OpenNF: Enabling Innovation in Network Function Control

Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl,
Junaid Khalid, Sourav Das, and Aditya Akella
University of Wisconsin-Madison
{agember,raajay,cprakash,rgrandl,junaid,souravd,akella}@cs.wisc.edu
http://opennf.cs.wisc.edu

ABSTRACT

Network functions virtualization (NFV) together with software-defined networking (SDN) has the potential to help operators satisfy tight service level agreements, accurately monitor and manipulate network traffic, and minimize operating expenses. However, in scenarios that require packet processing to be redistributed across a collection of network function (NF) instances, simultaneously achieving all three goals requires a framework that provides efficient, coordinated control of both internal NF state and network forwarding state. To this end, we design a control plane called OpenNF. We use carefully designed APIs and a clever combination of events and forwarding updates to address race conditions, bound overhead, and accommodate a variety of NFs. Our evaluation shows that OpenNF offers efficient state control without compromising flexibility, and requires modest additions to NFs.

1. INTRODUCTION

Network functions (NFs), or middleboxes, are systems that examine and modify packets and flows in sophisticated ways: e.g., intrusion detection systems (IDSs), load balancers, caching proxies, etc. NFs play a critical role in ensuring security, improving performance, and providing other novel network functionality [38].

Recently, we have seen a growing interest in replacing dedicated NF hardware with software-based NFs running on generic compute resources—a trend known as network functions virtualization (NFV) [14]. In parallel, software-defined networking (SDN) is being used to steer flows through appropriate NFs to enforce policies and jointly manage network and NF load [19, 22, 24, 27, 33].

Together, NFV and SDN can enable an important class of management applications that need to *dynamically redistribute packet processing across multiple instances of an NF*—e.g., NF load balancing [33] and elastic NF scaling [23]. In the context of such applications, "NFV + SDN" can help achieve three important goals: (**1**) satisfy tight service level agreements (SLAs) on NF performance or availability; (**2**) accurately monitor and manipulate network traffic, e.g., an IDS should raise alerts for *all* flows containing known malware; and (**3**) minimize NF operating costs. However, simultaneously achieving all three goals is not possible today, and fundamentally requires more control than NFV + SDN can offer.

To see why, consider a scenario where an IDS is overloaded and must be scaled out in order to satisfy SLAs on throughput (Figure 1). With NFV we can easily launch a new IDS instance, and with SDN we can reroute some in-progress flows to the new instance [19, 33]. However, attacks may go undetected because the necessary internal NF state is unavailable at the new instance. To overcome this problem, an SDN control application can wait for existing flows to terminate and only reroute new flows [24, 39], but this delays the mitigation of overload and increases the likelihood of SLA violations. NF accuracy may also be impacted due to some NF-internal state not being copied or shared.

In this example, the only way to avoid a trade-off between NF accuracy and performance is to allow a control application to *quickly and safely move the internal IDS state for some flows* from the original instance to the new instance, and *update network forwarding state alongside*. Similar needs arise in the context of other applications that rely on dynamic reallocation of packet processing: e.g., rapid NF upgrades and dynamic invocation of remote processing.

In this paper, we present OpenNF, a control plane architecture that *provides efficient, coordinated control of both internal NF state and network forwarding state* to allow quick, safe, and fine-grained reallocation of flows across NF instances. Using OpenNF, operators can create rich control applications that redistribute processing to optimally meet their performance, availability, security and cost objectives, thus avoiding the need to make undesirable trade-offs.

We address three major challenges in designing OpenNF:

C1: Addressing race conditions. This is the most basic issue that arises when reallocating in-progress flows: When some internal NF state is being moved, packets may arrive at the source instance after the move starts, or at the destination instance before the state transfer finishes. Unless care is taken, updates to NF state due to such packets may either be lost or happen out of order, violating move safety. Similarly, when state is copied across NF instances, updates occurring contemporaneously may cause state to become inconsistent. Depending on the NF, these issues may hurt its accuracy.

To account for race conditions, we introduce two novel constructs: (1) an event abstraction to externally observe and prevent local state changes inside NFs, and (2) a clever two-phase scheme for updating network forwarding state. We show how to combine the two to provably ensure state updates are not lost or reordered during state moves and shared state remains consistent.

C2: Bounding overhead. The second issue is ensuring that reallocation can be efficient. Moving and sharing state between NF instances consumes both NF CPU and network resources. Moreover, avoiding loss, reordering, and state inconsistency requires packet buffering, which introduces both latency and memory overhead. If these performance and resource overheads are unbounded, then we cannot satisfy tight SLAs or constrain operating costs.

To bound overhead, we propose a flexible *northbound API* that control applications use to *precisely* specify which state to move, copy, or share, and which guarantees to enforce (e.g., loss-free).

C3: Accommodating a variety of NFs with minimal changes. The final issue is ensuring that our framework is capable of accommodating a wide range of NFs in a largely non-intrusive fashion. Providing APIs for NFs to create/update state [35] is one approach, but it restricts how internal NF state is structured and may not accommodate the state allocation/access needs of some packet pro-

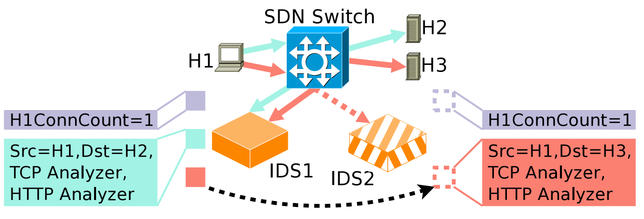


Figure 1: A scenario requiring scale-out and load balancing to satisfy SLAs on throughput are and minimize operating expenses. The IDS [32] processes a copy of network traffic to detect port scans and malware in HTTP flows. For each active flow, the IDS maintains a connection object with src/dst IPs, ports, etc. and several analyzer objects with protocol-specific state (e.g., current TCP seq # or partially reassembled HTTP payloads). It also maintains host-specific connection counters. If the red (darker) flow is reassigned to the second IDS instance to avoid SLA violations, then the SDN switch's flow table must be updated, the flow-specific state must be moved, and the host-specific state must be copied or shared to ensure no attacks go undetected.

cessing logic. Instead, we design a novel *southbound API* for NFs that allows a controller to request the export or import of NF state without changing how NFs internally manage state.

We have implemented our northbound API using Floodlight [7], and we have constructed several control applications that use this API. We have also augmented four NFs—Bro [32], Squid [17], iptables [10], and PRADS [15]—to support our southbound API (§7).

Our evaluation of OpenNF shows that: (1) OpenNF can eliminate spurious alerts and cut NF scale-in time by tens of minutes compared to using current control frameworks; (2) state can be moved, copied, and shared efficiently even when certain guarantees are requested—e.g., a loss-free move involving state for 500 flows takes only 215ms and imposes only 50ms of additional latency on packets received during the operation; and (3) additions to NFs to support OpenNF's southbound API increase code size by at most 9.8%, and packet processing time at NFs increases by less than 6% during state export or import.

2. WHY OpenNF?

When packet processing is being collectively handled by multiple instances of an NF, the NF deployment as a whole must typically meet three important goals: (1) satisfy tight NF service level agreements (SLAs) on performance or availability—e.g., aggregate throughput should exceed 1Gbps most of the time, and the time outdated/unpatched NFs are used to process flows should be less than 10 minutes per year; (2) accurately monitor and manipulate network traffic—e.g., an IDS should raise alerts for *all* HTTP flows containing known malware packages, and a redundancy elimination (RE) decoder should correctly restore redundancy removed by an RE encoder; and (3) operate with minimal cost—e.g., resources are shutdown when the extra capacity is not needed.

Simultaneously achieving all three goals is not possible today. In particular, we need additional control mechanisms, beyond those offered by combining NFV [14] and SDN [30]. Below, we describe several concrete examples and highlight how the aforementioned triumvirate of goals translate into control plane requirements. We also discuss how current NFV and SDN control frameworks, and simplistic enhancements to them, fall short in satisfying these needs.

2.1 Motivating Examples

Always up-to-date NFs. For maximum security, a cellular provider may want traffic to always be processed by the latest NF software. For example, an SLA may require that traffic is never processed by outdated NF instances for more than 10 minutes per year (goal #1). Fortunately, NFV allows us to launch an updated instance in a matter of milliseconds [29], and SDN allows us to reroute traffic to that instance just as quickly [19, 33]. However, this simple rerouting of traffic can compromise NF accuracy (goal #2) due to the absence of internal NF state at the new instance: e.g., rerouting active HTTP flows to a new IDS instance can cause the IDS to miss detecting some malware due to the lack of metadata for earlier packets in the flows. To overcome this issue, we can wait for existing flows to terminate and only reroute new flows [24, 39]. However, since flow durations are unbounded, this approach cannot guarantee the SLA will be satisfied: e.g., up to 40% of flows in cellular networks last longer than 10 minutes [37].[1] The only way to both satisfy the SLA and maintain NF accuracy is for the control plane to offer the ability to ***move NF state alongside updates to network forwarding state***. Furthermore, the ***operation must complete in bounded time***.

To guarantee NF accuracy (goal #2) during and after state transfer, it may be important that no packets or updates to state are lost and no re-ordering of updates happens. For example, IDS instances operating on a copy of traffic have no opportunity to request a packet retransmission if the copied traffic is dropped during state move; this can lead to missed alerts because only part of the data sent over a connection is checked for malware.[2] Likewise, the IDS may raise false alerts if it receives and processes SYN and data packets out of order. Thus, the control plane ***must offer support for key guarantees such as loss-freedom and order preservation.*** (We formally define *loss-freedom* and *order-preservation* in §5.1.)

High performance network monitoring. Performance is also a crucial concern for cellular providers. For example, an SLA may require NF deployment throughput to exceed 1Gbps most of the time. Meeting this SLA with a single NF instance can be challenging due to the complexity of packet processing. Fortunately, NFV enables NFs to be dynamically scaled-out as network load increases, and SDN enables flows to be rerouted to leverage the new capacity. However, as in the first scenario, flows must be rerouted *quickly*—waiting for flows to terminate can cause NF overload to persist and violate the SLA (goal #1)—and *safely*—rerouting flows without moving internal NF state (in a loss-free and order-preserving manner) can compromise NF accuracy (goal #2). Similarly, when network load decreases the NF should be scaled-in, with flows rerouted quickly and safely beforehand, to minimize operating costs (goal #3). To achieve this, we again need the ability to move NF state alongside updates to network forwarding state, and the move must occur within bounded time and with key guarantees.

