# Improving the Safety, Scalability, and Efficiency of Network Function State Transfers

Aaron Gember-Jacobson and Aditya Akella
University of Wisconsin-Madison
{agember,akella}@cs.wisc.edu

## ABSTRACT

Several frameworks have been proposed to orchestrate the transfer of internal state between network function (NF) instances. Unfortunately, these frameworks suffer from safety, efficiency, and scalability problems due to their excessive use of packet buffering. We propose two novel enhancements, packet reprocessing and peer-to-peer transfers, to address these issues. We show these enhancements reduce the average per-packet latency overhead by up to 92% and state transfer times by up to 70%.

## CCS Concepts

•**Networks** → **Middle boxes / network appliances;** *Network dynamics; Programmable networks;*

## Keywords

Network functions virtualization; peer-to-peer; software defined networking

## 1. INTRODUCTION

Network functions (NFs) tend to create and maintain complex internal state in order to perform intricate cross-packet and cross-flow analyses. Such rich packet processing can help improve the security, performance, and efficiency of a network and its applications. However, the stateful nature of NFs makes it difficult to reroute traffic without negatively impacting NF behavior.

For example, consider a scenario where an intrusion prevention system (IPS) is heavily loaded and must be scaled out in order to maintain suitable performance (e.g., low latency). Network functions virtualization (NFV) [3] allows us to launch another IPS instance and balance load between multiple instances to avoid performance bottlenecks. However, to ensure the IPS blocks all suspicious traffic, we must ensure that the NF state associated with existing flows (e.g., partial signature matches) is available at whichever IPS instance is now responsible for that traffic.



*HotMiddlebox'15, August 17-21, 2015, London, United Kingdom*
 ISBN 978-1-4503-3540-9/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785989.2785997

Prior work has shown that virtual machine (VM) replication is too coarse, and per-flow consistent updates [15] are too slow, to adequately maintain NF correctness and performance [7]. Instead, frameworks like Split/Merge [13] and OpenNF [7] have been designed to move and copy internal NF state at fine granularity. These frameworks allow us to maintain correct NF behavior amidst dynamic redistribution of packet processing: e.g., during elastic NF scaling, NF upgrades, or selective NF offload.

Unfortunately, frameworks like OpenNF and Split/Merge suffer from key safety, efficiency, and scalability problems. These issues arise due to limitations in how the frameworks provide certain safety guarantees. In particular, Split/Merge and OpenNF both rely on a central controller to buffer packets during a state transfer so that no packets, or state updates, are lost [7, 13]. This approach imposes high latency overhead on packets arriving during the transfer, and requires significant CPU and memory capacity at the controller. For example, consider a scenario where OpenNF is used to move 800 flows between two instances of the Bro IDS [11], while traffic is flowing at a rate of 20K packets/second (0.15Gbps). Buffering packets at the controller increases packet latencies by 93x (≈46.5ms), requires a buffer capacity of 2K packets, and consumes 6% of the controller's CPU[1]. Furthermore, there is no mechanism to address buffer overflow, which means loss-freedom is not guaranteed!

We present two enhancements to remedy these issues:

**1) Packet reprocessing** allows an NF instance to continue processing packets while a state transfer occurs. If a packet triggers an update to state, a copy of the packet is sent to the new NF instance, where it is reprocessed to bring the transferred state "up to speed." This design reduces the latency overhead imposed on packets, provides a mechanism for safely recovering from buffer overflow, and, for some NFs, reduces the amount of buffer space required. The key challenge is preventing the new NF instance from outputting packets or log entries that were already output by the original NF instance.

**2) Peer-to-peer (P2P) transfer** sends state and packets directly from one NF instance to another, without any buffering at the controller. This accelerates the state transfer and reduces the CPU and memory burden on the controller. The key challenge is correctly injecting packets into the new NF's input stream.

[1]Our controller machine is equipped with a 4-core 2.8GHz Intel Xeon CPU and 6GB of memory.

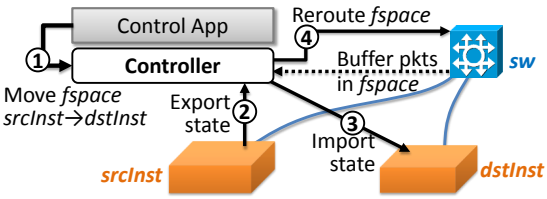



**Figure 1: Parts of an NF state management framework, and the steps for a move operation; packet buffering is required for loss-free moves**

Our preliminary evaluation with OpenNF and two NFs [2, 11] shows that packet reprocessing reduces the average per-packet latency overhead by up 92%, and P2P transfer reduces the total state transfer time by up to 70%.

