Centrally Controlled Distributed VNF State Management

Babu Kothandaraman
Royal Institute of Technology
Stockholm, Sweden
babuk@kth.se

Manxing Du
Acreo Swedish ICT AB
Stockholm, Sweden
mandu@acreo.se

Pontus Sköldström
Acreo Swedish ICT AB
Stockholm, Sweden
ponsko@acreo.se

ABSTRACT

The realization of increased service flexibility and scalability through the combination of Virtual Network Functions (VNF) and Software Defined Networks (SDN) requires careful management of both VNF and forwarding state. Without coordination, service scalability comes at a high cost due to unacceptable levels of packet loss, reordering and increased latencies. Previously developed techniques has shown that these issues can be managed, at least in scenarios with low traffic rates and optimistic control plane latencies. In this paper we extend previous work on coordinated state management in order to remove performance bottlenecks, this is done through distributed state management and minimizing control plane interactions. Evaluation of our changes show substantial performance gains using a distributed approach while maintaining centralized control.

CCS Concepts

•**Networks** → **Middle boxes / network appliances; Programmable networks;** Peer-to-peer protocols;

Keywords

Scalable network functions; middleboxes; software-defined networking

1. INTRODUCTION

There has recently been an interest in replacing dedicated hardware Network Functions (NFs, or middleboxes) with software-based virtualized counterparts, with the goal of achieving more flexible and scalable services[3, 5]. By combining flexible compute resource allocation through e.g. OpenStack[4] with the flexible traffic steering capabilities provided by SDN, elastic software-based middleboxes that can grow or shrink on-demand seems to be on the horizon. However, there are some technical challenges left before these systems are ready for a production environment, in this paper we focus on the challenge of coordinated network and VNF state management during a scale-in or -out event. As others have pointed out these two types of states has to be synchronized to avoid service degradation. For certain VNFs such as a stateless NAT, only the configuration state has to be taken into account by e.g. configuring appropriate address mapping rules before directing user traffic to the new instance. In a stateful NAT the transient state created by the traffic itself has to be transferred before user traffic reaches the VNF to avoid that existing connections over the NAT are disconnected. Other VNFs may in addition also be sensitive to packet loss or reordering during the transfer process, e.g. packet reordering and loss in an IDS VNF may trigger false positives or fail to trigger on real positives [7].

While there are several methods for dealing with network state or VNF state separately [6, 8, 10], there are only a few that handle them in a coordinated manner [7, 12, 11]. In this paper we focus on improving OpenNF [7] as it is the most feature rich of the methods, able to guarantee loss-free and order preserving packet and state transfers. Our implementation improves OpenNF by removing the OpenFlow / OpenNF controller from the critical path during state and traffic transfer, instead state and packets are transfered in a peer-to-peer fashion between VNFs. The next section gives a brief introduction to OpenNF, for more details see [7].

2. OPENNF ARCHITECTURE

The OpenNF architecture consists of two parts, 1) a shared library that is linked with the VNF application on the data plane and 2) a control application running on a controller. The shared library provides an API with methods for exporting and importing different types of state from a VNF instance and to enable generation of various events. The control application runs on an SDN controller and is responsible for coordinating the transfer of both network and VNF state using the OpenFlow protocol and OpenNF protocol respectively. In OpenNF all VNF state is associated with the flow(s) that updates a particular chunk of state, either as a one-to-one mapping for an individual flow, a group of flows or for all flows, these could e.g. be a packet counter for an individual IP address, for an IP subnet, and for all IP packets respectively. Grouping the state into these categories and associating the state with the flow(s) allows OpenNF to easily export the appropriate state when a certain set of traffic flows are to be moved to another instance.

2.1 Data plane VNF API

The data plane API is implemented using JSON over TCP, relevant commands are shortly summarized below.

HotMiddlebox'15, August 17-21, 2015, London, United Kingdom
© 2015 ACM. ISBN 978-1-4503-3540-9/15/08...$15.00
DOI: http://dx.doi.org/10.1145/2785989.2785996

Here *SrcVNF* refers to the source of the state being transferred and *DstVNF* where it should be placed.

