COCONUT: Seamless Scale-out of Network Elements

Soudeh Ghorbani and P. Brighten Godfrey
University of Illinois at Urbana-Champaign

Abstract

A key use of software-defined networking is to enable scale-out of network data plane elements. Naively scaling networking elements, however, can cause incorrect behavior. For example, we show that an IDS system which operates correctly as a single network element can erroneously and permanently block hosts when it is replicated.

In this paper, we provide a system, COCONUT, for *seamless scale-out* of network forwarding elements; that is, an SDN application programmer can program to what functionally appears to be a single forwarding element, but which may be replicated behind the scenes. To do this, we identify the key property for seamless scale out, weak causality, and guarantee it through a practical and scalable implementation of vector clocks in the data plane. We prove that COCONUT enables seamless scale out of networking elements, i.e., the user-perceived behavior of any COCONUT element implemented with a distributed set of concurrent replicas is provably indistinguishable from its singleton implementation. Finally, we build a prototype of COCONUT and experimentally demonstrate its correct behavior. We also show that its abstraction enables a more efficient implementation of seamless scale-out compared to a naive baseline.

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

Keywords Software Defined Networking, Virtualization, Network Functions, One Big Switch, Replication, One-to-many Mapping, Consistency, Correctness, Weak Causal Consistency

1. Introduction

An important use of software-defined networking (SDN) is to automate scaling of networks, so that individual network functions or forwarding elements can be replicated as necessary. Replication of network elements allows capacity to scale gracefully with demand [11], provides high availability [11], and assists function mobility [20, 63]. Multiple SDN systems replicate network elements, in different ways. Each tenant in a virtualized data center might be presented one logical "big switch" abstraction that in reality spans multiple physical hardware or software switches [11, 42, 46]. As another example, Microsoft Azure's host-based SDN solution leverages VMSwitches to build virtual networks where each host performs all packet-actions for its own VMs [4]; these VMSwitches act in parallel and independently despite the fact that they might form a single virtual network. Outside of virtualization, caching of forwarding rules is a form of replication; for example, [15, 32, 59, 62] cache rules at multiple locations in the network, and Open vSwitch [50] caches rules from user-space into kernel-space, which is critical to improve performance.

All these systems replicate logical network elements by duplicating forwarding rules across multiple locations, without coordination between them, which we call *simple replication*. Our work begins by asking: *Does simple replication for scaling out network elements preserve the semantics of a single element?* If the network elements are stateless, the simple replication approach taken by existing systems is enough (§3). But if a developer writes a network function or application such as a stateful firewall on top of a single virtual "big switch", is its functional behavior the same as if it were running on an actual single physical switch? We show that simple replication does indeed change the network's semantics: for example, a replicated firewall can erroneously and *permanently* block hosts. In fact, our experiments show there are scenarios that these problems occur frequently (§3).

How, then, could an SDN programmer deal with this problem? Living with the risk of incorrect functionality is unappealing, as critical infrastructure elements such as security appliances (firewalls, intrusion detection systems, *etc.*) are increasingly deployed in a scale-out manner. Alternately, the programmer could write her application so that it takes into account the distributed implementation of network elements and associated race conditions. But this is inconvenient for the programmer at best, and is infeasible at worst, when replication is explicitly hidden in the physical infras-

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04. . . 5.00

DOI: http://dx.doi.org/10.1145/3064176.3064201

tructure underneath a tenant's virtual network. Indeed, one lure of the virtualized cloud for tenants is the prospect of migrating their workloads and network applications to the cloud "as-is", *i.e.,* with no re-designing and re-architecting of their applications, with the expectation that they perform exactly akin to their non-virtualized networks [13, 37].

Our goal is thus to achieve **seamless scale-out** for network forwarding elements: *a system which guarantees that an SDN application writer can program to the semantics of a single device, but which utilizes multiple replicated elements behind the scenes.*

Achieving seamless scale-out is not easy. The most generic solution would be to synchronize replicas to provide a strongly consistent logical view, but the required locking would not achieve the performance necessary for the data plane [4, 62]. Recent work [30, 33, 43, 53] provides a form of consistency in the data plane in the sense of ensuring "trace" properties of a single packet's path, as in Consistent Updates (CU) [53]. But this is essentially orthogonal to our goal; seamless scale-out does not require per-packet path consistency, and systems that provide per-packet path consistency can even *cause* the correctness problems described above (§3.2). Also, the mechanisms used to implement CU assume a single atomic update point (the ingress switch of a packet's path). No such atomic update point exists in our setting, because we need to preserve the single-device semantics across a large number of flows across the whole network with potentially unspoken dependencies.

The system we present here, COCONUT ("COrrect COncurrent Networking UTensils"), provides seamless scale-out for network elements defined by a dynamically-updatable OpenFlow-like abstraction. To work towards a solution, we observe that the culprit of scale-out correctness problems is violation of what we call *weak causality*. For example, simple replication can cause a replicated firewall to miss the weak causal dependency between a client's outbound request and a server's inbound response, so it sees a seemingly-unsolicited inbound response first and permanently blocks traffic from that server.[1] We design a set of high-level algorithms to avoid weak causality violations, drawing on the classical concept of logical clocks [38] to track the state of each forwarding rule at each abstract network element. But providing a practical and scalable implementation of these high-level algorithms is challenging; switches do not directly implement logical clocks, and emulating a large vector of logical clocks in packet header fields is impractical. We provide a practical realization of those algorithms using OpenFlow-compatible switches, leveraging the distinguishing characteristics of SDNs and virtualized networks, and is thus suitable for deployment in the context of a modern virtualized data center with software switches in each physical host. Our design uses limited bits in header fields of a physical network to emulate logical clocks in the virtual network, while dealing with concurrent creations and changes of multiple virtual networks that may interleave with each other and compete for use of these bits.

To prove that COCONUT correctly provides seamless scale-out—that is, the scaled-out network is indistinguishable from a singleton network element—we need a new analytical framework that takes into account the sequence of observations made by the end-points, with potential interdependencies. We formalize this with a definition we call **observational correctness** that requires that any sequence of end-point observations in the scaled-out network is plausible for the singleton version. In tune with what applications expect from best-effort networks, this model is permissive of occasional packet drops and re-ordering, while prohibiting weak causality violations (breaking "happened before" relations [38] adopted for best-effort networks, §4.1) that could jeopardize applications' correctness. We formally prove that COCONUT provides observational correctness.

We implemented a prototype of COCONUT integrated with Floodlight [2], Open vSwitch [50], and OpenVirtex [7]. We evaluated COCONUT and several alternative schemes on a hardware SDN testbed arranged to emulate a 20-switch fat-tree topology and in Mininet [23] emulations up to 180 switches, with multiple topologies, load patterns, and SDN application scenarios. Our findings are as follows:

- A strawman solution, providing strong consistency (SC) similar to [20] by routing all data traffic through a controller during updates, would come at too high a cost: about 12 Gbps bandwidth overhead and a $20\times$ increase in user traffic latency even in a modest-sized network. COCONUT incurs no measurable data plane performance overhead, and has significantly lower overhead in terms of forwarding rule update delay ($3.5\times$ faster in a network with 128 hosts and 80 switches) and number of forwarding rules ($2\times$ lower).
- Unlike baseline simple replication, COCONUT correctly achieves seamless scale-out, and does so with modest overhead. For a 180-switch network, for example, the mean forwarding rule updates time is only $1.2\times$ slower than simple replication, and the mean number of forwarding rules increases by only $1.6\times$, with just 0.7% of that overhead persisting for longer than 100ms.
- We also compare with a natural implementation where the programmer avoids replication-related race conditions "manually" within the SDN application. COCONUT enables an implementation that is both more convenient for the programmer and provides $2.8\times$ lower mean latency for user data flow initiation due to its efficient logical clock-based approach.

In summary, our key contributions are:

[1] Note that even this simple example involves multiple flows entering the network at different points, illustrating the aforementioned insufficiency of using a single atomic update point as in CU [53].

1. We observe that simple replication breaks the semantics of a single network element and show experimentally that it causes application-level incorrect behavior.
2. We present COCONUT, a system for seamless scale-out in the context of OpenFlow forwarding elements in a virtualized data center, and prove that it correctly preserves a single-element abstraction.
3. We demonstrate experimentally that COCONUT achieves its goals with modest performance overhead.

We believe this lays the foundation for a practical and dependable service model for virtualized network infrastructure, and a powerful abstraction for programming SDNs.

2. Background

2.1 Basic Abstractions

COCONUT provides seamless scale-out for network elements. The abstraction of a *network element* that we work with here is essentially an SDN device such as an OpenFlow switch. Each element or switch has a table of *rules*, each rule containing a *priority*, a *match* on packet headers, and a list of *actions*. Although each individual rule is stateless, the system is not: the controller can dynamically update rules based on dataplane events, *e.g.,* failures. Upon receiving a packet, the switch executes the actions for the highest priority rule that matches the packet. These actions could result in changes in the packet, dropping it, or forwarding it.

