MANUSSCRIPT

A flow handover protocol to support state migration in softwarized networks

Manuel Peuster* | Hannes Küttner | Holger Karl

[1]Computer Networks Group, Paderborn University, Paderborn, Germany

Correspondence
*Manuel Peuster, Paderborn University, Warburgerstr. 100, 33098 Paderborn, Germany. Email: manuel.peuster@uni-paderborn.de

Summary

Softwarized networks are the key enabler for elastic, on-demand service deployments of virtualized network functions. They allow to dynamically steer traffic through the network when new network functions are instantiated or old ones are terminated. These scenarios become in particular challenging when stateful functions are involved, necessitating state management solutions to migrate state between the functions. The problem with existing solutions is that they typically embrace state migration and flow rerouting jointly, imposing a huge set of requirements on the on-boarded VNFs, e.g., solution-specific state management interfaces.
To change this, we introduce the *seamless handover protocol (SHarP)*. An easy-to-use, loss-less, and order-preserving flow rerouting mechanism that is not fixed to a single state management approach. Using *SHarP*, VNF vendors are empowered to implement or use the state management solution of their choice. *SHarP* supports these solutions with additional information when flows are migrated. In this paper, we present *SHarP*'s design, its open source prototype implementation, and show how *SHarP* significantly reduces the buffer usage at a central (SDN) controller, which is a typical bottleneck in state-of-the-art solutions. Our experiments show that *SHarP* uses a *constant* amount of controller buffer, irrespective of the time taken to migrate the VNF state.

KEYWORDS:
network function virtualization, software defined networks, flow migration, prototype implementation and testbed experimentation

1 | INTRODUCTION

The upcoming generations of networks will heavily rely on network softwarization concepts such as network function virtualization (NFV) and software-defined networking (SDN)[1]. Those concepts allow the dynamic deployment of virtualized network functions (VNFs) in different locations of the network[2,3]. Its main benefit is the possibility to add or remove additional resources on-demand, a process usually referred to as *(automated) scaling*. In such an elastic system, resources are not only added to existing VNF instances but new, replicated instances can also be started as needed (*horizontal scaling*). This leads to the problem that a NFV platform needs to dynamically reroute flows that are processed by the VNFs to distribute the load to new instances or to consolidate existing flows if instances are removed. While such traffic steering processes are executed, services should be interrupted as briefly as possible and no additional packet loss or reordering should occur[4].

Such elastic deployments become even more challenging when the involved VNFs are stateful and are required to maintain information about single or groups of flows, e.g., an intrusion detection systems (IDS) that maintains counters to keep track of the malicious packets it has seen. To tackle this problem, several state management solutions, like *Split/Merge*[5] or *OpenNF*[4], exists. They jointly manage the state migration between VNF instances and the traffic rerouting between them. The downside of these approaches is that they impose complex modifications of the VNF implementations in order to provide the required interfaces to extract and inject state information into the involved instances. We argue that this is a major obstacle for an interoperable and open NFV landscape. It requires VNF vendors to custom-tailor their VNFs to the NFV platform on which they should be on-boarded if they want to benefit from the state management solutions offered by these platforms.

To remove this obstacle, we present *SHarP*, a very lightweight traffic-steering solution for elastic VNF deployments that supports state management solutions (e.g. triggers for handover start) but leaves the actual choice of the state migration solution to the VNF vendor. The resulting system provides a clearer separation of concerns than existing solutions, making it a better fit for practical, real-world deployments.

The key contributions of this paper, which is based on a conference paper presented at IEEE NetSoft 2018[6], are as follows: We introduce our seamless flow handover protocol design that does not require a dedicated control interconnection between the SDN controller and the involved VNFs. Our handover protocol assigns the packet buffering tasks, required to provide a loss-free and order-preserving flow rerouting mechanism, to the destination VNF instances and thus reduces the load to the centralized SDN controller. In addition, we introduce the *handover support layer (HSL)*: a helper component that can easily be integrated into existing VNF implementations and requires fewer modifications than existing approaches, like the *FreeFlow* library used by *Split/Merge*[5]. Finally, we provide an extensive evaluation that first analyses the theoretic scaling behavior of our solution and compares it to *OpenNF*[4] before backing the theoretic expectations with a set of testbed experiments. These experiments verify that the controller buffer usage of the proposed approach scales well with the packet rate of the data plane and stays constant irrespective of the time required for state transfers between the VNFs. The results also show that our handover solution has only minimal impact, e.g., in terms of introduced delays, to the moved flows.

2 | RELATED WORK

Steering and moving flows between dynamically allocated VNFs is already well studied and several approaches, targeting different use cases like load balancing, service chaining, or scaling exist[7,8,9]. However, none of them provides supporting information and triggers to integrate with additional state management mechanisms and not all of them provide seamless handover mechanisms that do not introduce additional packet loss. As a result, the usefulness of these approaches for stateful VNFs is limited.

Other solutions that are designed to migrate state of virtual machine instances exist. But they come with a large overhead because they move much more state information than needed to operate a VNF[10]. In addition, more specific approaches that focus on joint traffic steering and state migration of VNFs have been proposed. The most prominent ones are *Split/Merge*[5] and its extension called *Pico Replication*[11], *OpenNF*[4] with its extensions[12,13], *CoGS*[14], as well as a novel approach called *SliM*[15] and a tagging-based solution presented in[16].

With *Split/Merge*[5], an orchestrator can migrate flows and move the corresponding function state using a simple API call. However, its failure recovery and migrate operation can cause lost or out-of-order state updates at the network function as flow processing is stopped during handover and arriving packets are dropped. In *Pico Replication*[11], the internal function state is cloned to other network functions at policy-defined intervals using modules that manage the packet flow of individual instances. The system uses OpenFlow to provide flow-level failure recovery by dynamically rerouting flows. Its focus is high availability rather than dynamic scaling of VNFs. While providing limited control over the desired functions, both systems fail at executing seamless handovers that are required to guarantee service availability and accuracy.

OpenNF[4] provides coordinated control of network function state and network forwarding rules. This framework consists of a central management application that uses a manager component to move state and flows from one instance to another. To integrate a VNF into the system, it has to implement a set of API functions that are used by the management component to pull and push state information. When the central control application decides to move a flow from one instance to another, it fetches the state from the source instance and pushes it to the destination instance. During this process, arriving packets are sent to a buffer at the controller until the state is transferred to the destination. The buffered packets are then forwarded to the SDN switch and from there to the destination instance. By using these mechanisms, *OpenNF* is able to perform loss-free and order-preserving flow move operations. A key concern with *OpenNF* is that it buffers the majority of the packets at the controller. The controller starts the buffering of all packets destined for the VNF as soon as the state is exported from the source VNF and only starts releasing the packets when the state is imported in the destination instance. This extensive usage of the controller as a proxy prevents *OpenNF* from scaling well with increasing amount of state and traffic volume. In the worst case, it can result in buffer overflow and lost packets at the controller, compromising the handover's safety.

To prevent buffer overflows the creators of *OpenNF* introduced an extension to *OpenNF* allowing the controller to drop packets from its internal buffer by utilizing a different method of packet buffering and state transfer[12]. Packets are duplicated at the source VNF, applied to the state, and sent to the controller to be buffered and then processed a second time at the destination VNF. This allows the controller to drop the packets from the buffer and simply restart the handover process since all packets are still applied to the state and the process of redirecting and buffering can be repeated. However, the reprocessing of the packets requires extensive modifications to the packet processing part of the VNF implementation as all output of the packet processing has to be suppressed[12] and therefore presents a bigger challenge for adopting the system than desirable. Additionally, duplicating packets at the source VNF and sending them to the controller uses the network path to the source VNF twice as much as it would previously. This can present a problem if the handover was executed to prevent a data plane overload at the source VNF.

DiST[13] improves on *OpenNF* by a peer-to-peer approach of transferring packets and states between VNFs. Instead of buffering packets and processing the state at the controller, the VNFs interact directly with each other over the data plane, reducing the controller link utilization to control messages only. This reduces the risk of overloading the controller or the control network. *DiST* uses the source VNF to redirect packets that cannot be applied to the state anymore to the destination instance where they are buffered. It generates additional load on the source VNF and the network plane as packets during the handover need to traverse the network path between source and destination VNF.

The authors of[16] present an in-depth analysis of OpenNF and propose small improvements to the system to reduce migration times. They also introduce a mechanism that follows similar ideas as *SHarP*. Their mechanism tags packets by utilizing the capability of SDN switches to modify unused packet header fields. The tags are used to identify affected flows and ensure a loss-free, order-preserving handover that only buffers packets at the VNFs. The number of parallel VNF migrations is however limited by the size of unused header fields that can be used for tagging. Their work is more theoretical and backs our findings of drastically reduced controller load when the majority of buffering tasks is moved to the destination VNF. In contrast to our *SHarP* prototype, their solution does not provide a flow detection mechanism to support the selection of the right parts of the overall state to be migrated. Further, the presented system relies on changes of the VNF implementations to export state, like OpenNF does, but its architecture appears to be compatible to the *handover support layer* approach introduced in this paper that removes this requirement.