State export/import To export state the command *getPerFlow(filter)* is sent to SrcVNF, the *filter* defines which flow(s) the command is referring to, and by extension which state should be exported. State chunks are returned to the controller in *statePerFlow* messages. To import state the message *putPerFlow(map<flowid, chunk>)* is sent to DstVNF, containing a map of flow identifiers and their state chunks. Similar commands exist for handling state associated with multiple or all flows, and to delete state.

Events Events are enabled and disabled by the controller to handle data traffic during the VNF state transfer. At the SrcVNF *enableEvents(filter, drop)* is used to encapsulate and redirect packets to the controller without further processing. At DstVNF *enableEvents(filter, buffer)* is used to redirect and buffer packets.

2.2 Controller functionality and API

The controller in turn uses the data plane API to provide different operations that control applications can utilize to manage VNF and network state, our focus is on the *Move(src, dst, filter, scope, prop)* operation which transfers both VNF state and traffic to another VNF instance. The *filter* parameter defines which network flows (and which state) the operation refers to, *prop* is used to select guarantees. The *Move* operation The different guarantees provided by *Move* are:

No guarantees (NG) All packets arriving during the state transfer are processed by the SrcVNF, state may be unsynchronized after the operation.

Loss-free (LF) Packets belonging to flow(s) currently being moved are redirected using *events(filter, drop)* to the controller where they buffered until the VNF state transfer finishes. When completed buffered packets are sent to DstVNF via the switch using the OpenFlow Packet-Out command. Finally the switch is updated to send traffic directly to DstVNF.

Order preserving (OP) Extends LF with packet stream synchronization using *event(filter, buffer)* at DstVNF and a two phase forwarding update that ensures that all redirected packets are processed at DstVNF before packets arriving from the switch.

The LF and OP versions of the *Move* operation achieves state transfer without packet loss or re-ordered packets which e.g. Split/Merge [12] could not. However, these guarantees come at a cost in packet latency and control plane overhead as data plane packets are redirected and buffered at the controller.

2.3 Optimizations

OpenNF implements three optimizations for the *Move* operation:

Parallelize (PZ) Immediately send received state to DstVNF without waiting for GetPerFlow to finish.

Late Locking (LL) Packets arriving at SrcVNF are redirected to the controller on a per-state basis, redirection only happens if the associated state has been sent to the controller.

Early release (ER) Packet redirection from the controller to DstVNF is performed on a per-connection basis. Instead of waiting for the full state transfer to complete, packets are sent to the DstVNF if corresponding state has been sent and acknowledged by DstVNF. LF *Move* with all optimizations can be seen in Figure 1.

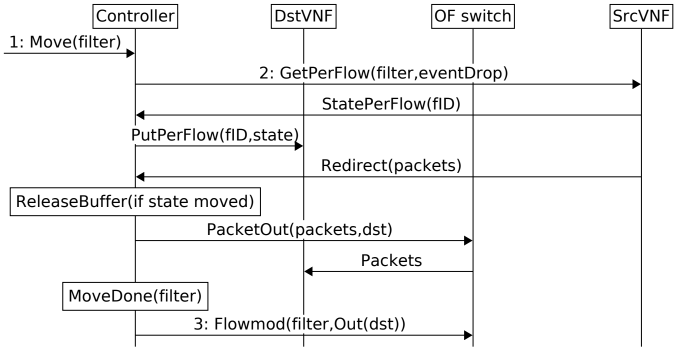

Figure 1: LF OpenNF*Move* with PZ|LL|ER.

Both LL and ER will change the order of packets between flows, while maintaining the order within flows. This reduces the order preserving guarantee to be order preserving only within flows, which may be significant for some VNFs.

2.4 Bottlenecks in OpenNF

Our main concern with the OpenNF protocol is the use of the controller as a proxy during VNF state transfer and redirection of data packets during both LF and OP *Move*. We foresee four problems with this approach, **1)** risk for controller and **2)** control network overload, **3)** increased data packet latency, and **4)** low scalability.

