Managing State for Failure Resiliency in Network Function Virtualization

Sameer G Kulkarni*†, K. K. Ramakrishnan†, and Timothy Wood‡
*IIT Gandhinagar, India, †University of California, Riverside, ‡George Washington University.

***Abstract*— Ensuring high scalability (elastic scale-out and consolidation), as well as high availability (failure resiliency) are critical in encouraging adoption of software-based network functions (NFs). In recent years, two paradigms have evolved in terms of the way the NFs manage their state - namely the Stateful (state is coupled with the NF instance) and a Stateless (state is externalized to a datastore) manner. These two paradigms present unique challenges and opportunities for ensuring high scalability and high availability of NFs and NF chains. In this work, we assess the impact on ensuring the correctness of NF state including the implications of non-determinism in packet processing, and carefully analyze and present the benefits and disadvantages of the two state management paradigms. We leverage OpenNetVM and Redis in-memory datastore to implement both state management paradigms and empirically compare the two. Although the stateless paradigm is desirable for elastic scaling, our experimental results show that, even at line-rate packet processing (10 Gbps), stateful NFs can achieve chain-level failover across servers in a LAN incurring less than 10% performance. The state-of-the-art stateless counterparts incur severe throughput penalties. We observe 30-85% overhead on normal processing, depending on the mode of state updated to the externalized datastore.**

***Index Terms*—Network Function Virtualization (NFV), Service Function Chaining (SFC), Fault-tolerance, Availability,**

I. Introduction

Software-based Network functions (NFs) have evolved significantly in recent years and have become an integral part of service provider, enterprise, and data center networks. These NFs are typically high speed packet processing engines functioning as a bump-in-the-wire on the data path and may need to process several million packets per second, as link rates scale up from 10G, to 40G, to 100Gbps. They support a variety of in-network services such as network address translation (NAT), firewalls (FW), intrusion detection and prevention (IDS/IPS), *etc.* Network flows typically pass through more than one NF, being processed in a specific order referred to as a Service Function Chain (SFC). Figure 1 shows a simple chain consisting of NAT, FW, IDS and rate limiter NFs.

NFs operate inline with the network forwarding datapath, and as such, NF failure or underlying hardware failures (server node, link) can significantly disrupt network operations. Hence providing NF failure resiliency is critical. Further, we observe that NFs differ in their computational complexity and can drastically vary in their packet processing rates [1]. In order to meet varying traffic demands and to meet chain-wide performance goals, the NFs in a chain may need to be elastically scaled – *i.e.,* networks have to dynamically adapt the number of NF instances and balance the load across them.

978-1-7281-8154-7/20/$31.00 ©2020 IEEE

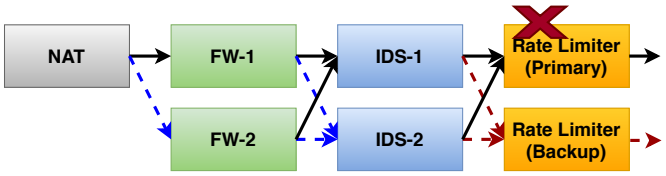


Fig. 1: NF chains comprising NAT, FW, IDS and Rate limiter NFs; Elastic scaling of FW and IDS NFs results in redirecting part of the traffic from NAT across scaled instances (blue dotted line); Failure of Rate limiter NF results in redirection of traffic to a backup rate limiter NF (red dotted lines).

Works such as Pico replication [2], FTMB [3] and Reinforce [4][1] make use of stateful NFs that maintain their state locally within the NF instance and share the state across multiple NF instances through message passing or other traditional operating system shared memory constructs. However, support for shared state in these works is limited to the instances within a single server node. In contrast, Stateless NFs are a new architectural approach for managing state in network function virtualization (NFV) platforms. Stateless NFs decouple the existing design of NFs into a stateless processing component and a data store layer. They break the tight coupling of state with the processing components, thus seeking to enable a more elastic and resilient network function infrastructure [5]. Works such as CHC [6] have followed such an approach and extended the state management aspects to facilitate very low cost (per-packet processing latency) chain-wide failure resiliency.

In this work, we compare stateful and stateless NF management and deployment paradigms. We implement and deploy stateful and stateless NFs and NF chains on OpenNetVM [7], a DPDK based high performance NFV platform. We provide both qualitative and quantitative analysis and results on the impact of incorporating stateful or stateless NFs on our NFV platform. We specifically target the impact on NF performance (failure-free operation) and their ability to support features such as elastic scaling and fault-tolerance while addressing non-determinism and having chain-wide consistent operations.