When rebalancing load, we must also account for the fact that NFs may depend on state that applies to more than one flow: e.g., an IDS maintains connection counters for each end-host. If traffic is balanced at the granularity of hosts or subnets, all flows for a host will traverse the same IDS instance, and the counters can be moved to that instance. However, when flows involving the same host are balanced to different instances, both instances must have the relevant counters. Furthermore, if one instance is later terminated and flows for a given host are re-routed to the same remaining instance, the counters from both instances should be merged. Thus, the control plane must offer the ability to ***move, copy or share, and combine NF state that applies to multiple flows***.

Fast failure recovery with low resource footprint. When an NF instance fails, we can minimize downtime (goal #1) by rerouting

[1]Prematurely terminating flows also violates SLAs.

[2]***Is loss-free important given the network already can drop packets?*** Note that end points recover from network-induced drops using retransmissions, and the IDS can eventually get a copy; but the IDS can never recover packets dropped during state transfer. A similar argument applies to order-preserving.

in-progress (and new) flows to a non-failed instance. For these flows to be accurately processed (goal #2), critical NF state must be available at the selected instance. One way to fulfil this is to periodically create a backup of all NF state; this consumes non-negligible CPU and memory bandwidth at the NF (violating goal #3), and the delay between copies will result in the backup containing significant amounts of stale state. A second approach would be to back up pieces of NF state as they are updated. This eliminates the stale state problem, and the resource footprint is proportional to the frequency of state updates and the amount of state being backed up. To support this, we need the ability to copy NF state, as well as the ability to ***track when/how state is updated***.

Selectively invoking advanced remote processing. Based on preliminary observations made by a local NF, an enterprise may want to employ deeper and more advanced processing of a subset of in-progress flows (variant of goal #2). For example, when an IDS detects that internal hosts are making HTTP requests for a blacklisted domain, the enterprise invokes additional packet processing to have the corresponding replies analyzed for malware. Due to limited resources at the local IDS instance, the enterprise may leverage a more powerful remote cloud-resident IDS. Further, to avoid the cost of redirecting all traffic to the cloud (goal #3), traffic from the remaining hosts should continue to be processed locally. This requires the support highlighted in earlier examples (e.g., moving flow-specific state with a loss-free guarantee). Additionally, more advanced processing typically requires maintaining more detailed state: e.g., the cloud-resident IDS may create additional state for the new flows to compare signatures to a large corpus of known attacks. Thus, the NF control plane ***should not restrict an NF's ability to create additional state***. Further, ***it should automatically capture this additional state*** if the processing of the flow is later transferred back to the original NF instance.

2.2 Related Work

Existing NF control planes such as PLayer [27], SIMPLE [33], Stratos [23], FlowTags [22], and connection acrobatics [31] only provide control over, and coordination of, traffic forwarding. As already discussed, forwarding changes alone are insufficient to satisfy multiple objectives without degrading NF accuracy.

VM [20] or process replication [6] only allows cloning of NF instances in their entirety. The additional, unneeded state included in a clone not only wastes memory, but more crucially can cause undesirable NF behavior: e.g., an IDS may generate false alerts (we quantify this in §8.4). Moreover, this approach prevents state from multiple NF instances from being moved and merged, precluding, e.g., fast elastic scale-down.[3] Because of their intrinsic limitations, combining existing control planes with techniques for VM migration/process replication does not address the above requirements.

Vendor-supplied controllers [5, 16] that move, copy, and share NF state between multiple NF instances can leverage knowledge about the internal workings of NFs. However, they cannot control network state in a way that fully satisfies all goals—e.g., it is hard to provide optimized load balancing across network links.

Split/Merge [35] and Pico Replication [34] are the only systems that provide some control over both internal NF state and network state. They provide a shared library that NFs use to create, access, and modify internal state through pre-defined APIs. In Split/Merge, an orchestrator is responsible for coordinating load balancing by invoking a simple *migrate (f)* operation that reroutes flow *f* and moves corresponding NF state. In Pico Replication, modules are added to

[3]Basic scale-down can be supported by assigning new flows to the "combined" instance and waiting for flows at the "old" instance to terminate; but this can take a long time.

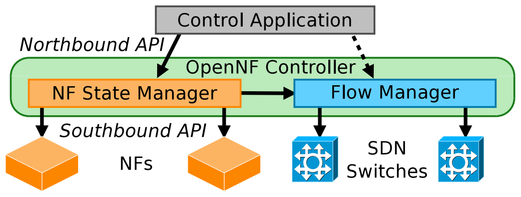


Figure 2: OpenNF architecture

an NF to manage the flow of packets in and out of each instance and to clone states at policy-defined frequencies.

Unfortunately, the migrate operation can cause lost or re-ordered NF state updates, since packets arriving at an NF instance after migrate is initiated are dropped, and a race exists between applying the network forwarding state update and resuming the flow of traffic (which is halted when migrate starts). Furthermore, the orchestrator and NF modules are targeted to specific problems, making them ill-suited to support other complex control applications. Finally, the API NFs must use to create and access states uses nondescript keys for non-flow-based state, making it difficult to know the exact states to move and copy when flows are rerouted, and the API only allows one state allocation per flow, requiring some internal NF state and packet processing logic to be significantly restructured. We discuss these issues in more detail later in the paper.

3. OpenNF OVERVIEW

OpenNF is a novel control plane architecture (Figure 2) that satisfies the aforementioned requirements and challenges. In this section, we outline our key ideas; §4 and §5 provide the details.

OpenNF allows control applications to closely manage the behavior and performance of NFs to satisfy high level objectives. Based on NF output or external input, control applications: (1) determine the precise sets of flows that specific NF instances should process, (2) direct the controller to provide the needed state at each instance, including both flow-specific state and state shared between flows, and (3) ask the controller to provide certain guarantees on state and state operations.

In turn, the OpenNF controller encapsulates the complexities of distributed state control and, when requested, guarantees loss-freedom, order-preservation, and consistency for state and state operations. We design two novel schemes to overcome underlying race conditions: (1) an *event abstraction* that the controller uses to closely observe updates to state, or to prevent updates but know what update was intended, and (2) a *two phase forwarding state update* scheme. Using just the former, the controller can ensure move operations are loss-free, and state copies are eventually consistent. By carefully sequencing state updates or update prevention (scheme 1) with the phases of scheme 2, the controller can ensure move operations are loss-free and order-preserving; we provide a formal proof in Appendix A. Lastly, by buffering events corresponding to intended updates and handling them one at a time in conjunction with piece-meal copying of state, the controller can ensure state copies are strongly or strictly consistent.

OpenNF's southbound API defines a standard NF interface for a controller to request events or the export or import of internal NF state. We *leave it to the NFs to furnish all state matching a filter* specified in an export call, and to determine how to merge existing state with state provided in an import call. This requires modest additions to NFs and, crucially, does not restrict, or require modifications to, the internal state data structures that NFs maintain. Furthermore, we use the well defined notion of a flow (e.g., TCP connection) as the basis for specifying which state to export

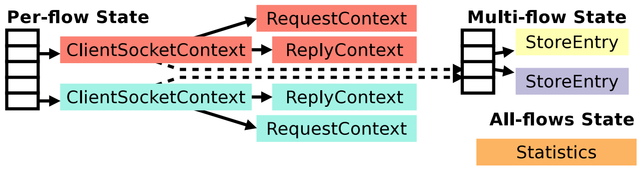


Figure 3: NF state taxonomy, with state from the Squid caching proxy as an example

and import. This naturally aligns with the way NFs already create, read, and update state.

4. SOUTHBOUND API

In this section, we describe the design of OpenNF's southbound API. To ensure a variety of NFs can be easily integrated into OpenNF, we must address two challenges: (1) account for the diversity of NF state and (2) minimize NF modifications.

4.1 State Taxonomy

To address the first challenge, we must identify commonalities in how internal state is allocated and accessed across various NFs. To this end, we examined several types of NFs from a variety of vendors, including: NATs [10], IDSs [32], load balancers [1, 8], caching proxies [17], WAN optimizers [18], and traffic monitors [13, 15].

We observe that *state created or updated by an NF while processing traffic applies to either an individual flow (e.g., TCP connection) or a collection of flows.* As shown in Figure 1, the Bro IDS maintains connection and analyzer objects for each TCP/UDP/ICMP flow and state for each host summarizing observations relating to all flows involving that host. Similarly, as shown in Figure 3, the Squid caching proxy maintains socket context, request context, and reply context for each client connection and cache entries for each requested web object. Most NFs also have state which is updated for every packet or flow the NF processes: e.g., statistics about the number of packets/flows the NF processed.[4]

Thus, as shown in Figure 3, we classify NF state based on *scope*, or how many flows an NF-created piece of state applies to—one flow (*per-flow*), multiple flows (*multi-flow*), or all flows (*all-flow*). In particular, per-flow state refers to structures/objects that are read or updated only when processing packets from the same flow (e.g., TCP connection), while multi-flow state is read or updated when processing packets from multiple, but not all, flows.

Thinking about each piece of NF-created state in terms of its association with flows provides a natural way for reasoning about how a control application should move/copy/share state. For example, a control application that routes all flows destined for a host H to a specific NF instance can assume the instance will need all per-flow state for flows destined for H and all multi-flow state which stores information related to one or more flows destined for H. This applies even in the case of seemingly non-flow-based state: e.g., the fingerprint table in a redundancy eliminator is classified as all-flows state, and cache entries in a Squid caching proxy are multi-flow state that can be referenced by client IP (to refer to cached objects actively being served), server IP, or URL.

Prior works on NF state management either draw no association between state and flows [26], or they do not distinguish between multi-flow and all-flows state [35]. This makes it difficult to know the exact set of state to move, copy, or share when flows are rerouted. For example, in the Squid caching proxy, cached web objects (multi-flow states) that are currently being sent to clients must be copied to avoid disrupting these in-progress connections, while other cached objects may or may not be copied depending on the SLAs a control application needs to satisfy (e.g., high cache hit ratio vs. fast scale out).[5] We quantitatively show the benefits of granular, flow-based control in §8.1.3.

We also discovered during our examination of NFs that they tend to: (1) allocate state at many points during flow processing—e.g., when the Bro IDS is monitoring for malware in HTTP sessions, it allocates state when the connection starts, as protocols are identified, and as HTTP reply data is received—and (2) organize/label state in many different ways—e.g., the Squid caching proxy organizes some state based on a traditional 5-tuple and some state based on a URL. Prior works [35] assume NFs allocate and organize state in particular ways (e.g., allocate state once for each flow), which means NFs may need significant changes to use these frameworks.

4.2 API to Export/Import State

We leverage our taxonomy to design a simple API for NFs to export and import pieces of state; it requires minimal NF modifications. In particular, we leverage the well defined notion of a flow (e.g., TCP or UDP connection) and our definition of state scope to allow a controller to specify exactly which state to export or import. State gathering and merging is delegated to NFs which perform these tasks within the context of their existing internal architecture.