## 2. BACKGROUND

Frameworks like Split/Merge [13] and OpenNF [7] facilitate fine-grained transfers of internal NF state to support fast and safe reallocation of flows across NF instances. In these frameworks (Figure 1), a scenario-specific control application decides: (*i*) when internal NF state should be moved—e.g., after a new NF instance is launched; (*ii*) what subset of state should be moved—this is usually defined in terms of a flow space (*fspace*), e.g., all state pertaining to flows originating from a particular subnet; and (*iii*) between which pair of NF instances the transfer should occur. A central controller then asks the source NF (*srcInst*) to export the state pertaining to flows in *fspace*. This state is provided to and imported by the target NF (*dstInst*). In Split/Merge, the state is transferred directly from *srcInst* to *dstInst*, while in OpenNF the state passes through the controller. In either case, NFs must be modified prior to run time to support such export/import operations. Finally, the controller updates the forwarding state in a software-defined networking (SDN) switch (*sw*), such that traffic in *fspace* is now forwarded to *dstInst*.

Since end-hosts may continue to generate packets during a move operation, these frameworks must carefully manage packet processing to avoid introducing inconsistencies in NF state. In the simplest case, OpenNF drops all packets in *fspace* that arrive at *srcInst* during a state transfer. However, if an NF is located "off-path" and processes a copy of all traffic, the drops will not trigger retransmissions from end-hosts,[2] and the NF's analyses may be incomplete: e.g., the Bro IDS may fail to raise some security alerts if only part of the data sent over a connection is checked for malware [7].

To address this issue, OpenNF and Split/Merge provide a *loss-free* move operation, in which packets in *fspace* are buffered at the controller until the state transfer completes. In Split/Merge the controller receives packet events from *sw*, while in OpenNF the packet events come from *srcInst*; the latter design ensures packets that are in-transit from *sw* to *srcInst* at the start of the move are not lost. OpenNF also supports an *order-preserving* move [7], but we exclude this from our discussion due to its complexity.

Due to OpenNF's stronger guarantees and open source licensing, the rest of the paper discusses problems and solutions primarily in the context of OpenNF. However, the same ideas extend to Split/Merge (Section 7).

[2]End-hosts will retransmit when network-induced drops occur prior to packet cloning.

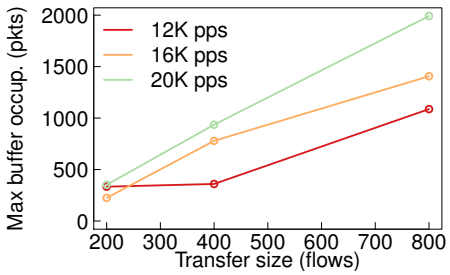



**Figure 2: Required buffer capacity when moving state for $n$ flows with a traffic rate of $r$ pps**

## 3. PROBLEMS WITH A LOSS-FREE MOVE

We now discuss the safety, efficiency, and scalability problems that arise from the way in which OpenNF performs a loss-free move of internal NF state. We also examine whether mechanisms used in virtual machine (VM) migration and SDN can address these issues.

**Controller Resources.** A loss-free move requires significant packet buffering. The controller must buffer all packets in *fspace* that arrive at *srcInst* between the start of state export (step 2 in Figure 1) and the end of state import (step 3). Assuming a move operation affects $n$ active flows, the state export and import operations take a total of $s$ seconds per flow, and the aggregate traffic rate for the $n$ flows is $r$ packets per second (pps), then the controller must buffer up to $s \times n \times r$ packets.

However, OpenNF introduced a *late-locking and early-release* (LLER) optimization to start buffering packets of a flow as late as possible and flush the buffer as early as possible. In particular, packets from a given flow are processed at *srcInst* until *srcInst* starts to export state pertaining to that flow. Similarly, the controller releases buffered packets from a flow as soon as *dstInst* has imported all state pertaining to the flow, even if state for other flows is not yet imported.