(How) are networking elements scaled-out today? Scaling out can be realized via *simple replication* or one-to-many mapping, where a logical rule is mapped to a distributed set of physical rules, each individually capable of fully implementing the logical rule. In this technique, before installing a rule in multiple physical flow tables, an entity such as the network hypervisor [37] typically rewrites the rule. For example, a rule that matches on virtual ports will need to be rewritten to refer to physical ports [11, 20]; virtual addresses may be translated to physical addresses or packets may be placed into tunnels [37]; and rules that match in part with wildcards could be replaced with multiple rules, each using only exact-match values [12, 37]. The latter mechanism is used in software switches, where wildcard rules in userspace are cached as exact match rules in the kernel to enhance performance [37].

Prior to COCONUT, a number of systems have provided simple replication, mostly for scaling out *static stateless* network elements, *i.e.,* those whose actions or presence in the network do not depend on the history of previous packets or actions [18]. Simple replication of stateless elements preserves the semantics of applications [18].

Modern programmable networks, however, are much more dynamic. This can come in the form of controllers adding, removing, or modifying forwarding rules dynamically in response to application traffic. The question is, can these stateful elements be replicated via simple replication?

Monitoring	
srcip=127.1.*.*	count
*	drop

Routing	
dstip=127.2.*.*	fwd(1)
*	drop

Monitoring + Routing	
srcip= 127.1.*.*, dstip=127.2.*.*	count,fwd(1)
srcip= 127.1.*.*	count
dstip=127.2.*.*	fwd(1)
*	drop

Figure 1: Composing monitoring and routing.

We list a few key existing applications of simple replication (§2.2), before showing that this technique may cause incorrect application behavior when used for implementing dynamic stateful network functions (§3). We also show that the existing works on correctness in the network not only do not solve this problem but can exacerbate it (§3).

2.2 Applications of Replication

Network virtualization: Simple replication is a key technique for building distributed virtual switches. Nicira's NVP [37] and OpenVirtex [7], for example, can provide a one-big-switch abstraction that connects VMs on the same virtual network even though they are located in different physical hosts or regions of the physical network, and whose locations may change due to spinning up VMs or VM mobility. This is implemented with simple replication from a single virtual switch onto a distributed set of software switches.

Composition of multiple virtual switches can also result in replication. Under existing composition techniques, multiple logical rules are jointly mapped to a set of physical rules where each physical rule is individually capable of implementing multiple logical rules [16, 28, 46]. For example, Figure 1 shows (a) a monitoring module that performs monitoring based on source address, (b) a destination-based routing module, and (c) a Monitoring+Routing application resulting from parallel composition of the previous two modules (rules ordered from highest to lowest priorities) [46]. The first rule of the monitoring, module, for example, is implemented with 2 rules in the composed application.

Network Function Virtualization (NFV): Performance is a critical consideration in NFV where software is used to implement network functions or applications such as firewalls, load balancers, *etc.* Simple replication, used in caching, is a key technique to enhance forwarding performance in software switches and NFVs [32, 37, 50, 59, 62].

Implementing higher level abstractions: In the context of network programming languages, Frenetic [16] provides high-level primitives such set difference, not directly supported by the hardware, by mapping those primitives to multiple OpenFlow rules, *e.g.,* a rule with the match field `src-IP=186.206.176.* OR src-IP=62.205.112.38`, is implemented via two rules.

In all of the above techniques, each physical instance or replica is *functionally equivalent* to a faithful implementation of one (or more) logical rules, *i.e.,* the replica performs identical actions as the logical rule. In a fully static net-

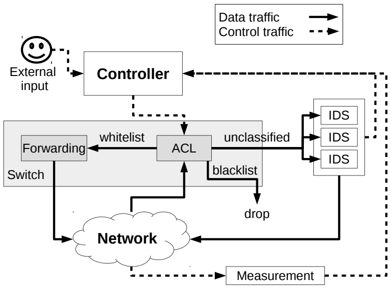


Figure 2: SDN-enabled security architecture.

work, packets traversing the physical network result in the same end-to-end fate as if they were processed directly by rules in the ideal, non-replicated implementation of the logical network. However, as network state changes over time, there may be inconsistent state among the multiple replicas that implement one logical element. Furthermore, this problem may become more serious if the changes are interdependent with application-level behavior (rather than simple route changes). We next see how this may cause application-level incorrect behavior.

3. What Can Go Wrong?

We show with a few examples that the simple replication can break the semantics of a single element and lead to incorrect application behavior. Per-packet consistency [53] does not fix the problem, and interestingly, can even trigger the problem in an otherwise-correct network.

3.1 Example 1: SDN-enabled Security

Network Intrusion Detection Systems (IDSes) and stateful firewalls perform complex traffic processing and analysis that are CPU-intensive and hard to implement at high speed. Performance can be improved significantly by programming faster devices like SDN switches to act as an initial triage filter [49]. As depicted in Figure 2, the switch *whitelists* traffic known to be benign, forwarding it directly to its destination; *blacklists* traffic known to be malicious, dropping it immediately; and sends only the remaining *unclassified* traffic to the IDS device for more expensive analysis (e.g., DPI). The controller uses external input, traffic measurement tools, and notices from the IDS cluster to craft whitelists and blacklists in the ACL table of the switch.

This concept is the crux of several security and DoS protection systems such as Radware's SDN-enabled DefenseFlow [48], and SciPass used in the TransPAC network and Indiana University [9]. At Lawrence Berkeley National Laboratory (LBNL) and NCSA, a similar system that whitelists GridFTP traffic, which is uninteresting from a security standpoint in such scientific environments, reportedly reduces the total traffic volume to their security appliance cluster by about 32-37% on a typical day [8].

This architecture results in frequent ACL changes on switches. Using custom setups that interface with the Bro and Snort IDSes, for instance, LBNL and Indiana University block an average of 6,000-7,000 and 500-600 IPs per day, respectively [8], and systems that whitelist GridFTP traffic at LBNL and NCSA result in a few hundred to several tens of thousands ACL operations per day [10].

The traffic which is unclassified is sent to a cluster of security appliances. Such devices usually ship with analyzers for many protocols and applications to detect protocol and application specific attacks. The `weird.bro` and `scan.bro` scripts in Bro, for instance, give notices when Bro observes data being transferred in a session without seeing the SYN ACK packet of that session, data being transferred without observing ACK, repeated SYN ACK packets for the same session, and failed connection attempts to multiple hosts over a time interval. The notices from the IDS are then sent to the controller application which might in turn install rules on the ACL to block IPs. In some systems, such as SciPass, this blocking is by default *permanent* [3]. Erroneous IP blocking is notoriously hard to debug; in most cases it requires the owner of the IP to call the network operator who then manually inspects the IDS logs [58].

However, this system can encounter a problem if the triage switch is replicated. Consider the following setup. The IDS cluster is set to analyze some protocols including TCP port 80, *i.e.,* if it receives a reply, it checks if the reply is solicited or not. If it is, it forwards the packet to its destination. Otherwise, it sends a notice to the controller to block the source of the traffic. A popular web service on the internal network receives a continual stream of incoming requests from clients on port 80.[2] Let $P1$ refer to the initial policy that TCP port 80 on the switch is unclassified. Next, the network operator chooses to update the policy from policy $P1$ to $P2$ where TCP port 80 is whitelisted. The only affected module is the ACL that should add a rule to forward TCP port 80 to the forwarding table instead of the IDS cluster.

Without replication, at any point during the update, if a server receives a request, it is allowed to reply: its solicited reply either traverses the forwarding table and reaches its destination, or through the IDS that already knows about the request—the request can only be forwarded to its destination by the IDS after the IDS observes the request.

With simple replication, however, the switch might be implemented using rules across multiple physical devices. For example, in a one-big-switch setup with OpenVirtex [7], the single rule that sends TCP port 80 traffic to the forwarding table is now translated into multiple rules, one residing on each physical edge switch that acts as part of the one-big-switch. These rules cannot be installed all at once.

Hence, the following race condition can happen: The new rule for $P2$ is installed at the edge switch connected to host A. Then, A sends a request to host B. The request is directly

[2] Similar problems arise if the service is external and the client is internal.

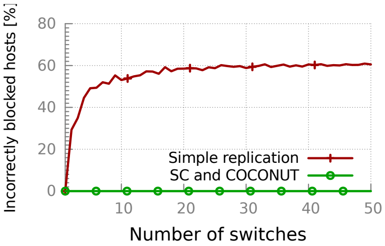


Figure 3: Simple replication causes incorrect blocking.

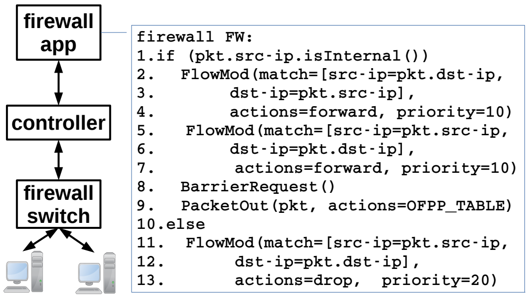
forwarded to B; therefore, the IDS does not observe the request. B replies, and its reply hits a *different* edge switch which still uses policy $P1$. Thus, B's reply is forwarded to the IDS. Since the IDS never saw the request, the IDS sends (false) notices to the controller informing it that the server is sending a stream of unsolicited replies. This will eventually cause the controller to block the server even though the traffic it is sending is already whitelisted and it is legitimately replying to requests it receives. In other words, the hosts observe the following invalid sequence of events: A sends a request, B receives it and replies, B is blacklisted.

