Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

Statelet-Based Efficient and Seamless NFV State Transfer

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer

***Abstract*—Network Functions Virtualization (NFV) environments can provide increased elasticity and flexibility for operators, as network-related services can be scaled and moved as needed. Such operations require a seamless transfer of state to provide a service without interruption or performance degradation. In this paper, we propose and analyze a novel approach for efficient and seamless NFV state transfers. Our approach is based on the concept of statelets, which are compact representations of information in incoming packets that change the state of a Virtualized Network Function (VNF). We present and describe SliM (Slim Migration), a system for efficient NFV state transfers using a statelet interface that we have implemented as an add-on to the Data Plane Development Kit (DPDK), a high-performance packet I/O library. We have evaluated SliM in a testbed and present results that show its benefits in terms of lower delays and lower packet-loss rates. Moreover, our analysis and evaluation results show that SliM offers three times as much shared link capacity to the dataplane as previous approaches, while reducing the duration of a state transfer to one third of their time.**

I. Introduction

Network Functions Virtualization (NFV) is a concept that applies cloud computing principles to the design, deployment, and management of network services. It enables fast and elastic provisioning of network functions (NFs), also known as *middleboxes*, and has the potential of significantly reducing the operator's cost and increasing flexibility by deploying NFs on multi-purpose commodity hardware instead of purpose-built hardware. An instance of a *Virtual Network Function (VNF)* deployed on an NFV infrastructure can be migrated from one physical node to another, e.g., to optimize traffic flows in the network. It can also be *split* into multiple instances, each deployed on a different physical node, e.g., to *scale out* the NF and better cope with increasing demands.

Migration techniques widely used for web services in cloud environments conceptually take a snapshot of a service instance, transfer the snapshot to the new destination, and resume the service from the new instance [1] [3]. To avoid state inconsistencies, the service is typically halted during the parts of this procedure and messages arriving during the migration are buffered and processed after the new instance resumes. This approach is badly suited for the migration of VNFs and their stringent requirements on jitter and packet loss; VNF migration requires *seamless* operation so that a user of a network service does not perceive any disruption nor performance degradation even during the migration process.

Therefore, existing VNF migration techniques commonly transfer a snapshot of the state of a servicing instance to a new instance and subsequently *resynchronize* the inconsistent state; a successful and timely resync process is crucial as it enables the destination instance to seamlessly continue the operation of the NF with its most recent state. However, existing approaches are not well-suited especially for the high-volume resync traffic of bandwidth-intensive NFs, which may slow down the resync process significantly, or even make it impossible to resync at all under adverse conditions. To make matters worse, both the resync traffic and the actual dataplane traffic of the NFs may compete for the same network bandwidth resources, and prioritizing one over the other does not improve matters. This problem becomes more apparent as we consider current trends towards NFV at the edge such as *provider edge clouds*, virtual CPEs, and *fog computing*, which require VNF migration across shared and bandwidth-constrained network links.

To address the lack of efficient solutions for a seamless NFV state transfer, we propose a novel approach based on *statelets*, which are compact representations of the information in incoming packets relevant for a VNF's state change during the migration process. This approach enables us to reduce the volume of the resync traffic, effectively leaving more bandwidth share to the datapath traffic and shortening the resync duration. This paper builds on and advances the basic idea and concept behind our statelet approach [16] with the following contributions:

- We describe the details of the statelet approach.
- We present and analyze the model of our SliM system and how it implements the statelet approach.
- We introduce a system implementation of SliM including a DPDK library for statelet announcement and installation, which allows VNF developers to benefit from DPDK's performance enhancements.[1]
- We implement two widely used VNFs to evaluate our system in a testbed, the results of which confirm our analysis and demonstrate that SliM NFs can achieve 3-fold increase in bandwidth utilization while reducing the duration of the state transfer to one third of previous work.

We explain the system model and the migration procedure in Section IV, and subsequently analyze and quantify the the problem and our proposed approach in Section V. After describing the system implementation and platform in Section VI, we discuss the evaluation setup and the evaluation results in Section VII, respectively.

[1] Available on GitHub: https://github.com/nokia/SliM

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

II. Related Work

Different network management tasks require solving the problem of resynchronizing a computing instance's changing state to another instance. *State migration* requires the state of a source to be transferred to a destination on demand, the location of the destination might be spontaneously selected, based on criteria such as network optimization. Commonly, a snapshot of the source instance's current state is transferred first, but is outdated when it reaches the destination, thus, it must be resynchronized to the current state of the source afterwards. Another task is *failure recovery*, [4], [22], [23]. Here, the physical location of a backup node required can be carefully planned and pre-defined. The initial state can be proactively transferred (or is empty at boot time), but as the state in the primary instance changes, it must be continuously resynchronized with the backup instance, so that the latter can take over operation with the current state in case of failure.

Improving the performance and costs of **state migration** in *platform virtualization* environments is a thoroughly studied topic which first came to larger attention through the work of Clark et al. [3]. To migrate a virtual machine according to their approach, a snapshot of all memory pages is copied to the destination in a first round, using shadow pages in successive rounds to determine and transmit any changes that have occurred during the transfer of the previous round (Figure 1 (1.)). The approach was tested on a highly utilized web server, while observing 165ms of downtime, which is acceptable regarding most web services. The downtime here is caused by the last round, which requires a VM halt until the last page differences have been transferred. Additionally, a service degradation regarding the capacity can be observed during the page diff transfer lasting 72s.

The evaluation was conducted in an intra-datacenter environment, leading to the assumption that the latency between the source and destination VM was small during the evaluation. Several further optimizations for delta-based migration techniques have been proposed, including compression and deduplication mechanisms [1]. These have been evaluated under very heterogeneous conditions and workloads, thus the spread of the performance results is significant. The total VM migration times range between $3.6s$ and $35min$ with downtimes between $200ms$ and $3s$. Raad et al. [19] achieve sub-second downtimes transferring virtual machines *over WAN links* using the Locator/Identifier Separation Protocol (LISP).

An alternative to state delta transmission is *deterministic replay*. Here, a transferred, outdated snapshot is updated by capturing system events (like network packets) at the source and reprocessing them at the destination. H. Liu et al. [14] leverage an event capture mechanism originally proposed for intrusion detection to recover a current system state at the destination instance.

The migration of virtual network functions can be classified as a domain-specific problem of state migration of virtualized entities. A variety of network functions and its applications must guarantee a certain throughput and latency without significant packet loss. While TCP-based best-effort services can absorb the downtime of the previous approaches with retransmits and rate control, an even small downtime or a degradation of capacity can lead to perceivable service disruption when processing real-time traffic for Voice-over-IP (VoIP) or video conferencing. Therefore, various approaches have emerged for *seamless* migration in NFV environments, [5], [7], [8], [17], [21]. Besides state migration mechanisms, frameworks for VM placement and orchestration exist [2], [6], which can make use of the former and profit from them. Finally, when moving network functions seamlessly, the underlying SDN must be migrated with them [9]. Most important related work is summarized in Table I.

Olteanu et al. [17] have implemented and evaluated a ClickOS-based framework for seamless elastic scaling of a carrier-grade NAT virtual middlebox. The authors chain NAT instances for state migration, their migration approach profits from the fact that after a socket is being established, the state of the NAT implementation used becomes immutable, it does not change anymore except that it may eventually time out. The VNF chain could increase the datapath length and thus bear the risk of jitter in higher-delay networks. L. Liu et al. [15] exploit the short life of a large fraction of flows. These short-lived mice flows will eventually end at the source before the migration is supposed to end. They therefore address the challenge of identifying long-lived elephant flows and provide a new API to the VNF to identify these flows.

OpenNF [8] enables a network function to seamlessly split or merge several sessions, or completely transfer its state to other physical resources by using the aforementioned *deterministic replay* of packets after snapshot transmission. We refer to this method as **duplication**-based state resynchronization, or a *double processing* of packets (Figure 1 (2.)). The OpenNF controller plays a significant role on the data plane by *buffering* packets. The framework furthermore defines a common interface to extract, delete and merge state. Therefore, it introduces a state classification, depending on distinct *flows*. A drawback of OpenNF is that the controller is utilized by packet buffering at the dataplane, which may reach high volumes. Therefore, Gember-Jacobson et al. [7] changed the OpenNF architecture in two significant ways: First, a packet is processed not only by the original NF, it is duplicated and sent to the newly instantiated NF just to keep its state up-to-date. Secondly, the communication between the original and the new network function to keep a synchronous state does not require an intermediate controller (peer-to-peer).

Several of the aforementioned frameworks for managing state require significant integration effort by software developers, especially regarding the classification of state as to be split, merged, or cloned for horizontal scaling. To overcome or mitigate this. Khalid et al. [13] recently developed a method for code analysis, which automates this kind of classification for network function developers. The mechanism was tested on widely-used implementations like Snort or OpenVPN.

