Let the state follow its flows: An SDN-based flow handover protocol to support state migration

Manuel Peuster
Paderborn University
manuel.peuster@uni-paderborn.de

Hannes Küttner
Paderborn University
hkuettne@mail.uni-paderborn.de

Holger Karl
Paderborn University
holger.karl@uni-paderborn.de

Abstract—**Dynamically steering flows through virtualized network function instances is a key enabler for elastic, on-demand deployments of virtualized network functions. This becomes particular challenging when stateful functions are involved, necessitating state management. 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.**

In this paper, we introduce the *seamless handover protocol (SHarP)*. It provides an easy-to-use, loss-less, and order-preserving flow rerouting mechanism that is not fixed to a single state management approach. This allows VNF vendors to implement or use the state management solution of their choice. *SHarP* supports these solutions with additional information when flows are migrated. Further, we show how *SHarP* significantly reduces the buffer usage at a central (SDN) controller, which is a typical bottleneck in existing solutions. Our experiments show that *SHarP* uses a *constant* amount of controller buffer, irrespective of the time taken to migrate the VNF state.

I. Introduction

The concept of network function virtualization (NFV) combined with software-defined networking (SDN) allows the dynamic deployment of virtualized network functions (VNFs) in different locations of the network [1], [2]. 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 [3].

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). To tackle this problem, several state management solutions, like *Split/Merge* [4] or *OpenNF* [3], 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 are as follows: We introduce a 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 detination 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* [4]. Finally, we provide an extensive evaluation that first analyses the theoretic scaling behavior of our solution and compares it to *OpenNF* [3] before backing the theoretic expectations with a set of testbed experiments. These experiments verify that controller buffer usage of the proposed approach scales well with the packet rate of the date plane and stays even constant irrespective of the time required for state transfers between the VNFs.

II. 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 [5]–[7]. 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 [8]. 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* [4] and its extension called *Pico Replication* [9], *OpenNF* [3] with its extensions [10], [11], *CoGS* [12], as well as a novel approach called *SliM* [13] and a tagging-based solution presented in [14].

With *Split/Merge* [4], 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* [9], 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 [3] 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 [10]. 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 [10] 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 [11] 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 [14] 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 flow 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* [15]; it works seamlessly with *SHarP*. Other distributed state management solutions, like the recently introduced *CoGS* [12] or *SliM* [13] 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.

III. 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.

A. 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 Fig. 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 (Fig. 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 Fig. 1 will be moved from VNF$_1$ to VNF$_n$.

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

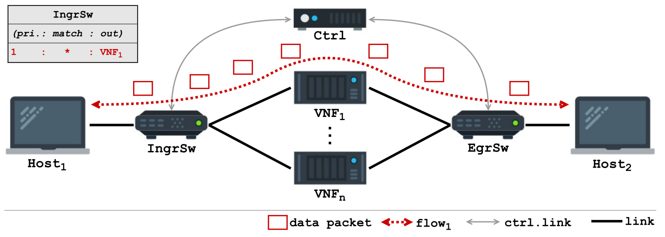


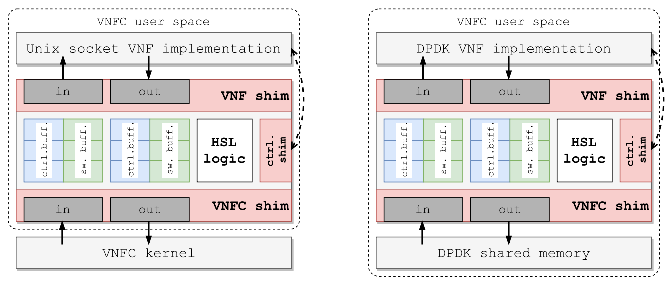
Fig. 1: Example network with multiple VNF instances, *ingress* and *egress switch* as wells as a data flow processed by *VNF*$_1$

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 Sec. III-B). We assume that the links of the example networks do not introduce any additional packet loss or packet reordering.

B. 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 [15], 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 Fig. 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, 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* [3].

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 Fig. 2a, more



(a) Unix socket-based setup (b) DPDK-based setup

Fig. 2: Handover support layer (HLS) sitting between VNFC and VNF implementation

NFV-specific implementations are possible. For example, HSL shims that are based on DPDK [16] as shown in Fig. 2b.

C. Handover procedure

SHarP's handover procedure can be split into three main phases. They are shown in Fig. 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.

At the beginning of the first phase, the scenario looks like the one shown in Fig. 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., a MANO system, 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., a MANO system, does not need to know by which VNFs the flows matching the request are currently processed (R1/R5). In the example given in Fig. 1, the handover request will move all flows from VNF_1 to VNF_n by using a wildcard ($\star$) 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.