For each scope we provide three simple functions: get, put, and delete. More formally, the functions are:

```
multimap<flowid, chunk> getPerflow(filter)
void putPerflow(multimap<flowid, chunk>)
void delPerflow(list<flowid>)
multimap<flowid, chunk> getMultiflow(filter)
void putMultiflow(multimap<flowid, chunk>)
void delMultiflow(list<flowid>)
list<chunk> getAllflows()
void putAllflows(list<chunk>)
```

A *filter* is a dictionary specifying values for one or more standard packet header fields (e.g., source/destination IP, network protocol, source/destination ports), similar to match criteria in OpenFlow [30].[6] This defines the set of flows whose state to get/put/delete. Header fields not specified are assumed to be wildcards. The `getAllflows` and `putAllflows` functions do not contain a *filter* because they refer to state that applies to all flows. Similarly, there is no `delAllflows` function because all-flows state is always relevant regardless of the traffic an NF is processing.

A *chunk* of state consists of one or more related internal NF structures, or objects, associated with the same flow (or set of flows): e.g., a chunk of per-flow state for the Bro IDS contains a `Conn` object and all per-flow objects it references (Figure 1). A corresponding *flowid* is provided for each chunk of per-flow and multi-flow state. The *flowid* is a dictionary of header fields and values that describe the exact flow (e.g., TCP or UDP connection) or set of flows (e.g., host or subnet) to which the state pertains. For example, a per-flow *chunk* from the Bro IDS has a *flowid* that includes the source and destination IPs, ports, and transport protocol, while a multi-flow *chunk* containing a counter for an end-host has a *flowid* that only includes the host's IP.

When `getPerflow` or `getMultiflow` is called, the NF is responsible for identifying and providing all per-flow or multi-flow

[4]NFs also have configuration state. It is read but never updated by NFs, making it easy to handle; we ignore the details in this paper.

[5]NF-specific state sharing features, such as inter-cache protocols in Squid, can also be leveraged, but they do not avoid the need for per-flow state, and some multi-flow state, to be moved or copied.

[6]Some NFs may also support extended *filter*s and *flowid*s that include header fields for other common protocols: e.g., the Squid caching proxy may include the HTTP URL.

state that pertains to flows matching the *filter*. Crucially, *only fields relevant to the state are matched against the filter*; other fields in the *filter* are ignored: e.g., in the Bro IDS, only the IP fields in a *filter* will be considered when determining which end-host connection counters to return. This API design avoids the need for a control application to be aware of the way an NF internally organizes state. Additionally, by identifying and exporting state on-demand, we avoid the need to change an NF's architecture to conform to a specific memory allocation strategy [35].

The NF is also responsible for replacing or combining existing state for a given flow (or set of flows) with state provided in an invocation of `putPerflow` (or `putMultiflow`). Common methods of combining state include adding or averaging values (for counters), selecting the greatest or least value (for timestamps), and calculating the union or intersection of sets (for lists of addresses or ports). State merging must be implemented by individual NFs because the diversity of internal state structures makes it prohibitive to provide a generic solution.

4.3 API to Observe/Prevent State Updates

The API described above does not interpose on internal state creations and accesses. However, there are times when we need to prevent an NF instance from updating state—e.g., while state is being moved—or we want to know updates are happening—e.g., to determine when to copy state.

OpenNF uses two mechanisms to prevent and observe updates: (1) having NFs generate packet-received events for certain packets—the controller tells the NF which subset of packets should trigger events—and (2) controlling how NFs should act on the packets that generate events—process, buffer, or drop them.

Specifically, we add the following functions to the API:

`void enableEvents(`*filter*`,`*action*`)`
`void disableEvents(`*filter*`)`

The *filter* defines the set of packets that should trigger events; it has the same format as described in §4.2. The *action* may be `process`, `buffer`, or `drop`; any buffered packets are released to the NF for processing when events are disabled. The events themselves contain a copy of the triggering packet.

In the next section, we discuss how events are used to realize important guarantees on state and state operations.

5. NORTHBOUND API

OpenNF's northbound API allows control applications to flexibly move, copy, or share subsets of state between NF instances, and to request important guarantees, including loss-freedom, order-preservation, and various forms of consistency. This API design appropriately balances OpenNF's generality and complexity: Not offering some guarantees would reduce complexity but make OpenNF insufficient for use with many NFs—e.g., a redundancy eliminator [18] will incorrectly reconstruct packets when re-ordering occurs (§5.1.2). Similarly, always enforcing the strongest guarantees would simplify the API but make OpenNF insufficient for scenarios with tight SLAs—e.g., a loss-free and order-preserving move is unnecessary for a NAT, and the latency increase imposed by these guarantees (§8.1) could cripple VoIP sessions.

The main challenge in supporting this API is designing suitable, low-overhead mechanisms to provide the necessary guarantees. In this section, we show how we use events together with fine-grained control over network forwarding to overcome this challenge. We first describe how we provide a loss-free and order-preserving `move` operation (we provide a formal proof of these guarantees in Appendix A), and what optimizations we use to improve efficiency. We then describe how OpenNF's `copy` and `share` operations provide eventual, strong, or strict consistency for state required by multiple NF instances.

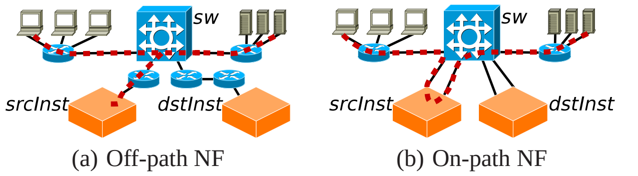


Figure 4: Assumed topologies for move operation

5.1 Move Operation

OpenNF's `move` operation transfers both the state *and* input (i.e., traffic) for a set of flows from one NF instance (*srcInst*) to another (*dstInst*). Its syntax is:

`move(`*srcInst*`,`*dstInst*`,`*filter*`,`*scope*`,`*properties*`)`

As in the southbound API, the set of flows is defined by *filter*; a single flow is the finest granularity at which a move can occur. The *scope* argument specifies which class(es) of state (per-flow and/or multi-flow) to move, and the *properties* argument defines whether the move should be loss-free (§5.1.1) and order-preserving (§5.1.2).

In what follows, *sw* denotes the last SDN switch through which all packets matching *filter* will pass before diverging on their paths to reach *srcInst* and *dstInst* (Figure 4). We assume the SDN controller keeps track of *sw*. We also initially assume that loss and reordering do not occur on the network paths from *sw* to *srcInst* or *sw* to *dstInst*; stronger versions of loss-free and order-preserving move that do not rely on this assumption are described in §5.1.3.

For a move without guarantees, the controller (1) calls `getPerflow` and `delPerflow` on *srcInst*, (2) calls `putPerflow` on *dstInst*, and (3) updates the flow table on *sw* to forward the affected flows to *dstInst*. To move multi-flow state as well (or instead), the analogous multi-flow functions are also (instead) called. For the rest of this section, we assume the *scope* is per-flow, but our ideas can easily be extended to multi-flow state.

With the above sequence of steps, packets corresponding to the state being moved may continue to arrive at *srcInst* from the start of `getPerflow` until after the forwarding change at *sw* takes effect and all packets in transit to *srcInst* have arrived and been read from the NIC and operating system buffers. A simple approach of dropping these packets when *srcInst* receives them [35] prevents *srcInst* from establishing new state for the flows or failing due to missing state. But this is only acceptable in scenarios where an application is willing to tolerate the effects of skipped processing: e.g., scan detection in the Bro IDS will still function if some TCP packets are not processed, but it may take longer to detect scans. Alternatively, an NF may be on the forwarding path between flow endpoints (Figure 4(b)), e.g., a Squid caching proxy, in which case dropped TCP packets will be retransmitted, although throughput will be reduced.

5.1.1 Loss-free Move

In some situations loss is problematic: e.g., the Bro IDS's malware detection script will compute incorrect md5sums and fail to detect malicious content if part of an HTTP reply is missing; we quantify this in §8.1.2. Thus, we need a move operation that satisfies the following property:

> **Loss-free:** *All state updates resulting from packet processing should be reflected at the destination instance, and all packets the switch receives should be processed.*

The first half of this property is important for ensuring all information pertaining to a flow (or group of flows) is available at the instance where subsequent packet processing for the flow(s) will occur, and that information is not left, or discarded, at the original instance. The latter half ensures an NF does not miss gathering important information about a flow.

In an attempt to be loss-free, Split/Merge halts, and buffers at the controller, all traffic arriving at *sw* while migrating per-flow state [35]. However, when traffic is halted, packets may already be in-transit to *srcInst*, or sitting in NIC or operating system queues at *srcInst*. Split/Merge drops these packets when they (arrive and) are dequeued at *srcInst*. This ensures that *srcInst* does not attempt to update (or create new) per-flow state after the transfer of state has started, guaranteeing the first half of our loss-free property. However, dropping packets at *srcInst* violates the latter half. While we could modify Split/Merge to delay state transfer until packets have drained from the network and local queues, it is impossible to know how long to wait, and extra waiting increases the delay imposed on packets buffered at the controller.

SDN consistency abstractions [28, 36] are also insufficient for guaranteeing loss-freedom. They can guarantee packets will be forwarded to *srcInst* or *dstInst*, but they do not provide any guarantees on what happens to the packets once they arrive at the NF instances. If *srcInst* processes the packets after state transfer has started, then the state installed at *dstInst* will not include some updates; if *srcInst* drops the packets instead, then some state updates will never occur.

What then should we do to ensure loss-freedom in the face of packets that are in-transit (or buffered) when the move operation starts? In OpenNF, we leverage events raised by NFs. Specifically, the controller calls `enableEvents(`*filter*`,drop)` on *srcInst* before calling `getPerflow`. This causes *srcInst* to raise an event for each received packet matching *filter*. The events are buffered at the controller until the `putPerflow` call on *dstInst* completes. Then, the packet in each buffered event is sent to *sw* to be forwarded to *dstInst*; any events arriving at the controller after the buffer has been emptied are handled immediately in the same way. Lastly, the flow table on *sw* is updated to forward the affected flows to *dstInst*.

Calling `disableEvents(`*filter*`)` on *srcInst* is unnecessary, because packets matching *filter* will eventually stop arriving at *srcInst* and no more events will be generated. Nonetheless, to eliminate the need for *srcInst* to check if it should raised events for incoming packets, the controller can issue this call after several minutes—i.e., after all packets matching *filter* have likely arrived or timed out.

5.1.2 Order-preserving Move

In addition to loss, NFs can be negatively affected by re-ordering. For example, the "weird activity" policy script included with the Bro IDS will raise a false "SYN_inside_connection" alert if the IDS receives and processes SYN and data packets in a different order than they were actually exchanged by the connection endpoints. Another example is a redundancy elimination decoder [18] where an encoded packet arriving before the data packet w.r.t. which it was encoded will be silently dropped; this can cause the decoder's data store to rapidly become out of synch with the encoders.