There are alternatives to state migration in low-delay NFV

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

	Per-flow SDN rules	Gen./Spec.	Ctrl. in Datapath	State Sync Info
VM Live Migration [3]	No SDN (L2 Gratuitous ARP)	Generic	No	Memory page diff
Split/Merge [21]	Yes	Generic	Yes	Flow State
Scalable CGN [17]	No	NAT-specific	No	NAT Information
OpenNF [8]	Not required (flowspace)	Generic	Yes	Packet
P2P-OpenNF [7]	Not required (flowspace)	Generic	No	Packet
SliM	No	Generic	No	Statelet

TABLE I
OVERVIEW OVER RELATED WORK

environments. Kablan et al. [11] suggest to use RAMClouds to share all state between VNF instances, avoiding the necessity of state migration. Other approaches suggest distributed state management [18], for example based on distributed hash tables (DHTs).

A use case of software-defined networks (SDNs) is to flexibly interconnect VNFs, for example in service chains. This is relevant, as when moving groups of network functions seamlessly, the underlying SDN must be migrated with them. Ghorbani et al. [9] propose an architecture to seamlessly move an underlying SDN, while minimizing the impact of congestions. Their claims are proven analytically, followed by a testbed evaluation.

Failure recovery mechanisms share commonalities with the aforementioned instance migration. The Remus [4] framework executes speculatively until a checkpoint has been reached, transfers memory deltas (Clark et al. [3]), and releases outgoing Tx packets held in a queue only after acknowledgement of the state update from the backup instance. Remus is evidently suitable for TCP flows, but likely not for delay-sensitive NFs in today's networks, as the periodic release of queued packets (at checkpoints every 25ms) may result in unacceptable delay and jitter effects, which could be aggravated by a WAN-delayed acknowledgement from the backup unit.

Mechanisms like FTMB [22] have been designed especially for NF failure recovery. Similar to Split/Merge [21], Rajagopalan et al. [20] exploit per-flow states to replicate a VNF's state step-by-step. Their approach achieves a significant cost reduction regarding state synchronization. However, besides the per-flow-state requirement, it requires frequent SDN controller interaction on the dataplane, depending on the number of flows.

III. Approach

In duplication-based mechanisms [7], [8], the traffic used for double-processing does not directly have an impact on the critical path of the application delay. Nevertheless, the bandwidth required for state transfers remains a significant cost factor. For example, a VNF may be required to be seamlessly transferred between datacenters, or from a datacenter to an enterprise site. Here, the double-processing traffic's path might often share a common physical link with the dataplane, which creates a bottleneck: the synchronization traffic competes with dataplane ingress traffic. State migration time will approach infinity as

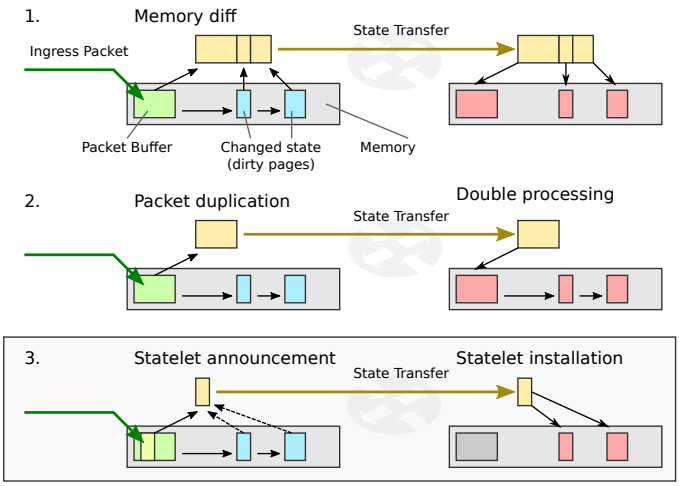

Fig. 1. Comparison of different approaches for state synchronization. 1. VM live migration (last round) [3], 2. OpenNF with peer-to-peer transfer [7], 3. Statelet-based transfer (our approach).

less and less bandwidth is available, and if available bandwidth is exhausted, migration will completely fail, assuming that dataplane traffic must not be dropped. Furthermore, as the amount of duplication traffic must also be buffered at the destination until VM startup, it has a negative impact on buffer memory consumption, and aggravates the risk of overflow.

The problem of insufficient bandwidth capacity also applies to traditional, transparent VM live migration techniques [1]: In order to achieve minimum or no downtime, the number of iterative rounds must be set to a high value. Besides taking a long time consuming a significant amount of bandwidth, the mechanism will probably never converge with an absence of dirty pages, since the network-intensive workload changes the memory on every incoming packet (like packet buffers or counters).

Due to the aforementioned appraisal that many network functions require only a small amount of information in every packet to update their state, like header fields [2], the idea is to identify this information with the help of the specific VNF application. Therefore, one of the contributions in this paper is the introduction of a novel interface for this task.

The approach assumes an NF which changes its state deterministically based on incoming packets. For every incoming

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

packet, the NF is able to actively pass all information it requires for internal state modification in a compact representation to an interface, referred to as a **statelet**. Furthermore, another VNF instance of the same type is able to change its internal state accordingly, based on an incoming statelet.

Compared to previous work proposing memory page diffs [3], which may contain large structures changed after every packet entering, or proposing duplication of whole packets with possible information irrelevant for state synchronization [7], [8], statelets are expected to produce less traffic (see Section V for details). Figure 1 compares different previous approaches with the statelet approach.

A statelet can be seen as a set of information *defining* the new state together with the previous state. For a formal definition, consider a current state $s \in S$ of a VM, and a packet $p \in P$ entering. Let the next state be defined with $s' = f(s,p)$. The task for a NF developer to make use of the statelet approach is to also define a function $g : S \times P \to L$ which returns[2] a statelet $l \in L$, and a function $h : S \times L \to S$ which installs the statelet at the destination, so that $\forall s,p : f(s,p) = h(s,g(s,p))$. For the statelet approach to be effective, the total information carried in statelets should be much smaller than in packets, i.e. $|L| \ll |P|$.

More specifically from an interface perspective, a statelet is assumed in the following as a variable-length byte vector. A statelet can be empty (only announcing a packet event, for example to increase packet counters)[3], or can be omitted (no information about a packet event required, no relevant state change). On the examples of several VNF types, we show how VNFs can make use of statelets (Table II):

- **Network Address Translation**: Besides translating and forwarding packets, the NF must maintain entries in a NAT table with the respective fields for mapping incoming and outgoing packets to translated connections. Commonly, a NAT creates an entry in this table upon a socket's first packet. A very simple NAT gateway only announces a statelet upon creation of a new NAT table entry (and whenever it is explicitly closed, e.g. due to a FIN packet, if available). The resulting statelet for entry creation just contains the source address/port and the destination address/port (12 bytes with IPv4). A more elaborate NAT function also gathers statistics like packet and byte counters, which require the corresponding information to be transferred as a statelet for every incoming packet, however still only comprising several bytes.
- **Signature-based Intrusion Detection**: A very basic intrusion prevention mechanism would just drop packets upon finding a signature, thus operating stateless. However, it is commonly required to look into the packet history to identify repeated attacks, for example SSH or HTTP dictionary attacks. Thus, a statelet is produced for any information that might be required to identify attacks in the future, like information about relevant login attempts and the origin address of dictionary attacks.
- **VPN Concentrator**: This NF generates various statelets during a user's connection establishment, announcing the current state of the authentication procedure for this user. For authenticated clients, any change of cryptographic state, like sequence numbers, AES keys, or rekeying events and corresponding information, is announced with a statelet.

We prove that a statelet is never required to exceed the packet size to make a consistent state update: If a statelet is larger than the packet which has triggered it, it must contain (1) redundancy, as per our previous assumption state change is only based on information in incoming packets, and/or (2) information about state which was not changed by the entering packet (which becomes redundant at the destination already having this information). Finally, in the extreme case that the redundancy can only be removed with larger computational overhead leading to degraded dataplane performance, the mechanism can fall back to transferring the packet as the statelet like in duplication-based approaches. This infers that the costs of duplication-based mechanisms constitute an upper bound for statelet-based ones.

The statelet announcement is an additional step in the dataplane process, so it is necessary to ensure that the overall dataplane performance is not significantly degraded. We will argue in the following why the concept of statelets only has a very small impact on VNF performance: First, the announcement can be (and should be) implemented as an asynchronous call. In our implementation, the dataplane is only required to point to allocated memory in order to announce a statelet, before continuing with normal dataplane operation, while any other operation on the statelets may be conducted by an out-of-band thread (Section VI). Secondly, statelet announcement is necessary only during a short period of the migration process. Therefore, the NF's dataplane thread checks a flag `upd_enable`, and announces statelets only if the flag is set.