Figure 2 shows the actual buffer capacity requirements for a loss-free move, with LLER, between two Bro IDS instances ($s \approx 8$ ms). Even with low values for $n$ and $r$, the controller must have buffer space for thousands of packets: e.g., transferring state for 800 flows while the traffic rate for these flows is just 20K pps (0.15Gbps) requires a buffer capacity of 2K packets. The same scenario requires a buffer capacity of 38K packets in Split/Merge, due to the lack of LLER.

Packet buffering also consume controller CPU resources: e.g., an additional 6% of the CPU is consumed when the traffic rate rises from 4K pps to 20K pps.

**Buffer Overflow.** As the traffic rate ($r$) increases beyond a few Gbps, or state export/import time ($s$) increases beyond a few tens of milliseconds, the controller may not be able to provide the required buffer capacity—especially if multiple moves are occurring simultaneously. Since packets are not processed by *srcInst* before being sent to the controller, any packets which cannot be buffered at the controller will never be processed, and all NF state updates that would have occurred from processing these packets will never occur. This violates the loss-free safety guarantee that prompted buffering in the first place!

**Latency Overhead.** Finally, we face the problem of inflated packet latencies. We measure latency based on the time a packet arrives at *sw* and the time an NF instance

outputs (a modified version of) the packet.[3] LLER reduces the average latency overhead due to buffering by up to 63%, but the average latency of packets arriving during a loss-free move may still be over 50ms. Furthermore, latency overhead grows as the controller receives more packets: e.g., when moving state for 800 flows between two IDS instances, the average per-packet latency overhead rises from 3ms to 47ms when the packet rate increases from 4K pps to 20K pps—a 15x increase in latency with just a 5x increase in traffic rate!

## 3.1 Potential Solutions

The above issues are reminiscent of challenges faced in VM migration and SDN. Below, we consider whether solutions from these domains can address the efficiency and scalability problems in OpenNF.

**VM Memory Deltas.** When VMs are migrated, packets must be buffered between the time the VM stops running at its old location and the time it starts at its new location. State-of-the-art VM migration techniques [5] minimize this downtime by sending multiple rounds of memory deltas before stopping the VM to capture and send a final delta. This same technique cannot be applied in OpenNF, because internal NF state is managed at sub-page granularity.

**Distributed SDN Controllers.** Highly reactive SDN applications (e.g., fine-grained data center traffic engineering [4]) can quickly overwhelm an SDN controller due to the high volume of control messages received from SDN switches. This is similar to the high volume of packets in OpenNF. Several distributed SDN controllers have been proposed [6, 9, 10, 16] to address this issue. However, scaling out the controller introduces new state consistency challenges.

**Our Solutions.** Given the insufficiency of prior solutions, we propose two novel enhancements to improve the safety, efficiency, and scalability of loss-free moves in OpenNF (and Split/Merge). We introduce *packet reprocessing* as a mechanism to reduce latency overhead, guarantee loss-freedom amidst buffer overflow, and, for some NFs, reduce buffer capacity demands (Section 4). Then we introduce *P2P transfers* to reduce the number of packets subjected to additional latencies and improve controller scalability (Section 5).

# 4. PACKET REPROCESSING

Packet reprocessing fundamentally changes the way OpenNF (or Split/Merge) handles packets that arrive during a loss-free move operation. First, *srcInst* continues processing all packets in *fspace* that arrive during the operation, even if processing a packet results in updates to state that has already been exported. This reduces latency overhead and provides a source of up-to-date state if packets are dropped due to buffer overflow. Second, *dstInst* imports a "snapshot" of state from *srcInst*, and then brings it "up to speed," rather than requiring the absolute latest state to be moved from *srcInst*. This provides an opportunity to reduce buffer demands when moving state for NFs where only some packets trigger state updates.

**Process Packets Twice.** Our key insight is to process packets twice—once at *srcInst* and once at *dstInst*. A packet in *fspace* that arrives at *srcInst* during a move operation is processed by *srcInst* to: (*i*) update state at *srcInst*, and (*ii*) obtain any output the NF creates as a result of receiving this packet. For example, a network address translator (NAT) rewrites packet headers and outputs the modified packet, while an IDS logs an alert if malicious traffic is detected. Then, the packet is re-processed by *dstInst* solely to *obtain any state updates* the NF requires to process future packets. For example, a NAT needs the mapping created during the processing of the first packet of a flow in order to correctly process future packets from the flow, and an IDS needs metadata from prior packets to detect attacks that span multiple packets or flows. We do not want *dstInst* to produce packet or log output during this reprocessing, because *srcInst* has already produced such output.