II. Background & Design Alternatives

A. NF State Management

Some NFs may be inherently stateless *i.e.,* they do not maintain any state associated with packet processing *e.g.,* stateless firewalls utilize static pre-configured access control rules to block certain packets. However, a large number of NFs are stateful and maintain flow/packet specific state information

[1]This work extends & complements [4] with results on Stateless NFs.

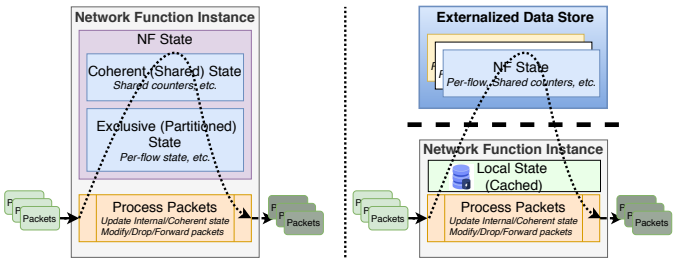


Fig. 2: Network Function State Management: (a) Stateful NFs with state maintained within the NF instance and (b) Stateless NFs where the NF state is maintained in the externalized data store and optionally the NFs can locally cache the state.

e.g., IDS, load balancer, *etc.* [3]. The state maintained by the NFs may correspond to i) per flow status - *i.e.,* state for each new flow *e.g.,* application delivery controllers and stateful firewalls; ii) per packet status - state associated with the processing of each individual packet by the NFs *e.g.,* IDS. In this work, we focus only on the stateful NFs.

In [8], they categorize the state maintained by the NFs into **i) Internal - ephemeral state**: it is of no consequence outside that NF instance's execution, *e.g.,* application logic, resource mappings (CPU core, configuration files), *etc.*
ii) External state - Partitioned and coherent state: NF state that is required for, and impacts, the packet processing. It includes *Partitioned state* - *e.g.,* per flow state, that is often specific to an NF instance and differs across different replicas, and *coherent state* - *e.g.,* shared global counters and state associated with a group of flows, which can be updated by different NF instances and need to be kept consistent across NF replicas. In addition, changes made to the packet due to the NF processing also correspond to the state *e.g.,* NAT, load balancer NFs modify the TCP/IP headers; Firewalls, IDS/IPS may modify the packet routing state across the NF chains. Figure 2 (a) shows the NF state partitioning and packet processing with the Stateful NFs.

B. Non-Determinism in NFs

Non-determinism (ND) is pervasive in NFs [3], [4]. Output from two identical NFs, even when provided with identical inputs, can differ due to non-determinism exhibited by the NFs in processing the packets. ND can be due to a) the local behavior of each NF *i.e.,* hardware dependence whose outcome cannot be predicted, such as hardware clocks, random number generators, etc., or race conditions in accessing shared variables among different NF threads; b) the behavior of the network connecting them, as well, *e.g.,* the order of packet arrival and subsequent processing either due to random packets being lost, dropped or marked for ECN [9].

For example, rate limiter NFs that restrict the maximum number of connections for different clients may end up rejecting or terminating different connections either due to a race condition among NF threads accessing and updating the shared connection counter variable or due to ordering of packet processing within the NF itself. Similarly, a load balancer NF that assigns each TCP connection to one server from a pool of backend servers may end up choosing different backend servers for the same flow across different NF instances when the backend server selection logic is based on system specific calls like `random()`. This also impacts the state at the external clients and may potentially disrupt network services as the clients may end up losing the connection/session state maintained at their respective ends. Thus, ND further complicates the state management of NFs when providing resiliency.

C. Stateless or Externalized state NFs

Recent work, such as StatelessNF [5] and CHC [6] have proposed an alternative approach to manage NF state by decoupling the NF state from the NF processing instance and externalizing that state to an in-memory database like Redis [10], RAMCloud [11] *etc.* as shown in Fig. 2 (b). Note: The Externalized data store is decoupled from the NF instance and can be implemented and run as a single/cluster of processes or containers on the same or different compute nodes. *Any state access operations from the NF instance to the externalized state* (e.g., *add/read/write/modify/delete) require inter-node or inter-process communication* e.g., *remote procedure call, HTTP/TCP socket etc., based on the location of the externalized state.* To avoid the communication overheads, most of the research works [5], [6] consider an in-memory data store that allows partitioning and storing of the database on the DRAM of the local node (*e.g.,* Redis client process). Despite, it should be noted that with user space NFs, the access to state through the database APIs typically incurs the overhead of an additional context switch to kernel space. Further, just as with stateful NFs, the externalized state can also be cached locally within the NF instance that can exploit the local cache and alleviate communication overheads by allowing the processing of a large batch of packets.