The introduction of a new interface also means an integration effort for developers, however, many approaches suggest state-aware network functions to allow for not only state transfer, but also seamless horizontal scaling [7], [8], which require the integration of novel APIs. Statelet announcement may also become a part of a future API unifying several state management features towards elastic VNFs [12]. Finally, code analysis techniques [13] may not only be used for state classification towards split/merge operations, but could also be applied to identify information in a packet required for generating a statelet in existing NF implementations.

IV. System Model

In this section, a model of a generic framework for VNF migration is proposed and described, which exploits the use of the statelet approach introduced in the last chapter. In our model, several hardware nodes with a *hypervisor* managing virtualized computing resources are given, which are

[2]Statelet omission (unchanged state) is not modeled here for brevity.
[3]In our implementation, the statelet also has a two-byte *type id*.

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

VNF Type	Example Behavior	Example Workload	Statelet size(s)	Relative statelet flow size σ*
NAT	Maintains a NAT table, an ARP table, and counters for keeping track of the internal hosts' traffic.	Every 15th workload packet opens a new socket. ARP requests are not required (steady state).	• 12+18 bytes for every new NAT socket • 10 bytes for every new ARP table entry • 12 bytes for every workload packet (counters)	$0,025$
Intrusion Detection	Tracks TCP/UDP flows for suspicious content, fully captures suspicious packets.	Multiple TCP/UDP flows, 5% of packets are suspicious.	• 20 bytes to track the current flow's session for every non-suspicious packet, • Entire content (512 bytes) for every suspicious packet	$0,057$
VPN Session	Works in AES-CBC (Cipher Block Chaining) mode, 256 bits block size. Uses counters to keep track of packet numbers.	Session data only.	For every packet: • 4 bytes for the current session identifier • 32 bytes for the ciphertext of the packet's last block (decryption and encryption) • 12 bytes for counters (bytes and packets)	$0,093$

* Assuming a mean packet size of 512 bytes, see Section V

TABLE II
EXAMPLES OF NETWORK FUNCTIONS, WORKLOADS, AND THE CORRESPONDING STATELETS

connected by a flexible software-defined network (SDN) with limited capacity and significant delay.

The nodes and network are managed by a logically centralized controller, which is also in charge of coordinating the migration process. The problem of if, when, and where to place a VNF is out of scope, but has been investigated in previous work [2], [6]. The controller is restricted to *control* traffic, it is not involved in any data traffic, including packet buffering. This makes the controller's bandwidth requirements independent from the VNF load, thus relieves it from the necessity to scale depending on the latter.

In the following, we assume that a migration is about to be conducted from a source (*srcInst*) to a destination instance (*dstInst*). Furthermore, we separate between **complete** and **partial** VNF migration. The latter will leave *dstInst* with handling only a *part* of the traffic that has been formerly processed by *srcInst*. Complete migration is desired to perform resource optimization or maintenance tasks, partial migration by scale-out mechanisms which need to "split" a VNF by redistributing a part of the traffic only. The mechanism for seamless state transfer as proposed in our work can be used for partial migration and full migration (see Section IV-D). In the remainder of the paper we focus on full migration.

Last, we assume that VM state snapshots can be taken locally at the nodes, and can be installed on destination nodes currently not in dataplane operation. A snapshot should exactly contain the VNF's state at the moment of triggering it. The process may need time, but there must not be VM halts longer than an accepted jitter tolerance (depending on the application). Thus, a mechanism is required allowing the VNF to operate without any service disruption while the snapshot is taken, and to guarantee that the snapshot is consistent at snapshot start. These mechanisms are known as *isolated snapshots* or *redirect-on-write* (see Section VI-B).

A possible approach is to consult shadow page tables on the hypervisor [3], which has the advantage that the snapshot creation is transparent to the guest VM. Alternatively, like in our proof-of-concept implementation (see Section VI), the snapshot can also be actively prepared by the VNF's software itself, allowing a freshly booted application instance to recover from it. This enables the VNF to select necessary parts of the dataplane state, but avoids state not necessary for taking up the dataplane operation at the destination.

A. Architecture

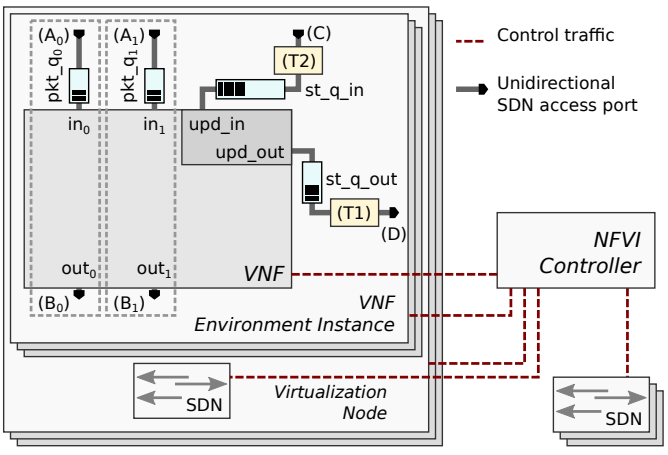

Fig. 2. Architecture overview, 2 interfaces

In the proposed abstraction (Figure 2), the VNF has one or multiple full-duplex network interfaces, and we instantiate several resources for each of them: two unidirectional dataplane

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

interfaces `in` and `out`, and a packet queue `pkt_q` (dashed rectangles)[4]. The statelet interfaces `upd_in` and `upd_out` are novel in this abstraction. They are not required to be network interfaces, they may as well be implemented using functions to pass statelets in the form of variable-length byte vectors (without addressing etc.), however they should operate in order and without risk of data loss.

State migration functionality is provided by the VNF Environment Instance (VEI), handling control tasks and providing status updates to the NFVI controller. The VEI maintains three queues: `pkt_q`, `st_q_out` and `st_q_in`. In all cases, `pkt_q` is directly attached to *in*. `pkt_q` buffers network packets to guarantee in-order processing during switchover (only for a short round-trip time), `st_q_out` and `st_q_in` buffer statelets. The ends of the queues not facing towards the VNF, as well as the `out` port (Figure 2, A-D), are flexibly assigned via the SDN, as described in the following. In the initial and final state of a migration, the SDN delivers all VNF dataplane traffic to `pkt_q`, while taking the processed traffic from `out` to its destination. The ports of `st_q_in` and `st_q_out` (C and D) are unused in the initial configuration.

Whenever a flag `upd_enable` is set, the VNF prepares statelets upon incoming packets as discussed in Section III, and passes them them to `upd_out`. A VNF instance receiving this vector via `upd_in` applies the respective changes to its internal state, but does no further action (like sending packets out).

B. State Migration Procedure

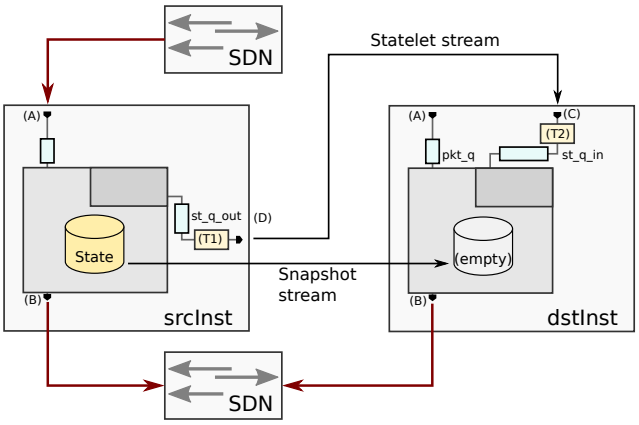

Fig. 3. Datapath during migration process before traffic redirect.

Figure 3 depicts the components in the datapath of the state migration before the traffic redirect to `dstInst` takes place. The following steps are conducted, some of them concurrently, for example if threads are opened for parallel execution.

1) The controller prepares `dstInst` on the destination hypervisor. The destination VEI does not have any state yet, and is not connected to any datapath. At `dstInst`, `st_q_in` and `pkt_q` are set into a mode where they only enqueue statelets, but do not dequeue and pass them to the VNF application for processing.

2) `srcInst` then connects to the freshly prepared `dstInst` using a reliable, connection-oriented protocol like TCP to send two data streams. `srcInst` then sets the `upd_enable` flag, and immediately opens a statelet dequeue thread and a snapshotting thread:

2.1) The statelet dequeue thread (T1) continuously waits for new statelets on `st_q_out` (which are now continuously generated by the VNF dataplane) and sends them out on the statelet stream. The dequeuing continues until explicitly interrupted in the following steps.

2.2) The snapshotting thread serializes the VNF's state, and sends it over the snapshot stream. The thread closes the snapshot stream afterwards, before closing itself.

Simultaneously, `dstInst` opens two threads upon connection attempt of `srcInst`, a statelet enqueue thread (T2) and a *synchronization* thread:

2.3) The statelet enqueue thread enqueues the byte vectors received over the statelet stream in `st_q_in`. As no dequeuing takes place yet, the size of `st_q_in` increases (statelets are buffered at the destination).