This introduces a key challenge: *how do we suppress an NF's output while reprocessing a packet?* There are many different points in an NF's code where packet or log output may be produced, and this output should be produced during normal processing, but not during reprocessing. Fortunately, NFs typically use standard libraries and system calls to produce packet and log output: either PCAP (`pcap_dump`, `pcap_inject`, etc.) or socket (`send`, `sendto`, etc.) functions for the former, and regular I/O functions (`write`, `printf`, etc.) for the latter.[4] We can thus easily replace all calls to these standard output functions—using a simple find and replace applied to the NF's code—with calls to wrapper functions. Each wrapper function checks whether a (thread-local) global variable has been set to indicate the current packet is being reprocessed, and, if not, calls the appropriate library/system function.

**Buffering.** Packet reprocessing still requires packets to be buffered at the controller, but this buffering is no longer part of the critical path for packet processing. For every packet in *fspace* that arrives at *srcInst* during a loss-free move, *srcInst* (*i*) sends the packet to the controller, and (*ii*) processes the packet normally. This means *packet processing is only delayed by the time it takes an NF to copy the packet.*

The controller buffers packets from *srcInst* if the state pertaining to the corresponding flow has not yet been imported by *dstInst*—as is always done with a loss-free move with LLER. Once *dstInst* acknowledges that it has imported a flow's state, the controller marks the flow's buffered packets with a "do-not-output" flag and sends them to *dstInst* for reprocessing. The flag informs *dstInst* that it should suppress packet and log output while it processes these packets.

Note that state is kept on *srcInst* until some time after the move operation completes, rather than deleting the state immediately after it is exported. This ensures *srcInst* has the necessary state for normal packet processing.[5]

**Handling Buffer Overflow.** In addition to reducing latency overhead, packet reprocessing allows us to *guarantee a move is loss-free, even when the controller's packet buffer overflows.* As discussed in Section 3, buffer overflow normally causes packets, and their corresponding updates to be lost. However, with packet reprocessing, the NF state on *srcInst* is always up-to-date, because any packet in *fspace* that arrives at *srcInst* during the move is both sent to the controller and processed by *srcInst*. Therefore, we

[3]For NFs which do not output packets, we use the time an NF instance finishes processing the packet.

[4]If NFs use non-standard output functions, we can easily extend our design to include these functions.

[5]We assume NFs (have been modified to) lock the appropriate state objects while a packet is being processed in order to prevent a partially modified object from being exported.

can always re-export state from *srcInst*, and import the updated state on *dstInst*, without worrying about processing buffered, or dropped, packets at *dstInst*. We formally prove the correctness of this algorithm in Section 6.

Careful buffer management can help minimize the number of buffer overflows and state re-transfers. The most advantageous drop policy is to identify the flow with the largest number of currently buffered packets and drop all buffered packets for this flow. Then, we re-export and import just the state pertaining to the affected flow. This policy will free up the largest amount of buffer space in relation to the number of state re-transfers.

**Reducing Buffer Capacity Demands.** Packet reprocessing also introduces an opportunity to reduce the number of packets that need to be buffered during a move operation. Recall that packets are reprocessed by *dstInst* solely to obtain any state updates the NF requires to process future packets. Thus, if no state is updated when a packet is processed,[6] we only need to process the packet at *srcInst* to obtain any output the NF creates as a result of receiving this packets; *dstInst* does not need to re-process packet, thereby eliminating the need to potentially buffer the packet.

Some NFs routinely update internal state for every packet—e.g., an IPS, IDS, or WAN optimizer—while others only update state for select packets—e.g., a stateful firewall only updates NF state when a TCP handshake completes or a connection finishes.[7] The latter can selectively clone packets to reduce buffer demands, and lighten the reprocessing load at *dstInst*. This is similar to Pico Replication's approach for eliminating unnecessary state cloning and output buffering at *srcInst* [12]; however, our goal is to reduce the burden on the controller and *dstInst*.

**Limitations.** Maintaining correct NF behavior in the presence of packet reprocessing requires an NF's execution to be deterministic. In other words, if an NF starts from some state $S_i$ and processes some packet, it is essential that the resulting state is always $S_{i+1}$ and the output (if any) is always $O_{i+1}$. Without this property, *dstInst* could end up in a different state ($S'_{i+1}$) than *srcInst* after reprocessing a packet, and the behavior of *dstInst* going forward could differ from the behavior that would have resulted if only *dstInst* had processed the packet or the move operation had never occurred. We can guarantee this property for many NFs by seeding an NF's random number generator, forcing packets to be processed in the order they arrived at *sw* [7], or including extra metadata with packet clones [14].