A key advantage with externalizing state is that when any NF instance fails, the state in the externalized data store is not impacted, and is still available for the replica NF to seamlessly failover, which only requires the flows to be redirected to a replica NF. *e.g.,* When the rate limiter NF fails, the flows from the IDS-1 and IDS-2 can be redirected to the new instance of the rate-limiter as shown in Fig. 1 without the need to re-synchronize and update the state at the replica instance. The approaches using stateless NFs with externalized state are better suited for a Microservices architecture [12]. Stateless NFs allow seamless scaling of NF instances and failover to different NF instances, without having the state replicated to a distinct replica that is selected a priori. In addition, since the state is decoupled from the NF instance, asymmetric routing of packets from different flows which may be common in multi-path routing [13] can be supported as well – *e.g.,* in Fig. 1, packets of flows processed at NF instances (FW-1, IDS-1) in the service chain may instead be processed by NF instances FW-2 and IDS-2 without any loss of information[2].

[2]This is based on the assumption that all the state is externalized, and no state is cached locally within the NFs

Architecturally having externalized state for NFs might seem a step in the right direction, but the challenges in terms of performance, addressing non-determinism and complexity of operation need to be carefully analyzed including the overhead of employing them for high speed packet processing NFs has to be empirically determined (as we do in this paper).

D. Elastic Scaling and Route Management

Elastic scaling is the ability to adapt NF instances to changing application traffic patterns with automatic scale-out (add) or scale-in (remove) of the NFs. With traditional stateful NFs, when the NFs are elastically scaled, it is essential to first update the associated state for the portion of the traffic that is distributed across these new NF instances before updating the route for the flows. Works such as [14], [8] ensure the corresponding NF state is replicated before migrating flows to the new instance. However, with stateless NFs (with no caching) *e.g.,* as in [5], flows can be instantly redistributed without the need to worry about any state update. But, cache-based stateless NFs [6] require the cached state to be flushed out and synchronized to an externalized data store before routing the traffic to the new NF instances. We analyze the penalty for such state migration for the stateful and stateless (with cache) approaches. Further, with NF chains, it is necessary to ensure the flow's affinity for an NF instance (upstream NFs in the chain) is maintained when any NF in the chain is elastically scaled. *e.g.,* in Fig. 1, when the FW and IDS NFs are scaled to a new FW-2 and IDS-2 instances, FW-2 needs to ensure that flows that have their state at IDS-1 are routed towards IDS-1 and only the flows that have their state migrated to new IDS instance are routed and served at IDS-2.

E. Addressing NF and NF chain failures

Unlike elastic scaling, providing resiliency for an NF or NF chain failure requires more careful consideration of NF state management. Accordingly, earlier works have distinguished two approaches with 'Active:Standby' mode of operation *viz.* i) checkpoint only[8], [2] and ii) replay based approaches [3], [6], [4]. In either case, it is necessary to setup a standby replica NF and NF chain instances a priori, and perform periodic state updates (synchronization) on the standby instances. Although most works [2], [5], [3] address fault tolerance, only [6], [4] specifically address chain-wide failure resiliency.

Therefore, addressing elastic scaling and fault tolerance for NFs is a major challenge - the solution needs to ensure: i) consistent state updates across NF instances, since any loss of state can not only degrade performance, but can also disrupt correct operation of the network service; ii) overcome non-determinism to ensure state consistency; and iii) have low-overhead on normal operation to ensure high packet processing rate and low packet-processing latency.

III. Related Work

Works [3], [4] consider network functions to be stateful and correspondingly provide support to migrate state across different NF instances to facilitate elastic scaling and failure resiliency, while the works [5], [6] consider the NFs to be stateless having externalized the state to a datastore.