Controller overload is a risk as potentially large amounts of state data and packet flows has to traverse the controller, placing a high load on it as incoming state and traffic has to be processed and/or buffered before being forwarded to the DstVNF. A high load is also placed on the control network, in particular the OpenFlow control channel which is used to send Packet-in/Packet-out messages, the OpenFlow control channel typically does not support high packet rates.

The additional hop between the VNFs through the controller and the low performance of the OpenFlow control channel adds latency to the transfers and increases the time it takes to transfer state and data, in turn causing more data packets to be buffered. This additional latency and buffering increases the latency experienced by data flows during the transfer, which could affect any latency and/or jitter sensitive applications generating the data flows. Finally, due to the reliance on the controller, OpenNF does not scale well with increasing amount state and traffic bandwidth.

3. DISTRIBUTED STATE TRANSFER

To address the four problems from the previous section we extended OpenNF with Distributed State Transfer (DiST) to avoid passing VNF state and data packets through the controller, which also removes the need for packet buffering in the controller. With DiST, the problems of OpenNF are addressed as follows:

1-2) DiST uses the control network only for signalling, state transfer and packet redirection is done directly between VNFs over data plane links. This reduces both the risk of controller and control network overload.

3) By using the data plane for state transfer and redirection

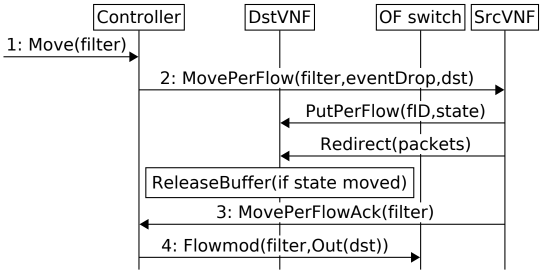
Controller
DstVNF
OF switch
SrcVNF
1: Move(filter)
2: MovePerFlow(filter,eventDrop,dst)
PutPerFlow(flD,state)
Redirect(packets)
ReleaseBuffer(if state moved)
3: MovePerFlowAck(filter)
4: Flowmod(filter,Out(dst))

Figure 2: LF DiST *Move* with PZ|LL|ER.

of packets, and by reducing number of messages involved in state transfer, DiST reduces the state transfer time and as well as the number of redirected packets compared to OpenNF. Redirection through the data plane and buffering at DstVNF instead of the controller reduces the latency experienced by moved data flows compared to OpenNF.

4) The controller only handles control messages to and from VNFs and switches, with no involvment on actual state or data packets, improving scalability.

3.1 DiST Protocol

To achieve peer-to-peer state transfer and redirection, DiST reused most functionality in OpenNF with minor changes. At controller side, it is simplified to only handling signallings to/from VNFs without being involved in state transfer. The data plane API has been extended to include DstVNF address in appropriate messages for state transfer and redirection. Buffering techniques are implemented at DstVNF to facilitate LF and OP guarantees. The changes are explained in detail for different guarantees below:

DiST No guarantees At controller, the initial *GetPerFlow(filter)* in OpenNF is replaced with a *MovePerFlow(filter, dst)* message which is sent to SrcVNF, it contains the IP of DstVNF. SrcVNF establishes a connection to DstVNF and uses *PutPerFlow* to transfer state. DiST avoids usage of statePerFlow messages as in OpenNF bypassing the controller for handling state. When all state for the filter has been transferred, SrcVNF acknowledges the *MovePerFlow* to the controller which updates forwarding in the switch.

DiST Loss-free with PZ The LF operation starts by sending an *enableEvent(filter, drop, DstVNF)* message to SrcVNF, which initiates a TCP connection with DstVNF and creates a filter for redirecting packets to DstVNF where they are buffered. The only change compared to original OpenNF is the inclusion of the DstVNF IP in the enableEvent message. Once *enableEvent* is acknowledged the controller sends a *MovePerFlow* message to SrcVNF and state transfer starts. SrcVNF redirects packets (as events) to DstVNF which buffers them in a hash table. Once all affected state has been transferred SrcVNF sends a ReleaseBuffer command to initiate processing of the buffered packets. SrcVNF finally acknowledges the *MovePerFlow* command to the controller, which updates forwarding in the switch.