In contrast to these approaches, which focus on joint state management and traffic steering, our approach (*SHarP*) focuses on the latter only. As a result, *SHarP* integrates much more flexibly by leaving the choice of the used state management approach to the VNF vendor instead of fixing it for the complete execution environment; even different state management schemes for different VNFs or groups of VNFs are possible. This simplifies the on-boarding of VNFs to different platforms since the platforms do not introduce any requirements for specific state-management interfaces. An example for a complementary state management solution is our *E-State*[17]; it works seamlessly with *SHarP*. Other distributed state management solutions, like the recently introduced *CoGS*[14], *SliM*[15] or *FogStore*[18] approaches, are also complementary to *SHarP* and could benefit from its loss-less flow migration procedures. In contrast to *OpenNF*, our system distributes the buffering process required for loss-less handovers to the destination VNF instances; this heavily reduces the controller load and provides better scalability.

Recent work by Yikai Lin et al.[19] proposes the concept of *programmable buffers* added to SDN switches to pause and resume network flows. They show how their solution can be used for mobility management as well as for flow migration in NFV scenarios. Future versions of *SHarP* could benefit from their concepts and programming abstractions by utilizing the additional buffering capabilities of the presented SDN switch solution. Sun et al.[20] focus on the question which flows are suitable for migration and propose control solutions for optimized migration decisions. This work focuses on the orchestration level and could utilize *SHarP* as migration mechanism.

This paper is an extended version of our conference paper[6]. Besides smaller improvements in all parts of the paper and the updated related work, we focused on extending our design and prototype descriptions in Section 3, highlighting *SHarP's* integration with the involved VNFs, its order-preserving features, and its support for bidirectional traffic. Finally, we substantially extended our evaluation in Section 4, analyzing not only the behavior of single handovers but also the overall system behavior to show that *SHarP* performs well even in agile environments in which services are reconfigured multiple times per second.

3 | SEAMLESS HANDOVER PROTOCOL (SHARP)

The design of our handover protocol follows two main goals. First, the flow handover mechanism has to explicitly support state migration procedures but should not mandate any specific state migration solution. Second, our solution will offer improved scalability compared to existing approaches, specifically it should reduce the load on the central controller by minimizing the number of packets the controller has to buffer to ensure a loss-less and order-preserving handover.

To achieve these design goals, we defined the following set of requirements: The first requirement for a handover mechanism is a *flexible flow selection (R1)* interface that allows to select single flows as well as groups of flows that shall be moved from one VNF to another. These handovers should be performed as fast as possible to *minimize service interruption times (R2)* and they have to ensure that they do *not introduce additional packet loss or packet reordering (R3)*. To be able to handle many flows, the *scalability (R4)* in terms of control load and buffer usage is important. Finally, a handover mechanism has to be designed for *compatibility (R5)* and not require specific modifications from VNF implementations to accommodate a wide range of different VNFs.

3.1 | Handover scenario

SHarP is designed to work with networks that contain at least two SDN switches: an ingress and an egress switch as shown in Figure 1. Our design extends to any number of switches, yet to simplify presentation, we limit ourselves here to only two switches; evaluation results do not depend on number of switches. Between the switches, multiple VNF instances are located and their dataplane interfaces are connected with one port to either switch. In addition to this, the VNFs are connected to a management network that allows them to exchange information in a peer-to-peer manner. Data flows enter the *SHarP-enabled* VNF deployment from a source ($Host_1$) through the ingress switch, traverse one VNF instance (or a chain of multiple VNF instances), and leave the system through the egress switch towards the destination ($Host_2$). Bi-directional flows in which packets are sent from the destination ($Host_2$) to the source ($Host_1$) are also supported (Figure 1). Flows can be moved between VNF instances using the proposed handover mechanism by triggering the handover procedure through the northbound API of the controller. For example, the flow shown in Figure 1 will be moved from VNF_1 to VNF_n.

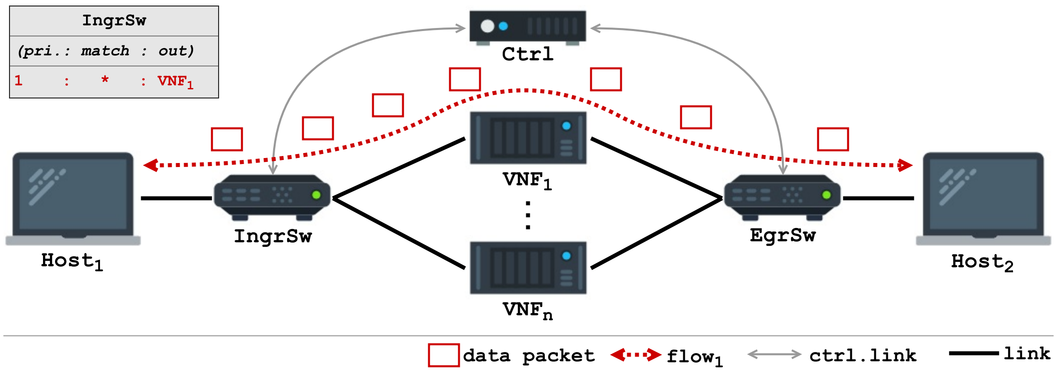
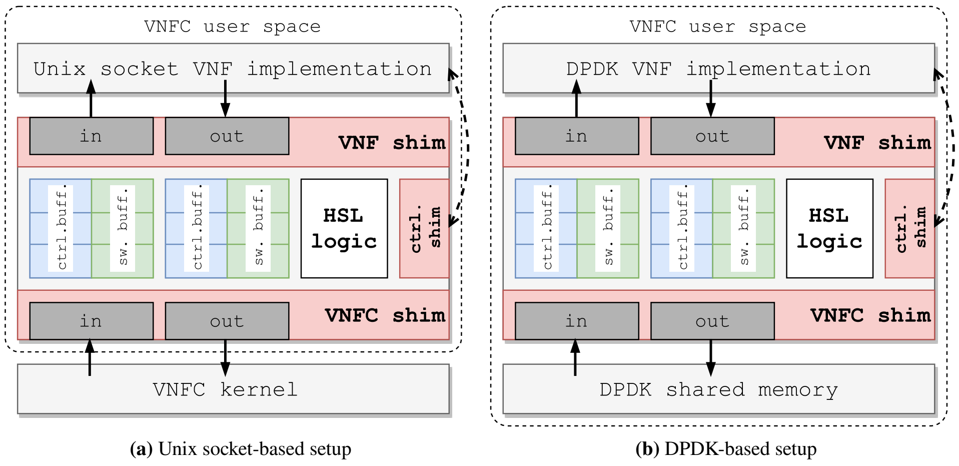


FIGURE 1 Example network with multiple VNF instances, *ingress* and *egress switch* as wells as a data flow processed by VNF_1

The involved VNFs do not need a direct connection to the controller as this is not commonly the case and thus would impose a needless requirement. Instead, control messages sent by the controller to the VNFs are forwarded by the switches and intercepted by an intermediate software layer that is running inside the VNF's container (or VM). This layer also buffers packets as required to ensure loss-free and order-preserving handovers (described in Section. 3.2). We assume that the links of the example networks do not introduce any additional packet loss or packet reordering.



(a) Unix socket-based setup

(b) DPDK-based setup

FIGURE 2 Handover support layer (HSL) sitting between VNFC and VNF implementation

3.2 | Transparency towards VNF and state management

One of the main requirements for *SHarP* is to be as transparent as possible towards VNF implementations that operate in a *SHarP-enabled* environment (R5). This also means that *SHarP* does not enforce the use of a particular state management or state sharing framework. Instead, it provides the means to assist state sharing solutions, like E-State[17], with functionalities to *pause and buffer* incoming flows or to *inform* the actual state migration solutions when a handover is performed by the network. This functionality is completely encapsulated in an additional software layer, called *handover support layer (HSL)*, that is located between the actual VNF implementation and the network interfaces of the VNF container (VNFC) as shown in Figure 2. This software layer acts as a bridge and is able to forward packets between the interfaces of the VNFC and the VNF implementation. In addition, it implements a control logic that intercepts control messages sent by the *SHarP* controller through an SDN switch over the data plane of the system. Those control messages allow the *SHarP* controller to trigger events, like preparing the destination VNF for a handover, without requiring a direct connection between controller and VNF. Besides this control logic, packet buffers are implemented and used to buffer incoming packets when the destination VNF is not yet ready to process them, i.e., the state transfer from the source VNF has not completed. Optionally, the HSL offers a control channel to the VNF implementation used to inform the VNF about the status of the handover, e.g., to trigger its state migration mechanism. We leave it to the VNF to prepare and migrate all state belonging to the flows that are handed over. This allows us to transparently handle multiple VNF implementations without needing information about the internal state structure, a major difference to *OpenNF*[4].

All interfaces of HSL are implemented as modular, plugin-like components (shims) that can easily be replaced to make the HSL agnostic to different data layer interfaces. Besides the standard UNIX socket shim shown in Figure 2a, more NFV-specific implementations are possible; for example, HSL shims that are based on DPDK[21] as shown in Figure 2b.

3.3 | Handover procedure

SHarP's handover procedure can be split into three main phases. They are shown in Figure 3 for handing over a single unidirectional flow from VNF$_1$ (source VNF) to VNF$_n$ (destination VNF); it also shows the forwarding table entries of the ingress switch (`IngrSw`). We decided to show the handover of an unidirectional flow to keep the figures clean and understandable. *SHarP* supports the handover of bidirectional flows by performing symmetric handover steps on the ingress and the egress switch at the same time as described in Section 3.7.