Fig. 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[1]. 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 Fig. 3a.

Once `Ctrl` receives the `ACKs` it enters the second phase of the handover procedure that is shown in Fig. 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`.

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

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. HLS 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 Fig. 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. 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 [17].

D. 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* [3], 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 (cf. Fig 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 (Sec. IV-B).

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).

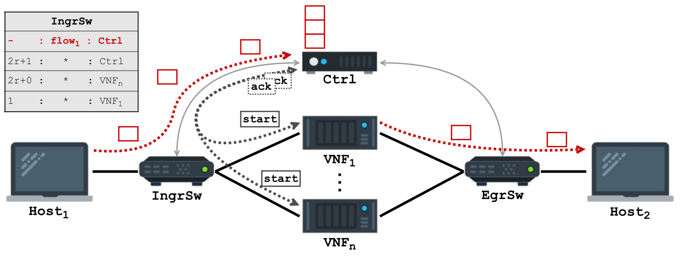
IV. Evaluation

We analyzed the performance of *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).

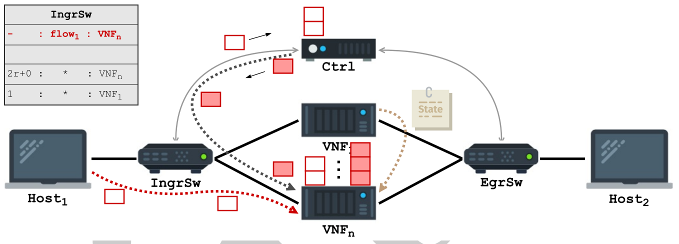
Our results present these metrics as a function of data plane data rate and the duration it takes the VNFs to migrate their state. The maximum packet delay is the main indicator for the delay introduced into the service as the handover is executed. The buffer usage at the controller and at the VNF indicate how well *SHarP* fulfills the claim that only a small amount of data has to be buffered and processed at the controller.

A. Theoretical evaluation

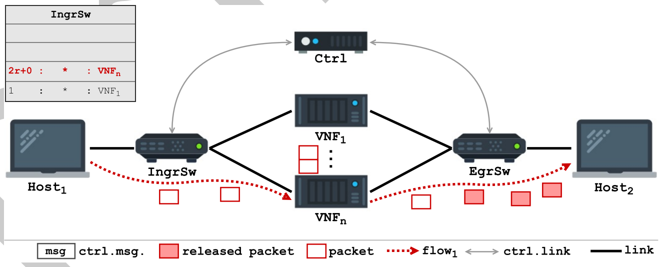
In *OpenNF*, a loss-free order-preserving handover requires a transmission of a total of $3N + 2R + C$ messages over



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

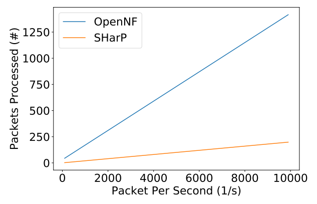


(b) Phase 2: Installed temp. forwarding entry to destination VNF$_n$. Buffering done at destination VNF$_n$ to allow early controller buffer release and load distribution. Trigger of state transfer solution.

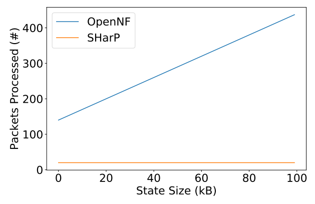


(c) Step 3: Final forwarding state reached and state migration finished. Release and replay of buffered packets at destination VNF$_n$.

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



(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

Fig. 4: Processed packets at the controllers of OpenNF and SHarP for a handover with a duration of 70ms

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, where V is the number of messages needed by the VNF implementation to synchronize state 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.

Fig. 4 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 Fig. 4a 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 [4]. 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 Fig. 4b, the number of packets the controllers of both protocols has to process is shown in relation to the 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 100 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.

B. Experimental evaluation

We implemented a prototype of the *SHarP* controller based on the *Ryu SDN Framework*[2]. 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 Fig. 2a. The use of Python limits the 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 [16] is planned as future work. Both prototypes (SHarP and HSL) are open source and available on GitHub[3].

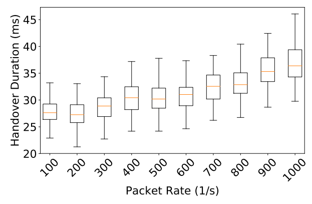
Using these prototypes, we executed a set of experiments to evaluate the performance of the proposed handover protocol. These experiments have been executed on a SDN testbed based on the emulation framework *Containernet* [18] running on a server with an Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz and 24GB memory. The used network topology was the same as shown in Fig. 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.