Elastic Scaling Split/Merge [8] defines state access APIs to read and update the internal state of virtualized NFs being moved across hosts. It relies on the ability to identify per-flow state to provide consistent migration. FlexNFV [15] is a DPDK based framework that periodically monitors NF load on a service chain, and performs timely scaling of NFs to evenly distribute the load among available instances. In work [16], authors propose a proactive approach to scale and provision NF instances ahead of time based on the estimated flow rates using an efficient online learning method.

Fault tolerance and high availability: Pico Replication [2] relies on flow-group based NF state transfers *i.e.,* application level NF state check-pointing to address high availability for the stateful NFs. During the check-pointing, to ensure correctness, it pauses the packet processing of the flows and buffers all the input and output packets which results in significant throughput and latency overhead during the failure-free operation. On the other hand, FTMB [3] relies on packet replay and periodic (coarse-grain) check-pointing of the NF state. It logs all the input packets and the per packet access log for the shared variables in the NF that account for non-determinism, which are necessary to replay and restore the state correctly on the replica NF. However, both do not address fault tolerance for NF chains and do not provide any NF chain-wide consistent recovery. REINFORCE [4] fills this gap with an efficient chain level replication scheme which does not excessively impact the normal operation as well does not place any restrictions on replay mode to ensure correctness of the replica state.

IV. Implementation

Stateful NFs: We leverage our previous work on REINFORCE [4] to support stateful NFs. REINFORCE is built on OpenNetVM [7] - a DPDK based high performance NFV platform. Each NF maintains a 64KB local memory block to maintain NF state *e.g.,* per flow state information. In addition 64MB of a shared memory block is provided to maintain the global shared state across multiple instances of the same kind of network function *e.g.,* global counters.

Stateless NFs: We implement stateless NFs with the assistance of Redis [10] as a backend data-store. We used Redis version 2.8.4. and the latest version of Hiredis - a minimalist 'C' client library to integrate with our NFs. Since we built REINFORCE on OpenNetVM, we leverage the same base platform to build the stateless NFs as in CHC [6].

We customized and built the existing NFs (Basic monitor (BM), Vlan Tag (QoS), Load balancer (LB) and Deep Packet Inspection (DPI)) to read and export the state variables to Redis, so that core processing logic of the NFs is unchanged, and only the relevant state access operations are modified. Note, we do not change any of the dynamic memory and in-packet processing functionality. Only the static counter variables (Svs) are exported to Redis.

We tested for varying state update patterns to compare the impact of using synchronous (sync) and asynchronous (async) state update operations. Further, we enabled a local cache of

variables for each of the NF state variables for both the synchronous (sync+c) and asynchronous (async+c) state updates. These state variables are updated on the backend database only after performing a batch of packet processing operations. In addition, we did not wait for acknowledgements in the case of asynchronous operations (for both async, async+c cases) so that packet processing is not stalled and can be performed concurrently with state update operations.

We experimented with several in-memory databases and narrowed our implementation to leverage Redis as the preferred datastore due to performance, tuning support for different configuration parameters, stability and 'C' plugin availability that enables easy integration with our platform. Further, in order to improve performance, we tuned the Redis configuration parameters as follows:

1. We enabled TCP keep-alives so that once the connection is setup by an NF, it is reused for the entire session, without requiring the TCP connection to be setup for every request.

2. We disabled the transparent huge pages and RDB persistence options to avoid the overhead of disk operations.

3. Also, to avoid excessive logging overheads, we set the log-level to 'warnings-only' mode.

V. Evaluation

We use an experimental testbed consisting of five Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz servers, each with 157GB RAM, (two sockets with 28 cores each), running Ubuntu SMP Linux kernel 3.19.0-39-lowlatency. Additionally, we have a source and sink node at either end. We deploy the Redis datastore on both predecessor and primary nodes. For these experiments, nodes were connected back-to-back with dual-port 10Gbps DPDK compatible NICs to avoid any switch-induced overheads. We keep a dedicated 10Gbps link for Redis traffic, while the NFs communicate over a separate 10Gbps DPDK port. We use the DPDK-based high speed traffic generator, Moongen [17] to generate line rate traffic (14.88Mpps). We vary the traffic rate as needed for each of the different experiments. For the NF chain scenario, we deploy the entire NF chain on single (primary) node.

We compare REINFORCE with CHC [6]. We implement a simplified version of CHC, where the NF state is externalized to a Redis datastore [10]. The NFs cache the state locally and perform asynchronous state update operations after processing a batch of 256 packets. All our experiments with REINFORCE use a small batch size of 32 packets. However for CHC, we set batch size to 256 packets, as the smaller batch (32) limited CHC's throughput to less than 3Mpps[3] .