At the beginning of the first phase, the scenario looks like the one shown in Figure 1 in which all flows between Host$_1$ and Host$_2$ are processed by VNF$_1$. A handover is triggered by a request to the northbound API of the *SHarP* controller (`Ctrl`) and contains an OpenFlow-like matching rule for a flow (or a group of flows) to be moved, a priority r for the handover request, as well as the identifier (e.g., MAC address or switch port ID) of the destination VNF to which the flows should be moved. The priority r allows our system to organize the handover procedure among multiple handover requests and allows the user of the system, e.g., an NFV orchestrator, to overwrite existing handover rules. A *SHarP* handover request does not require any further knowledge about the state of the network, in particular, the requesting external entity, e.g., an NFV orchestrator, does not need to know by which VNFs the flows matching the request are currently processed (R1/R5). In the example given in Figure 1, the handover request will move all flows from VNF_1 to VNF_n by using a wildcard ($*$) in its match field.

Once the request arrives at `Ctrl`, it installs a so-called *flow detection table entry* on `IngrSw` that matches all flows specified by the handover request and forwards their packets to `Ctrl`. The priority of this entry p_t is set to $p_t = 2r + 1$ so that there is room for another table entry belonging to this handover request with priority r. Using this fixed mapping of handover rule priorities r to forwarding table entry priorities t_p on the switch ensures a clear separation of forwarding entries belonging to different handover requests. Next, a second table entry is installed that matches the same flows but forwards their packets to the destination VNF_n. This entry has priority $2r + 0$ such that it will only be used once the *detection table entry* is removed.

Figure 3a shows how incoming packets from $Host_1$ are matched and forwarded to `Ctrl`, which buffers them. Packets that are still processed by VNF_1 leave the system via `EgrSw`. In this state, the controller learns about all flows that are affected by the handover and can generate exact match entries for each of these flows to hand them over one by one. To do so, one exact table entry for each flow is installed in `IngrSw` which forwards all packets of this particular flow to `Ctrl`. These exact entries implicitly have the highest priority since no wildcard fields are used anymore[†]. The detection phase stays active until a *maximum silence time*, which is set as the idle timeout of the *detection entry* is reached and the *detection entry* is removed from `IngrSw`. Flows that have not been detected during this time are treated as new flows by our system. They are directly forwarded to VNF_n by the table entry with priority $2r + 0$. When the detection phase is over, `Ctrl` sends `START_HO` messages to the involved VNFs using a `PACKET_OUT` event on `IngrSw` to inject them into the data plane. The controller knows the destination VNF from the handover request and the source VNF by utilizing the controller internal knowledge about the previous network configuration. The HSL in the VNFs intercepts the control message and can, e.g., trigger the preparation of the state transfer before replying with acknowledgments as shown in Figure 3a.

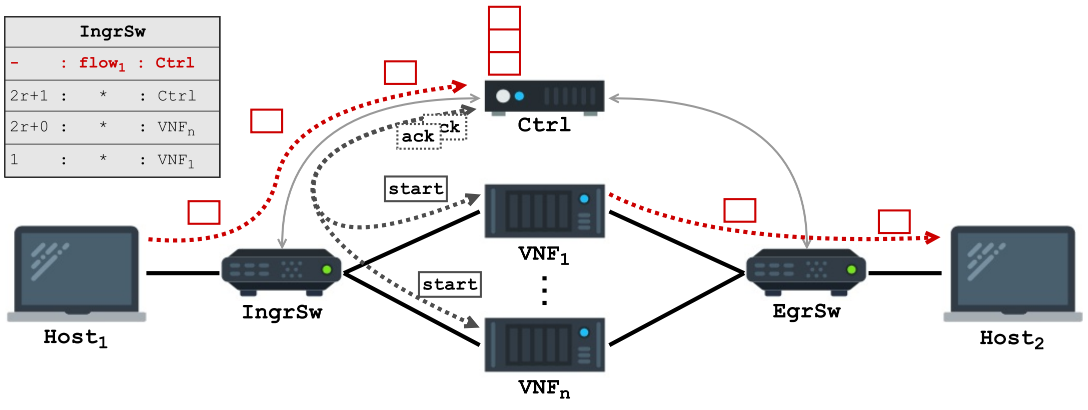
Once `Ctrl` receives the `ACKs` it enters the second phase of the handover procedure that is shown in Figure 3b. Immediately after this phase has started, `Ctrl` starts to mark (e.g. by VLAN tag or encapsulation) and release the packets from its buffer and sends them towards the destination VNF_n via `IngrSw`. VNF_n detects the marked packets and puts them in its internal `ctrl_buff` because it knows that they have been buffered at `Ctrl` before. At the same time, `Ctrl` updates the exact forwarding table entry to forward all new packets of the flow arriving at `IngrSw` directly to VNF_n. At VNF_n, the packets are buffered in the internal `sw_buff` of the VNF to not mix them up with the packets previously buffered at the controller (important for R3).

One problem at this point is that `Ctrl` needs to know when it has received all packets that are not already forwarded to VNF_n. But there may be packets that are still in flight between `IngrSw` and `Ctrl`. To solve this, `Ctrl` instructs `IngrSw` for a short time to duplicate and flag packets (`BUFFER_FOLLOW_UP`) that are forwarded to VNF_n and to send the flagged copy of them also to `Ctrl`. In this configuration, `Ctrl` can inject a test packet into the data plane at `IngrSw` and will immediately know that it has seen all packets not yet forwarded to VNF_n once it sees the test packet. Thus, `Ctrl` knows that it does not need to buffer any new packets and removes the packet duplication configuration from `IngrSw`.

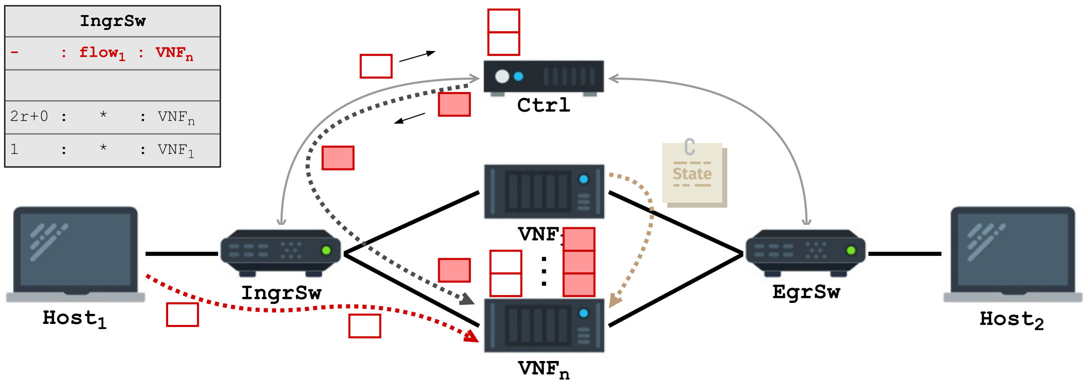
During the entire second phase shown in Fig 3b, no traffic is processed by any of the VNFs and all arriving packets are buffered in the two buffers of the destination VNF_n. In this state, the VNFs can trigger their state management solutions, which can transfer the VNF's internal states in a peer-to-peer fashion over the management network between the VNFs. HSL can support these state management solutions by giving them information about the source and destination VNF as well as the exact flow identifier.

The third phase of the handover, shown in Figure 3c, is entered once `Ctrl` has released all its buffered packets and the state management mechanism at the VNFs indicates that all state has been moved. The HSL then immediately starts to release the buffered packets towards the VNF implementation of VNF_n to be processed using the state that has been moved from VNF_1 to VNF_n in the previous step. It first releases its `ctrl_buff` and afterwards its `sw_buff` to ensure that all packets are processed by VNF_n in the same order as they have entered the *SHarP* system (see Section 3.6). Finally, `Ctrl` can remove the additional handover table entries from `IngrSw` and reach a stable system state in which all flows involved in the handover are processed by VNF_n. More details, like control packet formats and handover rule removal procedures are described in[22].

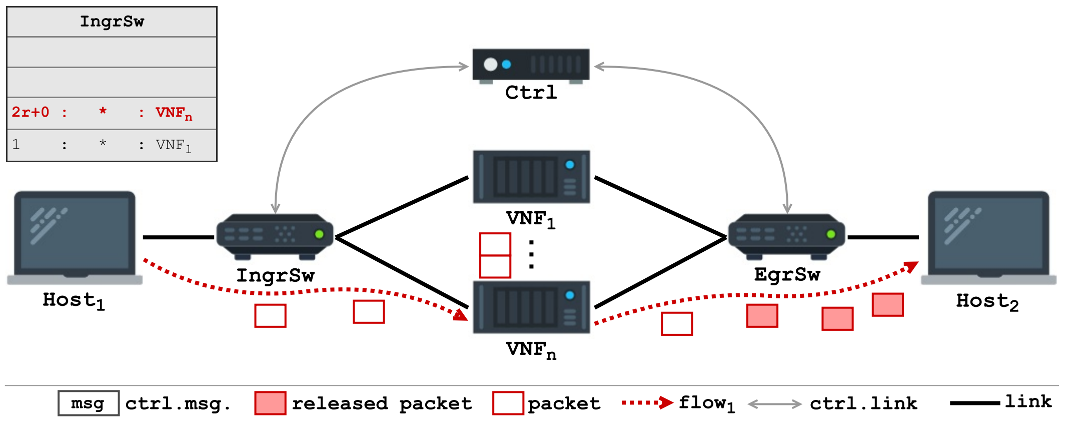
[†]http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt



(a) Phase 1: Flow detection and handover initialization. New packets are buffered at the controller.



(b) Phase 2: Installed temp. forwarding entry to destination VNF$_n$. Buffering done at destination VNF$_n$ to allow early controller buffer release and load distribution. Trigger of state transfer solution.



(c) Step 3: Final forwarding state reached and state migration finished. Release and replay of buffered packets at destination VNF$_n$.