This problem is troublesome to resolve. Even though the controller knows a certain type of traffic was whitelisted, it is difficult for the controller to realize the mistake, because a host with some valid traffic might still have sent malicious traffic as well. If the server owner realizes a mistake and phones the network operator, the problem would be hard to resolve as the IDS logs indicate a suspicious server activity (sending unsolicited replies).

To determine how frequent this error can be, we implemented a tree topology with up to 50 leaf switches acting as the logical ACL. Each leaf switch is connected to 5 hosts in Mininet. Each host sends requests to randomly selected hosts with flow interarrival times and sizes drawn from the web-server workload information in [55]. Control delays are drawn from the measurements of HP Procurve switches in [26]. Figure 3 shows the percentage of hosts incorrectly blocked following a single $P1 \rightarrow P2$ policy change, averaged over 100 trials. The percentage of incorrectly blocked hosts rapidly increases with scale, *e.g.,* with a medium-sized network of 20 switches, it approaches 60%. An alternative approach of using symmetric paths for all related flows in that example imposes great overhead for some applications such as GridFTP, used in both NCSA and LBNL [8], that depend on many flows.

3.2 Example 2: Logical Firewall

Imagine that an enterprise network has a 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 policy could be achieved as follows, using a single switch and a firewall application FW running on the controller (Figure 4). Initially, FW installs in the switch a low priority flow table entry that matches all client and server traffic and sends the packet to the controller. When FW receives a packet from a client, it instructs the switch to do three things: (1) install rules to allow bidirectional communication between the client and the server, bypassing the controller, (2) wait for these rules to take effect, via a `BarrierRequest` message, and (3) process the original packet again using the new rules. When FW receives a packet from a server, it must have been unsolicited, so it blacklists the server by installing a permanent high priority rule that drops packets from the server. This rule provides the desired property of safeguarding clients from connecting to malicious servers, even if the client tries to connect.



```
firewall FW:
if (pkt.src-ip.isInternal())
  FlowMod(match=[src-ip=pkt.dst-ip,
        dst-ip=pkt.src-ip],
        actions=forward, priority=10)
   FlowMod(match=[src-ip=pkt.src-ip,
        dst-ip=pkt.dst-ip],
         actions=forward, priority=10)
  BarrierRequest()
  PacketOut(pkt, actions=OFPP_TABLE)
else
   FlowMod(match=[src-ip=pkt.src-ip,
         dst-ip=pkt.dst-ip],
         actions=drop,  priority=20)
```

Figure 4: Replicated firewall incorrectly blocks communication.

With simple replication, *i.e.,* if that logical switch is in reality mapped to more than one physical switch, the client-to-server traffic could traverse one physical switch, s_1, and the resulting server-to-client traffic traverses a different physical switch s_2. In this case, the response traffic may reach s_2 before the rules for bidirectional communication are installed on it, intuitively because the `BarrierRequest` now waits for rules to take effect at only one switch, rather than all. The packet, therefore, will be handled by the default rule which sends it to the firewall application, which proceeds to install a high priority rule D to block all traffic for that flow—an undesirable outcome and something that would not happen without replication. Note that even when the rules that allow client-server communication are installed on s_2, the switch continues dropping traffic due to rule D, since it has a higher priority. In an experiment with similar setup as §3.1, when the client and server are connected to two separate replicas, we found the communication is incorrectly blocked 21% of the time. This example is similar to the previous IDS example in its effect, but here it is triggered by normal client-server traffic rather than an administrator's policy change.

3.3 Shortcomings of Existing Approaches

Per-packet Consistency Is Not Enough. A line of work has preserved properties of a single packet's journey, even during network updates — for example, avoiding loops and black holes [30, 43] or preserving *per-packet (or per-flow) consistency*, wherein every packet (or flow) traversing the network is processed by exactly one global network configuration and never by a mix of multiple configurations [53]. These properties do not achieve seamless scale-out, because they do not preserve dependencies *across* different packets or flows. In the IDS example, each source-to-destination flow was processed by only a single policy; the problems are only visible *across* flows, violating the request/reply orderings that the IDS policy depends on. This is a critical distinction, because Consistent Updates (CU) [53] implements per-packet consistency by relying on each packet's entry point as a single point of atomic update. Seamless scale-out involves behavior of packets across many flows from multiple entry points and potentially flowing through endhosts; no atomic update is possible.

In fact, application-level incorrectness can occur *because* of deploying CU [53] to guarantee per-packet consistency. This is because the two-phase update algorithm of CU itself causes replication. We return to the example of §3.1. In the non-replicated setting, if the network uses CU to update the policies to whitelist TCP port 80 traffic, the update will no longer be a single step, because the flows using rule $R1$ on the ACL need to be updated one by one. Suppose that the flow from A to B is updated, but other flows (including the one from B to A) are not yet updated. In this case, the ACL will have 2 rules corresponding to $R1$ on the ACL in Figure 2 (not shown): an old rule to match traffic using old tags (old policy traffic) and the new rule with new tags (new policy traffic). Now, host A sends TCP port 80 traffic with the new tag, which is forwarded to B (new policy). B receives the packet, and replies. Its reply to A, however, *is* delivered to the IDS since it has the old tag. The IDS consequently blocks B given that it has not seen the request, something that would not happen without CU. The underlying problem in this case is that CU maps a single logical rule to multiple physical rules with different tags.

Strong Consistency Is Cost Prohibitive. SDN switches do not directly provide primitives to preserve strong consistency, but one can implement it using the controller [17, 20, 39]: when a rule needs to be updated, direct all related flows to the controller, which temporarily emulates the switches' behavior; perform an atomic rule update at the controller; update the switches; and finally shift traffic back to the switches. This technique would correctly achieve seamless scale-out. But we show in §5 that it has dramatic performance penalties, *e.g.,* 20× increase in delay for the IDS example. Shifting traffic to the controller is thus appropriate for relatively rare virtual network migration events supported by [17, 20, 39] but not for the ongoing process which we hope to support.

4. Design of COCONUT

The previous examples demonstrate that simple replication does not provide seamless scale-out. In this section, we begin by presenting the intuition of what logical property the network requires to achieve seamless scale out. We call this property weak causal correctness, formalize it (§4.1), show the intuition behind our design with simple (but impractical) algorithms to preseve weak causal correctness (§4.2), and finally present a practical realization of the design (§4.3).

4.1 Not All Orderings Are Created Equal

Causality violations in §3, *e.g.,* receiving a response before or without the request that caused the response, are caused by inconsistent state among replicas of one single logical rule—a packet is handled by a new instance of a logical rule and another packet that "comes after" it is handled by an old instance of the same logical rule. In the IDS example in §3.1, for instance, the request packet is handled by a new instance of the ACL rule, but the reply that it triggers is handled by the old instance of the same logical rule. Thus, the IDS receives the reply packet first, missing its dependency (the request).

On the surface, it might seem counter-intuitive that the ordering between those packets is a problem that could compromise application correctness, since even in non-replicated best-effort networks, packets can be reordered or dropped. The sublety here is that even in best-effort networks some orderings, that we call weak causality, are always preserved. For example, no amount of reordering or packet loss will change the fact that with a standard TCP implementation, receiving a SYN packet always happens before sending the first SYN ACK.

We use this intuition to formally define weak causality and observational correctness. We first formalize network events and define networks' behaviors.

The endpoints interact with the network with `send` and `receive` *events*. These are the only events we are ultimately interested in because they are the only *externally visible* events, *i.e.,* while the network could have multiple internal events such as rule lookup, packet rewrite, *etc.*, those internal events are not visible to the endpoints. The distinction between internal and external events is a common technique for defining correct behavior of state machines [41]. The notation $r_{h,i}(pkt)$ and $s_{h,i}(pkt)$ are used to respectively refer to the event of receiving and sending packet *pkt* by endpoint p_h where this event is the ith event happening at p_h. Each sequence of external events is a *trace*. The *behavior* of a system is the set of all plausible traces in that system [41]. In a system with n endpoints, $p_{i \in \{0,\ldots,n-1\}}$, a *local history* of endpoint p_i, denoted by L_i, is a sequence of $e_{i,j}$s, where $e_{i,j}$ is the jth external event that happens at p_i, *i.e.,* the system's behavior observable by p_i. A *history* H=$\langle L_0, L_2, ..., L_{n-1}\rangle$ is a collection of local histories, one for each endpoint.