Thus, we need a move operation that satisfies the following:

> **Order-preserving:** *All packets should be processed in the order they were forwarded to the NF instances by the switch.*

This property applies within one direction of a flow (e.g., process SYN before ACK), across both directions of a flow[7] (e.g., process SYN before SYN+ACK), and, for moves including multi-flow state, across flows (e.g., process an FTP get command before the SYN for the new transfer connection).

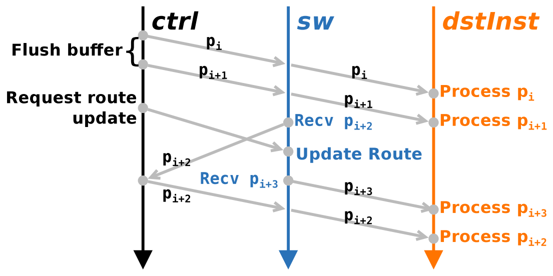


Figure 5: Order-preserving problem in Split/Merge

Unfortunately, neither Split/Merge nor the loss-free move described above are order-preserving. The basic problem in both systems is a race between flushing packets buffered at the controller and changing the flow table at *sw* to forward all packets to *dstInst*. Figure 5 illustrates the problem in the context of Split/Merge. Even if all buffered packets (p_i and p_{i+1}) are flushed before the controller requests a forwarding table update at *sw*, another packet (p_{i+2}) may arrive at *sw* and be forwarded to the controller before *sw* applies the forwarding table update. Once the update is applied, *sw* may start forwarding packets (p_{i+3}) to *dstInst*, but the controller may not have received the packet p_{i+2} from *sw*. Thus, the packet p_{i+2} will be forwarded to *dstInst* after a later packet of the flow (p_{i+3}) has already been forwarded to *dstInst*.

We use a clever combination of events and a two-phase forwarding state update to guarantee a loss-free *and* order-preserving move. Figure 6 has psuedo-code for the steps.

```
1  eventReceivedFromSrcInst (event)
2      if shouldBufferEvents then
3          eventQueue.enqueue (event.packet)
4      else
5          sw.forward (event.packet, dstInst)

6  packetReceivedFromSw (packet)
7      if lastPacketFromSw== null then
8          signal (GOT_FIRST_PKT_FROM_SW)   // wait @ 24
9      lastPacketFromSw ← packet

10 eventReceivedFromDstInst (event)
11     if event.packet == lastPacketFromSw then
12         signal (DST_PROCESSED_LAST_PKT)  // wait @ 26

13 moveLossfreeOrderpreserve (srcInst, dstInst, filter)
14     shouldBufferEvents ← true
15     srcInst.enableEvents (filter, DROP)
16     chunks ← srcInst.getPerflow (filter)
17     srcInst.delPerflow (chunks.keys)
18     dstInst.putPerflow (chunks)
19     foreach event in eventQueue do
20         sw.forward (event.packet, dstInst)
21     shouldBufferEvents ← false
22     dstInst.enableEvents (filter, BUFFER)
23     sw.install (filter, {srcInst, ctrl}, LOW_PRIORITY)
24     wait (GOT_FIRST_PKT_FROM_SW)
25     sw.install (filter, dstInst, HIGH_PRIORITY)
26     wait (DST_PROCESSED_LAST_PKT)
27     dstInst.disableEvents (filter)
```

Figure 6: Pseudo-code for loss-free and order-preserving move

We start with the steps used for a loss-free move, through calling `putPerflow` on *dstInst*. After `putPerflow` completes we extract the packet from each buffered event, mark it with a special

[7]If packets in opposite directions do not traverse a common switch before reaching the NF—e.g., a NAT is placed between two switches—then we lack a vantage point to know the total order of packets across directions, and we cannot guarantee such an order unless it is enforced by a flow's end-points—e.g., a server will not send SYN+ACK until the NAT forwards the SYN from a client.

"do-not-buffer" flag, and send it to *sw* to be forwarded to *dstInst*; any events arriving at the controller after the buffer has been emptied are handled immediately in the same way. Then, we call `enableEvents(`*filter*`,buffer)` on *dstInst*, so that any packets forwarded directly to *dstInst* by *sw* will be buffered; note that the packets marked with "do-not-buffer" (discussed above) are not buffered.

Next, we perform the two phase forwarding state update. First, we update the forwarding entry for *filter* on *sw* to forward matching packets to both *srcInst* and the controller.[8] The controller waits for at least one packet from *sw*, and always stores the most recent packet it receives. Second, we install a higher priority forwarding entry for *filter* on *sw* to forward matching packets to *dstInst*. Through this two phase update, the controller can become aware of the *last* packet sent to *srcInst*.[9]

Finally, we need to ensure that *dstInst* processes all packets forwarded to *srcInst* before processing any packets that *sw* directly forwards to *dstInst*. We achieve this with the following sequence of steps: (1) wait for an event from *srcInst* for the last packet sent to *srcInst*—this is the packet we stored during the two phase forwarding state update; (2) send the packet contained in the event to *sw* to forward to *dstInst*; (3) wait for an event from *dstInst* for the packet; and (4) call `disableEvents(`*filter*`)` on *dstInst* to release any packets that had already been sent to *dstInst* by *sw* and were buffered at *dstInst*.

The additional waiting required for order-preserving does come at a performance cost (we quantify this in §8.1.1). Thus, we offer applications three versions of move (loss-free and order-preserving, loss-free only, and no guarantees) so they can select the most efficient version that satisfies their requirements.

5.1.3 Guarantees with Lossy Network Paths

While the above mechanisms guarantee a loss-free and order-preserving move when no loss or reordering occurs on the paths from *sw* to *srcInst* and *sw* to *dstInst* (proof in Appendix A), these properties are not guaranteed with lossy network paths. If the path from *sw* to *dstInst* is lossy, a packet from *srcInst* sent by the controller to *sw* to be forwarded to *dstInst* could be dropped after it is forwarded by *sw*.[10] Packets could also be re-ordered after being forwarded by *sw*. If the path from *sw* to *srcInst* is lossy, a packet forwarded by *sw* after the first phase route update would reach the controller but might not reach *srcInst*; this means *srcInst* would never raise an event for the packet and the packet would never be sent to *dstInst*. Hence, the controller would wait indefinitely for an event from *dstInst* for the packet (line 26 in Figure 6).

To cope with loss and re-ordering on the path from *sw* to *dstInst*, we simply need to send packets from events directly from the controller to *dstInst* via a TCP-based control channel, rather than sending the packets to *sw* to be forwarded to *dstInst*. This change allows us to satisfy our loss-free property.

To satisfy our order-preserving property, we also need to cope with loss on the *sw* to *srcInst* path. Our solution is to skip the first phase forwarding update and use special "tracer" packets to determine when the last "regular" packet has been processed by *srcInst*. The revised steps are shown in Figure 7. Lines 13 to 22 are the same as the steps in Figure 6. After enabling buffering on *dstInst*, we update the forwarding entry for *filter* on *sw* to forward matching packets to *dstInst*. We then send a tracer packet to *sw* to be forwarded to *srcInst*. Since we have already re-routed the relevant traffic to *dstInst*, the tracer packet should be the last packet to arrive at *srcInst*. Thus, when we get an event from *srcInst* containing the tracer packet, we know that *srcInst* has processed the last regular packet (lines 1-3); this last regular packet should have previously been stored (line 5). Lastly, we wait (line 26) for an event from *dstInst* that it has received and processed the last regular packet (lines 10-12) before releasing the buffered packets on *dstInst*[11] (line 28). Since the tracer packet could be lost on the path from *sw* to *srcInst*, we wait with a timeout and retransmit the tracer packet if necessary (lines 24-27).

```
 1 eventReceivedFromSrcInst (event)
 2     if event.packet == tracerPkt then
 3         srcProcessedLastPkt ← true
 4     else
 5         lastPktFromSrc ← event.packet
 6         if shouldBufferEvents then
 7             eventQueue.enqueue (event.packet)
 8         else
 9             sw.forward (event.packet, dstInst)
10 eventReceivedFromDstInst (event)
11     if event.packet == lastPktFromSrc ∧ srcProcessedLastPkt then
12         signal (DST_PROCESSED_LAST_PKT) // wait @ 26
13 moveLossfreeOrderpreserve (srcInst, dstInst, filter)
14     shouldBufferEvents ← true
15     srcInst.enableEvents (filter, DROP)
16     chunks ← srcInst.getPerflow (filter)
17     srcInst.delPerflow (chunks.keys)
18     dstInst.putPerflow (chunks)
19     foreach event in eventQueue do
20         sw.forward (event.packet, dstInst)
21     shouldBufferEvents ← false
22     dstInst.enableEvents (filter, BUFFER)
23     sw.install (filter, dstInst)
24     repeat
25         sw.forward (tracerPkt, srcInst)
26         signaled ← wait (DST_PROCESSED_LAST_PKT,
           TIMEOUT)
27     until signaled
28     dstInst.disableEvents (filter)
```

Figure 7: Pseudo-code for loss-free and order-preserving move when network paths are lossy

In Appendix A, we formally prove that this sequence of steps is loss-free and order-preserving even when network paths are lossy. Providing these guarantees when there is reordering on the path from *sw* to *srcInst* remains an open problem.

5.1.4 Optimizations

Supporting the above guarantees may impose additional latencies on packets arriving during the move operation. In particular, when a move involves multiple flows, we halt the processing of those flows' packets from the time `enableEvents` is called until after `putPerflow` completes.

One way to reduce these latencies (and reduce drops in the case of a move without guarantees) is to reduce the total time taken to complete the move operation. To achieve this, an application could issue multiple pipelined moves that each cover a smaller portion of the flow space. However, this requires more forwarding rules in *sw* and requires the application to know how flows are divided among the flow space. Instead, we can leverage the fact that `getPerflow` and `putPerflow` operations can be, at least partially, ex-

[8]We use existing SDN consistency mechanisms [28, 36] to ensure the update is atomic and no packets are lost.

[9]The controller can check the counters on the first flow entry in *sw* against the number of packets it has received from *sw* to ensure the packet it currently has stored is in fact the last packet.

[10]The packet will not be dropped between the controller and *sw* because the packet is sent to *sw* over a TCP-based control channel.

[11]Packets arriving at *dstInst* continue to be buffered until the buffer has been emptied.

ecuted *in parallel*. Rather than returning all requested states as a single result, the *srcInst* can return each *chunk* of per-flow state immediately, and the controller can immediately call `putPerflow` with just that *chunk*. The forwarding table update(s) at *sw* occurs after the `getPerflow` and all `putPerflow` calls have returned.