FIGURE 3 Three phases of *SHarP's* handover procedure for a flow moved from VNF$_1$ to VNF$_n$

3.4 | Handover from a VNF's perspective

As described in the last section, *SHarP* relies mainly on the destination VNF instance to buffer incoming packets during state transfers. To do so, it uses the HSL, which is able to communicate with `Ctrl` over control messages exchanged using the data plane of the system. We present this in more detail using Figure 4 which shows the detailed flow of our example handover from the perspective of the involved VNFs. The figure shows how the involved components interact with each other, starting with `Ctrl` that informs the involved VNFs about the start of the handover, by sending them `START_HO` messages, once the flow detection is completed and all incoming packets are buffered at the controller (see Phase 1 in Figure 3). When the control message

START_HO arrives at VNF$_1$ (1), the HSL running inside that VNF (VNF1_HSL) intercepts it and notifies the VNF implementation (VNF1_IMPL) about the handover and about the flows that will be moved away from VNF$_1$ (2). VNF1_IMPL can then freeze its state, prepare it for migration (3), and acknowledge this action. At this point, no more packets are arriving at the VNF and thus no new updates to the internal state appear. At the same time, Ctrl also informs VNF$_n$ about the upcoming handover (6), which also prepares its VNF implementation VNFn_IMPL for the upcoming state transfer, i.e., triggers VNF$_n$ to request the state from VNF$_1$ in the next pahse.

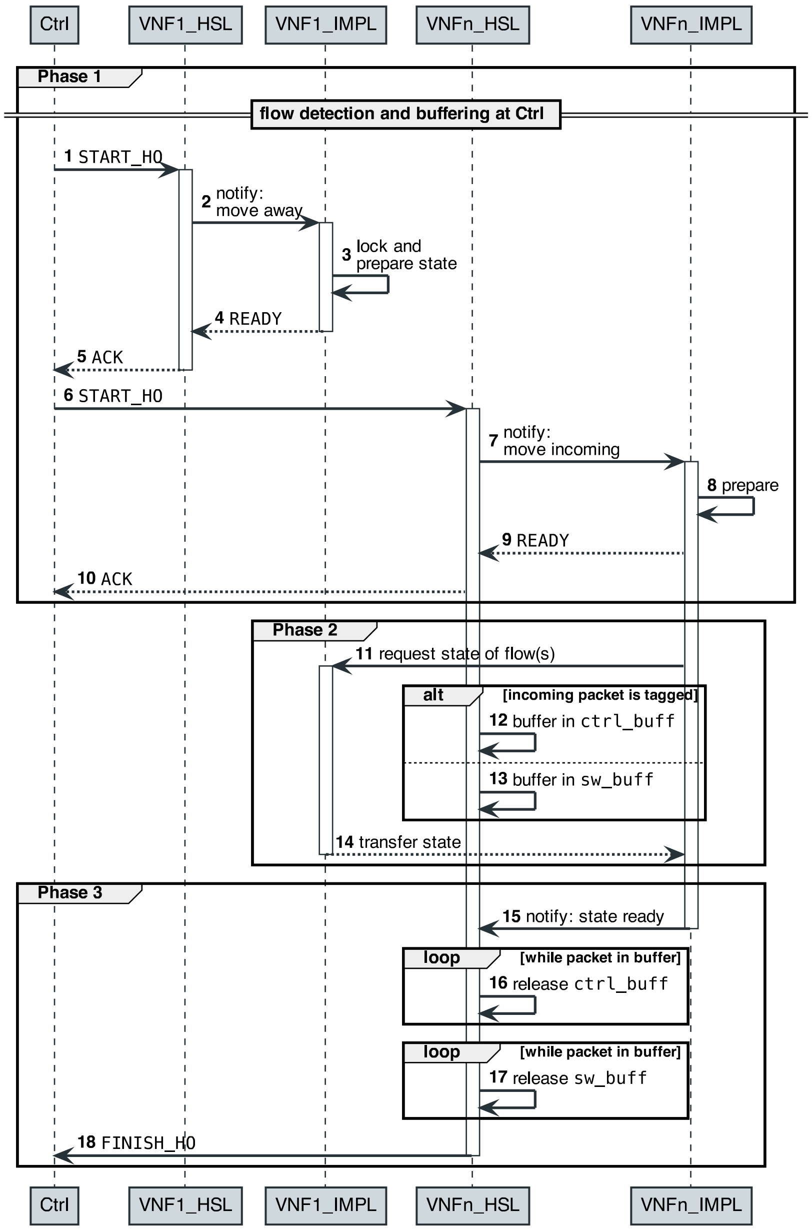


FIGURE 4 Handover procedure from the perspective of the involved VNFs and their HSLs.

Once Ctrl received all ACKs, it enters the second phase of the handover in which VNF$_n$ buffers all incoming packets inside its HSL. Those buffered packets are distinguished by tags so that VNFn_HSL knows whether a packet was previously buffered at Ctrl or not. Based on this, the packets are either buffered in ctrl_buff (12) or sw_buff (13). During this phase, VNFn_IMPL triggers the actual state migration by requesting a state transfer from the source VNF$_1$ (11), which transfers the state to the destination VNF$_n$ (14). This state transfer might take an arbitrary amount of time, depending on, e.g., the used state management solution and network conditions, in which the `VNFn_HSL` keeps buffering incoming packets. Once the state migration is done, the `VNFn_IMPL` notifies the HSL (15) and *SHarP* enters its third phase. The `VNFn_HSL` now starts to release the buffered packets, first from `ctrl_buff` (16) and then from `sw_buff` (17) to prevent reordering of packets. When both buffers have released all packets, VNF_n can continue with its normal operation and finally notify `Ctrl` about the finalized handover (18).

One of the benefits of *SHarP* is that it does not require a dedicated control channel between `Ctrl` and the involved VNFs. Instead, all control messages are injected/fetched from the switch and sent over the data plane to the VNFs, where the HSL intercepts them (R5). The used control messages are encapsulated inside standard Ethernet frames using `EtherType=0x821c` not to interfere with other protocols. The payload of each control message contains a command code to express its functionality, an identifier used to reference the handover to which the control message belongs, and a sequence of type-length-values (TLV) fields. Those TLVs hold additional context information, like the matching rule used for the handover or its priority. If control messages are sent from `Ctrl` to a VNF, the destination VNF's Ethernet address is used as destination address of the control message. If control messages are sent from a VNF to `Ctrl`, the address field is left empty, as the switch detects all control messages matching their `EtherType` and forwards them to `Ctrl` using `PACKET_IN` events.

3.5 | Removing buffer load from the controller

For a seamless handover, packets need to be buffered while the state is synchronized between the VNF instances and no state updates can be performed. Later, the buffered packets can be released to the destination instance to be applied to the state. In *OpenNF*[4], packet buffering takes place completely at the controller which may lead to performance issues. The controller can quickly be overloaded if the amount of packets to be buffered is large, i.e., because of a long-lasting state transfers. Our system design, in contrast, reduces the buffer load of the controller by moving the responsibility to buffer incoming packets during a state transfer to the destination VNF instance. The *SHarP* controller only needs to buffer packets during the small period of time in which the handover is initialized (Figure 3a) and tries to release this buffer as early as possible (R2). In particular, the buffer is released before the actual state transfer is started, which makes the controller buffer usage of *SHarP* independent of the state transfer. We will show this property in more detail in our evaluation (Section. 4).

Buffering most of the packets directly at the destination instance has the additional advantage of using the capacity of the destination VNF instance. A VNF only needs to buffer the packets belonging to flows that are redirected to that instance and not of all handovers in the network, further improving scalability of the entire system (R4).

3.6 | Preserving packet order

One of our key requirements is to not introduce additional packet reordering (R3) into the data plane of the system when flows are moved between VNFs. This requirement is important because VNFs might apply changes to their internal state in the wrong order, causing inconsistencies or even information loss when their state is migrated to another instance. Assuming that the network links between the hosts, the switches, and the VNFs as such do not introduce any packet reordering, SHarP preserves packet order as follows.

Let $(p_1, \dots, p_k, p_{k+1}, \dots, p_l, p_{l+1}, \dots, p_m, p_{m+1}, \dots, p_n)$ be n packets sent in a flow from Host_1 to Host_2. The subset containing packets $(p_1, \dots, p_k)$ is sent to VNF_1 before the handover takes place. They arrive in the order they were sent from Host_1 and are applied to the state $S\langle\rangle$, resulting in state $S\langle p_1, \dots, p_k\rangle$ in VNF_1.

Once the handover is triggered and the flow detection table entry is in place, packets $(p_{k+1}, \dots, p_l)$ are matched by the detection entry and are sent to the controller, which buffers them while VNF_1 is preparing for the handover (Phase 1, Figure 3a). Assuming a FIFO buffer and taking into account the fact that the packets are encapsulated into OpenFlow `packet_in` messages delivered over TCP, the controller will maintain the order of packets $(p_{k+1}, \dots, p_l)$ in its buffer.