2.4) The synchronization thread receives the snapshot over the stream, and installs it for `dstInst`. Then, it immediately starts to dequeue statelets from `st_q_in`, and applies the respective changes to the snapshot. If the thread notices that `st_q_in` has reached an underrun of d statelets for the first time, it sends a *redirect* message to the controller. In our implementation, d was set to 0, thus the message is sent whenever the queue is empty for the first time after dequeueing start. If the statelet stream has been closed **and** `st_q_in` is empty, the dataplane of `dstInst` is activated by letting the dataplane continuously dequeue and process packets in `pkt_q`. In the latter queue, packets will have arrived after switchover, and wait during the drain-packet round-trip in order to be processed after the last statelet. In this case, the migration has finished from the perspective of `dstInst`.

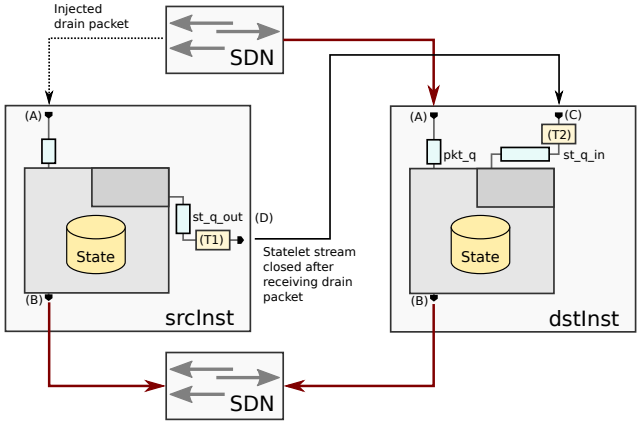

Fig. 4. Datapath during migration process after traffic redirect.

[4]In the following, we simplify the process description and depict only one network interface.

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

3) Upon receiving the redirect message (Figure 4), the controller redirects the ingress flow of `srcInst`'s dataplane input to the one of `dstInst`[5]. `dstInst` cannot yet start with the dataplane operation, as there could still be some statelets from `srcInst` underway.

To ensure `srcInst` has completed all dataplane operations, and to guarantee in-order delivery, a special *drain packet* is now injected by the SDN controller on the datapath where the switchover was made, but towards `srcInst`. If the network prioritizes packets, it has the lowest priority used on the datapath to `srcInst`. The drain packet is sent as an indicator that likely there will be no additional packets on the former path to `srcInst`. Upon receiving this packet, srcInst flushes `st_q_out` and closes the statelet stream (the latter causes the synchronization thread of `dstInst` to start the dataplane). Here, the migration has finished from the perspective of `srcInst`.

C. Mutual Exclusion

Except in the case of making a snapshot at `srcInst` which requires isolated snapshotting (Section VI-B), the process ensures mutual exclusion between dataplane and synchronization operations on the state. At *srcInst*, statelets are quickly written to *st_q_out* before continuing dataplane operation, so that the statelet dequeue thread can process them from the buffer without any further interaction with the dataplane. At `dstInst`, the first operation is the installation of the snapshot before any buffered statelets are applied to it. Similarly, the `pkt_q` is drained only after the statelet stream has been closed and *st_q_in* has been drained, thus the last statelets have been installed before dataplane operation continues at *dstInst*.

D. Extending to partial state migration

In the last section, a complete state transfer to a destination VNF was described. For seamless scale-out/in operations, a *partial* state migration might be desired which offloads parts of a VNF's instance's workload to another instance (split), or combine the workload on a single instance (merge). Traffic control for split and merge was handled in previous work [8]. Given that the VNF is able to define a pattern by which the flowspace can be split, as well as the corresponding state, this section extends the full state migration scheme to allow partial migration of a VNF's state, which allows to use it for split and merge operations.

First, the source VM omits state that does not need to be transferred to `dstInst` during snapshot serialization. Secondly, statelets are only announced if they are required for the state which has been included in the serialized snapshot. Thirdly, the SDN controller redirects only relevant flows to the destination instance. Finally, upon receiving the drain packet, the source instance does not stop the dataplane operation. [6]

[5]To minimize packet loss during the redirect operation, flows can be prepared in a previous step, so that only a single *switchover* operation is needed on one forwarding device.

[6]For example, when iterating a hashtable for snapshotting, every odd IP address is skipped, only statelets for odd addresses are announced, and the SDN controller applies a wildcard bit mask to redirect only odd IP addresses.

While a state migration for an instance split only requires minimum modifications to the *complete* migration method, a merge is challenged by the operational dataplane of `dstInst`, which must not be interrupted. The first problem is that the snapshot must be merged with the operational state of the destination, furthermore, mutual exclusion between statelet installation and the dataplane must be ensured. Therefore, data structures which support split and merge must be used, in the sense that a memory structure in the snapshot transferred does not interfere with a structure used by the current dataplane of the destination instance during merge.

We therefore propose the following solution: If a source instance responsible for flowspace A (e.g. odd IP addresses only) shall be merged with a destination instance responsible for flowspace B (e.g. even addresses), the snapshot data structures (e.g. the hashtables) are not merged along with the VNF instances. Instead, the snapshot of the source is written to a separate memory region of the destination, and the subsequent statelets are installed here, as well. When the dataplane of the destination instance eventually takes over operation for the source instance's datapath (after the statelet stream closes), the dataplane selects the structure to use based on the flowspace description, ingress packets of flowspace B are processed on the destination instance's original data structure, while packets from flowspace A operate on the received and updated snapshot. The proposed mechanism only has minor impact on dataplane performance, as it requires one additional match operation only.

V. Analysis

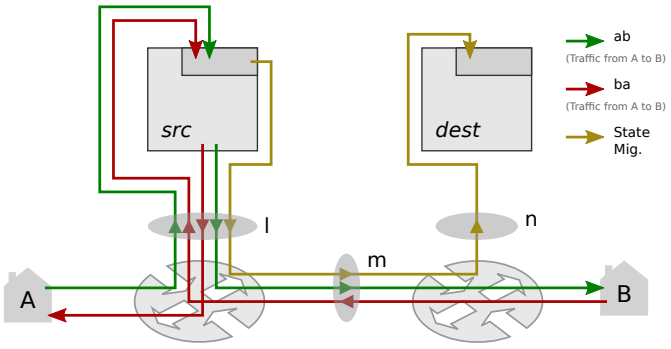

Fig. 5. Flow and physical link model of an infrastructure for a state migration.

In this section, the problem of state migration in NFV environments with restricted bandwidth capacity, which was motivated in the introduction, is quantified. Therefore, a flow and physical-link model is applied to the model of the state migration mechanism we have described in Section IV. Besides supporting problem motivation, the goal of this section is to allow assessing of requirements and properties on state migration mechanisms in a network with specific physical properties. The analysis in this section puts a focus

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

on interdependencies of *migration completion time*, *bandwidth usage*, and the *statelet factor* required to successfully complete the migration in finite time.

For any data stream being handled by an NF in any time interval, the **statelet factor** σ is defined in the following as the ratio of the average statelet traffic volume caused by the packets of the stream and the average volume of the packets themselves. For example, if a stream consists of 500-byte packets only, and the size of the statelets generated is 50 bytes per packet in average, σ is 0.1. A stream with $\sigma = 1$ corresponds to the performance of duplication-based mechanisms. Examples for statelets are shown in Table II.

Figure 5 depicts the model of a generic NF inside bi-directional traffic between A and B. The NF is about to be moved from *src* to *dest* over a common datapath which is shared with the traffic between A and B. This especially happens when moving an NF between datacenters: Instead of having a dedicated, strong link between *src* and *dest* for state transfer, here, the inter-datacenter WAN link may become the bottleneck. Even more restricted WAN links must be used by migrations to or from virtual customer premises equipment (vCPE).

For reducing complexity, the model neglects control traffic, WAN delay including WAN-related queueing delay, and the effect of TCP flow control mechanisms. The model assumes that if packet loss occurs caused by exceeded bandwidth requirements, the service becomes unstable and cannot be fulfilled. The bandwidth of internal interfaces between the hypervisor and its VMs (PCI and memory buses) are also assumed much larger than the inter-VM links. However, the aforementioned parameters neglected in this model are taken into account in the evaluation in Section VII.

The model assumes three full-duplex physical paths, *l*, *m* and *n*. *l* is the common sub-path of the paths (src, A) and (src, B) (depicted in Figure 5). *m* is the common sub-path of the paths $(src, dest)$ and (A, B), while *n* is the common sub-path of the paths $(dest, A)$ and $(dest, B)$. On these physical links, several flows are established: *ab* (user traffic from A to B), *ba* (user traffic from B to A) and a state migration flow from src to dest (in the model, a reverse ACK path is neglected.).