DiST Loss-free with PZ|LL|ER Each state for which transfer has started is marked and arriving packets associated with marked state are redirected to DstVNF. Packets associated with state for which transfer has not yet started are processed directly by SrcVNF. DstVNF in turn maintains a hash table that buffers packets per-flow. When all state for a flow has been transferred DstVNF starts processing packets from that flows bucket in the hash table. If state for a flow has been transferred and there are no packets in the buffer, redirected packets are not buffered but processed immediately. A message sequence chart for this flavor of *Move* is depicted in Figure 2.

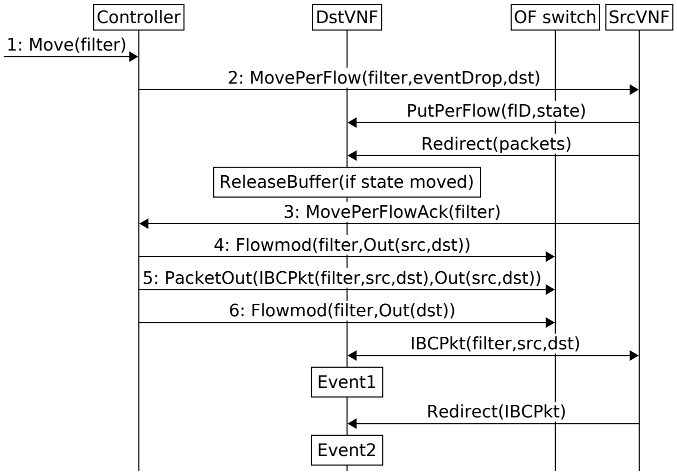
Controller
DstVNF
OF switch
SrcVNF
1: Move(filter)
2: MovePerFlow(filter,eventDrop,dst)
PutPerFlow(flD,state)
Redirect(packets)
ReleaseBuffer(if state moved)
3: MovePerFlowAck(filter)
4: Flowmod(filter,Out(src,dst))
5: PacketOut(IBCPkt(filter,src,dst),Out(src,dst))
6: Flowmod(filter,Out(dst))
IBCPkt(filter,src,dst)
Event1
Redirect(IBCPkt)
Event2

Figure 3: OP DiST *Move* with PZ|LL|ER.

DiST Order-preserving with PZ|LL|ER Extends the LF procedure with a two-phase forwarding update to ensure that redirected packets from SrcVNF are processed before packets forwarded from the switch. Synchronizing the two packet streams (one redirected from SrcVNF and the other forwarded from the switch) is done using an "InBand Control packet" (IBCPkt). The IBCPkt must be crafted differently for each *filter* and also depends on how the VNF processes packets, the IBCPkt must match the filter definition in both VNFs but also be distinguishable from normal data packets. A similar problem exists in many OAM protocols in which probe packets must be fate-sharing with the traffic flow they measure but still be distinguishable from user traffic. The two-phase update starts by updating the switch forwarding rule for *filter* to duplicate traffic to both Src- and DstVNF, followed by a Packet-Out message sending the IBCPkt to both VNFs at once. Assumes that the IBCPkt is inserted at the same place in both packet streams, this may depend on the OpenFlow switch implementation. Finally the switch forwarding rule is updated to send *filter* traffic only to DstVNF.

DstVNF will now receive two IBCPkts, one directly from the switch and the other redirected by SrcVNF. If the IBCPkt arrives first from the switch (Event1) DstVNF starts buffering packets from the switch (Packets matching *filter* coming from the switch before IBCPkt are dropped, these will also arrive from SrcVNF) while still processing redirected packets from SrcVNF, until the second IBCPkt arrives. At that point processing of redirected packets stops and processing of buffered packets from the switch starts. If an IBCPkt arrives from SrcVNF before the IBCPkt from the switch (Event2) processing of redirected packets from SrcVNF is stopped. Once the second IBCPkt arrives processing of packets from the switch is started. The duplication of traffic may seem unnecessary here, however it is needed to avoid packet loss for packets arriving between the Packet-Out sending the IBCPkt and the final rule update

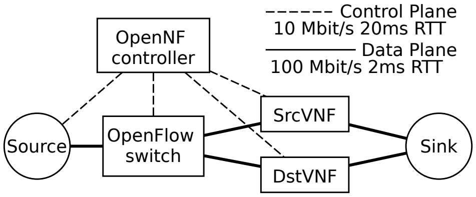

Figure 4: Testbed setup

getting (even though SrcVNF redirects them, they arrive after IBCPkt so they are dropped at DstVNF). A message sequence chart for order-preserving *Move* is depicted in Figure 3.