After the update of the flow table entries, at the beginning of Phase 2, packets $(p_{l+1}, \dots, p_m)$ and all subsequent packets are forwarded from Host_1 to VNF_n and not to the controller anymore. Based on our initial assumptions about the network infrastructure, these packets arrive at the HSL of VNF_n in-order and are buffered in the *switch buffer* (FIFO) of the HSL as shown in Figure 2. At the same time, packets $(p_{k+1}, \dots, p_l)$ that have been buffered at the controller during Phase 1 are now released and sent to VNF_n as well. These packets are released in-order, encapsulated into OpenFlow messages, sent to the ingress switch, and finally delivered to VNF_n over the normal data plane. Thus, it can be assumed that they arrive in order at the HSL of VNF_n, where they are buffered (FIFO) in a second buffer (*control buffer* shown in Figure 2) to ensure that they are not mixed with packets $(p_{l+1}, \dots, p_m)$. In this phase, state $S\langle p_1, \dots, p_k\rangle$ is migrated from VNF$_1$ to VNF$_n$ so that VNF$_n$ is ready to continue the packet processing starting with packet p_{k+1}.

Once the state migration procedure has finished, *SHarP* enters Phase 3 and the HSL of VNF$_n$ starts to release the buffered packets. At first, packets $(p_{k+1}, \dots, p_l)$ are released from the *controller buffer* and are processed by VNF$_n$ which updates the migrated state accordingly resulting in state $S\langle p_1, \dots, p_k, p_{k+1}, \dots, p_l\rangle$. After that, the packets from the *switch buffer* of the HSL are released and also processed by VNF$_n$ resulting in $S\langle p_1, \dots, p_k, p_{k+1}, \dots, p_l, p_{l+1}, \dots, p_m\rangle$. At this stage the handover is complete and newly incoming packets $(p_{m+1}, \dots, p_n)$ are directly processed by VNF$_n$, resulting in the complete, ordered state $S\langle p_1, \dots, p_k, p_{k+1}, \dots, p_l, p_{l+1}, \dots, p_m, p_{m+1}, \dots, p_n\rangle$ in VNF$_n$.This shows how *SHarP* maintains the correct oder of packets by releasing the involved buffers one after each other. This ensures that the VNF state is correctly updated during the handover procedure (R3).

3.7 | Bidirectional handover

As previously mentioned, we use a unidirectional flow to describe the handover procedure for presentation reasons. Nevertheless, *SHarP* also supports migrating bidirectional flows between VNFs, as we show in our evaluation. Handing over a bidirectional flow is effectively a combination of two unidirectional flow handovers and offers the same features as the previously described unidirectional case. Design-wise, the controller needs two (instead of one) packet buffers to independently buffer packets from both switches (`IngrSw`, `EgrSw`), which can be independently released during the handover procedure. To trigger a bidirectional handover, the same API call with an additional flag (`bi_ho:bool`) is used. The controller then automatically generates the matching rules for both switches by swapping the source and destination fields of the rules installed on `EgrSw`. For the example given in Figure 3, this means that `IngrSw` is responsible to move the packets sent from Host$_1$ to Host$_2$ and `EgrSw` is responsible to move the packets sent from Host$_2$ to Host$_1$. The rules are then installed on both switches in the same way as it is done in the unidirectional case. It might happen that the installation of rules on one switch is delayed and does not happen simultaneously with the rule installation on the other switch. This has no impact on *SHarP* since all arriving packets are already buffered at `Ctrl` at this stage. Next, `Ctrl` informs the involved VNFs about the upcoming handover; this has to be done over both possible paths, namely by duplicating the control messages and sending them over `IngrSw` and `EgrSw` at the same time. The involved VNFs then wait until they received control messages on both interfaces before they acknowledge the handover. This ensures that no packets are in flight between switches and VNFs when the next phase is entered in which the handover is executed as described in Section 3.3. Another requirement for the bidirectional handover is the availability of four (two for each direction) buffers in the HSL as already depicted in Figure 2.

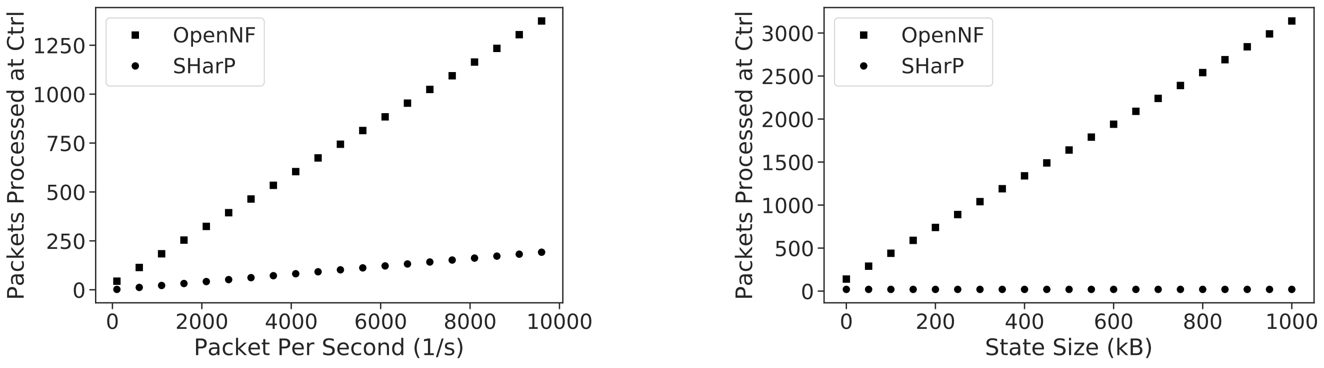
4 | EVALUATION

We analyzed *SHarP* to highlight the improvements compared to *OpenNF*. This theoretical analysis is than backed by a set of experiments performed with our *SHarP* prototype and validates that our handover protocol behaves like expected, e.g., no packet loss or reordering occurs and the controller buffer usage remains constant even when the state migration time increases. We used the following metrics to characterize the performance of our system: The handover duration (1), maximum packet delay introduced by handover (2), controller buffer usage (3), VNF buffer usage (4), packet loss (5), and packet reordering (6).

Our results present these metrics as a function of data plane data rate and the duration it takes the VNFs to migrate their state. The maximum packet delay is the main indicator for the delay introduced into the service as the handover is executed. The buffer usage at the controller and at the VNF indicate how well *SHarP* fulfills the claim that only a small amount of data has to be buffered and processed at the controller. Further, we show how *SHarP* behaves under load when multiple handover requests for many flows arrive within less than a second.

4.1 | Theoretical evaluation

In *OpenNF*, a loss-free order-preserving handover requires a transmission of a total of $3N + 2R + C$ messages over the control plane, where N represents the number of state messages, R the number of redirected packets, and C a constant number of control messages. In SHarP, only $2T + C$ messages need to be sent over the control channel, where $T \leq R$ is the subset of the redirected packets that is buffered at the controller. Additionally, $V + 2R + C$ messages are transmitted over the data plane,



(a) Processed packets as a function of packets per second with a fixed state size of 10kB

(b) Processed packets as a function of state size with a constant packet rate of 1000 pps

FIGURE 5 Processed packets at the controllers of OpenNF and SHarP for a handover with a duration of 70ms

where V is the number of messages needed by the VNF implementation to migrate state between VNFs which is not under our control. We can assume that V is close to N if the state transfer is implemented efficiently. In both approaches, the overall cost of a handover scales with the amount of state to transfer and the number of redirected packets. In SHarP, however, that cost is incurred mostly at the destination VNF and only partly at the controller. Furthermore, T, the number of packets processed at the controller in SHarP, only depends on the packet rate and the network delay, not on the state transfer duration, as packet buffering is outsourced to the destination VNF after a short period of time.

Figure 5 shows the comparison of the amount of packets a controller in *OpenNF* and *SHarP* has to process during a handover with an execution time of 70 ms and an initial signalling period of 10 ms. In Figure 5a the size of the state that has to be transferred is set to 10 packets of 1000 bytes each, which is a realistic estimate for state sizes[5]. The packet rate of the flow during the handover is increased from 100 packets per second to 10 000 packets per second. It can be seen that with a higher packet rate the increase of packets processed at the *OpenNF* controller is vastly higher than the packets processed in *SHarP*. In Figure 5b, the number of packets the controllers of both protocols have to process is shown in relation to the migrated VNF state size. The packet rate of the flow during the handover is fixed to 1000 packets per second while the state size is increased from 1 to 1000 kilobytes. The difference of both approaches is clearly visible as the number of processed packets in *OpenNF* increases linearly while it is constant in *SHarP*. This is a significant advantage of *SHarP* over *OpenNF*, as it, from a controller perspective, allows exceedingly better scalability independent of the VNF state size.

The network delay on the control path and between switch and VNFs in combination with the packet rate also has a non-negligible impact on the amount of packets processed by the controller. As the previous estimation showed, the amount of packets is directly linked to the packet rate and the time it takes the controller to notify the participating VNFs. This initial time is composed of the delay between the controller and the switch and two times the delay between controller and VNFs plus the time it takes the controller to release its buffer.

Figure 6 shows the processed packets as a function of total network path delay at different packet rates. For estimating the release time of the controller buffer, we assume that the controller can release an average of 10 000 packets per second with a packet size of 1000 bytes, which saturates about 1% of a 10 Gbit/s interface. The total network delay shown in the figure corresponds to the sum of all delays that occur during the initial signaling period. More specifically, it includes the delay between `Ctrl` and `IngrSw` and the delay between `IngrSw` and the VNFs. The figure clearly shows that the network delay has a strictly linear influence on the number of packets the controller has to process (as was to be expected).