Observational correctness: For a physical network, P, to be an observationally-correct implementation of a logical or abstract network, L, any trace in P's history should be a plausible trace in the history of an ideal, non-replicated implementation of L. That is, the possible behavior of P is a subset of the possible behavior of a non-replicated implementation of L. We see in §3 that this condition does not hold under simple replication, *e.g.,* the following trace that happens in the replicated network in the example in §3.1 is not plausible in the non-replicated networks: *A sends a request, B receives the request, B sends a reply, IDS receives the reply*. (*i.e.,* the trace misses the event of the IDS receiving the request that triggers the reply).

Weak causality: Event $e_{k,l}$ has *weak causal dependency* on event $e_{i,j}$, shown by $e_{i,j} \rightarrow e_{k,l}$, if one of the following cases hold:

R1: local dependencies. This applies when i=k (*i.e.,* both events happen in the same endpoint), $j < l$ (*i.e.,* $e_{i,j}$ comes before $e_{k,l}$), and $e_{k,l}$ is a send event. In the example above, '*B sends a reply*' is locally dependent on '*B receives the request*'. Note that we replace the traditional "program order" [5] with local dependencies in rule $R1$. This is done to account for the fact that a best-effort network can reorder packets. The above condition on $e_{k,l}$ is what distinguishes our notion of weak causality from the original definition of causality in [5].

R2: sends-to. $e_{i,j}$ and $e_{k,l}$ are respectively the events of sending and receiving the same packet.

R3: transitivity. There is some other $e_{r,t}$ event such that $e_{i,j} \rightarrow e_{r,t} \rightarrow e_{k,l}$.

If an event $e_{k,l}(q)$ involving packet q has weak causal dependency on an event $e_{i,j}(p)$ involving packet p, we say that q has weak causal dependency on p, denoted by $p \rightarrow q$. Events and packets with no weak causal dependencies are called *concurrent*.

While best-effort networks can drop packets and reorder concurrent packets, they preserve weak causality. For instance, if concurrent packets $pkt1$ and $pkt2$ are sent to endpoint p_i, receiving them with any order or not receiving one or both of them are permissible, *e.g.,* $\varnothing$, $\langle r_{i,j}(pkt1)\rangle$, and $\langle r_{i,j}(pkt2), r_{i,j+1}(pkt1)\rangle$ are plausible traces. However, a host always receives a SYN ACK packet after sending a SYN packet (its weak causal dependency). Receiving a SYN ACK without sending a SYN packet, or receiving it before sending a SYN packet, therefore, are not plausible traces.

Unlike non-replicated networks, replicated ones can violate weak causality, *e.g.,* the IDS in §3.1 receives a reply while missing its dependency. This implies that replicated networks can have traces (those that violate weak causality) that are not plausible in logical networks that they intend to implement, and consequently are not correct.

Root cause of weak causality violation: It is not hard to see that if no rule changes, then any trace in the replicated network is a plausible trace of the logical network. The fact that a replicated network can have implausible traces, therefore, results from handling packets with inconsistent instances of rules. Intuitively, handling concurrent packets with inconsistent instances does not result in a implausible trace. Even in non-replicated networks, it is permissible to handle two concurrent packets with inconsistent state while the network state is changing. The problem happens when orderings of packets are known, *e.g.,* $p \rightarrow q$. In non-replicated networks, p cannot be handled by a newer state compared to q. Under simple replication, in contrast, this property does not automatically hold because the instances handling p and q could be different. Therefore, p might be handled by a newer state compared to q. In the IDS example, for instance, the event of the IDS receiving the reply (e_3) happens after the event of B sending the reply (e_2), which in turn happens after the event of B receiving the request (e_1). Yet, even though $e_1 \rightarrow e_3$, the packet associated with e_1 (request) is handled by a newer instance compared to the instance that handles the packet associated with e_3 (reply). We provide algorithms to ensure that with COCONUT's replication, for any two packets p and q where $p{\rightarrow}q$, applying a logical rule on q implies that no newer version of the same logical rule is applied on p. We show in [19] that preserving this property is sufficient for observational correctness:

Theorem 1: Any behavior of COCONUT's implementation of replicated networks could have happened in the logical network.

The intuition behind the proof is to show that COCONUT is weak causality-aware[3] (Lemma 2 in [19]) and this is sufficient for observational correctness.

4.2 COCONUT's High-level Algorithms

In an implementation of a logical network with m logical rules, $LR_{0\leq i<m}$, one single logical rule, LR_i, is mapped to multiple physical instances, $PR_{i,j}$, where j is the ID of the switch hosting the $PR_{i,j}$ instance.

Changes to a logical rule should be replicated across all the physical rules that implement it. Without enduring the prohibitive cost of synchronization for atomically updating all the physical rules at once and in unreliable networks where elements can fail, inevitably, there exist instances when different physical replicas are in different and inconsistent states. Fortunately, this different network state usually does not cause anomalous application behaviors — unless endpoints' applications receive packets from the network, they are unaware of the network state. The problem happens when the packet is handled by a new version of the rule and then triggers a causal sequence of events leading to some packet (perhaps the same or newly generated packet) being handled by an old version of the rule.

We leverage this observation and the classical concepts of logical and vector clocks to prevent such weak causality violations. We use *logical clocks for tracking network state changes and restricting the space of executions to those that are weakly causally consistent.* Endpoints affix vectors of logical clocks to packets that show their latest observed network state. These clocks prohibit switches from applying outdated rules that might violate weak causal correctness,

[3] A network is *weak causality aware* iff for any two packets p and q and for any logical rule R, $p \rightarrow q$ implies that the version of R that handles q is at least as recent as the one that handles p.

Algorithm 1 Ideal Switch sw

```
1: procedure UPDATE(rule $PR_{i,sw}$)
2:     $VC_{sw}[i]$++
3:     regular-update($PR_{i,sw}$)

4: procedure RECEIVE(packet $pkt$, port $ip$)
5:     rule $PR_{i,sw}$ =lookup($pkt, ip$)
6:     while ($VC_{sw}[i] < VC_{pkt}[i]$) do
7:         update($PR_{i,sw}$)
8:     $VC_{pkt}[i] :=$ max($VC_{sw}[i], VC_{pkt}[i]$)
9:     regular-apply($PR_{i,sw}, pkt, ip$)
```

and prompt them to update their rules before applying them to packets.

More specifically, in a network with m logical rules, each packet pkt carries an m-dimensional vector of logical clocks, VC_{pkt}, in which $VC_{pkt}[j]$ shows the latest version number of logical rule LR_j that pkt has "observed"—that is, the latest version known at the sender of pkt when it was sent, or the version applied to pkt along its path (whichever is more recent). As an example, the switch that handles a packet p with the second version of the logical rule LR_j sets its $VC_p[j]$=2, and the endpoint that receives p sets $VC_q[j]$=2 for a packet q that it sends after receiving p. We assume that switches are preloaded with all versions of rules, similar to the way that switches can be preloaded with failover rules.

The reader will have already realized that in large-scale multi-tenant datacenters hosting $10K$s of virtual networks [14, 57], storing a clock value for every rule in every packet, performing operations on these VCs, and preloading switches with all rules are infeasible. Our goal in this section is to convey the intuition behind our design and reason about its correctness. Later, §4.3 presents a scalable and OpenFlow-compatible, but slightly more complex, emulation of these algorithms. Three types of entities—switches, shells, and the controller —work with the vector clocks carried by packets. We describe the role of each next.

Switch operations: Each physical switch sw has a logical clock $VC_{sw}[j]$ for each logical rule $PR_{j,sw}$ hosted at the switch. This clock stores the current version number of the rule that the switch will apply to matched packets. Note that one logical rule can be hosted at multiple physical switches, and these may have different clock values while the rule is being updated. When a switch needs to update a rule, it also increments its corresponding logical clock (procedure `UPDATE` in Algorithm 1; `regular-update` is the regular rule update operation without COCONUT).

When receiving a packet pkt on input port ip (procedure `RECEIVE` in Algorithm 1), the switch sw looks up the rule that needs to be applied on the packet, $PR_{i,sw}$. If $VC_{pkt}[i] > VC_{sw}[i]$, the packet or a packet that *happened before* was already handled by a newer version of LR_i than the one currently active on sw. Hence, applying the outdated version risks weak causality violations once pkt is received by any endpoints. So at this point, sw is required to update the rule before handling pkt. The `update(`$PR_{i,sw}$`)` function has the switch update $PR_{i,sw}$ using the preloaded rules, its clock for this rule, and the packet's clock for this rule, $VC_{pkt}[i]$, to show the latest version number. Finally the switch acts on pkt by applying the rule (line 9 in Algorithm 1).

Deleting a rule $PR_{j,sw}$ is a special case of updating it: the logical clock of the deleted rule, $VC_{sw}[j]$, is incremented and its value is set to $\varnothing$ (a special value) dictating sw to apply other rules for matching packets.

Algorithm 2 $Shell_i$

```
procedure RECEIVE(packet $pkt$)
    for $j \in VC_i$ do
        if $VC_{pkt}[j] > VC_i[j]$ then
            $VC_i[j] := VC_{pkt}[j]$
    remove-VC($pkt$)
    regular-fwd-to-host($pkt$)

procedure SEND(packet $pkt$)
    add-VC($pkt, VC_i$)
    regular-send-to-net($pkt$)
```

Controller's operations: The *controller* sits between the network hypervisor and the network, and is tasked with installing the physical rules, such as those sent by the network hypervisor to it, on switches.