In the following, $C_{<link>}$ returns the total available capacity of the physical path of `<link>`. F_{ab} or F_{ba} return the current bandwidth consumption of the flows ab or ba. Based on these definitions, Equation 1 defines the remaining bandwidth capacity available for state transfer between src and dst.

$$C_{rem} = \min\{C_l - F_{ab} - F_{ba}, C_m - F_{ab}, C_n\} \quad (1)$$

In the system model, the packets are immediately sent to and held by the destination hypervisor until the state snapshot is established at the destination. T_{all} is comprised of the time for snapshotting, the time required to transfer the state over the remaining bandwidth, and the time taking over the dataplane operation at *dest*. T_B is the combined local snapshotting time at *src* and installation time at *dest*. Finally, S is the snapshot size. The condition to be able to keep up with the state while transferring the snapshot is given in 2.

$$\sigma \cdot (F_{ab} + F_{ba}) < C_{rem} \quad (2)$$

As the buffer is populated at *dest* the time to transfer the snapshot over the remaining bandwidth while simultaneously transferring the statelets, the total migration time, is modeled with Equation 3.

$$T_{all} = T_B + \frac{S}{C_{rem} - \sigma \cdot (F_{ab} + F_{ba})} \quad (3)$$

To find out the maximum statelet factor accepted for given network and load parameters, Equation 4 can be used.

$$\sigma < \frac{C_{rem} - \frac{S}{T_{all} - T_B}}{F_{ab} + F_{ba}} \quad (4)$$

In Figure 6, the maximum allowed statelet factor is plotted on an example with $C_m = 1000MBit/s$, and the capacities of l and n set to $10Gbit/s$. The snapshot size was chosen with $S = 20MByte$, local snapshotting/installation time T_B with $0.3s$. The graph contains a curve each for different desired migration times t_{all}. At the points where the curves hit the X axis, the bandwidth is insufficient even to transfer the snapshot in the desired total migration time t_{all}. We can also observe that, compared to packet duplication ($\sigma = 1$), the bandwidth utilization can almost be tripled with very small statelets, which confirms our testbed evaluation results.

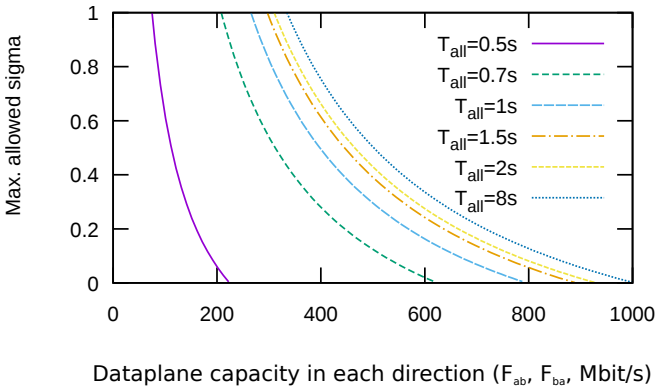

Fig. 6. Analytical determination of the maximum allowed statelet factor which is required for a successful seamless migration of a VNF. Duplication corresponds to a statelet factor σ of 1. Note that the dependency between the total migration time T_{all} and the dataplane capacity is non-linear.

A. Impact of dataplane load on traditional VM migration

Traditional memory-delta-based state resynchronization mechanisms [3], [4] are not seamless, they have a significant downtime during the last round of memory delta transfer (Section II, [1]). It is difficult to quantify the amount of invalidated memory, which depends on the type of NF and implementation details. However, it is possible to model a ring

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

buffer containing received packets, the least element a high-performance NF will have: Whenever a packet arrives, a part of the ring buffer having (at least) the size of the packet is dirtied. As ring buffers eventually wrap around and overwrite their last entries, not more memory than the size of the ring buffer can be dirtied in one round of delta transfers.

r_{DP} is the current dataplane load, r_c the capacity of the link. m_0 is the initial snapshot size, the total memory of the VM. We determine the time to transfer memory dirtied in the last round m_i with $t_i = \frac{m_i}{r_c - r_{DP}}$. The memory which has dirtied and must be transferred in the next round is defined with $m_{i+1} = min\{r \cdot t_i, c_{mbuf} \cdot bufsz\}$. We set $bufsz$ to $1400bytes$ and $c_{mbuf} = 64K$, like in our implementation.

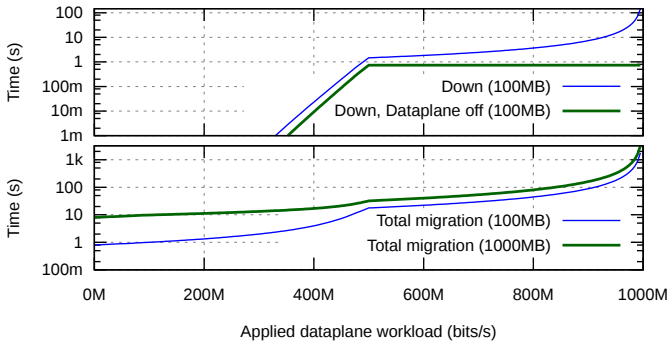

Fig. 7. Numerical calculation of total migration times and last-round downtimes, based on an optimistic model of delta-based VM live migration.

Figure 7 shows the results of calculating 10 rounds of delta transfer for different dataplane utilization. At the top, the duration of the last round is depicted during which the VM is paused (Downtime). A variant with dataplane traffic switched off during the last round is also considered ($r_{DP} = 0$), so that the link capacity is available solely for state transfer. We can observe a cut in the increasing downtime and total migration time at 500M, which is caused by the ring buffer wrapping around and overwriting already invalidated pages.

The model is optimistic and must be considered as a *lower bound* of downtime and duration. It ignores TCP slow start phases, control and protocol overhead, delay, signaling overhead, and CPU bottlenecks which would further decrease performance in a testbed evaluation. As SliM performs even better than the optimistic model, we did not conduct a testbed evaluation for comparison of delta-based migration with SliM.

VI. Evaluation Platform

Our SliM proof-of-concept implementation (Figure 8) has been designed for the KVM/QEMU/libvirt virtualization environment, but should be easily adaptable to other hypervisor technologies. The implementation is prepared to be integrated into OpenStack in the future, but uses a custom Python-based NFVI controller at this point in time. Every VNF has one or multiple dataplane interfaces, as well as one management interface. In our deployment, the latter is connected to a single subnet shared between all VNFs, while the dataplane interfaces are connected to an SDN-controlled domain.

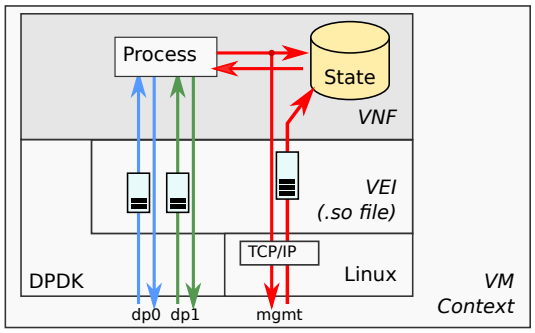

Fig. 8. Software architecture of the proof-of-concept and evaluation implementation on a NFVI hypervisor.

The *Data Plane Development Kit (DPDK)*[7] is used for the dataplane interfaces in order to reduce kernel stack overhead and make use of accelerated features. The management interface operates on Linux kernel sockets for convenience. We chose to implement the VEI inside the VM, as the effort to implement a reliable VM/host interface can thus be avoided without any obvious drawback.

The VEI is loaded by the network function as a shared object (`.so`) library. It depends on DPDK for required datapath functionality, but passes the majority of DPDK's API up to the VNF, with some exceptions e.g. on `lcore`[8] assignment and burst reception/transmission. The VNF may obtain multiple `lcores` for its dataplane operation (as usual in DPDK), where statelets announced by any lcore to `st_q_in` are dequeued by a single dequeue thread. To offload the dataplane operation from overhead, the VEI just requires the dataplane to pass the pointer to the statelet with *slim_notify_statelet(...)*, which is enqueued on a `rte_ring` instance before immediately returning.

A. Statelet Stream

The statelet dequeue thread takes the byte vectors and sends them over a TCP stream in a type-length-value (TLV) format, where the type is a VNF-specific identifier. The type and length fields (in bytes) of every statelet are each 2 bytes long, leading to a maximum statelet size of 65536 bytes.

As the statelet path is not time-critical until the stream is closed, statelets are buffered until the maximum segment size (MSS) of TCP has been reached. For example assuming a MSS of 1400 and 12-byte statelets (similar to our proof-of-concept VNF implementation, including the type), 100 statelets can be transferred with only one TCP segment. Compared to packet duplication, this additionally reduces overhead due to headers and inter-frame gaps, and offloads the intermediate path from switching small packets.