4.2 | Experimental evaluation

We implemented a prototype of the *SHarP* controller based on the *Ryu SDN Framework*[23]. Our prototype offers an easy-to-use, RESTful northbound interface that offers the required functionalities to trigger handover procedures between arbitrary VNF instances. In addition to the controller prototype, we implemented a Python-based HSL prototype that acts as a bridge between the VNFC and the actual VNF implementation using standard Unix sockets as shown in Figure 2a. The use of Python limits the

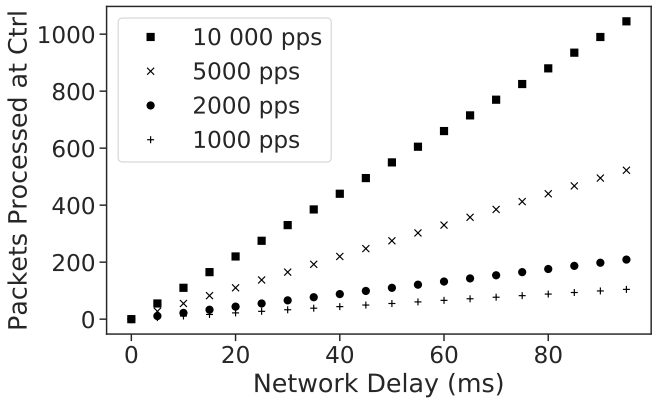


FIGURE 6 Influence of the total network path delay between controller, switch and VNF on the number of packets processed at the SHarP controller at different packet rates

throughput of the HSL prototype but still allows us to evaluate *SHarP* in terms of buffer usage and handover performance. A high-performance implementation of the HSL using DKDK[21] is planned as future work. Both prototypes (SHarP and HSL) are open source and available on GitHub[24].

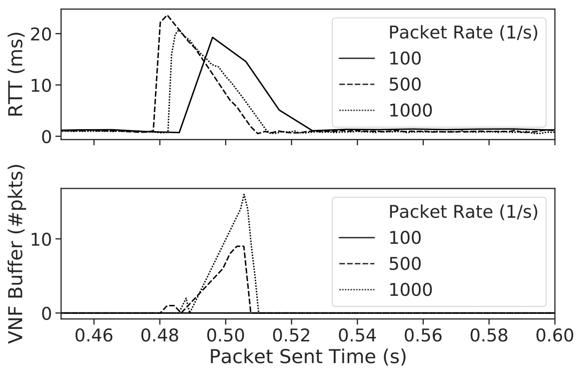
Using these prototypes, we executed a set of experiments to evaluate the performance of the proposed handover protocol. These experiments have been executed on an SDN testbed based on the emulation framework *Containernet*[25] running on a server with an Intel(R) Core(TM) i7 CPU 960 @ 3.20 GHz and 24 GB memory. The used network topology was the same as shown in Figure 1 consisting of two hosts, two switches, and two VNFs that are able to forward arbitrary traffic between their input and output interfaces. Both the hosts and VNFs are represented by Docker (1.12.3) containers connected to the emulated network created by Containernet (2.3.0d1) containing two Open vSwitches (2.5.2). Our prototype controller is implemented on top of Ryu 4.13.

4.2.1 | Handover characteristics

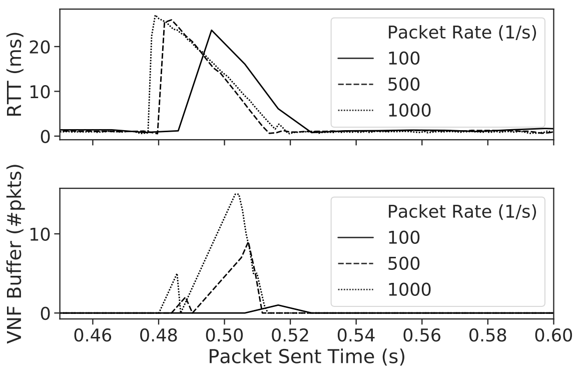
The first part of our experimental evaluation analyses handovers performed with our prototype and they impact the rerouted packets and the involved buffers. During the experiments, a constant UDP traffic flow is generated on $Host_1$ and sent to $Host_2$ over the first VNF. $Host_2$ receives the packets and sends them back to *Host1*, creating a bidirectional traffic flow which is then handed over to VNF_n by our *SHarP* controller. During this procedure, we collect the metrics mentioned before as follows: First, each of the packets is identified by a unique sequence number so that any lost, reordered, or duplicated packet can be easily identified. Second, the round trip time (RTT) of the packets is measured at $Host_1$ to identify packet delays that are introduced by the execution of a handover. Third, we measure the buffer usage at the VNFs as well as at the *SHarP* controller during the entire experiment. Finally, the total handover duration, which is defined as the time taken between the initial handover request and the final migration of the flow to the destination VNF, is measured at the controller.

The first set of experiments focuses on a single handover procedure, with state transfer duration set to 0 s, to analyze what happens to the packets and VNFs during a flow migration. The upper parts of Figure 7 show packet delays over experiment time with the handover happening at about 0.4 s. The results show how the delay of the packets quickly increases to about 25 ms before they normalize again after about 5 ms. Except for smaller variation, this effect remains the same for different packet rates and packet sizes. The lower parts of Figure 7 show the number of packets stored in the destination VNF buffer during the handover. Depending on the packet rate, different numbers of packets need to be buffered. All of them are quickly released, once the handover is done. These results show that *SHarP* provides a stable and predictable handover solution.

In the second set of experiments, we executed handovers with different packet rates, packet sizes, and state transfer durations. Each configuration was executed 100 times, each with a fully restarted network and controller setup to eliminate side effects from previous runs. All error bars in this paper show 95% confidence intervals. The goal of these experiments is to analyze the general behavior of *SHarP* under different conditions. The first set of results given in Figure 8 shows the handover performance as a function of the data rate of the moved flow given as packets per second. The results shown in Figure 8 are based on measurements using a packet sizes of 58 bytes and 1000 bytes. As shown by the figures, the packet size has no impact on the handover duration



(a) Handover behavior for different packet rates using 58 byte packets



(b) Handover behavior for different packet rates using 1000 byte packets

FIGURE 7 Packet delay and VNF buffer state during a single handover for different packet rates and packet sizes

or packet delays, but obviously accounts for more buffer usage if larger packets are used. During all experiments, no packets were lost, reordered or duplicated, which verifies the seamless nature of our handover mechanism.

Figure 8a shows that the overall handover duration yields a linear increase with the increased packet rate, since more packets need to be processed. The maximum packet delay introduced by the handover procedure is shown in Figure 8b. It starts by increasing linearly and ends with the delay stagnating around 27 ms. This maximum delay has an upper limit in the time it takes the controller to notify the VNF about the handover and the VNFs to synchronize the state. If there is no state to be exchanged the packet delay stagnates towards the end since the round-trip time between controller and VNF does not increase. The buffer usage of the controller and the VNFs is shown in Figure 8c and Figure 8d, respectively. As the packet rate increases, the entire system has to buffer more packets; this results in a linear increase in buffer usage at both the controller and the VNF. However, the controller buffer usage is lower by a factor of about five than the VNF buffer usage, contributing to the scalability of the system since the VNF buffer usage is distributed across the involved VNFs.

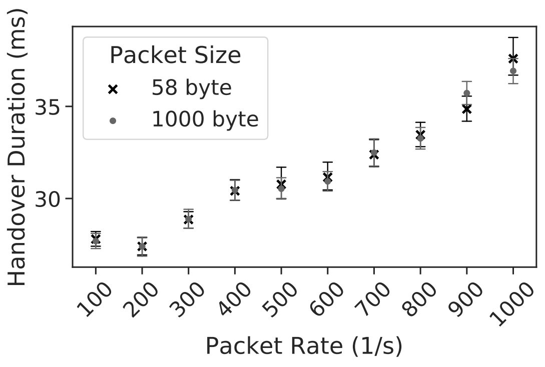
The handover performance as a function of state transfer duration is shown in Figure 9. The increase in the state transfer duration is achieved by artificially introducing a delay after which the VNFs signal the completion of the state transfer. The experiments are executed with a fixed packet rate of 1000 packets per second and 58 byte as well as 1000 byte packets, while the state transfer duration was increased by 100 ms every step, ranging from 0 ms to 1000 ms. Figure 9a shows that the handover duration increases linearly with the additional time introduced by the state transfer, as expected. The maximum packet delay shown in Figure 9b is only offset by a small constant delay from the state transfer duration it experiences; this shows that the packets are indeed released from the buffers as soon as possible and that the service delay is directly influenced by the state size and transfer duration.

The most important results of our evaluation are given in Figure 9c and Figure 9d. They present the buffer usage at the controller as well as at the VNF and highlight the reduced controller load of *SHarP*. Even though the total amount of packets buffered in the system increases with the state transfer duration, the number of packets buffered at the controller remains constant. As predicted in Section. 4.1, this produces a significantly lower workload for the controller compared to *OpenNF* which is achieved by buffering the majority of the packets during the state transfer at the VNF, as the graph in Figure 9d attests.

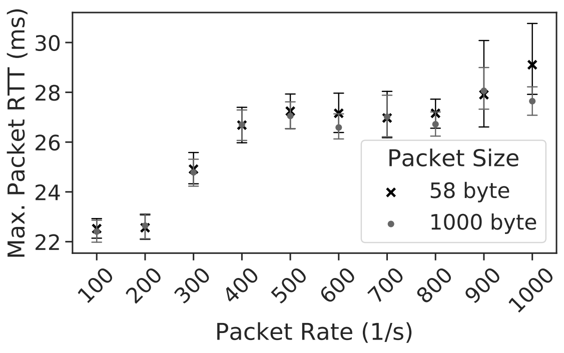
Further, Figure 10 shows the packet delay distributions of all packets sent during a single handover experiment, using 58 byte packets. The left part of the figure shows that the packet rate has only minor impact on the delay and only few packets are delayed by the handover. The right part, in contrast, shows the impact of the state transfer duration to the delay experienced by the packets. It clearly shows that longer state transfer durations lead to a high number of delayed packets and is thus critical for the overall performance of elastic VNF deployments.