## 5. P2P TRANSFER

Packet reprocessing can reduce the latency overhead imposed on packets that arrive during a loss-free move, but it cannot reduce the number of packets that must be processed during the move. To achieve the latter, we must reduce the time it takes to complete a state transfer.

Our key insight is to *avoid triangular routing*. In other words, we transfer state and packets directly from *srcInst* to *dstInst* without passing through the controller.

[6]Or the updated state is not critical for achieving the NF behavior we desire: e.g., only statistics are updated.

[7]A stateful firewall may reset a timeout value every time it receives a packet, but this timeout is generally much longer than a move operation, such that we can wait to reset the timeout until after *dstInst* starts processing all packets.

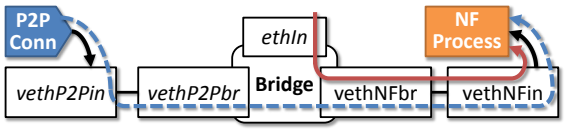



**Figure 3: Setup for injecting packets from P2P connection into *dstInst*'s input stream; solid red path is taken by normal packets, dashed blue path is taken by packets that require reprocessing**

**New Functions.** Our P2P transfer algorithm relies on two new NF-facing API functions:

```
void transfer(dstInst,fspace)
void accept(srcInst)
```

The latter is invoked on *dstInst* to indicate it should: (*i*) listen for an incoming socket connection from *srcInst*, (*ii*) import any state received over the connection, and (*iii*) buffer and reprocess any packets received over the connection. The former is invoked on *srcInst* to indicate it should: (*i*) connect to *dstInst*, (*ii*) export and send any state pertaining to active flows in *fspace*, and (*iii*) send packets that need to be reprocessed. Both functions can be implemented in an NF-agnostic manner using the existing export (`get`), import (`put`), and event raising functions included in OpenNF's original NF-facing API [7].

**Injecting packets.** Although the new functions are relatively simple, we face a key systems engineering challenge on *dstInst*: *how are packets arriving on the P2P connection injected into dstInst's packet input stream?* Normally *dstInst* reads raw packets directly from a network interface (*ethIn*). OpenNF's original controller-driven design takes advantage of the packet output capabilities of SDN switches to inject packets into the link to *dstInst* such that the packets arrive at *ethIn*. However, unless *srcInst* or *dstInst* establishes a control channel with the switch, we cannot leverage this same design in a P2P transfer.

Instead, we rely on virtual Ethernet (veth) interfaces and bridging to inject packets from the P2P connection into *dstInst*'s input stream (Figure 3). When an NF starts, we create a bridge and two veth pairs—*vethP2Pin*/*vethP2Pbr* and *vethNFbr*/*vethNFin*. The interfaces in a veth pair are virtually wired together, such that a packet sent on one veth interface is received by the other veth interface. We add *vethP2Pbr*, *vethNFbr*, and *ethIn* to the bridge. We then configure the NF to read packets from *vethNFin*.

When a packet arrives on *ethIn*, it passes through the bridge to *vethNFbr*, causing it to be received by *vethNFin* and read by the NF (solid red path in Figure 3). When a packet arrives on a P2P connection—or a packet that previously arrived on a P2P connection is released from a buffer—we send the packet on *vethP2Pin*; the packet is received by *vethP2Pbr*, sent across the bridge to *vethNFbr*, received by *vethNFin*, and read by the NF (dashed blue path).

## 6. EVALUATION

We now evaluate the efficiency, scalability, and safety improvements resulting from packet reprocessing and P2P transfer. We answer three important questions: (*i*) How much does packet reprocessing reduce latency overhead? (*ii*) Can P2P transfer reduce state transfer times? (*iii*) Does our algorithm for recovering from buffer overflow provably guarantee loss-freedom?