To offload the datapath from state migration overhead and to avoid jitter and peaks caused by it, the aforementioned work is handled in a separate thread both on the sender and receiver side. While snapshots can be simply serialized by

[7] http://dpdk.org
[8] Logical CPU cores for dataplane operation

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

another thread, statelet *announcements* must be conducted by the dataplane thread itself, however they immediately return after placing the statelet's pointer in a ring buffer, which is then further processed by a dequeuing thread. Currently, in our proof-of-concept implementation, the stream uses the Linux kernel TCP stack for convenience, however, implementations like `mtcp` [10] may be used for statelet transfer in future versions, providing even higher performance.

A possible extension to the statelet mechanism would be that if a packet overwrites the state change a previous packet has caused, the statelet of the previous packet, if still buffered at the source and waiting to be sent (e.g. in a maximum-sized TCP segment), could be discarded in the buffer. However, to the best of our knowledge, mapping a statelet to the ones that will be overwritten would require an API extension and additional processing power on the dataplane, and the expected traffic reduction might not justify it. To shrink the statelet stream's size close to a maximum extent in future implementations, run-length encoding (RLE) or stream compression formats like *gzip* could be used. A compression of the statelet stream should be effective, as the latter is supposed to have a low entropy, and does not directly affect dataplane performance as statelets are packet in a separate thread. Nevertheless, it should be carefully used in order not to deteriorate the statelet thread performance of SliM.

B. Isolated Snapshots

In Section IV, a mechanism to do *isolated snapshots* of the VNF state was assumed. Therefore, in the proposed implementation, we developed a lightweight, well-tested mechanism to make isolated snapshots of `rte_hash`, a hashtable implementation in DPDK. If a snapshot must be taken, the hashtables can be *freezed*. When reading from the table, an argument is passed whether to read from the freezed or the most current state. Thus, snapshot serialization can use the freezed state not including subsequent write actions, while the dataplane can operate on the most current state.

The hash map maintains a struct with two values a and b for every key stored in it, $key => (a, gen_a, b, gen_b)$, where gen_a and gen_b are positive integer metadata values, referred to as *generations*. Additionally, a table-wide generation t exists, which is initially 0. On every *unfreeze*, which occurs after the shapshot has finished, t is increased by 1. The value m is $t+1$, if the table is currently freezed, $m = t$ otherwise. Informally, m is always one ahead of t while the table is freezed, t catches up when unfreezing.

If an entry e is to be added, the value of the struct is initialized with $k => (e, m, null, -1)$. Whenever an entry is present and must be overwritten, the entry is written to either a or b, depending on which generation is smaller: $k => (e, m, *^{9}, *)$, if $gen_a < gen_b$, or $k => (*, *, e, m)$ otherwise. If an entry is read, the value of either a or b is taken, depending on which value has the higher corresponding generation. However, only if we want to read from the freezed state for snapshotting, the value with the highest generation is taken *not exceeding* t. If such a value does not exist, the value is assumed as not present in the map. Finally, if an entry is to be deleted, the entry struct is completely removed if the table is not currently freezed. If the table is freezed, a new value is created containing $null$, non-isolated read actions therefore assume this value is not present.

The implementation supports multiple subsequent freezes and unfreezes, allowing to resume normal operation after a failed migration and retry it at a later point in time, or allowing for multiple split and merge operations in future implementations supporting partial migration. For entries deleted during a freeze, cleaning jobs must currently delete the backing structs after unfreeze, however, these can be placed in a queue during the freeze, so the cleanup processes are not required to iterate over the whole table. Finally, snapshot serialization can be conducted parallel to write actions, as no backing `rte_hash` entries are deleted while in freezed state breaking the linked serialization list. The correct operation was verified in an extensive test case.

C. VNF Example Implementations

For evaluation and proof of concept, a **Network Address Translation (NAT)** VNF was implemented. It uses two `lcores` for two duplex dataplane interfaces dedicated to the interior and exterior network, where packets from the interior to the exterior network are *masqueraded* behind the exterior IP address of the VNF. The network function supports ARP requesting/responding, and TCP/UDP NAT, but currently IP fragmentation and ICMP are unsupported. The VNF additionally keeps track of the interior network hosts' traffic volume by using counters.

The VNF's state is comprised of the NAT, ARP and counter table. If the VEI requests state updates, a statelet is generated for every new NAT socket created (18 bytes), for every new ARP table entry (10 bytes), and for every change of a host's packet counter (12 bytes), where the latter occurs on every data-plane packet processed. The counters are also used to detect out-of-order statelets or packets, which did not occur in the evaluation so far.

To demonstrate the feasibility of implementing other network functions with the SliM framework, we have also implemented a simple (non-standard) **mobile packet gateway (PG)** with DPDK and SliM. Every packet coming from the load generator is tagged with a 2-byte ID, resembling a mobile cell to which the sender, a mobile subscriber, is currently associated. Beneath the cell ID, it is also announced if the cell attempts to hand the user over to another cell. The gateway is required to tag every packet coming from the echoserver (the outside network) with the tag of the cell to which the mobile subscriber is currently associated, and therefore looks it up with the incoming packet's destination IP and port.

The challenge of the packet gateway NF, generating statelets for every handover (50 bytes), is to instantly tag packets from the exterior network with the new cell ID (thus redirecting

[9]The asterisk denotes that no change is made to this field.

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

them to the new cell) as soon as a handover is announced from the interior, even when a large number of handovers are conducted during state migration.

VII. Evaluation

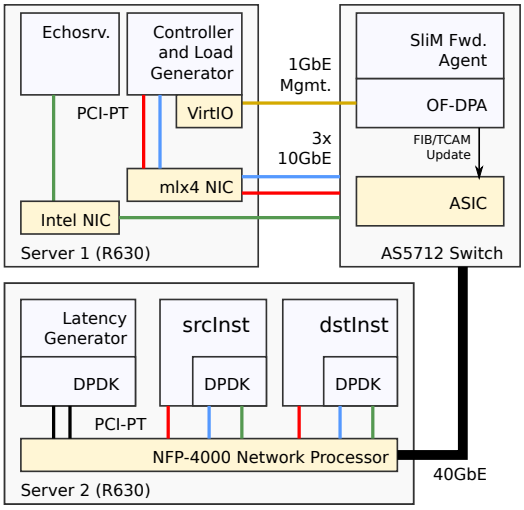

Fig. 9. Hardware and virtualization architecture used in the evaluation setup.

In this section, the evaluation setup is described, before the results of the proof-of-concept platform are presented and discussed. The evaluation is conducted on two PowerEdge R630 servers[10], operating on Linux and as a Kernel-based Virtual Machine (KVM) hypervisor. The servers each comprise two CPU sockets with 10 physical cores on each socket. Figure 9 depicts the hardware and virtualization architecture used. Server 1 is equipped with two different dual-port network interfaces (NICs), a *Mellanox ConnectX-3 (mlx4)* and an *Intel 82599ES* card. Server 2 uses a single *Netronome NFP-4000* network processor (NPU) card with two 40GbE ports. Besides the two servers, an *Edge-Core AS5712-54x bare-metal* switch is involved in the experiment. The switch runs *Open Network Linux* and the *OpenFlow Dataplane Abstraction (OF-DPA)* layer[11] to be able to control its forwarding behavior in a flow-like abstraction. Although OF-DPA can be used via the OpenFlow protocol over the network, we decided to directly link against the layer with a C-based SliM forwarding agent running as a Linux application on the switch, to avoid delayed responses to dataplane events.

On Server 1, two VMs are set up, a VM combining a controller and load generator with dedicated access to the dual-port `mlx4` NIC via PCI Passthrough (PCI-PT), and a second VM, the echoserver, having dedicated access to one of the Intel NIC's ports. Connected to PCI *virtual functions* spawned by the NFP-4000 NPU, Server 2 comprises both `srcInst` and `dstInst`. To establish a realistic simulation scenario however, if the instances communicate with each other, their whole traffic including management must traverse the AS5712 hardware switch first as if they were placed on different machines. The aforementioned traversal of the inter-instance traffic – like all traffic from or to the VNF instances under test – then involves metering and latency simulation like explained in the following.

Figure 10 schematically illustrates the flows used in the evaluation setup. A switched management VLAN (red) provides basic IP connectivity between both VNF instances and the controller. Furthermore, two dataplane paths, `dp0` and `dp1` are configured. Upon initialization of `srcInst`, all traffic originating from the load geneator's `dp0` interface is forwarded to the respective interface of `srcInst` and vice versa (blue), like the traffic between the echoserver and `srcInst`'s `dp1` interface (green). To implement the switchover command of SliM, the two dataplane paths can be redirected by the AS5712 switch to the respective interfaces of `dstInst`.

SliM is supposed to be able to migrate for example between inter-datacenter links, thus the setup must consider the datapath to have a delay which is typical for this scenario. Therefore, an additional latency was set to 2.5ms with a DPDK-based *latency generator* NF[12] connected to the NPU. The latency, which is applied into both directions of all traffic from or to the VNFs, results in a round-trip time (RTT) of 10ms between the VNFs themselves, and in an RTT of 5ms between each VNF and the controller / load generator. This value approximately resembles the delays of typical intra-European datacenter links[13].