Shell's operations: A *shell* is a shim layer sitting between each endpoint and the network, which can run in the hypervisor. Shells hide VCs from the endpoints by performing the necessary logical clock operations on their behalf. For each endpoint p_i, its shell $shell_i$ keeps an m-dimensional vector VC_i of logical clocks. $VC_i[j]$ contains the *max* version number of logical rule j observed in the logical clock of any packet p_i has received.

For each incoming packet, pkt, $shell_i$ updates VC_i if the packet carries any newer information, *i.e.,* $\forall j, VC_i[j]{=}max(VC_i[j], VC_{pkt}[j])$. It then removes VC_{pkt} from the packet before passing it to the endpoint (procedure `RECEIVE` in Algorithm 2). For any outgoing packet pkt, $shell_i$ appends its local VC, VC_i, to the packet before sending pkt (procedure `SEND` in Algorithm 2). This VC prevents switches from handling pkt with outdated rules that could violate weak causality.

4.3 OpenFlow-compatible Implementation

Having a scalable implementation of the simple algorithms in §4.2 is challenging. A major scalability bottleneck is the size of the time vectors. In general, in a distributed computation with N processes, causality can only be characterized by vector timestamps of size N, *i.e.,* the causal order has in general dimension N [56]. For implementing weak-causally consistent SDNs, where the vector timestamp tracks the version of every forwarding rule in the network, it would be overly burdensome (in terms of bandwidth and CPU) for packets to carry such large vectors and endpoints, switches, and controllers to operate on them. Another scalability challenge is preloading switches with all versions of rules. In addition to these *scalability* challenges, there is a *feasibility* challenge: vector clocks and their related operations cannot be readily implemented with the match/action operations on commodity switches today.

To overcome the feasibility challenge, we note that the weak causality problem that VCs solve only arises when a logical rule is *in flux*: there are both old and new physical instances of the rule in the network. Vector operations are not needed for *stable* rules that are not in flux (*i.e.,* before or after updates). Even when rules are in flux, their exact version numbers are not necessary for preserving weak causality. As long as the old versions of a rule are eliminated from the network, it is sufficient to know that the rule is being updated which can be sufficiently characterized by one single bit, which we call a *tag bit* (TB), to identify the current and new versions. Switches and endpoints then need to "mark" the TBs of the packets that are handled by such rules or any packet after them (by a tagging operation which can be implemented in existing switch hardware), and for in-flux rules, switches need to apply their updated versions to the tagged packets (*e.g.,* by having the updated rules as higher priority rules that match on the tag). These simple tricks enable us to emulate vector operations for updating a logical rule by reserving a TB for it and deploying regular match-action operations, thus solving the feasibility challenge. Concurrent updates could use separate update TBs.

The fact that only the in-flux rules require tags for correct operations, along with coordination at the SDN controller, also aids us to sidestep the scalability challenge: once an update operation terminates, *i.e.,* once the controller learns that all the physical instances of a logical rule LR are updated, it can re-use its TB for updating other rules. We can thus concurrently update as many logical rules as the number of bits dedicated to TBs. While this is likely to be sufficient for a single virtual network, it will still be a scalability bottleneck for cloud providers that host $10K$s of virtual networks and should support millions of concurrent updates of all of these networks per day [14, 57]. We resolve this by capitalizing on the fact that virtually all network virtualization platforms [7, 37, 50, 60] isolate traffic within each virtual network, so that traffic cannot leak between two virtual networks. Packets carrying extra bits disjoint from the bits used by the hypervisor and rules matching on them do not violate this property. Hence, multiple virtual networks can concurrently use the same TBs. Furthermore, the controller can preload switches with only the necessary rules.

We describe the practical implementation of COCONUT's algorithms as well as its failover operations after explaining the notations and requirements.

Requirements: In addition to requiring traffic isolation between virtual networks, COCONUT requires that the TB bits are dedicated to COCONUT's operations, *i.e.,* no other entity (such as the tenants or the network hypervisor) is allowed to use these bits. For simpler presentation, we further assume that arbitrary bitmask (supported since OpenFlow 1.1, early 2011) is supported for the header-field used for TBs. COCONUT requires that the network hypervisor should not cause *ambiguity*, *i.e.,* it should not install multiple rules with overlapping match fields and identical priority on a switch. Moreover, assuming that by default rule priorities are integer values between 0 and max-priority, COCONUT requires the priorities of the physical rules that the network hypervisor sends to the controller to be integers between 0 and $\lfloor(\text{max-priority})/2\rfloor$, *i.e.,* COCONUT uses half the priority-space to "pre-install" rules to accelerate the update process without causing ambiguity. As we will see, for any rule P with priority x, the priority of the stable rule that COCONUT eventually installs is $2x$ and the priority of the pre-installed rules for P is $2x + 1$. This implies that for any two rules P and L, where x=*P.priority* and y=*L.priority*, if $y \geq x+1$, then L's priorities ($2y$ and $2y + 1$) will be strictly larger than P's priorities ($2x$ and $2x + 1$) throughout.

Algorithms: For updating a set of physical rules corresponding to a logical rule of a virtual network *v-net*, the network hypervisor sends a set called the *rule-batch*, the identifier of *v-net*, and the identifiers of the *v-net*'s shells to the controller (arguments of the `UPDATE` procedure in Algorithm 3). Each element of the rule-batch set, b, is a tuple that includes the new rule that needs to be installed *b.new-rule*, and the old rule that is being replaced, *b.old-rule*. Also, *rule-batch.new-rules* and *rule-batch.old-rules* show, respectively, the set of all new and old rules in the *rule-batch*. For any given physical rule, R, we denote the match, action, priority, and the switch hosting R by, respectively, *R.match*, *R.action*, *R.priority*, and *R.sw*. We show the action of installing a set of rules SR by `install(SR)`, the action of updating SR by overwriting value *val* on the *var* header field by `update(SR,var,val)`. For instance, updating the priority values of all rules in SR to 10 is shown by `update(SR,priority,10)`.

Algorithm 3 starts by installing a set of temporary rules T that are identical to the new rules, except: (1) they have higher priorities; (2) they match on an unused TB, tag=1, in addition to the rules' existing match requirements; and (3) the action sets tag=1 in addition to the rules' existing actions (line 9 in Algorithm 3). Note that a single tag bit is used for all rules in the batch. The temporary rules T will gradually be updated and eventually turn into the new rules. Initially, these rules are invisible because no transmitted packets have tag=1. But once packets do start using the new tag (*i.e.*, the rules' increased virtual clock value), the switches are prepared and thus will not have to pay the expensive [54] cost of relaying packets to the controller while the new rule is

Algorithm 3 Controller Update Algorithm

```
1:  procedure UPDATE(set rule-batch, set shells, id v-net)
2:      TB $tag$ := get-tag($v$-$net$)
3:      map $T$
4:      for $b \in batch$ do
5:          $T[b] := b.new\text{-}rule$
6:          $T[b].match := (T[b].mtach)\&(tag = 1)$
7:          $T[b].priority := 2 \times T[b].priority + 1$
8:          $T[b].action := (tag = 1)\&(T[b].action)$
9:      install($T$)
10:     wait-conf($T$); update($T, match, T.match\&(tag = *)$)
11:     wait-conf($T$); update($T, action, T.action\&(tag = *)$)
12:     delete($rule\text{-}batch.old\text{-}rule$)
13:     wait-conf($T$); stop-tagging($shells, tag$)
14:     wait-conf($rule\text{-}batch.old\text{-}rule$)
15:     update($T, priority, T.priority$-1)
16:     wait-conf($shells$); wait-conf($T$)
17:     release-tag($v\text{-}net, tag$)
```

"paged in". Specifically, since the rules have higher priority than the old rules, if a packet matches both a T and an old rule, the action of the new rule will be applied on it.

Once confirmations are received, the T rules are updated not to need the TB=1 for matching packets (line 10 in Algorithm 3). This makes the update visible as endpoints now can receive packets matched and handled by these rules. After receiving the confirmations (`wait-conf(T)`), every instance of the rule is ready to handle packets with or without TBs. So packets do not need to be marked any longer (line 11 in Algorithm 3) and the old rules can be deleted, since higher priority rules are already installed (line 12 in Algorithm 3).

After receiving confirmation that the old rules are deleted, the priorities of T rules are converted into the stable value (line 15 in Algorithm 3). Note that this operation turns the T rules into the stable new rules. Finally, once the controller receives the confirmations from the shells that they no longer tag packets with the TB and switches have installed the new non-tagging rules, it can release the tag for v-net after waiting for the *flush time*, the time for in-flight packets and the buffered packets (that might be tagged) to be delivered or expired and dropped (line 17 in Algorithm 3).