A. Choosing the Externalized datastore.

We experimented with standard benchmark tools available with the Redis [10] Aerospike [18] in-memory cluster databases that can be used to externalize NF state. With Redis, we observed that we could achieve a maximum of 1.65 million read and 1.32 million writes transactions per second (tps) respectively on a single node, for transferring 8 bytes of data each time. Note: For highest performance, we tuned the 'parallel connections' and 'pipeline (in-flight requests)' parameters, and set the parallel connections to 200 and pipeline (in-flight requests) to 256. In fact, the default Redis parameters (parallel connections $= 50$, pipeline $= 1$) result in less than 100K read/write operations per second. With Aerospike, we observed $350K$ read and $370K$ write tps respectively. We chose Redis because it was easy to integrate using the Hiredis 'C' plugin with our NFV platform.

[3]Note: Our results (throughput) for CHC are better than those presented in the CHC paper [6], and the results may depend on the actual CHC implementation and its optimized datastore.

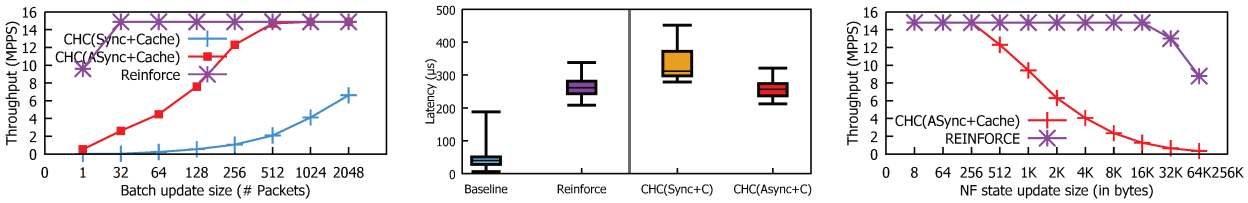
B. Performance impact of Externalizing NF State.

First, we profile the impact on throughput for using the external state store in both synchronous and asynchronous updates. We use a simple forwarder NF with a packet counter variable exported to the externalized state store as a toy NF, to demonstrate the impact of synchronous and asynchronous modes of state update, with local caches, for different batch sizes. The state update to the data store is carried out after processing a batch of packets.

From Figure 3a, we observe that with no caching, where the state variable is updated to the datastore after processing each and every single packet (i.e., with batch size=1, update operations are effectively operating with no local cache), both synchronous and asynchronous modes results in very low throughput (0.01 and 0.5Mpps respectively). Although, this mode ensures strict correctness, it incurs a very high penalty on throughput and latency. We gradually increase the batch size for updating the datastore. Here, variables are cached locally and are updated to the datastore only after processing a fixed batch of packets. We observe that even with a very large batch size of 2048, synchronous state updates can at-best achieve a performance of 6Mpps, while the asynchronous state updates (without waiting for acknowledgements) can achieve line rate throughput (14.88Mpps) for batch sizes above 1024. However, asynchronous state updates require the platform to take additional measures to ensure correct state recovery in the event of failures. This includes, at the very least during normal operation, tracking of failed updates (based on the acknowledgements), and ensure correct versioning of the states to be maintained both at the datastore and the local cache of the NF. Further, as suggested by CHC [6], to achieve correct recovery in the event of failures, it is necessary to keep track of the previous version of the state variables in the datastore along with additional metadata (last NF, and the packet responsible for state update) that can help suppress any duplicate updates. Figure 3b shows the latency profile for the baseline, REINFORCE and CHC with synchronous and asynchronous modes of operation with the local cache, and a state update batch size of 256. The round-trip latency for CHC with synchronous mode incurs high penalty, while the asynchronous mode is only marginally better than REINFORCE.

C. Elastic Scaling

With stateful NFs, elastic scaling requires to synchronize/-transfer state to the newly instantiated NFs. Hence, we profile



(a) Throughput for a Simple forwarder NF updating a single (8 byte) state variable.

(b) Round-trip latency (5%ile, 25%, median, 75% and 99%ile).

(c) Throughput for different size of NF state update.

Fig. 3: Throughput and Latency comparison for stateful (REINFORCE) and stateless (CHC) approaches including synchronous and asynchronous modes of externalized state. We use batch size of 32 for REINFORCE and 256 for CHC.

the state Synchronization overhead for the stateful NFs.