The total traffic from or to each VNF instance under test is furthermore restricted by a *meter*, dropping packets above a rate of $1Gbit/s$ (burst size $20ms$) per each direction. Although the SliM VNF implementations have been successfully tested with much higher data rates, the target is to hereby investigate the effect of external bandwidth limitation described in the introduction. As stated in the introduction, the traffic prioritization strategy decides in an overload situation, whether a state migration over a shared link will try to keep up with the resynchronization process to the cost of migration time and possible migration success, or drop dataplane packets to succeed in a shorter or finite amount of time. In our evaluation, the shared bandwidth is provided to all flows on a *best-effort* basis, no traffic of any flow is prioritized. This configuration should ensure eventual success even in an overload situation, however to the cost of packet loss.

The evaluation uses the NAT and the PG VNF implementation introduced in Section VI-C. The experiment starts with an active `srcInst` at relative time s. `dstInst` is booted up quickly thereafter (inactive with no state), and at $s+5\ sec$, the state migration is triggered. Performance and traffic metrics are collected during $[s, s+15\ sec]$. The measurement is conducted for every setup variant described, and for different loads. For every setup and load combination, the measurement was repeated 13 times to obtain results with a high confidence. For comparison, a *duplication-based* resynchronization *mode* was

[10] 2x Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz, 2x64GB RAM
[11] http://opennetlinux.org, https://github.com/Broadcom-Switch/of-dpa
[12] https://github.com/tudps/sloth-latency-gen/
[13] http://www.verizonenterprise.com/about/network/latency/, March 2017

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

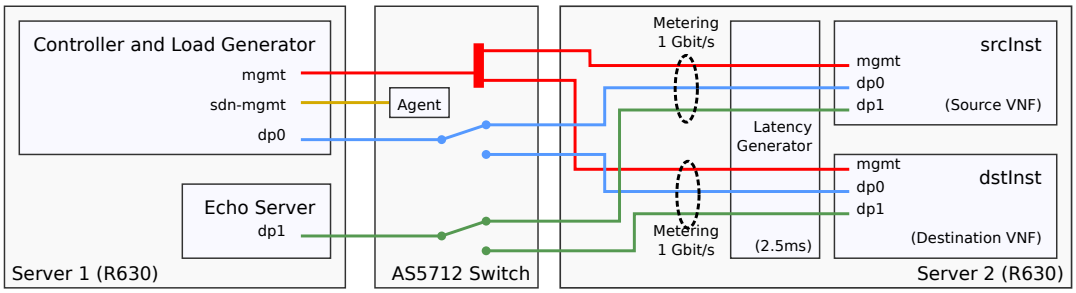

Fig. 10. Schematic illustration of flows in the evaluation setup. All depicted flows are bi-directional.

implemented in SliM, to ensure that results are comparable and not biased by implementation details. When in duplication mode, the whole ingress packet is copied and sent to the destination VM instead, where it is processed [7].

A typical workload of an interior network communicating with external servers is applied to the VNF datapath. For evaluating the NAT VNF, a packet generator opens 128 UDP sockets to 4 echo addresses through the NAT VNF, the latter reply with same-sized packets from the exterior datapath. Therefore, both the original and the reply packets pass the NAT function, where their source address and port number are translated. For evaluating the PG NF, 256 subscribers are being connected with a very fast mean handover frequency of 2s each. Furthermore, stricter requirements are applied for measuring packet loss, a packet is also considered as lost if it is sent to a wrong cell, with the exception if the last handover was 7,5 milliseconds ago (150% of the round-trip time between load generator and NF).

For both NFs under test, the load generator includes a counter and a timestamp in every packet to identify out-of-order packets, loss, and the round-trip time (RTT). Measurements are conducted with 1400-byte and 512-byte packets, the former is almost the maximum transmission unit (MTU) in the Internet, while the latter is close to the median of the Internet's packet size distribution[14]. 20MB of filling zeroes were added to the snapshot to simulate additional workload for larger states. If a future NF in a carrier-grade environment maintains hundreds of thousands of sessions, the latter snapshot size is a realistic assumption, even if only connection tables are maintained (50 bytes of state per connection).

A. Results

The absence of any service downtime significant to the application is a requirement for seamless migration. Caused by the best-effort scheduling when restricting the dataplane to 1Gbit/s, state migration traffic can supplant dataplane traffic if required to finish the migration in finite time and thus succeed[15], leading to dataplane packet loss, which can therefore be considered as the primary indicator for migration failure

[14]https://www.caida.org/research/traffic-analysis/pkt_size_distribution/graphs.xml

[15]Note that any prioritization of dataplane traffic would lead to overloaded migration attempts to completely fail, as their state cannot converge.

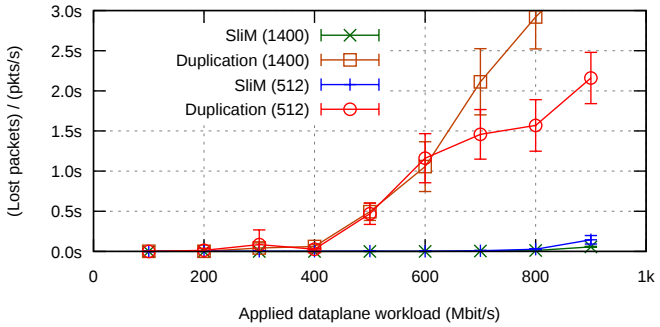

Fig. 11. "Seconds of packet loss", number of packets lost divided by the packets to process every second. NAT NF. The last point for Duplication (1400) beyond the y-axis range is at $3.9s$.

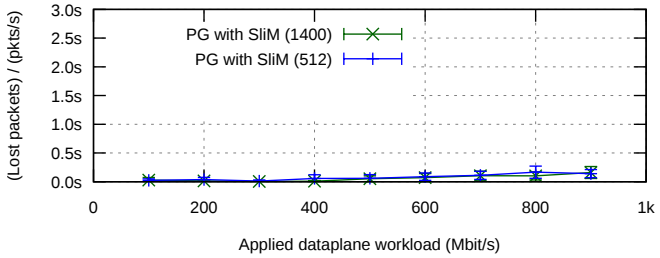

Fig. 12. "Seconds of packet loss", PG NF.

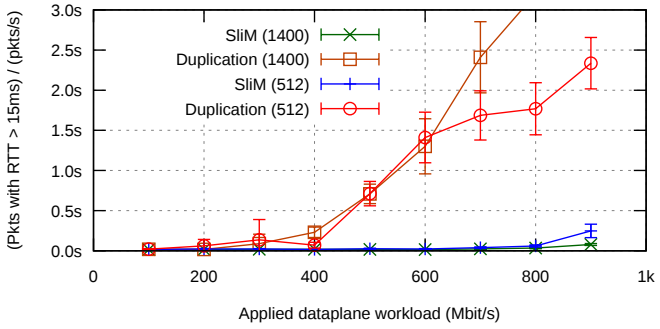

Fig. 13. Packets exceeding a RTT larger than $15ms$ divided by the packets per second, NAT NF. The last point for Duplication (1400) beyond the y-axis range is at $3.8s$. Beyond 200 Mbit/s, SliM clearly outperforms Duplication.

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

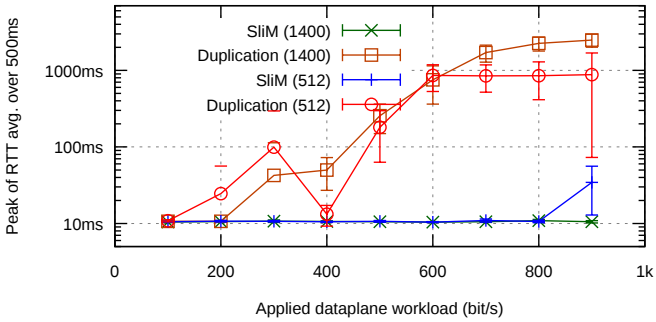

Fig. 14. NAT NF, 500ms intervals with max. mean RTT, NAT NF.

due to overload. Therefore, Figure 11 and 12 show the packet loss during the experiment in *seconds of loss*. The latter metric is the total number of packets that have been lost during one instance migration divided by the current packet rate per second[16], which makes it especially adequate to compare packet loss between different workload traffic rates. Besides packet loss, another important criterion for seamless behavior is latency. Figure 13 depicts the *seconds of RTT larger than 15ms*. It is defined like the seconds of loss, but also includes packets which have arrived later than $15ms$ (note that we have a baseline delay of $10ms$ caused by the load generator).