4. DiST EVALUATION

We evaluate our initial implementation of DiST in the testbed configuration shown in Figure 4. Based on an assumption of roughly one order of magnitude performance difference between control and data plane networks the data plane links between switches and VNFs are limited to 100 MBit/s bandwidth with a 1 ms one-way delay, control plane links are set to 10 MBit/s with 10 ms one-way delay. As VNF we use the PRADS implementation from the authors of [7]. Currently only the NG and LF DiST extensions to *Move* are implemented, there we compare these with the original OpenNF versions by replaying a live network traffic trace using tcpreplay[2] and after 20 seconds we initiate a *Move*. We replayed the traffic at 100, 500, 1000, 2500, and finally 5000 pps. At 2500 pps the OpenNF SrcVNF starts dropping incoming packets during *Move*, the same happens in DiST at 5000 pps. Both are likely due to locking of shared data structures between the packet processing and state transferring threads. For a fair comparison we therefore focus on the results where no packet loss occours, at 1000 pps.

4.1 Controller load

In the original OpenNF LF/OP *Move* the total amount of messages sent is approximately $3N + 2R + C$ (including ACKs) where N represents the number of states, R the number of redirected packets, and C a constant for each *Move* type (typically < 10). All these messages are either sent or received by the controller. With the DiST extension the number of messages is approximately $2N + R + C$, since we don't need two messages to transfer a state chunk nor two messages to redirect a packet. However, in DiST only C messages concern the controller, all other messages are sent between VNFs. While the cost of the DiST *Move* still scales with the amount of state to transfer and the number of redirected packets, that cost is placed at the VNFs, keeping the cost at controller constant.

4.2 Operation Time

Some of the data we collected is shown in Table 1, these are results at 1000 pps which corresponds to about 4 MBit/s. *MoveTime* is the time from start of the *Move* operation on the controller until the switch forwarding update, as expected the MoveTime for the original OpenNF solution is about 3 to 6 times higher than DiST for the PZ|LL|ER case. This is mostly explained by the latency differences on the control vs. data plane, but the reduced number of messages in DiST contributes as well. To separate the contribution from latency versus number of messages we also ran OpenNF with control plane latency same as data plane, this case is marked with an asterisk. MoveTime in LF mode with PZ|LL|ER versus number of state chunks transferred for the different pps values is shown in Figure 5 on the left side, on the right side is the ratio between OpenNF and DiST MoveTime. As can be seen, DiST is always faster than OpenNF. Looking at the ratio for OpenNF* it is clear that the reduced number of message makes DiST about 50% faster even with a control plane as fast as the data plane.

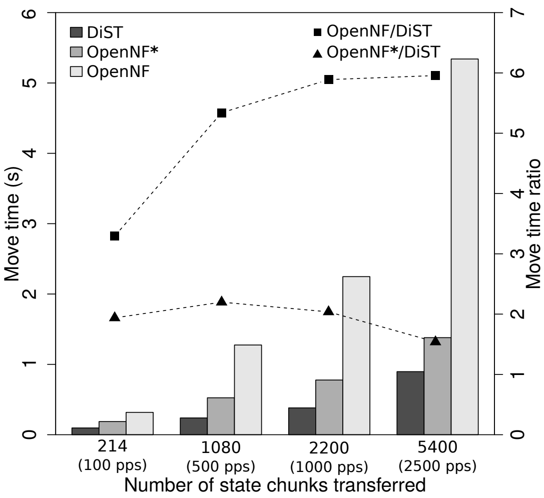

Figure 5: MoveTime for different number of state chunks (left). Ratio between OpenNF and DiST (right), all for LF with PZ|LL|ER.