The additional latency imposed on redirected packets can be further reduced by following an *early release and late locking strategy*. For late-locking, the controller calls `getPerflow` on *srcInst* with a special flag instructing *srcInst* to enable events for each flow just before the corresponding per-flow state is prepared for export (avoiding the need to call `enableEvents` for all flows beforehand). Also, once `putPerflow` for a specific *chunk* returns, the controller can release any events pertaining to that *chunk*.[12]

The *parallelizing* optimization can be applied to any version of move, and the *early-release* optimization can be applied to a move of either per-flow or multi-flow state, but not a move involving both.

5.2 Copy and Share Operations

OpenNF's `copy` and `share` operations address applications' need for the same state to be readable and/or updateable at multiple NF instances and, potentially, for updates made at one instance to be reflected elsewhere. For example, in a failure recovery application (§2) a backup NF instance needs to keep an updated copy of all per-/multi-/all-flows state. Similarly, a load balancing application that distributes an end-host's flows among multiple IDS instances needs updates to the host connection counter at one instance to be reflected at the other instances to effectively detect port scans.

In particular, `copy` can be used when state consistency is *not required* or *eventual* consistency is desired, while `share` can be used when *strong* or *strict* consistency is desired. Note that eventual consistency is akin to extending our loss-free property to multiple copies of state, while strict consistency is akin to extending both our loss-free and order-preserving properties to multiple NF instances.

5.2.1 Copy Operation

OpenNF's copy operation clones state from one NF instance (*srcInst*) to another (*dstInst*). Its syntax is:

`copy(`*srcInst*`,`*dstInst*`,`*filter*`,`*scope*`)`

The *filter* argument specifies the set of flows whose state to copy, while the *scope* argument specifies which class(es) of state (per-flow, multi-flow, and/or all-flows) to copy.

The `copy` operation is implemented using the get and put calls from the southbound API (§4.2). No change in forwarding state occurs as part of copy because state is not deleted from *srcInst*, allowing *srcInst* to continue processing traffic and updating its copy of state. It is up to control applications to separately initiate a change in forwarding state where the situation warrants (e.g., by directly interacting with the SDN controller, or calling `move` for some other class of state).

Eventual consistency can be achieved by occasionally re-copying the same set of state. As described in §4.2, an NF will automatically replace or combine the new and existing copies when `putPerflow`, `putMultiflow`, and `putAllflows` are called. Since there are many possible ways to decide when state should be re-copied—based on time, NF output, updates to NF state, or other external factors—we leave it to applications to issue subsequent copy calls. As a convenience, we do provide a function for control applications to become *aware* of state updates:

`void notify(`*filter*`,`*inst*`,`*enable*`,`*callback*`)`

[12]Although state chunks get transferred and events get processed via the controller in our current system, they can also happen peer to peer.

When invoked with *enable* set to true, the controller calls `enableEvents(`*filter*`, process)` on NF instance *inst*, otherwise it calls `disableEvents(`*filter*`)` on *inst*. For each event the controller receives, it invokes the provided *callback* function.

5.2.2 Share Operation

Strong and strict consistency are more difficult to achieve because state reads and updates must occur at each NF instance in the same global order. For strict consistency this global order must match the order in which packets are received by *sw*. For strong consistency the global order may differ from the order in which packets were received by *sw*, but updates for packets received by a specific NF instance must occur in the global order in the order the instance received the packets.

Both cases require synchronizing reads/updates across all NF instances (`list<`*inst*`>`) that are using a given piece of state. OpenNF's `share` operation provides this:

`void share(list<`*inst*`>,`*filter*`,`*scope*`,`*consistency*`)`

The *filter* and *scope* arguments are the same as above, while *consistency* is set to `strong` or `strict`.

Events can again be used to keep state strongly consistent. The controller calls `enableEvents(`*filter*`,drop)` on each instance, followed by a sequence of get and put calls to initially synchronize their state. When events arrive at the controller, they are placed in a FIFO queue labeled with the *flowid* for the flow group to which they pertain; flows are grouped based on the coarsest granularity of state being shared (e.g., per-host or per-prefix).

For each queue, one event at a time is dequeued, and the packet it contains is marked with a "do-not-drop" flag and forwarded to the originating NF instance. The NF instance processes the packet and raises an event, which signals to the controller that all state reads/updates at the NF are complete. The controller then calls `getMultiflow` (or `getPerflow`, `getAllflows`) on the originating NF instance, followed by `putMultiflow` (or `putPerflow`, `putAllflows`) on all other instances in `list<`*inst*`>`. Then, the next event is dequeued and the process repeated.

Since events from different NFs may arrive at the controller in a different order than packets were received by *sw*, we require a slightly different approach for strict consistency. The controller must receive packets directly from the switch to know the global order in which packets should be processed. We therefore update all relevant forwarding entries in *sw*—i.e., entries that both cover a portion of the flow space covered by *filter* and forward to an instance in `list<`*inst*`>`—to forward to the controller instead. We then employ the same methodology as above, except we invoke `enableEvents` with *action* set to `process` and queue packets received from *sw* rather than receiving packets via events.

It is up to control applications to determine the appropriate consistency requirements for the situation, recognizing that strong or strict consistency comes at a significant performance cost (§8.1.1). Applications should also consider which multi-/all-flows state is required for accurate packet processing, and, generally, invoke copy or share operations on this state prior to moving per-flow state.

6. CONTROL APPLICATIONS

Using OpenNF, we have written control applications for several of the scenarios described in §2. The applications are designed for the environment shown in Figure 8. In all applications, we use the Bro IDS, but different applications place different requirements on both the granularities of state operations and the guarantees needed; despite these differences, the applications are relatively simple to implement. We describe them below.

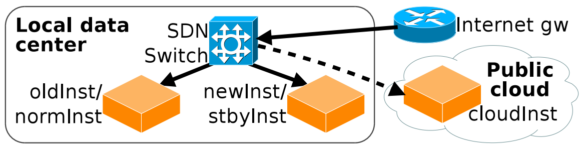


Figure 8: The Bro IDS runs on VMs in both a local data center and a public cloud. An SDN switch in the local data center receives a copy of all traffic from the Internet gateway for the local network and routes it to an IDS instance. The local IDS instances monitor for port scans and HTTP requests from outdated web browsers. The cloud instances additionally check for malware in HTTP replies.

```
movePrefix (prefix, oldInst, newInst)
    copy (oldInst, newInst, {nw_src: prefix}, MULTI)
    move (oldInst, newInst, {nw_src: prefix}, PER, LOSSFREE)
    while true do
        sleep (60)
        copy (oldInst, newInst, {nw_src: prefix}, MULTI)
        copy (newInst, oldInst, {nw_src: prefix}, MULTI)
```

Figure 9: Load balanced network monitoring application

High performance network monitoring. The first application (Figure 9) monitors the CPU load on the local Bro IDS instances and calculates a new distribution of local network prefixes when load becomes imbalanced. If a subnet is assigned to a different IDS instance, the `movePrefix` function is invoked. This function calls `copy` to clone the multi-flow state associated with scan detection, followed by `move` to perform a loss-free transfer of the per-flow state for all active flows in the subnet.

We copy, rather than move, multi-flow state because the counters for port scan detection are maintained on the basis of ⟨external IP, destination port⟩ pairs, and connections may exist between a single external host and hosts in multiple local subnets. An order-preserving move is unnecessary because re-ordering would only potentially result in the scan detector failing to count some connection attempts, and, in this application, we are willing to tolerate moderate delay in scan detection. However, to avoid missing scans completely, we maintain eventual consistency of multi-flow state by invoking copy in both directions every 60 seconds.

Fast failure recovery. The second application (Figure 10) maintains a hot standby for each local IDS instance with an eventually consistent copy of all per-flow and multi-flow state. The `initStandby` function is invoked to initialize a standby (`stbyInst`) for an IDS instance (`normInst`). It notes which `normInst` the standby is associated with and requests notifications from `normInst` for packets whose corresponding state updates are important for scan detection and browser identification—TCP SYN, SYN+ACK, and RST packets and HTTP packets sent from a local client to an external server. The copy is made eventually consistent when these key packets are processed, rather than recopying state for every packet. In particular, events are raised by `normInst` for these packets and the controller invokes the `updateStandby` function. This function copies the appropriate per-flow state from `normInst` to the corresponding `stbyInst`. When a failure occurs, the forwarding table in the switch is updated to forward the appropriate prefixes to `stbyInst` instead of `normInst` (code not shown).

Selectively invoking advanced remote processing. The third application (code not shown) monitors for outdated browser alerts from each local Bro IDS instance, and uses the cloud to check for malware in connections triggering such alerts.

When a local IDS instance (`locInst`) raises an alert for a specific flow (`flowid`), the application calls `move(locInst, cloudInst,flowid,perflow,orderpreserving)` to

```
standbys ← {}
initStandby (normInst, stbyInst)
    standbys[normInst] ← stbyInst
    notify ({nw_proto: TCP, tcp_flags: SYN}, normInst, true,
        updateStandby)
    notify ({nw_proto: TCP, tcp_flags: RST}, normInst, true,
        updateStandby)
    notify ({nw_src: 10.0.0.0/8, nw_proto: TCP, tp_dst: 80},
        normInst, true, updateStandby)
updateStandby (event)
    normInst ← event.src
    stbyInst ← standbys[normInst]
    filter ← extractFlowId (event.pkt)
    copy (normInst, stbyInst, filter, PER)
```

Figure 10: Fast failure recovery application

transfer the flow's per-flow state and forward the flow's packets to the IDS instance running in the cloud. The move must be loss-free to ensure all data packets contained in the HTTP reply are received and included in the md5sum that is compared against a malware database, otherwise malware may go undetected. Multi-flow state in this case, i.e., the set of scan counters at the local IDS instance, does not matter for the cloud instance's actions (i.e., malware signature detection), so it is not moved or copied.

7. IMPLEMENTATION

Our OpenNF prototype consists of a controller that implements our northbound API (§5) and several modified NFs—Bro, PRADS, Squid, and iptables–that implement our southbound API (§4).

The OpenNF controller is written as a module atop Floodlight [7] (≈4.7K lines of Java code). The controller listens for connections from NFs and launches two threads—for handling state operations and events—for each NF. The controller and NFs exchange JSON messages to invoke southbound functions, provide function results, and send events. Packets contained in events are forwarded to NFs by issuing OpenFlow packet-out control messages [30] to the SDN switch (*sw*); flow-mod messages are issued for route updates. The interface with control applications is event-driven.

We implemented NF-specific handlers for each southbound API functions. The NFs use a shared library for communicating with the controller. We discuss the NF-specific modifications below, and evaluate the extent of these modifications in §8.2.2.