For the NAT NF, regarding both metrics, we can observe that no or only minor loss occurs when using SliM and Duplication at low rates. At 300M(bit/sec), Duplication starts to cause downtimes for both packet sizes, SliM however can avoid loss up to 800M, where Duplication already causes an outage of $\sim 1,5s$. SliM only starts to fail around 900M. A small loss occurs for the PG NF beyond 500MB (Figure 12), which is caused by the workload model's strict requirement that a handover must be in effect $7,5ms$ after announcement, otherwise packets are considered as lost.

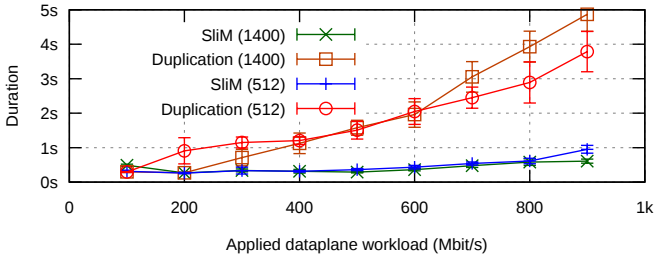

Fig. 15. Migration duration for 1400-byte and 512-byte packets, NAT NF.

The higher bandwidth consumption of duplicated packets negatively impacts queuing delays. In Figure 14, the large RTTs of Duplication are caused by packets enqueued in `pkt_q`, still waiting for the stream of duplicated packets to end for in-order delivery. This also applies to SliM, however

[16]For example, a loss rate of $1s$ can mean that for 1 second, no packet has been returned to the load generator, or that only half of the packets have been returned to it for 2 seconds.

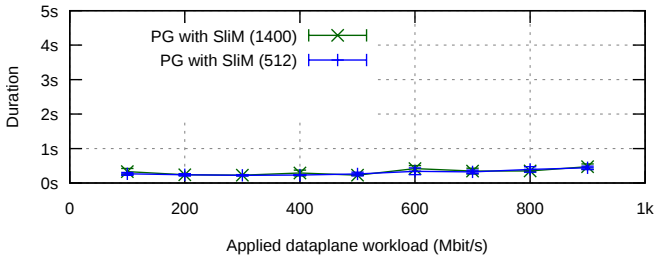

Fig. 16. Migration duration for 1400-byte and 512-byte packets, PG NF.

only beyond 850M, and results in RTTs not larger than 100 milliseconds in average. Figures 15 shows the total duration of an instance migration for the PG NF, where SliM can finish in a third of the time of Duplication at high data rates. Regarding the migration duration, both require less than $1s$ up to 100M, however, Duplication requires much more time with increasing workload and reaches $5s$ at 800M, where SliM still can finish in under $1s$. In summary, compared to Duplication, the performance results show that the link utilization can almost be tripled when using SliM, while maintaining seamless behavior. For the PG NF (Figure 16), the migration does not appear to be influenced by the dataplane workload as much as with the NAT NF, caused by statelets not being generated per packet, but only during cell transitions[17].

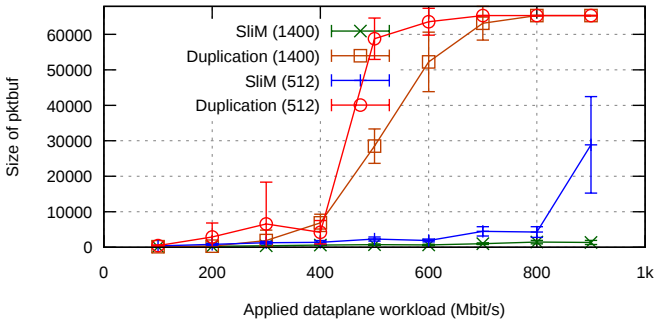

Fig. 17. The length of `pkt_q` for different workload bandwidths, NAT NF.

Figure 17 shows the maximum number of elements in `pkt_q` during the experiment. In our implementation, we had to limit the size of the buffer for `pkt_q` to 64K packets. If migration success is considered as an absence of any `pkt_q` overflow (which results in large packet loss), Duplication fails beyond 300M. Finally, the state transfer traffic for the NAT NF is shown in Figure 18, having a baseline requirement of 20MBytes for the snapshot size (the traffic for the PG NF has been measured with a nearly constant value of $\approx 23MB$, again, independent from the dataplane workload). We conclude that, compared to Duplication, SliM significantly reduces the traffic which must share capacity with the dataplane during migration.

[17]A very small tendency of longer duration after 600M could be explained with congestion on the link shared with the dataplane.

Leonhard Nobach, Ivica Rimac, Volker Hilt, David Hausheer: *Statelet-Based Efficient and Seamless NFV State Transfer.*
In: IEEE Transactions on Network and Service Management (TNSM), vol. 14, no. 4, p. 964-977,
December 2017. ISSN 1932-4537.

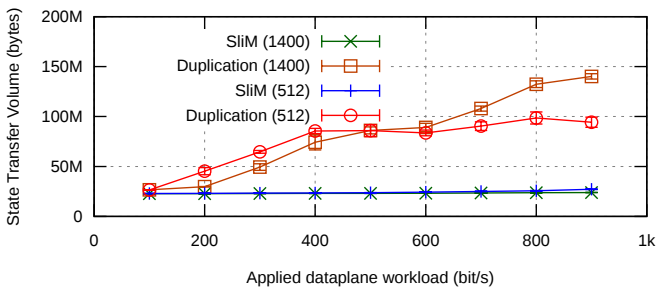

Fig. 18. State transfer traffic volume of the NAT NF, 20MB additional snapshot size.

VIII. Conclusion and Future Work

We have presented a new mechanism for bandwidth-efficient state migration, SliM, which only transfers the information in a packet required for the state synchronization between two or more instances of a VNF. We have implemented the proposed mechanism as a framework for DPDK, together with a NAT and a mobile packet gateway VNF implementation as a proof of concept. Evaluation results show a significantly higher performance of SliM compared to the state-of-the-art duplication-based approach with increasing bandwidth: With minor performance penalties caused by the migration, SliM is able to operate over links utilized up to three times the level at which duplication mostly fails.

For future work, we plan to extend the implementation with support for partial migration to allow for split and merge operations, as well as a small optimization to further reduce delay and jitter during switchover. In this work, we treated payload and state flows as best-effort, however, QoS strategies could decrease packet loss at the expense of increased migration time, or vice versa. We also plan implementing different types of VNFs over SliM, like an IDS, a CDN node, and a VPN. Furthermore, future development activities may include an evaluation on a large-scale distributed testbed, or the integration of SliM into a larger NFVI cloud platform like OpenStack.

Acknowledgements

This work has been supported by a Ph.D. internship at Nokia Bell Labs, and in part by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 - MAKI.

References

[1] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of Network and Computer Applications*, 52:11–25, 2015.

[2] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: Enabling innovation in middlebox applications. In *HotMiddlebox*. ACM, 2015.

[3] 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*, pages 273–286. USENIX, 2005.

[4] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, pages 161–174. USENIX, 2008.

[5] T. Dietz, R. Bifulco, F. Manco, J. Martins, H.-J. Kolbe, and F. Huici. Enhancing the BRAS through virtualization. In *NetSoft*, pages 1–5. IEEE, 2015.

[6] A. Gember, A. Krishnamurthy, S. S. 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.

[7] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *HotMiddlebox*. ACM, 2015.

[8] 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*. ACM, 2014.

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

[10] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level tcp stack for multicore systems. In *NSDI*, pages 489–502. USENIX, 2014.

[11] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions. In *HotSDN*, pages 49–54. ACM, 2015.

[12] J. Khalid, M. Coatsworth, A. Gember-Jacobson, and A. Akella. A standardized southbound API for VNF management. In *HotMiddlebox*, pages 38–43. ACM, 2016.

[13] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr. In *NSDI*, pages 239–253. USENIX, 2016.

[14] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *HPDC*, pages 101–110. ACM, 2009.

[15] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han. U-haul: Efficient state migration in NFV. In *SIGOPS*, page 2. ACM, 2016.

[16] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer. SliM: Enabling efficient, seamless NFV state migration. In *ICNP*, pages 1–2. IEEE, Nov 2016.

[17] V. Olteanu, F. Huici, and C. Raiciu. Lost in network address translation: Lessons from scaling the world's simplest middlebox. In *HotMiddlebox*. ACM, 2015.

[18] M. Peuster and H. Karl. E-State: Distributed state management in elastic network function deployments. In *NetSoft*, pages 6–10. IEEE, 2016.

[19] P. Raad, S. Secci, D. C. Phung, A. Cianfrani, P. Gallard, and G. Pujolle. Achieving sub-second downtimes in large-scale virtual machine migrations with LISP. *TNSM*, 11(2):133–143, 2014.

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

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

[22] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback recovery for middleboxes. In *SIGCOMM*. ACM, 2015.

[23] D. Williams and H. Jamjoom. Cementing high availability in OpenFlow with RuleBricks. In *HotSDN*, pages 139–144. ACM, 2013.