Algorithm 3 is for updating rules. Algorithms for deleting and adding new rules are similar: for **deleting** a set of rules DR, we set $rule\text{-}batch.new\text{-}rule$ and $rule\text{-}batch.old\text{-}rule$, respectively, to the set of rules that should match packets after DR's deletion, and DR. The deletion procedure is identical to the update procedure except for line 15 in Algorithm 3, where instead of updating the priorities of T, T is deleted since switches already host the rules that should match packets after DR is removed with their correct priorities. For **adding** a set of new rules, $rule\text{-}batch.old\text{-}rule$=∅, and the deletion of old rules (line 12 in Algorithm 3) and waiting for its confirmation should be skipped.

Shell operations are identical to the operations explained in §4.2 except that each shell i keeps a VC_i for the TB bits tag (and not all the logical rules), shown with $VC_i[tag]$, a timer associated with each TB bit, shown with $timer(tag)$. If the shell receives a `stop-tagging(TB tag)` command from the controller , it sets $VC_i[tag]$=0, resets tag's timer, *i.e.,* $timer(tag)$=0, and sends a confirmation to the controller . Shells honor the `stop-tagging(TB tag)` commands for the flushtime. If shell i receives a packet with tag=1 after the flush timer for tag has elapsed, it assumes it to be related to a different update batch and sets $VC_i[tag]$=1.

4.3.1 Handling Failures

We assume that different components of the system might experience **crash failure**, but not Byzantine failure. We further assume that each endpoint and its shell share fate, *i.e.,* they fail together. Switches and the controller are assumed to have reliable channels between them, similar to the main control channel in OpenFlow. Updates related to failed links are carried out similar to regular updates. Non-responsive switches (those not reacting to controller commands within a threshold) are assumed to have failed. When a switch fails, other switches and endpoints connected to it are populated with *detour* rules to reroute the traffic originally sent to the failed switch, and drop traffic they receive from it (*failover operations*). Dropping this traffic is essential for preserving safety; if the controller loses control over a switch, the switch's behavior, *e.g.,* its tagging operations, will be unknown. When a failed switch recovers, it communicates with the controller which populates it with correct version of rules (including the possible transient rules) before undoing the failover operations, *i.e.,* removing the rules that drop the traffic received from the failed switch from the network and endpoints as well as deleting the detour rules.

5. Evaluation of Prototype

We implemented a prototype of COCONUT (§5.1) and evaluated it in both a hardware SDN testbed and a Mininet emulation with multiple SDN applications and workloads (§5.2). We compared COCONUT's performance with a number of baselines: simple replication (SR), a strawman solution which provides strong consistency (SC), and CU. In summary, we found SC to be cost-prohibitive; even in modest-sized networks, it causes 12 Gbps bandwidth overhead and a 20× increase in user traffic latency, while COCONUT, CU, and SR[4] incur no measurable data plane performance overhead (§5.3). In terms of forwarding rule update delay and rule overhead, COCONUT has significantly lower overhead compared to SC (3.5× and 2× times lower respectively in a 80-switch network), and CU (1.5× and 245×, respectively). This overhead is only 1.2× and 1.3×, respectively, higher than SR (§5.4). Moreover, COCONUT's extra temporary rules are likely to be evacuated from the network faster

[4] Note that CU and SR do not guarantee observational correctness.

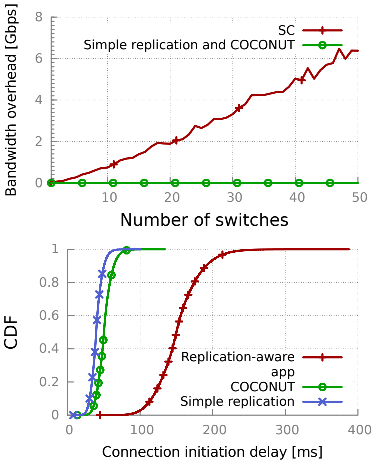


Figure 5: (a) SC causes significant bandwidth overhead. (b) Replication-aware app increases delay.

(§5.5). This result should be expected: switch update time is known to vary significantly [27, 29], with the *99th* percentile 10 times larger than the median in some cases [29]. Thus, by updating much fewer switches, COCONUT runs a lower risk of encountering stragglers. In some cases, the application developer can prevent replication-related race conditions by rewriting her applications to take the network replication into account. We show that in addition to offering programming simplicity, *i.e.,* enabling developers to use their applications "as-is", COCONUT's efficient logical clock-based approach of tracking causality results in 2.8× lower mean latency for user data flow initiation compared to this application-level approach (§5.3). We give more details about each of these conclusions next.

5.1 Prototype Implementation

Our COCONUT prototype consists of approximately 4K lines of Java and Python code and integrates a number of third party libraries and tools. We prototyped the controller using the Floodlight platform [2]. Floodlight runs a series of modules (*e.g.,* user applications), each supplied with mechanisms to control and query an SDN network. The COCONUT controller is implemented as a layer (which is itself a module) residing between the Floodlight Virtual Switch, a simple network virtualization developed as a Floodlight application, and the controller platform. Our prototype exposes much the same interface as the Floodlight platform. Hence, modules such as the virtualization applications that wish to be Floodlight clients simply use its interface instead. The COCONUT controller instruments the rules received from client modules and coordinates with shells to maintain correctness. We use OVS [50] to implement shells at the hosts with a bridge through which passes all traffic between the network and hosts.

5.2 Experimental Setup

Environment: For the physical network, we use the Ocean Cluster for Experimental Architectures in Networks (OCEAN) [1] which includes 13 Pica8 SDN Pronto 3290 switches, having a total of 676 switch ports. We "sliced" these ports to emulate fat-tree topologies with various sizes (up to 20 switches). To test COCONUT at scale, we also use the Mininet emulator [23] and simulated fat-tree [6] and VL2 [21] topologies with a few hundred switches in it. Switches' delays to apply and confirm application of updates (hereafter called *control delay*) are drawn from [26] in which the authors measure the performance of several commercial switches (HP Procurve, Fulcrum, and Quanta). We emulate the behavior of the HP Procurve switches in our Mininet experiments. We draw job allocation, flow interarrival times, and flow sizes from [12, 55].

Controller and Applications: We used two network virtualization platforms, OpenVirtex [7] and Pyretic [46], to create one-big-switch abstractions over physical fat-tree [6] and VL2 [21] networks of various sizes. Tenants of the network use several canonical applications to insert and update rules on their virtual one-big-switches. For OpenVirtex, the tenant runs the Floodlight controller [2] and its existing applications such as the learning switch and firewall, as well as the applications explained in §3. When these applications install, remove, or update a rule on the one-big-switch, OpenVirtex translates that to possibly multiple `FlowMod` messages and sends them to the physical network. For Pyretic, we use the parallel composition of the firewall and MAC learning implementations provided in [46]. The graphs in this section, unless stated otherwise, show the results for the ACL application running over an OpenVirtex's one-big-switches over fat-tree networks with parameter k={2,...,10}, *i.e.,* networks with (2 hosts, 5 switches), (16 hosts, 20 switches) ..., and (432 hosts, 180 switches), and the workload from [55]. Over these one-big-switches, the tenant's applications redirect a stream of traffic to a different host. These logical rules are then mapped to many physical rules: one rule for each port that connects to a host. We then update all those rules concurrently. Unless stated otherwise, we observe similar trends for other explained settings.

Scale-out schemes evaluated: COCONUT, SR, SC, CU. In addition to simple replication (SR) as a baseline, we use an implementation of Strong Consistency (SC) in SDNs [20]. For updating a rule, SC first installs temporary tunneling rules to direct all traffic that would be affected by the change to the controller (where it is handled by a single, strongly consistent, version of the logical rule), and from the controller to its destination. It then updates the rule at the controller; next it updates switches with the new rule and tears down the tunnels.

As another comparison point, we implemented a version of consistent updates (CU) which provides per-packet or per-flow consistency (§3.2). Of course, CU and COCONUT

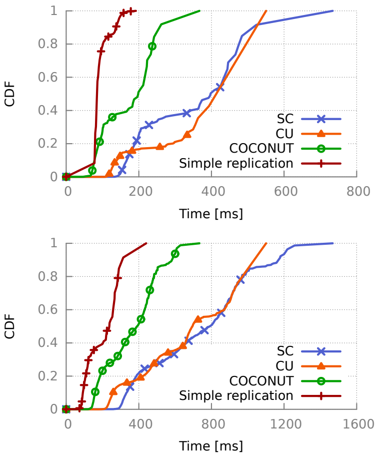


Figure 6: Testbed's (a) update initiation & (b) termination delays.

provide different correctness properties. The goal of this comparison is to evaluate whether COCONUT is expensive relative to the most powerful previously-studied notions of correctness[5]. Note that CU fundamentally operates at the granularity of a *flow*: (a subset of) traffic between ingress and egress switches [29]; it installs rules that are tagged to be specific to that flow. However, the abstraction we work with here operates on *forwarding rules* in a virtual network, and a single rule may apply to multiple flows. To translate CU to this rule-based abstraction, we implemented a module that duplicates rules so each flow using the rule has its own copy. It then runs CU to update each of those flows in parallel.