Bro IDS [32] performs a variety of security analyses defined by policy scripts. The `get`/`putPerflow` handlers for Bro lookup (using linear search) and insert `Connection` objects into internal hash tables for TCP, UDP, and ICMP connections. The key challenge is serializing these `Connection` objects and the many other objects (>100 classes) they refer to; we wrote custom serialization functions for each of these objects using Boost [2]. We also added a *moved* flag to some of these classes—to prevent Bro from logging errors during `delPerflow`—and a mutex to the `Connection` class—to prevent Bro from modifying the objects associated with a flow while they are being serialized. Lastly, we added library calls to Bro's main packet processing loop to raise events when a received packet matches a filter on which events are enabled.

PRADS asset monitor [15] identifies and logs basic information about active hosts and the services they are running. The `get`/`putPerflow` and `get`/`putMultiflow` handlers for PRADS lookup and insert `connection` and `asset` structures, which store flow meta data and end-host operating system and service details, respectively, in the appropriate hash tables. If an `asset` object provided in a `putMultiflow` call is associated with the same end-host as an `asset` object already in the hash table, then the handler

merges the contents of the two objects. The `get`/`putAllflows` handlers copy and merge, respectively, a global statistics structure.
Squid caching proxy [17] reduces bandwidth consumption by caching and serving web objects requested by clients. The per-flow state in Squid includes sockets, making it challenging to write `get`/`putPerflow` handlers. Fortunately, we are able to borrow code from CRIU [6] to (de)serialize sockets for active client and server connections. As with Bro, we wrote custom serialization functions, using Boost [2], for all objects associated with each connection. The `get`/`put`/`delMultiflow` handlers capture, insert, and remove entries from Squid's in-memory cache; entries are (de)serialized individually to allow for fine-grained state control.
iptables [10] is a firewall and network address translator integrated into the Linux kernel. The kernel tracks the 5-tuple, TCP state, security marks, etc. for all active flows; this state is read/written by iptables. We wrote an agent that uses libnetfilter_conntrack [11] to capture and insert this state when `get`/`putPerflow` are invoked. There is no multi-flow or all-flows state in iptables.

8. EVALUATION

Our evaluation of OpenNF answers the following key questions:

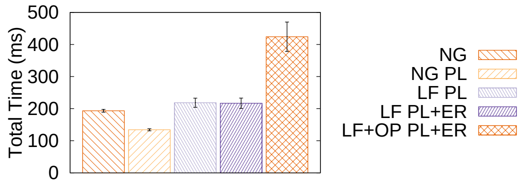
- Can state be moved, copied, and shared efficiently even when guarantees on state or state operations are requested by applications? What benefits do applications see from the ability to move, copy, or share state at varying granularities?
- How efficiently can NFs export and import state, and do these operations impact NF performance? How much must NFs be modified to support the southbound API?
- How is OpenNF's efficiency impacted by the scale of an NF deployment?
- To what extent do existing NF control planes hinder the ability to satisfy a combination of high-level objectives?

The testbed we used for our evaluation consists of an OpenFlow-enabled HP ProCurve 6600 switch and four mid-range servers (Quad-core Intel Xeon 2.8GHz, 8GB, 2 x 1Gbps NICs) that run the OpenNF controller and modified NFs and generate traffic. We use a combination of replayed university-to-cloud [25] and data-center [21] network traffic traces, along with synthetic workloads.

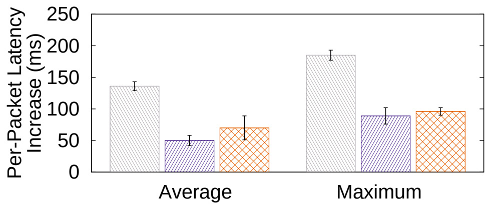
8.1 Northbound Operations

8.1.1 *Efficiency with Guarantees*

We first evaluate the efficiency of our northbound operations when guarantees are requested on state or state operations. We use two PRADS asset monitor instances ($PRADS_1$ and $PRADS_2$) and replay our university-to-cloud trace at 2500 packets/second. We initially send all traffic to $PRADS_1$. Once it has created state for 500 flows ($\approx$80K packets have been processed), we `move` *all* flows and their per-flow state, or `copy` *all* multi-flow state, to $PRADS_2$; we evaluate finer granularity operations in §8.1.3. To evaluate sharing with strong consistency, we instead call `share` (for all multi-flow state) at the beginning of the experiment, and then replay our traffic trace. During these operations, we measure the number of dropped packets, the added latency for packets contained in events from $PRADS_1$ or buffered at $PRADS_2$, and the total operation time (for move and copy only). Although the specific values for these metrics vary based on the NF, scope, filter granularity (i.e., number of flows/states affected), and packet rate, the high-level takeaways still apply.
Move. Figure 11 shows our results for `move` with varying guarantees and optimizations; we use the weaker versions of move presented in §5.1.1 and §5.1.2.



(a) Total `move` time



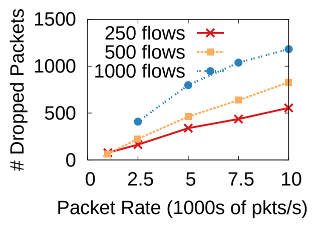
(b) Per-packet latency increase

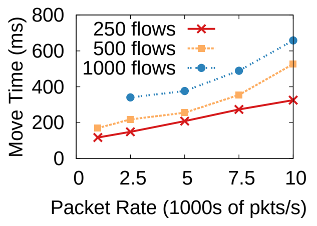
Figure 11: Efficiency of `move` with no guarantees (NG), loss-free (LF), and loss-free and order-preserving (LF+OP) with and without parallelizing (PL) and early-release (ER) optimizations; traffic rate is 2500 packets/sec; times are averaged over 5 runs and the error bars show 95% confidence intervals

A move without any guarantees or optimizations (NG) completes in 193ms. This time is primarily dictated by the time required for the NF to export (89ms) and import (54ms) state; we evaluate the southbound operations in detail in §8.2. The remaining 50ms is spent processing control messages from the NFs and performing the route update. Our parallelizing optimization (§5.1.4) can reduce the total time for the move operation (NG PL) to 134ms by exporting and importing state (mostly) in parallel. However, even this faster version of move comes at a cost: *225 packets are dropped!* Figure 12(a) shows how the number of drops changes as a function of the packet rate and the number of flows whose state is moved. We observe a linear increase in the number of drops as the packet rate increases, because more packets will arrive in the time window between the start of `move` and the routing update taking effect.

A parallelized loss-free move (LF PL) avoids drops by raising events. However, the 410 packets contained in events may each incur up to 185ms of additional latency. (Packets processed by $PRADS_1$ before the move or $PRADS_2$ after the move do not incur additional latency.) Additionally, the total time for the move operation increases by 62% (84ms). Figure 12(b) shows how the total move time scales with the number of flows affected and the packet rate. We observe that the total time for a parallelized loss-free move increases more substantially at higher packet rates. This is because more events are raised, and the rate at which the packets contained in these events can be forwarded to $PRADS_2$ becomes limited by the packet-out rate our OpenFlow switch can sustain. The average and maximum per-packet latency increase for packets contained in events also grows with packet rate for the same reason: e.g., the average (maximum) per-packet latency increase is 465ms (573ms) for a parallelized loss-free move of 500 flows at a packet rate of 10K packets/sec (graph not shown).

While we cannot decrease the total move time without using more rules in SDN switches, our early-release optimization (§5.1.4) can decrease the additional packet latency. At a rate of 2500 packets/sec, the average per-packet latency overhead for the 326 packets contained in events drops to 50ms (LF PL+ER in Figure 11(b)), a 63% decrease compared to LF PL; at 10K packets/sec this overhead drops to 201ms, a 99% decrease. Forwarding packets in events directly to $PRADS_2$, rather than sending packet-out commands to the OpenFlow switch, can likely reduce this latency even further.





(a) Packet drops during a parallelized `move` with no guarantees (b) Total time for a parallelized loss-free `move`

Figure 12: Impact of packet rate and number of per-flows states on parallelized `move` with and without a loss-free guarantee

Alert	Baseline	NG	LF	LF+OP
Incorrect File Type	26	25	24	26
Malware Hash Registry Match	31	28	27	31
MD5	116	111	106	116
Total	173	164	157	173

Table 1: Effects of different guarantees

Metric	Ignore	Copy Client	Copy All
Hits on $Squid_1$	117	117	117
Hits on $Squid_2$	Crashed	39	50
MB of multi-flow state transfered	0	3.8	54.4

Table 2: Effects of different ways of handling multi-flow

In addition to added packet latency, a loss-free move also introduces re-ordering: 657 packets (335 from events + 322 received by $PRADS_2$ while packets from events are still arriving) are processed out-of-order with a parallelized loss-free move. However, this re-ordering can be eliminated with an order-preserving move.

A fully optimized loss-free and order-preserving move (LF+OP PL+ER in Figure 11) takes 96% (208ms) longer than a fully optimized loss-free-only move (LF PL+ER) due to the additional steps involved. Furthermore, packets buffered at $PRADS_2$ (100 packets on average), while waiting for all packets originally sent to $PRADS_1$ to arrive and be processed, each incur up to 96ms of additional latency (7% more than LF PL+ER). Thus, applications can benefit from choosing an alternative version of move if they do not require both guarantees.

Copy and Share. A parallelized copy takes 111ms, with no packet drops or added packet latency, as there is no interaction between forwarding state update and this operation. In contrast, a share operation that keeps multi-flow state strongly consistent adds at least 13ms of latency to every packet, with more latency incurred when a packet must wait for the processing of an earlier packet to complete. This latency stems from the need to call `getMultiflow` and `putMultiflow` on $PRADS_1$ and $PRADS_2$, respectively, after every packet is processed, because our events only provide hints as to whether state changed but do not inform us if the state update is significant. For example, every packet processed by the PRADS asset monitor causes an update to the last seen timestamp in the multi-flow state object for the source host, but only a handful of special packets (e.g., TCP handshake and HTTP request packets) result in interesting updates to the object. However, adding more PRADS asset monitor instances (we experimented with up to 6 instances) does not increase the latency because `putMultiflow` calls can be issued in parallel. In general, it is difficult to efficiently support strong consistency of state without more intrinsic support from an NF, e.g., information on the significance of a state update.

8.1.2 *Importance of Guarantees*

We next evaluate the importance of the guarantees offered by our northbound API. Our methodology is similar to our experiments above, except we use the Bro IDS with a malware detection script [3], and we replay a trace of malware traffic [12] at 1000 packets/second. We issue a move operation (with the parallelize optimization) after 14K packets have been processed.