State Synchronization overhead: We profiled the cost of memory scan and updating of the dirty state for a 64KB memory and 1KB chunks to be 55-80 CPU ticks. The copy overhead for a 4KB page is measured to be 2315-2590 CPU ticks. Note: Copy operation for a batch of processed packets (32 packets) drastically reduces the overhead during normal processing and allows to achieve line-rate packet processing. However, the stateless NFs (with no cache) incur zero overhead for state update, while the cache based stateless NFs require datastore update operation (making the NF context switch from user to kernel space). We observed the overheads to be extremely excessive resulting in a maximum throughput of $\sim 4Mpps$ (details in §V-E).

D. Overhead Analysis on Normal Operation

Normal (Failure-free) Operation: For a number of different NFs, we compare: i) the overhead during normal operation by measuring the throughput; and ii) additional latency of packet processing (for state update operation), for individual NF instances. We observed the performance of REINFORCE to be on-par with the baseline (no resiliency case), achieving near baseline throughput for the monitor NF (line rate$\sim$13.5Mpps) and also for the NF chains, providing 2-3$\times$ better than CHC, although the per-packet processing latency 99%ile were comparable. Note: More details on the throughput and latency are provided in our earlier work [4].

Impact of Non-Determinism rate: REINFORCE performs a 2-phase commit of the chain-wide packet-processing progress and NF states. To ensure correctness, any non-deterministic updates result in chain-wide NF state checkpointing. We also analyze the impact on performance for different non-determinism rates. REINFORCE is able to provide near line-rate processing for the non-deterministic rate that is less frequent than one every $250\mu sec$, while more frequent non-determinism reduces the throughput, eventually dropping to 0.22Mpps. This is due to the round-trip latency of 2-phase commit, causing the NF processing to stall if a previous non-deterministic batch of packets has to be processed and state committed to the standby. In the case of stateless NFs (with local cache), performance impact depends on the memory update operations to the datastore.

E. Impact of bulk memory update operations

The size of the NF state for different types of NFs vary. We analyze the impact of updating the state, with different amount of variables, ranging from a total size of 8 bytes to 64 Kbytes. We compare the memory replication strategy of REINFORCE with the externalized state approach of CHC, where the state is replicated to the Redis datastore. For this experiment, we use simple forwarder NF (that has almost no computation overhead for packet processing) with a single state variable (set as an array of integers). We vary the size of this variable array for each run. For CHC, we use the asynchronous mode of operation with a local cache and the state update is done once for every batch of 256 packets. We also do not wait for the acknowledgements of the state update. Note: The local cache of the state variable in CHC (array) is read at initialization and the NF continues to update the local cache, while the state is replicated to datastore after processing a batch of 256 packets. With REINFORCE, the modified region of the variable array is updated after processing a batch of 32 packets and also replicated to the remote node as detailed in our earlier work [4]. Figure 3c shows the impact on throughput for both REINFORCE and CHC with increasing amounts of NF state being updated. With the externalized datastore of CHC, state updates above the size of 256 byte start to degrade the throughput and for large sizes (more than 16K) the throughput drops below 1Mpps. However, with REINFORCE, we see that even with 16K data size, it is able to keep up with near line-rate packet processing, and throughput drops for larger data sizes, but even then, only to 8.7Mpps for a 64K data size.

VI. Evaluation Summary and Future Directions

The stateless NF approach of decoupling processing and state management simplifies the elastic scaling of NFs and fits well with the microservices architecture. Externalized state allows any NF instance to start processing packets without any state update overheads or any loss of state. Further, this decoupling of the NF state allows us to easily support asymmetric routing (*e.g.,* equal-cost-multiple-path (ECMP)) where packets can take different paths, potentially traversing through different instances of the NFs. This is not easily implemented with the traditional stateful NF designs. However, with the current state-of-the-art in-memory databases like Redis, there are still significant performance challenges and limitations in externalizing the NF state.

As NFs act as a bump-in-the-wire, their packet processing rates have to be very high, up to several million packets per second ($\sim$14Mpps for a line-rate of 10Gbps). The processing rate with stateless NFs may be limited by the read/write transactions achievable with the external database. Our evaluation with Redis (with asynchronous and parallel execution) to read

and update 8 bytes of data achieves a maximum of 1.65 million read and 1.32 million write transactions per second (Tps) on a single node. The overhead for using in-memory databases comes primarily from the socket I/O (read/write system calls), that involves user-space to kernel-space context switching, and thus reduced performance for DPDK-based user-space NFs.