4.2.2 | Multi-handover performance

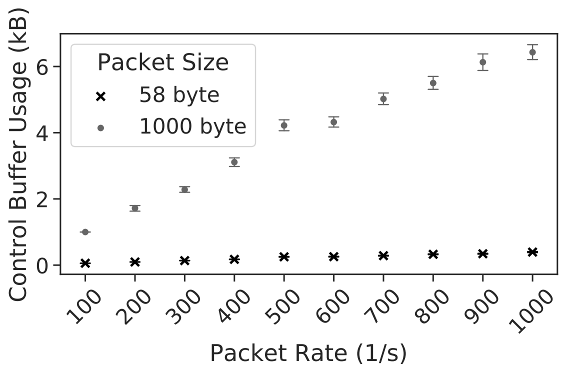
The second part of our experimental evaluation focuses on the overall behavior of our *SHarP* prototype and how it behaves in an environment in which an NFV orchestrator requests many handovers, e.g., because large parts of a service are reconfigured. We again use our previously described experiment setup and send bidirectional UDP traffic between Host_1 and Host_2. Each



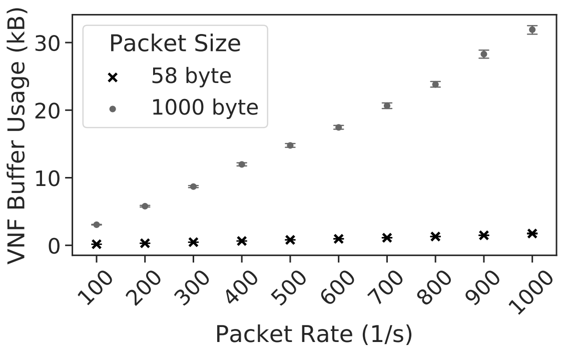
(a) Handover duration



(b) Maximum packet RTT



(c) Controller buffer usage

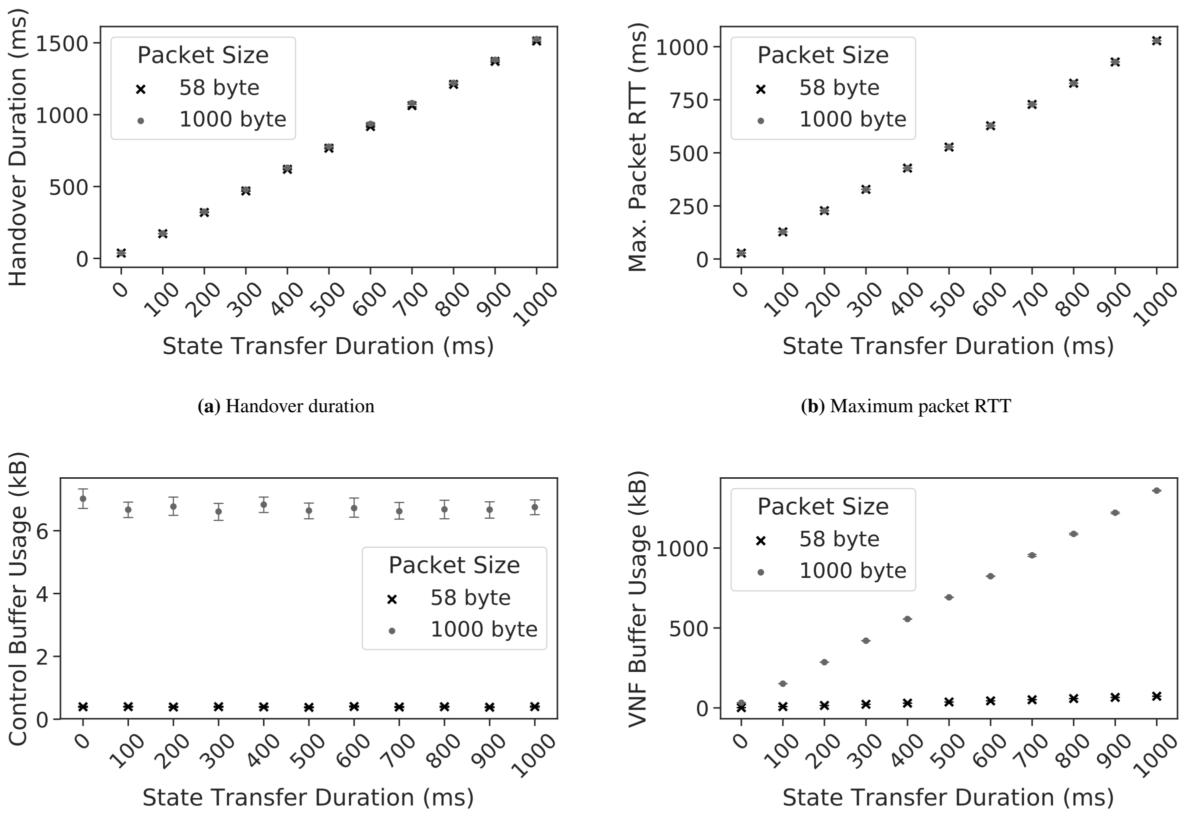


(d) VNF buffer usage

FIGURE 8 Handover performance of SHarP dependent on UDP packets per second with a packet sizes of 58 and 1000 bytes

experiment is again executed 100 times. Instead of focusing on the handover of a single flow, multiple flows (up to 100) are now generated in parallel, each with a packet rate of 50 packets/s, and moved from VNF_1 to VNF_2 during the experiments. We generate handover requests using a Poisson arrival process with a rate of 2, 5, or 10 handover requests per second ($\lambda = 2, \lambda = 5, \lambda = 10$) and sent them to *SHarP's* northbound interface. This simulates an environment in which the NFV orchestrator reconfigures the service multiple times per second, showing that *SHarP* is already designed for future, cloud-native NFV deployments in which reconfigurations may happen on a sub-second basis, which is usually not the case in today's VM-based deployments.

During the experiment, the handover durations are measured to see the impact of handover request number and rate on the performance of individual handovers. Figure 11 shows the results of these experiments for different numbers of handover requests, request arrival rates, and flows with small (58 byte) and large (1000 byte) packets. Figures 11a and 11c show the behavior of the handovers for an increasing number of performed handovers. They show that the handovers become slightly slower when more of them are executed. This effect is a bit stronger when larger packets are used (Figure 11c), which can be explained by the generally higher load in the system due to higher buffer usage at the controller. The handover request rate also impacts the handover duration as shown in Figures 11b and 11d. With a higher rate, the handovers become slightly slower. The size of the packets in the moved flows have almost no impact on this, which is an important property of *SHarP* because the handover performance does not depend on the nature of the moved traffic. In general, 90 % of the handovers in the experiment are complete in less than 120 ms; this shows that *SHarP* can deal with multiple handovers per second without substantial performance degradation.



(a) Handover duration

(b) Maximum packet RTT

(c) Controller buffer usage

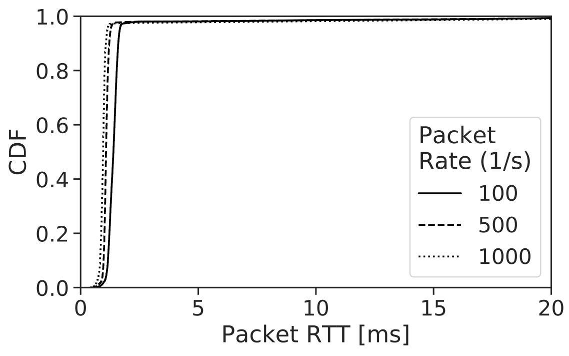
(d) VNF buffer usage

FIGURE 9 Handover performance of SHarP dependent on the state transfer duration with 1000 UDP packets per second and packet sizes of 58 bytes and 1000 bytes

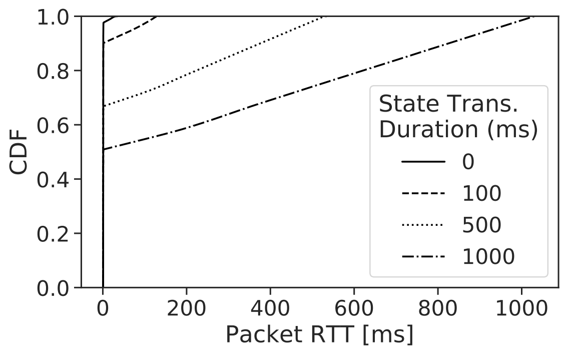
5 | CONCLUSION

We introduced *SHarP*, a novel flow handover mechanism that provides loss-free and order-preserving flow migration functionality for both unidirectional and bidirectional flows. We showed how *SHarP* preserves the order of packets by using different FIFO buffers and subsequently releasing them. In contrast to existing approaches, *SHarP* does not come with an integrated state management solution but provides the means to support any state management solution implemented by a given VNF by sending triggers to it whenever flows are migrated. We believe that this is a much more practical separation of concerns since it leaves the choice of the used state management mechanism to the VNF vendors.