5.3 Data Plane Performance Impact

While the correctness problems discussed in §3 can be avoided via preserving strong consistency instead of weak causal consistency, doing so comes at a great data plane performance cost. Figure 5(a) shows the aggregate bandwidth overhead imposed by SC on the controller, already a bottleneck in SDNs [12, 24, 27, 37, 54], for the IDS example of §3.1. In addition to bandwidth overhead, this practice imposes added latency to flows. The overhead is prohibitive and rapidly increases with scale, *e.g.,* for networks of size 100, over 5,000 flows experience an average of 20× increase in latency due to an ACL rule update (not shown). Instead of redirecting all traffic affected by the change to the controller, one can alternatively redirect the traffic to a random switch. This approach, hereafter called *SC-switch*, initially installs temporary tunneling rules from a randomly selected switch to the destination of the affected traffic and then tunneling rules from switches to that switch where traffic will be processed with a single, strongly consistent, version of the logical rule. After updating the rule at this switch, SC-switch updates switches with the new rule and finally removes the tunnels. While reducing the overhead on the controller, redirecting traffic to a single switch may cause significant congestion at that switch, and setting up and deleting tunnels may increase its CPU overhead. Plus, compared to SC, SC-switch results in longer paths (and consequently more tunneling rules to setup and remove) because, unlike the controller in this experiment, the switch is not always directly connected to the destination of traffic. As a result, rule updates with SC-switch may degrade the performance of a significantly larger number of flows. The ACL rule update for a 100-switch network, for example, results in an average of 7× increase in latency for more than 83K flows traversing the randomly selected switch under SC-switch. SC and SC-switch, therefore, are not practical and scalable.

In some cases, the application developer can rewrite her applications to take the network replication into account. In the example of §3.2, for instance, if the firewall application developer is aware of the underlying replication, she could ensure correctness by preserving the orderings of installed rules on not just one switch but across *all* replicas. In particular, after receiving a packet from a client, the application could send the rules for allowing bidirectional communication to all replicas, followed by `BarrierRequests` (line 8 in Figure 4) and *wait* to receive the `BarrierReplies` from all replicas before releasing the packet. This approach, however, increases the delay of communication. With the previous experimental setup, for instance, more than half of the sessions experience an increase of 2.8× or higher in their connection initiation latency compared to COCONUT. Figure 5(b) shows the CDF of connection initiations' delays caused by this approach over 100 runs. In addition to the performance penalty, in this approach, the programmer needs to be aware of the underlying replication and rewrite her applications to account for it. Note that while COCONUT's delay is slightly higher than SR, unlike SR, it prevents incorrect blocking.

5.4 How Long Are Updates Delayed?

When network state changes, SC installs tunneling rules to and from the controller; COCONUT and CU start with a phase that installs some initially-invisible rules. These operations cause delay before the change starts to become visible to data traffic (*update initiation delay*), and before all switches have informed the controller their update is complete (what we call *update termination delay*).

For a given "target" rule R being updated, COCONUT and SC only install rules that are co-located with R (here, the edge rules produced by OpenVirtex). CU, in its standard implementation, updates all rules along the paths of flows passing through R. In our evaluation, as an optimization for CU, we limit this to flows that have active traffic.

First, we measure update delays on the testbed sliced to emulate a 20-switch fat-tree topology. Figure 6 shows that

[5] Recent works on optimizing CU require special rule-formats [29, 40, 43], *e.g.,* each rule is exact-match on a single flow [29]. Such assumptions usually hold in the network core as the rules that violate them are moved to the network edge [29, 51]. This makes them more suitable for flow-based traffic engineering for the network core. Thus, CU remains the most appropriate comparison for our setting.

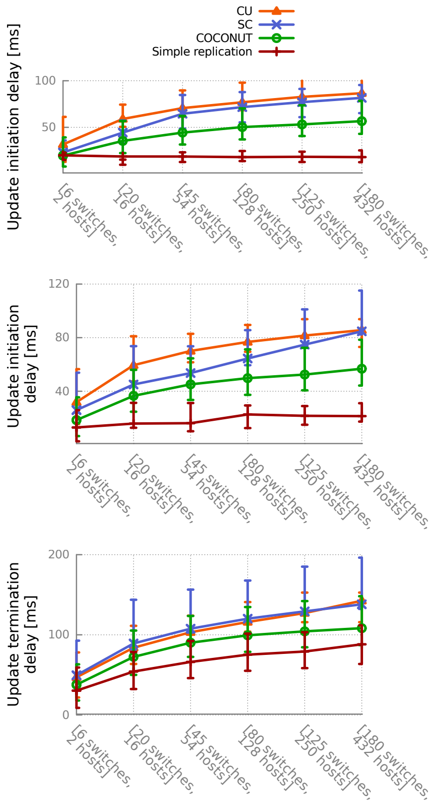


Figure 7: How long does it take to initiate and finish updates? Top: initiation delays for the firewall app; middle and bottom: initiation and termination delays for the IDS app.

while compared to SR, COCONUT increases the delay (*e.g.,* 1.4× increase in the median update termination time), it reduces the delay of SC and CU (2× and 1.8× lower median update termination delay, respectively). We use Mininet to measure this metric at scale and observe similar trends. Figure 7 shows mean values; error bars show 1st and 99th percentile over 100 runs. We observe similar trends for the IDS and firewall applications. Note that SC's cost rapidly increases with scale due to the overhead on the controller. Compared to SC, SC-switch results in higher costs, because in addition to the overhead on the switch, it needs to install and remove larger numbers of tunneling rules. For a 180-switch network, for instance, SC-switch is 47% slower than SC to finish an update (not shown).

The impact of the topology: In addition to the fat-tree networks, we experimented with the VL2 network [21] in Mininet. We found the number of edge switches, which are the switches that need to be updated, is the key player in COCONUT's speed, with little variation across these topology types. For example, the mean initiation delay for the IDS application was **47.8 ms** on a VL2 network with 25 edge switches (35 switches total and 500 hosts), which is very close to the delay on fat-tree networks of similar size: **45.1 ms** with 18 edge switches (45 switches total and 54 hosts) and **49.8 ms** with 32 edge switches (80 switches total and 128 hosts). Similarly, COCONUT's mean termination delays were 95.4 ms, 90.2 ms, and 99.3 ms on those three networks. SR and SC were similarly unaffected by the topology change, and CU worsened; we omit the results for brevity.

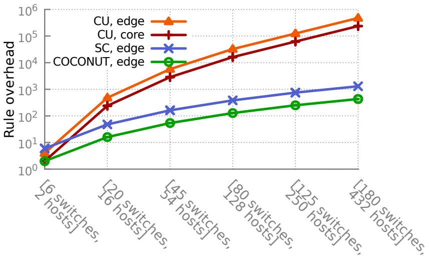


Figure 8: How much rule-overhead is imposed and where?

5.5 How Much Rule Overhead Is Imposed and Where?

COCONUT, SC, and CU all require installing some temporary extra rules. Since the number of rules switches can support is limited [32], it is important to keep this cost low. We measure the amount, location, and lifespan of this overhead.

By installing only one set of temporary rules, Ts, and morphing them into the final desired rules, COCONUT keeps the number of extra rules minimal. Plus, similar to SR and SC, COCONUT imposes this rule overhead only on the switches directly hosting the rules in the update batch. This implies that if COCONUT is used in conjunction with the common systems that place virtualized rules at the edge of the network [4, 37, 42], then only edge switches need to tolerate this overhead. In contrast, CU imposes this overhead on all the switches hosting the rules of the associated flows, possibly including core switches. Figure 8 shows the rule overhead (number of extra rules) and its location. Unlike CU, COCONUT and SC only have overhead at edge switches. Even for edge switches, COCONUT's overhead is singnificantly lower than SC's and CU's, *e.g.,* in a 80-switch network, respectively 2× and 245× lower.[6]

How Long Does Rule-overhead Persist? The extra rules installed by SC, CU, and COCONUT are supposed to be short-lived and all techniques remove those rules in their clean-up operations. Figure 9 shows that only 0.7% of COCONUT's rule overhead persists in the network for more than 100ms compared to 80.6% for SC and 60.7% for CU. This can again be explained by the fact that CU and SC update a significantly larger number of rules and impose a greater load on switches and controllers.

5.6 Can Header Bits Become a Scalability Bottleneck?

COCONUT's ability to handle concurrent updates is limited by the number of header bits available to it; if there are too many concurrent updates, COCONUT will have to queue the

[6] Note that we measure only CU's overhead *on top of* the rules we duplicated to move from a flow-based to a rule-based abstraction (§5.2).