Further, a major challenge arises in ensuring the correctness and consistency of the externalized state with respect to failed NFs that might have partially processed the packets and updated the state locally or synced only a portion of the state updates to the database, before crashing. In such scenarios, both the NFs and the database need to maintain additional version control for each state update, so that state updates can be validated before being committed. This would further reduce the NF processing capacity [6]. Moreover, with the Stateless NF approach there is a need to instrument and refactor the NF code to externalize the NF state. It requires all the internal NF state entities to be expressed in a well-defined key-value store mode. While this may be easily dealt with for per-flow state, it can be difficult to express shared per-session state as well as internal state variables in this manner. Also, typically NFs allocate and release memory dynamically (via alloc and free callback functions as in nDPI). Although ephemeral, these states may also need to be externalized to ensure operational correctness, which can result in significant state update overhead. On the other hand, the complexity of supporting non-determinism and chain-wide correctness in both the stateful and stateless NFs is non-trivial. To ensure correctness, stateless NFs would require additional support from the externalized databases to provide version control and roll back of the committed state.

Stateless NFs with externalized state is promising, decoupling state and processing for NFV. However, the performance challenges with externalized data stores suggest that they need to be adopted with care for high speed packet processing NFs.

VII. Conclusion

In this paper, we analyzed two NF state management (*i.e.,* traditional in-memory stateful NFs and stateless NFs) approaches that have been proposed for addressing elastic scaling and fault tolerance. Ensuring correctness and consistent state update and recovery for NF chains face similar challenges (addressing non-determinism and chain-wide consistency) for both state management approaches. Stateless NFs, although promising, fall short of achieving line-rate packet processing capabilities and stateful NFs offer much higher performance and correctness under non-deterministic packet processing. Advancements in userspace in-memory databases and persistent storage can continue to help externalizing state for specialized applications contexts, especially when NF processing is entirely deterministic.

Acknowledgement: This work was supported by US NSF grants CRI-1823270, CNS-1763929, and CRI-1823236.

References

[1] ETSI-GS-NFV-002, "Network Functions Virtualization (NFV): Architectural Framework," http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf, 2013, [ONLINE].

[2] 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

[3] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-recovery for middleboxes," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 227–240, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2829988.2787501

[4] S. G. Kulkarni, G. Liu, K. K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, "Reinforce: Achieving efficient failure resiliency for network function virtualization-based services," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 695–708, 2020.

[5] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 97–112. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[6] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 501–516. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/khalid

[7] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "Opennetvm: A platform for high performance network service chains," in *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ser. HotMIddlebox '16. New York, NY, USA: ACM, 2016, pp. 26–31. [Online]. Available: http://doi.acm.org/2940147.2940155

[8] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 227–240. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/rajagopalan

[9] Y. Velner, K. Alpernas, A. Panda, A. Rabinovich, M. Sagiv, S. Shenker, and S. Shoham, "Some complexity results for stateful network verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 811–830.

[10] redislabs, "Redis," 2019. [Online]. Available: https://redis.io

[11] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 29–41. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043560

[12] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*, 1st ed. O'Reilly Media, Inc., 2016.

[13] Y. Ganjali and A. Keshavarzian, "Load balancing in ad hoc networks: single-path routing vs. multi-path routing," in *IEEE INFOCOM 2004*, vol. 2. IEEE, 2004, pp. 1120–1125.

[14] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," *CoRR*, vol. abs/1305.0209, 2013. [Online]. Available: http://arxiv.org/abs/1305.0209

[15] X. Fei, F. Liu, H. Jin, and B. Li, "Flexnfv: Flexible network service chaining with dynamic scaling," *IEEE Network*, pp. 1–7, 2020.

[16] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive vnf scaling and flow routing with proactive demand prediction," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 486–494.

[17] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: a scriptable high-speed packet generator," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. ACM, 2015, pp. 275–287.

[18] V. Srinivasan, B. Bulkowski, W.-L. Chu, S. Sayyaparaju, A. Gooding, R. Iyer, A. Shinde, and T. Lopatic, "Aerospike: Architecture of a real-time operational dbms," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1389–1400, 2016.