During the experiments, a constant UDP traffic flow is generated on *Host1* and sent to *Host2* over the first VNF. *Host2* 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 *Host1* 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 experiments have been executed 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.

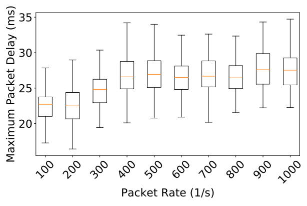
The first set of results given in Fig. 5 shows the handover performance as a function of the data rate of the moved flow given as packets per second. The results shown in Fig. 5 are based on measurements using a packet size of 1000 bytes. Measurements with a small packet size of 58 bytes

[2]Ryu SDN Framework, https://osrg.github.io/ryu/

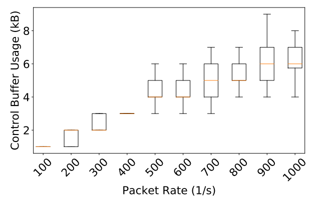
[3]SHarP prototype, https://github.com/CN-UPB/sharp



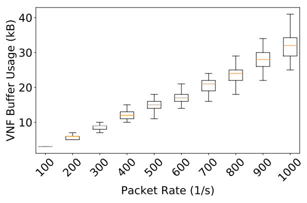
(a) Handover duration



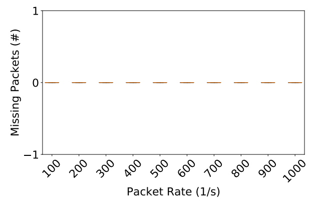
(b) Maximum packet delay



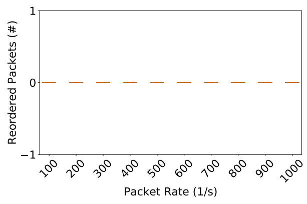
(c) Controller buffer usage



(d) VNF buffer usage

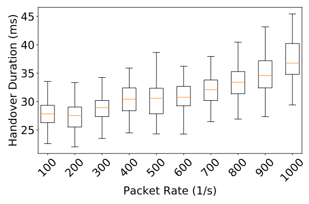


(e) Packet loss

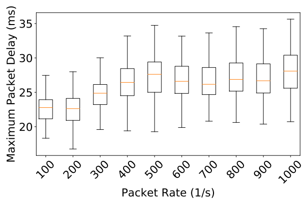


(f) Packet reordering

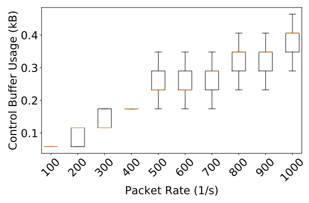
Fig. 5: Handover performance of SHarP dependent on UDP packets per second with a packet size of 1000 bytes



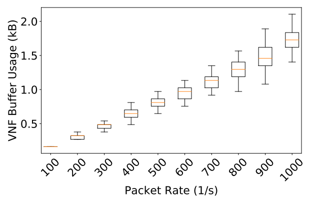
(a) Handover duration



(b) Maximum packet delay



(c) Controller buffer usage



(d) VNF buffer usage

Fig. 6: Handover performance of SHarP dependent on UDP packets per second with a packet size of 58 bytes

produce similar results with smaller absolute buffer usage numbers (Fig. 6). During all experiments, no packets were lost, reordered or duplicated, which verifies the seamless nature of our handover mechanism (Fig 5e and Fig 5f).

Fig. 5a 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 Fig. 5b. 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 Fig. 5c and Fig. 5d, 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 Fig. 7. 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 1000 byte packets, while the state transfer duration was increased by 100 ms every step, ranging from 100 ms to 1000 ms. The experiments are repeated with a packet size of 58 bytes as before and the results are virtually identical compared to the 1000 byte packets as shown in Fig. 8.

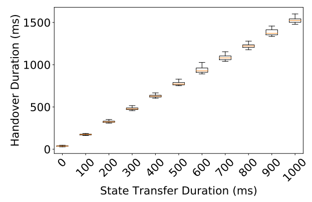
Fig. 7a shows that the handover duration increases linearly with the additional time introduced by the state transfer, as expected. The maximum packet delay shown in Fig. 7b is only offset by a small constant delay from the state transfer duration it experiences; this shows that the packets are indeed 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 Fig. 7c and Fig. 7d. 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 Sec. IV-A, 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 Fig. 7d attests.