Our experimental evaluation clearly shows that with *SHarP* the maximum packet delay that constitutes the service interruption time is kept to a minimum as it mostly depends on the initial time required to signal the VNF plus the state transfer duration. The interruption time only increases slightly with an increased packet rate and does not worsen at higher packet rates. The evaluation of the controller buffer at increasing packet rates and state transfer durations shows that with *SHarP*, the controller's buffer usage, and thus the amount of processed packets, only depends on the round-trip time between controller and VNFs and on the packet rate. It does not depend on the time taken for the state transfer process that is usually hard to predict and heavily depends on the VNF implementation. This gives *SHarP* a major advantage over similar handover approaches. Our results also show that *SHarP* is ready for future, cloud-native NFV deployments with sub-second reconfiguration times. We published the *SHarP* prototype as open source software on GitHub[24] to make it available for integration with different state management solutions.



(a) Packet RTT for different packet rates



(b) Packet RTT for different state transfer durations

FIGURE 10 Distribution of packet delays during different handover experiments using a packet size of 58 bytes

ACKNOWLEDGMENTS

This work has been partially supported by the SONATA project, funded by the European Commission under Grant number 671517 through the Horizon 2020 and 5G-PPP programs (www.sonata-nfv.eu), the 5GTANGO project, funded by the European Commission under Grant number H2020-ICT-2016-2 761493 through the Horizon 2020 and 5G-PPP programs (http://5gtango.eu), and the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing" (SFB 901).

References

1. Karl H, Dräxler S, Peuster M, et al. DevOps for network function virtualisation: an architectural approach. *Transactions on Emerging Telecommunications Technologies* 2016; 27(9): 1206–1215.

2. ETSI . NFV White Paper. Online at https://portal.etsi.org/NFV/NFV_White_Paper.pdf; . Accessed at 12/2017.

3. Han B, Gopalakrishnan V, Ji L, Lee S. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* 2015; 53(2): 90-97. doi: 10.1109/MCOM.2015.7045396

4. Gember-Jacobson A, Viswanathan R, Prakash C, et al. OpenNF: Enabling Innovation in Network Function Control. *SIGCOMM Comput. Commun. Rev.* 2014; 44(4): 163–174. doi: 10.1145/2740070.2626313

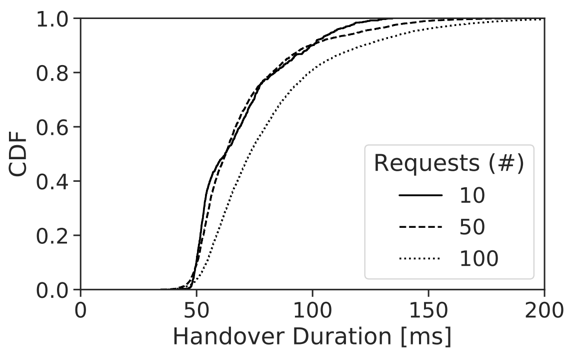
5. Rajagopalan S, Williams D, Jamjoom H, Warfield A. Split/ Merge: System Support for Elastic Execution in Virtual Middleboxes. In: USENIX . , ed. *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*nsdi'13. USENIX Association; 2013; Berkeley, CA, USA: 227–240.

6. Peuster M, Küttner H, Karl H. Let the state follow its flows: An SDN-based flow handover protocol to support state migration. In: IEEE ., ed. *4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*; 2018: 97-104

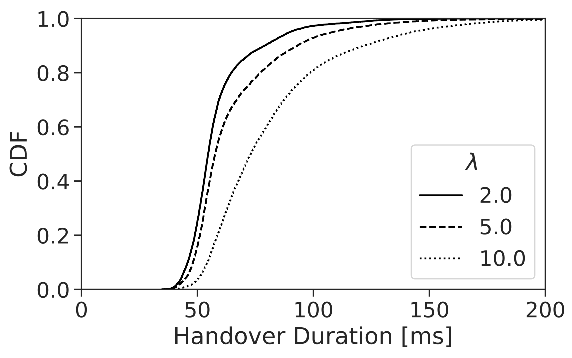
7. Joseph DA, Tavakoli A, Stoica I. A policy-aware switching layer for data centers. In: ACM ., ed. *ACM SIGCOMM Computer Communication Review*. 38. ACM. ; 2008: 51–62.

8. Qazi ZA, Tu CC, Chiang L, Miao R, Sekar V, Yu M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. *SIGCOMM Comput. Commun. Rev.* 2013; 43(4): 27–38. doi: 10.1145/2534169.2486022

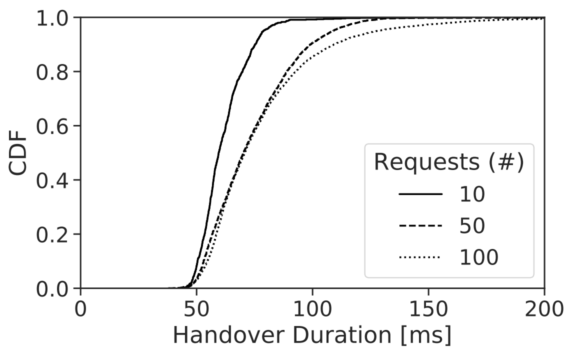
9. Fayazbakhsh SK, Chiang L, Sekar V, Yu M, Mogul JC. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In: USENIX ., ed. *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*NSDI'14. USENIX Association; 2014; Berkeley, CA, USA: 533–546.



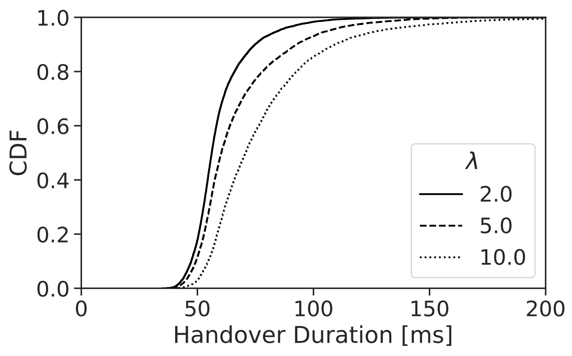
(a) Handover durations for 10 to 100 requests with fixed arrival rate ($\lambda = 10$) moving flows with 58 byte packets



(b) Handover durations for 100 requests and changing arrival rates (λ) moving flows with 58 byte packets



(c) Handover durations for 10 to 100 requests with fixed arrival rate ($\lambda = 10$) moving flows with 1000 byte packets



(d) Handover durations for 100 requests and changing arrival rates (λ) moving flows with 1000 byte packets

FIGURE 11 Distribution of handover durations for multiple handovers using different numbers of handover requests, request arrival rates, and flows with small and large packets

10. Liu J, Li Y, Jin D. SDN-based Live VM Migration Across Datacenters. In: ACM ., ed. *Proceedings of the 2014 ACM Conference on SIGCOMM*SIGCOMM '14. ACM; 2014; New York, NY, USA: 583–584

11. Rajagopalan S, Williams D, Jamjoom H. Pico Replication: A High Availability Framework for Middleboxes. In: ACM ., ed. *Proceedings of the 4th Annual Symposium on Cloud Computing*SOCC '13. ACM; 2013; New York, NY, USA: 1:1–1:15

12. Gember-Jacobson A, Akella A. Improving the Safety, Scalability, and Efficiency of Network Function State Transfers. In: ACM ., ed. *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*HotMiddlebox '15. ACM; 2015; New York, NY, USA: 43–48

13. Kothandaraman B, Du M, Sköldström P. Centrally Controlled Distributed VNF State Management. In: ACM ., ed. *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*ACM. ; 2015: 37–42.

14. Shao X, Gao L, Zhang H. CoGS: Enabling distributed network functions with global states. In: IEEE ., ed. *2017 IEEE Conference on Network Softwarization (NetSoft)*; 2017: 1-9

15. Nobach L, Rimac I, Hilt V, Hausheer D. Statelet-Based Efficient and Seamless NFV State Transfer. *IEEE Transactions on Network and Service Management* 2017; PP(99): 1-1. doi: 10.1109/TNSM.2017.2760107

16. Wang W, Liu Y, Li Y, Song H, Wang Y, Yuan J. Consistent State Updates for Virtualized Network Function Migration. *IEEE Transactions on Services Computing* 2017.

17. Peuster M, Karl H. E-State: Distributed state management in elastic network function deployments. In: IEEE ., ed. *IEEE NetSoft Conference and Workshops (NetSoft)*; 2016: 6-10

18. Mayer R, Gupta H, Saurez E, Ramachandran U. FogStore: toward a distributed data store for fog computing. In: IEEE. ; 2017: 1–6.

19. Lin Y, Kozat UC, Kaippallimalil J, Moradi M, Soong AC, Mao ZM. Pausing and Resuming Network Flows Using Programmable Buffers. In: SOSR '18. ACM; 2018; New York, NY, USA: 7:1–7:14

20. Sun C, Bi J, Meng Z, Yang T, Zhang X, Hu H. Enabling NFV Elasticity Control with Optimized Flow Migration. *IEEE Journal on Selected Areas in Communications* 2018: 1-1. doi: 10.1109/JSAC.2018.2869953

21. Linux Foundation . Data Plane Development Kit (DPDK). Website; 2017. Online at http://dpdk.org.

22. Kuettner H. Seamless SDN-based handover for virtualized network functions. Bachelor's Thesis. Paderborn University. 2017.

23. Ryu SDN Framework Community . Ryu. https://osrg.github.io/ryu/; . Accessed at 08/2017.

24. M. Peuster and H. Küttner and H. Karl . SHarP Prototype Repository. Online at https://github.com/CN-UPB/sharp; . Accessed at 10/2018.

25. Containernet Project . Containernet a Mininet Fork adding Container Support to Network Emulations. online at: https://containernet.github.io; 2017. Accessed 12/2017.

□