Looking at Table 1 it is clear that the LL and ER optimizations are very effective at reducing the number of redirected packets. In DiST these optimizations reduce the MoveTime by about 20% at 1000 pps, whereas for OpenNF the reduction is about 70%. However, the effectiveness of these optimizations depends on the composition of the incoming traffic.

RedirTime is the time from initiating state transfer at the SrcVNF until the last packet is redirected. We measured RedirTime in order to see how long time packets are still being redirected after the VNF state has been moved. This can be significant if we e.g. shut down SrcVNF once we believe *Move* is completed. As can be seen, in some cases RedirTime is longer than MoveTime indicating that RedirTime should be considered when determining if the *Move* has completed or not.

We also measured *Ser* and *Deser* which is the percentage of MoveTime spent (de-)serializing data in the VNF during state export and import respectively. As can be seen a large part of the MoveTime in DiST is spent on serializing, showing another bottleneck to be removed.

4.3 Traffic Pattern at Sink

Figure 6 depicts the traffic pattern of the processed packets at Sink for LF *Move* with PZ|LL|ER at 1000 pps. The dotted boxes indicate the start and end of *Move*. With PZ|LL|ER the SrcVNF should continue to process packets during the operation and hence the packet rate should not be affected, which is observed for DiST. In original OpenNF the traffic rate drops to zero in the beginning of *Move*, likely due to shared resources being locked by the state transfer thread when the *Move* is initiated. This in turn causes incoming packets to be queued, the release of the lock explains

Config	# States	# Redirected	MoveTime(s)	RedirTime(s)	Ser%	Deser%
OpenNF LF PZ	2186±1	6678±179	7.44±0.21	8.11±0.17	2.63	1.95
DiST LF PZ	2184±2	489±39	0.51±0.06	0.82±0.04	40.22	33.01
OpenNF LF PZ|LL|ER	2186±1	17±5	2.25±0.04	1.51±0.05	7.99	6.67
DiST LF PZ|LL|ER	2204±2	6±1	0.41±0.04	0.41±0.21	48.63	40.81
(OpenNF* LF PZ|LL|ER)	2187±1	6±2	0.78±0.02	0.96±0.17	26.83	22.07

Table 1: Data gathered from experiments at 1000 pps, with 95% confidence intervals. OpenNF* indicates equal latency in data and control plane.

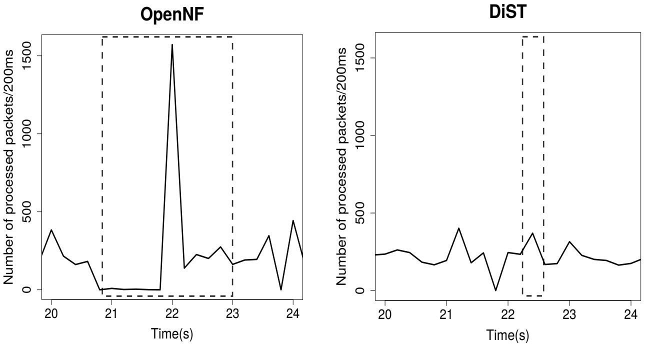

Figure 6: Traffic rates at Sink for OpenNF and DiST, with PZ|LL|ER. The dotted boxes indicate when the *Move* operation was active.

the spike in the traffic pattern. Moreover, these packets updated the state in SrcVNF instead of DstVNF, leading to unsynchronized state. Table 1 shows that OpenNF transfers the same number of states regardless of optimization while DiST PZ|LL|ER also transfers states created at SrcVNF during the operation.

5. CONCLUSIONS

Distributing the Move operation significantly reduces the amount of messages exchanged during the operation, halving the amount of messages per redirected packet and removing a third of the messages per state chunk transferred. Additionally, only a small, constant, number of messages traverse the control network to put load the controller, increasing the scalability of the system as a whole. In a scenario with a control plane that has less performance than the data plane these changes show a substantial performance gain, being roughly 3 times faster at 100 pps, 5 times at 1000 pps, and 6 times at 2500 pps. These performance values are however depending on the assumption of 10 times higher latency on the control plane, how accurate this assumption is depends on many factors. There are however good reasons to be suspicious of control plane performance, e.g. in some switches Barrier messages can cause up to 400 ms control plane latency [9].

