe that searching for simple, general solutions with care-
fully measured trade-offs warrants further and deeper investigation.

We set the question of what abstractions could be virtualized and took the initial steps to identify the problems with using a basic virtualization and caching technique, and demonstrated that existing notions of consistency do not capture those problems caused by the one-to-many mapping. While we started investigating the root cause of those problems, better understanding of the causes and possible solutions requires a solid new framework for reasoning about correctness. Those problems also demonstrate the need for new virtualization systems with provably correct behaviors.

7. REFERENCES