We compare the alerts raised by the Bro IDS when no move is performed (baseline) versus when a no guarantee (NG), loss-free (LF), or loss-free plus order-preserving (LF+OP) move is performed. Table 1 shows the type and number of alerts raised under each scenario. We observe that 5% and 9% of the alerts are missed with a no guarantee or loss-free move, respectively. More alerts are missed with loss-free because re-ordering is introduced. In contrast, no alerts are missing with a loss-free plus order-preserving move. Thus, the guarantees offered by our northbound API are essential to accurately monitoring and manipulating network traffic when packet processing is dynamically redistributed.

8.1.3 *Benefits of Granular Control*

Although the `move`, `copy`, and `share` operations above encompassed all flows, the northbound API allows applications to invoke these operations at any granularity, down to as fine as a single flow. We now examine the benefits this flexibility enables by using the `copy` operation with the Squid caching proxy. We generate 100 requests (drawn from a logarithmic distribution) for 40 unique URLs (objects are 0.5–4MB in size) from each of two clients at a rate of 5 requests/second. Initially, all requests are forwarded to $Squid_1$. After 20 seconds, we launch a second Squid instance ($Squid_2$) and take one of three approaches to handling multi-flow state: do nothing (*ignore*), invoke `copy` with the second client's IP as the filter (*copy client*), or invoke `copy` for all flows (*copy all*). Then, we update routing to forward all in-progress and future requests from the second client to $Squid_2$.

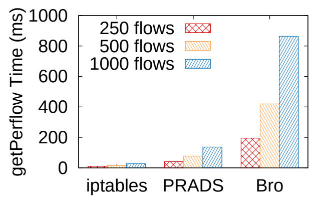
Table 2 shows the number of cache hits at each instance, and the bytes of multi-flow state transfered, under the three different approaches for handling multi-flow state. In all three approaches, the number of cache hits for $Squid_1$ are the same because all the unique objects were cached before the copy. Ignoring multi-flow state entirely causes the second instance to *crash*, as the objects currently being served to the second client are not available. Copying multi-flow state for the second client's flows avoids the crash, but skipping the other multi-flow state results in a 28% lower cache hit ratio at $Squid_2$ compared to copying all multi-flow state (i.e, the entire cache). However, the latter requires a 14.2x larger state transfer. OpenNF's APIs allows each application to make the appropriate trade-offs in such respects when selecting the granularity at which to invoke operations.

8.2 Southbound API

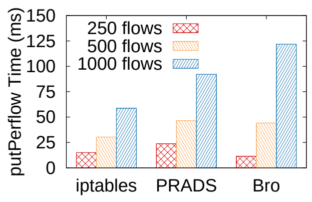
The time required to export and import state at NFs directly impacts how quickly a `move` or `copy` operation completes and how much additional packet latency is incurred when `share` is used. We thus evaluate the efficiency of OpenNF's southbound operations for several of the NFs we modified. We also examine how much code was added to the NFs to support these operations.

8.2.1 *API Call Processing*

Figures 13(a) and 13(b) show the time required to complete a `getPerflow` and `putPerflow` operation, respectively, as a function of the number of flows whose state is exported/imported. We observe a linear increase in the execution time of `getPerflow` and `putPerflow` as the number of per-flow state chunks increases. The time required to (de)serialize each *chunk* of state and send it to (receive it from) the controller accounts for the majority of the



(a) Time for `getPerflow`



(b) Time for `putPerflow`

Figure 13: Efficiency of state export and import

NF	LOC added for serialization	Total LOC added	Increase in NF code
Bro IDS	2.9K	3.3K	4.0%
PRADS asset monitor	0.1K	1.0K	9.8%
Squid caching proxy	5.0K	7.8K	4.2%
iptables	0.6K	1.0K	n/a

Table 3: Additional NF code to implement OpenNF's southbound API

execution time. Additionally, we observe that `putPerflow` completes at least 2x faster than `getPerflow`; this is due to deserialization being faster than serialization. Overall, the processing time is highest for Bro because of the size and complexity of the per-flow state. The results for multi-flow state are qualitatively similar; we exclude them for brevity. We are working on techniques for further improving the efficiency of southbound API calls.

We also evaluate how NF performance is impacted by the execution of southbound operations. In particular, we measure average per-packet processing latency (including queueing time) during normal NF operation and when an NF is executing a `getPerflow` call. Among the NFs, the PRADS asset monitor has the largest relative increase—5.8% (0.120ms vs. 0.127ms), while the Bro IDS has the largest absolute increase—0.12ms (6.93ms vs. 7.06ms). In both cases, the impact is minimal, implying that southbound operations do not significantly degrade NF performance.

8.2.2 NF Changes

To quantify the NF modifications required to support our southbound API, we counted the lines of code (LOC) that we added to each NF (Table 3). The counts do not include the shared library used with each NF for communication with the controller: ≈2.6K LOC. At most, there is a 9.8% increase in LOC[13], most of which is state serialization code that could be automatically generated [4]. Thus, the NF changes required to support OpenNF are minimal.

8.3 Controller Scalability

Since the controller executes all northbound operations (§5), its ability to scale is crucial. We thus measure the performance impact of conducting simultaneous operations across many pairs of NFs.

To isolate the controller from the performance of individual NFs, we use "dummy" NFs that replay traces of past state in response to `getPerflow`, simply consume state for `putPerflow`, and infinitely generate events during the lifetime of the experiment. The traces we use are derived from actual state and events sent by PRADS asset monitor while processing our cloud traffic trace. All state and messages are small (202 bytes and 128 bytes, respectively) for consistency, and to maximize the processing demand at the controller and minimize the impact due to network transfer.

Figure 14 shows the average time per loss-free `move` operation as a function of the number of simultaneous operations. The average time per operation increases linearly with both the number of simultaneous operations and the number of flows affected.

[13]We do not calculate an increase for iptables because we wrote a user-level tool to export/import state rather than modifying the Linux kernel.

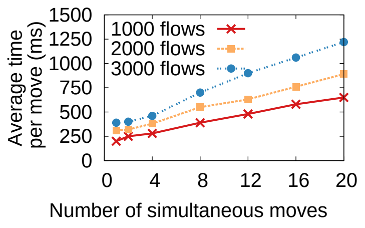


Figure 14: Performance of concurrent loss-free `move` operations

We profiled our controller using HPROF [9] and found that threads are busy reading from sockets most of the time. This bottleneck can be overcome by optimizing the size of state transfers using compression. We ran a simple experiment and observed that, for a `move` operation for 500 flows, state can be compressed by 38% improving execution latency from 110ms to 70ms.

8.4 Prior NF Control Planes

Lastly, we compare the ability to satisfy the objectives of an elastic/load balanced network monitoring application using OpenNF versus existing approaches [6, 20, 24, 27, 33] (§2.2). We start with one Bro IDS instance (Bro_1) and replay our data center traffic trace at a rate of 2500 packets/sec for 2 minutes. We then double the traffic rate, add a second Bro IDS instance (Bro_2), and rebalance all HTTP flows to Bro_2 (other flows remain at Bro_1); 2 minutes later we scale back down to one instance.

VM Replication. This approach takes a snapshot of the current state in an existing NF instance (Bro_1) and copies it to a new instance (Bro_2) as is. Since, VM replication does not do fine-grained state migration, we expect it to have unneeded states (§2.2) in all instances. We quantify unneeded state by comparing: a snapshot of a VM running the Bro IDS that has not yet received any traffic (*base*), a snapshot taken at the instant of scale up (*full*), and snapshots of VMs that have only received either HTTP or other traffic prior to scale up (*HTTP* and *other*). *Base* and *full* differed by 22MB. *HTTP* and *other* differed from *base* by 19MB and 4MB, respectively; these numbers indicate the overhead imposed by the unneeded state at the two Bro IDS instances. In contrast, the amount of state moved by OpenNF (i.e., per-flow and multi-flow state for all active HTTP flows) was 8.1MB. More crucial are the correctness implications of unneeded state: we found 3173 and 716 incorrect entries in conn.log at the two Bro IDS instances, arising because the migrated HTTP (other) flows terminate abruptly at Bro_1 (Bro_2).

Scaling Without Re-balancing Active Flows. Control planes that steer only new flows to new scaled out NF instances leave existing flows to be handled by the same NF instance [24]. Thus, Bro_1 continues to remain bottlenecked until some of the flows traversing it complete. Likewise, in the case of scale in, NFs are unnecessarily "held up" as long as flows are active. We observe that ≈9% of the HTTP flows in our cloud trace were longer than 25 minutes; this requires us to wait for more than 25 minutes before we can safely terminate Bro_2, otherwise we may miss detecting some attacks.

9. CONCLUSION

Fully extracting the combined benefits of NFV and SDN requires a control plane to manage both network forwarding state and internal NF state. Without such joint control, applications will be forced to make trade-offs among key objectives. Providing such control is challenging because we must address race conditions and accommodate a variety of application objectives and NF types. We presented a novel control plane architecture called OpenNF that addresses these challenges through careful API design informed by the ways NFs internally manage state today, and clever techniques that ensure lock-step coordination of updates to NF and network state. A thorough evaluation of OpenNF shows that: its joint control is generally efficient even when applications have certain stringent requirements; OpenNF allows applications to make suitable choices in meeting their objectives; and NFs need modest changes and incur minimal overhead when supporting OpenNF primitives.

10. ACKNOWLEDGEMENTS

We would like to thank Vivek Pai (our shepherd), Katerina Argyraki, Tom Anderson, David Cheriton, Vimalkumar Jeyakumar, Arvind Krishnamurthy, Ratul Mahajan, Jennifer Rexford, and the anonymous reviewers for their insightful feedback. This work is supported in part by a Wisconsin Alumni Research Foundation (WARF) Accelerator Award and National Science Foundation grants CNS-1302041, CNS-1314363 and CNS-1040757. Aaron Gember-Jacobson is supported by an IBM PhD Fellowship.