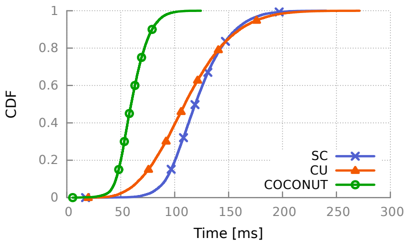


Figure 9: How long does the rule-overhead persist?

requests. With this in mind, can COCONUT handle modern network dynamics? A campus network may experience up to 18K updates per month [36], but the rate is significantly larger and more bursty in cloud environments where customers continuously deploy, delete, and migrate services, with an average of 12K updates per day in a typical cluster, peaking at one update per second [47]. To test COCONUT's rate of applying updates, we reserve 12 header bits (the number of bits of the VLAN tag, the header field reserved for the update operations in CU [53]), 19 header bits (the number of bits in one MPLS label), and 4 header bytes (the smallest possible option length in Geneve [22]) for COCONUT and modify the IDS application to send to COCONUT 12K update requests, equivalent to the average number of updates in *one day* in a cloud environment of [47]. We run this experiment on a fat-tree with 180 switches and measure the time COCONUT consumes to apply all the updates. Over 20 runs of this experiment, COCONUT applies these updates in respectively 2.4, 1.3, and 0.9 minutes on average. Its 90^{th} percentile update time is respectively, 2.6, 1.8, and 1.4 minutes, *i.e.,* more than 90% of the time its rate is $76\times, 112\times$, and $144\times$ faster than the peak update rate cited in [47]. Thus, we believe the existing header fields for carrying meta-data are more than sufficient for COCONUT's operations [47].

6. Related work

Sequence planning techniques synthesize an ordering of updates to preserve certain invariants (verified by verification tools [34, 35]) during updates [31]. Finding such orderings is NP-complete [31] and there does not always exist a sequence that preserves invariants such as loop freedom and congestion freedom [25, 53]. Thus, CU proposes an alternative approach for updating the network that guarantee to preserve trace properties [53]. CU formalized *trace properties* characterizing the paths individual packets take through the network, introduced per-packet consistency, and used a 2-phase update algorithm to implement it. As discussed previously, per-packet consistency does not preserve the weak causal correctness that is of interest to us (but also, COCONUT does not attempt to preserve per-packet consistency; to achieve these, the network provider could choose to run CU). While there are a few aspects of technical similarity in mechanism between CU and COCONUT (*e.g.,* version numbers and preloading initially invisible rules), COCONUT also has quite different mechanisms, in particular a vector of virtual clocks, each implemented as a single bit.

A few recent studies try to improve CU's efficiency, with various restrictions — either preserving narrower properties such as loopfreedom that are *subsets* of per-packet consistency or with constraints on forwarding rules [29, 33, 40, 43, 44]. None of those works provide guarantees that are stronger than CU's per-packet consistency.

The recently added atomic update operation of OVS and the bundle capability of OpenFlow [45] enable atomic update of a *single* switch and cannot be extended to multiple replicas at different locations. Session guarantees, originally developed for replicated storage [61], aim to present each *individual* application with a view of the logical element that is consistent with *its own* actions. Hence, they do not prevent the problems in §3 that happen because of the dependencies of actions of different users and applications. For the short-lived replication caused by migrations of middleboxes or virtual networks, OpenNF, LIME, and Split/Merge strive to retain strong consistency (SC) by heavy-weight operations[7] such as dropping packets or redirecting them to the controller [17, 20, 52]. In COCONUT, the dataplane continues processing packets during the update, *i.e.,* packets are not buffered (unlike [17, 52]), not redirected to the controller (unlike [17, 20, 39]) which is already a scalability bottleneck in SDNs [12, 24, 27, 37, 54], and not dropped (unlike [20, 52]).

Our earlier workshop paper observed the need for taking hosts' observations into account for defining correctness, but did not provide actual algorithms and systems to realize that vision [18].

7. Conclusion

We demonstrated that current network scale-out techniques do not preserve the semantics of the native network, leading to application-level incorrectness, and presented COCONUT, a system that solves this problem by preserving weak causal correctness. Some practical challenges remain, *e.g.,* requiring modification at endhost hypervisors. However, COCONUT appears to be surprisingly feasible, and represents a promising first step in an area that we believe will become increasingly important with roll-out of network virtualization and NFV.

Acknowledgments: We would like to thank our shepherd, Edouard Bugnion, Jennifer Rexford, and the reviewers for their feedback. This work was supported by a VMware Graduate Fellowship, by NSF CNS Award #1513906, and by the Maryland Procurement Office under Contract No. H98230-14-C-0141.

[7] OpenNF's implementation of SC, for instance, adds 10s of ms latency to each packet (*avg.* RTT<1ms in datacenters). The added latency rapidly increases with traffic rate and number of flows [17].

References

[1] Ocean Cluster for Experimental Architectures in Networks (OCEAN). http://ocean.cs.illinois.edu/.

[2] Project Floodlight. www.projectfloodlight.org/floodlight/.

[3] SciPass: IDS load balancer and science DMZ. https://github.com/GlobalNOC/SciPass/releases/tag/1.0.4.

[4] ONS 2014 Keynote: A. Greenberg, Microsoft Azure. http://www.youtube.com/watch?v=8Kyoj3bKepY, 2014.

[5] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *CCR*, 38(4):63–74, 2008.

[7] 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.

[8] J. Amann and R. Sommer. Providing dynamic control to passive network security monitoring. In *RAID*, 2015.

[9] J. Amann and R. Sommer. SDN based DDoS detection using SciPass and Bro. In *TNC*, 2015.

[10] S. Campbell and J. Lee. Intrusion detection at 100G. In *State of the Practice Reports*, 2011.

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

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

[13] Facebook Networking @Scale. SDN and NFV: Now for the enterprise community: Mark Russinovich, Microsoft Azure. https://www.youtube.com/watch?v=NVGeYDvoHQ8&feature=youtu.be, 2015.

[14] Facebook Networking @Scale. Synchronous geo-replication over Azure tables: A. Greenberg, Microsoft Azure. https://code.facebook.com/posts/1421954598097990/networking-scale-recap/, 2015.

[15] J. Fietz, S. Whitlock, G. Ioannidis, K. Argyraki, and E. Bugnion. VNToR: Network virtualization at the top-of-rack switch. In *SoCC*, 2016.

[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, 2011.

[17] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[18] S. Ghorbani and B. Godfrey. Towards correct network virtualization. In *HotSDN*, 2014.

[19] S. Ghorbani and P. B. Godfrey. COCONUT: Seamless replication of network elements. Technical report, 2017. http://webhost.engr.illinois.edu/~ghorban2/papers/coconut_tr.pdf.

[20] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker. Transparent, live migration of a software-defined network. In *SoCC*, 2014.

[21] 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.

[22] J. Gross, T. Sridhar, P. Garg, C. Wright, I. Ganga, P. Agarwal, K. Duda, D. Dutt, and J. Hudson. Geneve: Generic network virtualization encapsulation. *IETF draft*, 2014.

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

[24] K. He, J. Khalid, S. Das, A. Akella, E. L. Li, and M. Thottan. Mazu: Taming latency in software defined networks. Technical report, 2014. http://minds.wisconsin.edu/handle/1793/68830.

[25] 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.

[26] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, 2013.

[27] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[28] X. Jin, J. Gossels, J. Rexford, and D. Walker. Covisor: A compositional hypervisor for software-defined networks. In *NSDI*, 2015.

[29] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, 2014.

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

[31] B. G. Józsa and M. Makai. On the solution of reroute sequence planning problem in mpls networks. *Computer Networks*, 42(2):199–210, 2003.

[32] N. P. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite cacheflow in software-defined networks. In *HotSDN*, 2014.

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

[34] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[35] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[36] H. Kim, T. Benson, A. Akella, and N. Feamster. The evolution of network configuration: a tale of two campuses. In *IMC*, 2011.

[37] 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.

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

[39] W. Liu, R. B. Bobba, S. Mohan, and R. H. Campbell. Inter-flow consistency: Novel SDN update abstraction for supporting inter-flow constraints. In *SENT*, 2015.

[40] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. HotNets, 2014.

[41] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

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

[43] R. Mahajan and R. Wattenhofer. On consistent updates in software-defined networks. In *HotNets*, 2013.

[44] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. In *PLDI*, 2015.

[45] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *CCR*, 38(2), 2008.

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

[47] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. In *CCR*, volume 43, 2013.

[48] I. Pepelnjak. DefenseFlow NetFlow and SDN based DDoS attack defense. `http://www.radware.com/Products/DefenseFlow`.

[49] I. Pepelnjak. Real-life SDN/OpenFlow applications. `http://blog.ipspace.net/2013/06/real-life-sdnopenflow-applications.html`.

[50] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of Open vSwitch. In *NSDI*, 2015.

[51] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined internet architecture: decoupling architecture from infrastructure. In *HotSDN*, 2012.

[52] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

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

[54] Rob Sherwood. Modern OpenFlow and SDN. `http://bigswitch.com/blog/2014/06/02/modern-openflow-and-sdn-part-ii`, 2015.

[55] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[56] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing*, 7(3), 1994.

[57] SDN for the cloud. SIGCOMM 2015 Keynote: A. Greenberg, Microsoft Azure, 2015.

[58] A. Sharma. Bro: Actively defending so that you can do other stuff. In *BroCon*, 2014.

[59] N. Shelly, E. Jackson, T. Koponen, N. McKeown, and J. Rajahalme. Flow caching for high entropy packet fields. In *HotSDN*, 2014.

[60] 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.

[61] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

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

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