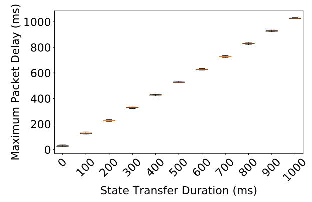
V. Conclusion

We introduced *SHarP*, a novel flow handover mechanism that provides loss-free and order-preserving flow migration functionality. In contrast to existing approaches, it 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.

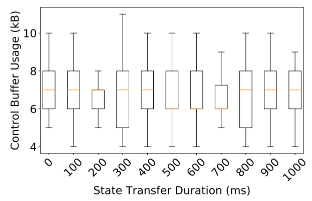
Our experimental evaluation clearly shows that with *SHarP* the maximum packet delay that constitutes the service inter-



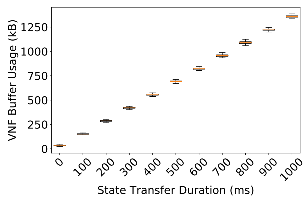
(a) Handover duration



(b) Maximum packet delay

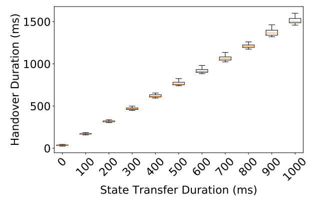


(c) Controller buffer usage

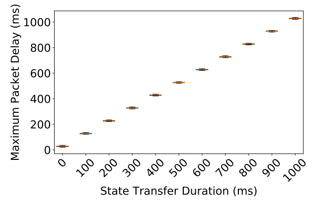


(d) VNF buffer usage

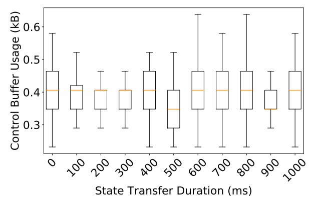
Fig. 7: Handover performance of SHarP dependent on the state transfer duration with 1000 UDP packets per second and a packet size of 1000 bytes



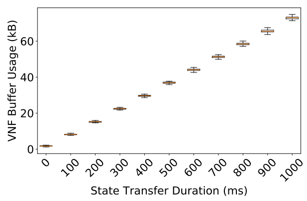
(a) Handover duration



(b) Maximum packet delay



(c) Controller buffer usage



(d) VNF buffer usage

Fig. 8: Handover performance of SHarP dependent on the state transfer duration with 1000 UDP packets per second and a packet size of 58 bytes

ruption 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 rate and state transfer duration 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. We published the *SHarP* prototype as open source software on GitHub to make it available for integration with different state management solutions.

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 5G TANGO 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] ETSI, "NFV White Paper," https://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *Communications Magazine, IEEE*, vol. 53, no. 2, pp. 90–97, 2015.

[3] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2740070.2626313

[4] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 227–240. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482649

[5] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," Tech. Rep., 2008.

[6] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 27–38, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534169.2486022

[7] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 533–546. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616497

[8] J. Liu, Y. Li, and D. Jin, "Sdn-based live vm migration across datacenters," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 583–584. [Online]. Available: http://doi.acm.org/10.1145/2619239.2631431

[9] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:15. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523635

[10] A. Gember-Jacobson and A. Akella, "Improving the safety, scalability, and efficiency of network function state transfers," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ser. HotMiddlebox '15. New York, NY, USA: ACM, 2015, pp. 43–48. [Online]. Available: http://doi.acm.org/10.1145/2785989.2785997

[11] B. Kothandaraman, M. Du, and P. Sköldström, "Centrally controlled distributed vnf state management," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, 2015, pp. 37–42.

[12] X. Shao, L. Gao, and H. Zhang, "Cogs: Enabling distributed network functions with global states," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, July 2017, pp. 1–9.

[13] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "Statelet-based efficient and seamless nfv state transfer," *IEEE Transactions on Network and Service Management*, vol. PP, no. 99, pp. 1–1, 2017.

[14] W. Wang, Y. Liu, Y. Li, H. Song, Y. Wang, and J. Yuan, "Consistent state updates for virtualized network function migration," *IEEE Transactions on Services Computing*, 2017.

[15] M. Peuster and H. Karl, "E-state: Distributed state management in elastic network function deployments," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 6–10.
[16] L. Foundation, "Data plane development kit," Website, 2017, online at http://dpdk.org.
[17] H. Kuettner, "Seamless sdn-based handover for virtualized network functions," Bachelor's Thesis, Paderborn University, 2017.
[18] M. Peuster, H. Karl, and S. van Rossem, "MeDICINE: Rapid Prototyping of Production-ready Network Services in Multi-PoP Environments," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2016, pp. 148–153.