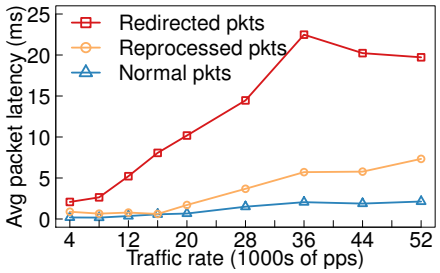



**Figure 4: Improvements in latency overhead**

We conduct our evaluation using a modified version of OpenNF [1] and two popular NFs—Bro IDS [11] and PRADS asset monitor [2]. Our testbed has an OpenFlow-enabled HP ProCurve 6600 switch and a dozen low-end servers (4-core Intel Xeon 2.8GHz CPU, 6GB RAM, 2 x 1 Gbps NICs). We generate traffic by replaying a university-to-cloud network traffic trace [8].

## 6.1 Packet Reprocessing

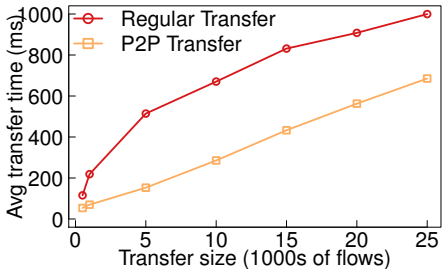
We evaluate the latency benefits of packet reprocessing using a slightly modified version of the Bro IDS that outputs packets after they are analyzed. We conduct a loss-free move affecting 800 active flows while replaying traffic at varying rates—up to 52K packets per second (0.4Gbps). We measure per-packet latency as the time elapsed between a packet arriving at *sw* and a packet begin output by an IDS instance. We compare the latency of: (*i*) packets processed normally by *srcInst* prior to the move, (*ii*) packets in *fspace* that are both processed and output by *srcInst* and reprocessed by *dstInst*, and (*iii*) packets in *fspace* that arrive during the move but are only processed by *dstInst*. The last case captures the latency overhead imposed by OpenNF's original design.

Figure 4 shows the average packet latency in each of the three cases—labeled *normal*, *reprocessed*, and *redirected*, respectively. We first observe that the latency for normal packets plateaus, and the latency for redirected packets dips, when the traffic rate is above 36K pps (0.3Gbps). This is the point where we exceed the packet processing capacity of a single Bro instance. More importantly, below this rate, we observe that packet reprocessing offers a 58% to 92% reduction in packet latency overhead compared to OpenNF's original design. Although, packets in *fspace* that arrive during a move operation still incur an ≈2x inflation in latency compared to packets processed prior to the move. This overhead stems from the NF using some of its resources to export state.

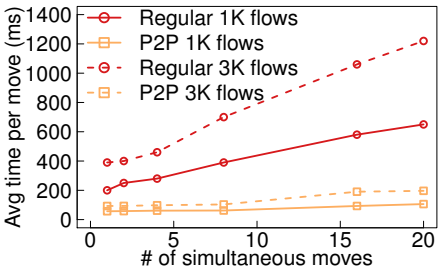
## 6.2 P2P Transfer

We now evaluate the benefits of a P2P transfer using the PRADS asset monitor [2]. We move state for varying numbers of flows between two PRADS instances and measure the time required to transfer state for all flows.

We first examine the improvements for move operations conducted in isolation. Figure 5(a) shows the time to complete the state transfer for varying numbers of flows with and without P2P transfer; each data point is averaged across three iterations. We observe that P2P transfer reduces the average state transfer time by at least 31%, and up to 70%, depending on the amount of state transferred.




(a) Single operation




(b) Simultaneous operations

**Figure 5: Improvements in state transfer time**

Next, we examine the improvements for move operations conducted simultaneously. Figure 5(b) shows the average sate transfer time per operation when between 1 and 20 operations occur simultaneously. We observe the transfer time per move is near constant with P2P transfers, regardless of the number of flows affected by each move operation. In contrast, without P2P transfers, the state transfer time steadily increases as the number of simultaneously operations increases.

## 6.3 Safe Buffer Overflow

Lastly, we prove that our algorithm for handling buffer overflow (Section 4) is loss-free. We assume *srcInst* and *dstInst* have some CPU and network headroom to accommodate the transfer of state and packets. We believe this is a reasonable assumption since networks generally run well below 100% utilization.

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 state for $f$ after processing $\langle p \rangle_{i,j}$ starting from initial state $S_{init}$.