6. FUTURE WORK

We observed packet losses at 5000 pps (about 20 MBit/s) with DiST, while the PRADS VNF seems to be capable of at least 15000 pps (about 60 MBit/s) during normal execution. We believe the reason for these losses is lock contention between the packet processing thread and the state transferring thread, combined with the extra load of packet redirection. One solution could be to change the order of the steps in *Move* and start by re-routing traffic to DstVNF and perform buffering there, and then transfer VNF state. This would reduce contention as the packets associated with the state we are transferring would arrive at DstVNF rather than at SrcVNF. Another benefit is that SrcVNF does not have to redirect packets, reducing its load. An OP version of this solution is illustrated in Figure 7.

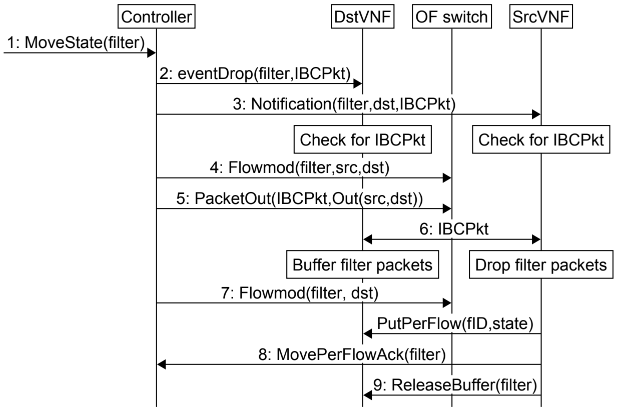

Figure 7: OP DiST *Move* with redirection and buffering first.

One negative effect of redirecting traffic before moving state is that we cannot implement the Late Locking optimization, packets that with the LL optimization active would be processed at SrcVNF will instead be buffered at DstVNF. The Early Release optimization can however be implemented even in this scenario.

Without Late Locking in this alternative solution we risk buffering flows at DstVNF for a long time while waiting for their VNF state to be transferred, inducing long latencies for those flows. Individually transferring smaller flows instead of grouping them into a single large *Move* could reduce the impact of this (e.g. performing 255 *Move* operations on /16 IP subnets instead one *Move* operation on a /8 IP subnet). The order of moving flows within larger *Move* operation could also be done based on amount of state associated with the matched flows, for e.g. flows with less state can be moved and traffic can be rerouted before moving flows with more state. Even network conditions can be used for deciding which flows should be moved first, e.g. moving flows with a larger traffic load first in order to offload the SrcVNF quickly. However, many *Move* operations would consume more flow rule entries in the switch and likely increase the total time.

Acknowledgment

This work was conducted within the framework of the FP7 UNIFY project, which is partially funded by the Commission of the European Union. In addition one author recieved funding from a Swedish Institute scholarship for studies at KTH.

7. REFERENCES

[1] Comparing various aspects of serialization libraries. https://code.google.com/p/thrift-protobuf-compare/wiki/BenchmarkingV2. Accessed: 2015-04-01.

[2] Tcpreplay: Pcap editing and replay tools for* nix. http://tcpreplay.synfin.net. Accessed: 2015-04-01.

[3] The UNIFY project. http://fp7-unify.eu. Accessed: 2015-04-01.

[4] Openstack: Open source cloud computing software, 2014.

[5] ETSI. White Paper: Network Functions Virtualisation (NFV), 2013.

[6] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. *arXiv preprint arXiv:1305.0209*, 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 *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 163–174. ACM, 2014.

[8] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. *ACM SIGCOMM Computer Communication Review*, 38(4):51–62, 2008.

[9] M. Kuzniar, P. Peresini, and D. Kostic. What you need to know about sdn control and data planes. Technical report, 2014.

[10] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 27–38. ACM, 2013.

[11] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 1. ACM, 2013.

[12] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, pages 227–240, 2013.