11. REFERENCES

[1] Balance. `http://inlab.de/balance.html`.
[2] Boost C++ libraries. `http://boost.org`.
[3] Bro 2.1 documentation: detect-mhr.bro. `http://bro.org/sphinx-git/scripts/policy/frameworks/files/detect-MHR.bro.html`.
[4] C++ Middleware Writer. `http://webebenezer.net`.
[5] Check Point Software: ClusterXL. `http://checkpoint.com/products/clusterxl`.
[6] CRIU: Checkpoint/Restore In Userspace. `http://criu.org`.
[7] Floodlight OpenFlow Controller. `http://floodlight.openflowhub.org`.
[8] HAProxy: The reliable, high performance TCP/HTTP load balancer. `http://haproxy.1wt.eu/`.
[9] HPROF. `http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`.
[10] iptables. `http://netfilter.org/projects/iptables`.
[11] libnetfilter_conntrack project. `http://netfilter.org/projects/libnetfilter_conntrack`.
[12] Malware-traffic-analysis.net. `http://malware-traffic-analysis.net`.
[13] nDPI. `http://ntop.org/products/ndpi`.
[14] Network functions virtualisation: Introductory white paper. `http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf`.
[15] Passive Real-time Asset Detection System. `http://prads.projects.linpro.no`.
[16] RiverBed Steelhead Load Balancing. `http://riverbed.com/products-solutions/products/wan-optimization-steelhead/wan-optimization-management`.
[17] Squid. `http://squid-cache.org`.
[18] A. Anand, V. Sekar, and A. Akella. SmartRE: An architecture for coordinated network-wide redundancy elimination. In *SIGCOMM*, 2009.
[19] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford. A slick control plane for network middleboxes. In *HotSDN*, 2013.
[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.
[21] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.
[22] S. K. Fayazbakhsh, L. Chaing, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *NSDI*, 2014.
[23] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-Class Entities. Technical Report TR1771, University of Wisconsin-Madison, 2012.
[24] A. Gember, A. Krishnamurthy, S. St. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical Report arXiv:1305.0209, 2013.
[25] K. He, L. Wang, A. Fisher, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in EC2 and Azure. In *IMC*, 2013.
[26] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network*, 2008.
[27] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, 2008.
[28] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. In *HotNets*, 2013.
[29] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *NSDI*, 2014.
[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2), 2008.
[31] C. Nicutar, C. Paasch, M. Bagnulo, and C. Raiciu. Evolving the internet with connection acrobatics. In *HotMiddlebox*, 2013.
[32] V. Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security (SSYM)*, 1998.
[33] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM*, 2013.
[34] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A high availability framework for middleboxes. In *SoCC*, 2013.
[35] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.
[36] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.
[37] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang. A first look at cellular machine-to-machine traffic: Large scale measurement and characterization. In *SIGMETRICS*, 2012.
[38] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM*, 2012.
[39] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE*, 2011.

APPENDIX

A. PROOF OF LOSS FREE AND ORDER-PRESERVING

In this appendix, we prove that move is loss-free and order-preserving. We first consider the mechanisms described in §5.1.1 and §5.1.2, and assume no loss or reordering occurs on network paths. We then relax these assumptions and consider the mechanisms described in §5.1.3.

A.1 No Loss or Reordering on Network Paths

Let p_i be the i^{th} packet for a flow f that arrives at *sw* and $\langle p\rangle_{i,j}$ be the sequence of packets from i to j. Also, let $S_{i,j} = \phi_{S_{init}}(\langle p\rangle_{i,j})$ be the value of the per-flow state for f after processing $\langle p\rangle_{i,j}$ starting from initial state S_{init}.

The controller issues all southbound API calls and route updates so we can definitively order the actions in time:

- t_1 Enable events and packet dropping for f on *srcInst*
- t_2 Get per-flow state S from *srcInst*
- t_3 Put per-flow state S to *dstInst*
- t_4 Extract packets from events buffered on controller and send to *sw* to forward to *dstInst*
- t_5 Enable events and packet buffering for f on *dstInst*
- t_6 Change the route for f on *sw* to forward to *srcInst* and the controller
- t_7 Change the route for f on *sw* to forward to *dstInst*
- t_8 Disable events and release packet buffer for f on *dstInst*

We use the time points to refer to the completion of each action from the perspective of the node on which the action is performed.

Loss-free. Let p_k $(1 < k < n)$ be the first packet for f to arrive at *sw* after t_7. Then, $\langle p\rangle_{1,k-1}$ will be forwarded to *srcInst*, and $\langle p\rangle_{k,n}$ will be forwarded to *dstInst*.

Of the packets forwarded to *srcInst*, let p_j $(1 < j < k)$ be the first packet for f to be dequeued at *srcInst* after t_1. Then, $\langle p\rangle_{1,j-1}$ will be processed at *srcInst* before t_1, resulting in per-flow state $S_{1,j-1}$. In contrast, $\langle p\rangle_{j,k-1}$ will be sent to the controller in events and dropped at *srcInst*.

Since no packets are processed at *srcInst* after t_1, the state $S_{1,j-1}$ will be exported from *srcInst* at t_2 and imported on *dstInst* at t_3. Also, since no packets are forwarded to *dstInst* before t_7, there won't be state S to overwrite or combine during the import at *dstInst*.

Events for $\langle p\rangle_{j,k-1}$ may arrive at the controller anytime after t_1. If they arrive before t_3, they are buffered. Starting at t_3, packets are extracted from the buffered events and sent to *sw* to be forwarded to *dstInst*. If events from *srcInst* arrive at the controller after the buffer is empty (i.e., after t_4), the packets they contain are immediately sent to *sw* to be forwarded to *dstInst*. Thus, all $p_i \in \langle p\rangle_{j,k-1}$ will be forwarded to *dstInst* and processed.

After t_7, $\langle p\rangle_{k,n}$ will arrive at *dstInst*. They will be dequeued and processed after t_8. In summary, $\langle p\rangle_{1,j-1}$ will be processed at *srcInst*, with S_{init} as the initial per-flow state, and $\langle p\rangle_{j,n}$ will be processed at *dstInst*, with $S_{1,j-1}$ as the initial per-flow state, implying move is loss-free.

Order-preserving. As above, let p_k be the first packet for f to arrive at *sw* after t_7 and p_j be the first packet for f to be dequeued at *srcInst* after t_1. Then, $\langle p\rangle_{1,j-1}$ will arrive and be processed at *srcInst* in order before t_1. Similarly, $\langle p\rangle_{k,n}$ will arrive (after t_7) and be processed (after t_8) at *dstInst* in order. To guarantee order-preserving, *dstInst* must have $S_{1,k-1}$ by t_8. From above, we know *dstInst* will have $S_{1,j-1}$ by t_3. Thus, we need to show that *dstInst* will receive and process $\langle p\rangle_{j,k-1}$ in order after t_3 but before t_8. Since, $\langle p\rangle_{j,k-1}$ is sent to a controller through a TCP channel, their order is preserved at the controller.

Let p_m ($j < m < k$) be the first packet for f to arrive at *sw* after t_6. This implies *sw* forwards $\langle p\rangle_{m,k-1}$ to both *srcInst* and the controller. The controller will remember the last packet in the sequence, p_{k-1}.

Events for $\langle p\rangle_{j,m-1}$ may arrive at the controller anytime after t_1. Events arriving before t_3 are buffered, while events arriving after t_4 are handled immediately. The controller will extract the packets from these events, mark the packets with a "do-not-buffer" flag, and send them to *sw* to be forwarded to *dstInst*. They will be processed at *dstInst* as they arrive, resulting in per-flow state $S_{1,m-1}$ at *dstInst*.

Since $\langle p\rangle_{m,k-1}$ are not forwarded to *srcInst* until after t_6, the controller will not receive events for these packets until after t_6. The controller will extract, mark, and send these packets to *dstInst* as above. Events are enabled on *dstInst* at t_5, so *dstInst* will raise an event for $\langle p\rangle_{m,k-1}$.

Since p_{k-1} is the last packet the controller receives from *sw*, it knows p_{k-1} was the last packet forwarded to *srcInst*. Furthermore, once the controller receives an event for p_{k-1} from *dstInst*, it knows *dstInst* has processed $p_{1,k-1}$ and has the state $S_{1,k-1}$. Therefore, the controller can guarantee that *dstInst* has state $S_{1,k-1}$ by t_8 and move is order-preserving.

The proof can be extended to moves involving multi-flow state by expanding the notion of flow to actually refer to a group of flows; we omit this extension for brevity.

A.2 Lossy Network Paths

We now remove the assumptions that no loss occurs on network paths and no reordering occurs on the path from *sw* to *dstInst*; we still assume there is no reordering on the path from *sw* to *srcInst*. We use the same notation as above. In accordance with the mechanisms described in §5.1.3, we modify the actions at time points t_4, t_6, and t_7 and add a time point as follows:

- t'_4 Extract packets from events buffered on controller and send to *dstInst*
- t'_6 Do nothing
- t'_7 Change the route for f on *sw* to forward to *dstInst*
- $t'_{7.5}$ Send a tracer packet to *sw* to forward to *srcInst*

Loss-free. The only change from above is the following:

Events for $\langle p\rangle_{j,k-1}$ may arrive at the controller anytime after t_1. If they arrive before t_3, they are buffered. Starting at t_3, packets are extracted from the buffered events and sent to *dstInst*. If events from *srcInst* arrive at the controller after the buffer is empty (i.e., after t'_4), the packets they contain are immediately sent to *dstInst*. Since the control channel from the controller to *dstInst* is reliable, all $p_i \in \langle p\rangle_{j,k-1}$ will arrive at *dstInst* and processed.

Order-preserving. As above, $\langle p\rangle_{1,j-1}$ will arrive and be processed at *srcInst* in order before t_1. Similarly, $\langle p\rangle_{k,n}$ will arrive (after t'_7) and be processed (after t_8) at *dstInst* in order. We thus need to show that *dstInst* will receive and process $\langle p\rangle_{j,k-1}$ in order after t_3 but before t_8.

Events for $\langle p\rangle_{j,k-1}$ may arrive at the controller anytime after t_1. Events arriving before t_3 are buffered, while events arriving after t'_4 are handled immediately. The controller will extract the packets from these events, mark the packets with a "do-not-buffer" flag, and send them to *dstInst*. Since $\langle p\rangle_{j,k-1}$ is sent to the controller through a TCP channel and to *dstInst* through a TCP channel, the order of packets within this sequence is preserved and none will be lost. The packets will be processed at *dstInst* as they arrive, resulting in per-flow state $S_{1,k-1}$ at *dstInst*.

Since a tracer packet is not sent to *sw* to be forwarded to *srcInst* until after t'_7, no other packets for f will arrive at *srcInst* after the tracer packet (assuming no reordering occurs on the path from *sw* to *srcInst*). Thus, all events from *srcInst* for $\langle p\rangle_{j,k-1}$ will arrive at the controller before an event for the tracer packet. Furthermore, an event for packet p_{k-1} (or an earlier packet in $\langle p\rangle_{j,k-1}$ if p_{k-1} is dropped on the path from *sw* to *srcInst*) will be the last event to arrive before an event for the tracer packet. This allows the controller to know that p_{k-1} was the last packet forwarded to *srcInst*.

Once the controller receives an event for p_{k-1} from *dstInst*, it knows *dstInst* has processed $p_{1,k-1}$ and has the state $S_{1,k-1}$. Therefore, the controller can guarantee that *dstInst* has state $S_{1,k-1}$ by t_8 and move is order-preserving.