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

Of the packets sent to *srcInst*, let $p_{h-1}$ $(1 < h < k)$ be the last packet for $f$ processed by *srcInst* before exporting state for $f$. Then, $S_{1,h-1}$ will be transferred to *dstInst*, and all packets $\langle p \rangle_{h,k-1}$ will be sent to *dstInst* for reprocessing. If no buffer overflow occurs, all packets $\langle p \rangle_{h,k-1}$ will arrive at *dstInst* and be processed, with $S_{1,h-1}$ as the initial flow state. The resulting state will be $S_{1,k-1}$. Packets $\langle p \rangle_{k,n}$ will arrive directly from *sw* and be processed, resulting in the desired state $S_{1,n}$.

Now assume buffer overflow occurs. Let $p_{i-1}$ $(h < i < k)$ be the last packet for $f$ buffered at *dstInst* prior to overflow. This implies *dstInst* will reprocess $\langle p \rangle_{h,i-1}$, with $S_{1,h-1}$ as the initial flow state, resulting in state $S_{1,i-1}$ at *dstInst*. This also implies *srcInst* has processed $\langle p \rangle_{h,i-1}$, resulting in state $S_{1,i-1}$ at *srcInst*.

Let $p_{j-1}$ $(i < j < k)$ be the last packet for $f$ processed by *srcInst* before re-exporting state for flow $f$. Then, $S_{1,j-1}$ will be transferred to *dstInst* and replace whatever state *dstInst* currently has for $f$. Now, all packets $\langle p \rangle_{j,k-1}$ will be sent to *dstInst* for reprocessing. If no further buffer overflow occurs, all packets $\langle p \rangle_{j,k-1}$ and $\langle p \rangle_{k,n}$ will arrive at *dstInst* and be processed, with $S_{1,j-1}$ as the initial flow state. The resulting state will be $S_{1,n}$, implying move is loss-free. □

# 7. DISCUSSION

Although we have focused on OpenNF [7], the same ideas apply to frameworks like Split/Merge [13]. Split/Merge uses a central controller to buffer packets during a move operation, so high controller resource demands, safety violations due to buffer overflow, and high latency overhead are all problems in this framework. Packet reprocessing can be implemented in Split/Merge by: (*i*) instructing the SDN switch to send packets to both *srcInst* and the controller during the move operation, and (*ii*) modifying NFs, as described in Section 4, to suppress output while reprocessing packets released from the controller's buffer. Split/Merge already transfers internal NF state in a P2P manner, but we can add P2P-like transfer of packets by instructing the SDN switch to mark and send packets to *dstInst*, rather than the controller, during a move operation.

# 8. CONCLUSION

Frameworks like Split/Merge [13] and OpenNF [7] make it possible to move internal NF state to safely accommodate dynamic redistribution of packet processing. However, we have shown these frameworks suffer from safety, efficiency, and scalability problems due to their reliance on packet events and packet buffering. To address these issues, we presented two novel enhancements: (*i*) *packet reprocessing* processes packets twice to avoid delaying NF output, while still ensuring future packets are processed using the latest NF state; and (*ii*) *P2P transfers* avoid triangular routing by directly transferring state and packets between NF instances. These two mechanisms reduce packet latency overhead by up to 92% and reduce the average state transfer time by up to 70%. These critical enhancements pave the way for broader adoption of NF state management frameworks.

# 9. ACKNOWLEDGEMENTS

We would like to thank Mark Coatsworth and Chaithan Prakash for their assistance in implementing our enhancements in OpenNF. We would also like to thank the anonymous reviewers for their insightful feedback. This work is supported by the Wisconsin Institute on Software-Defined Datacenters of Madison and National Science Foundation grants CNS-1302041, CNS-1330308 and CNS-1345249. Aaron Gember-Jacobson is supported by an IBM PhD Fellowship.

# References

[1] OpenNF code. http://opennf.cs.wisc.edu/code.

[2] Passive Real-time Asset Detection System. http://prads.projects.linpro.no.

[3] Network Functions Virtualisation – Introductory White Paper. http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf, 2012.

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

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

[6] A. A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. R. Kompella. Towards an elastic distributed SDN controller. In *HotSDN*, 2013.

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

[8] 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.

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

[10] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson. Pratyaastha: An efficient elastic distributed SDN control plane. In *HotSDN*, 2014.

[11] V. Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security (SSYM)*, 1998.

[12] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A high availability framework for middleboxes. In *SoCC*, 2013.

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

[14] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Macciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback recovery for middleboxes. In *SIGCOMM*, 2015.

[15] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE*, 2011.

[16] S. H. Yeganeh and Y. Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *HotSDN*, 2012.