Industrial-scale Stateless Network Functions

Márk Szalay*†, Máté Nagy†, Dániel Géhberger†, Zoltán Kiss†, Péter Mátray†,
Felicián Németh*, Gergely Pongrácz†, Gábor Rétvári*‡, László Toka*‡
*Budapest University of Technology and Economics, Hungary
†Ericsson, Hungary
‡MTA-BME Information Systems Research Group

***Abstract*—While the industry is still struggling to embrace the network function virtualization paradigm, recently a novel approach has appeared with the promise of improving the state-of-the-art: stateless virtualized network functions. Rooted in cloud-native computing, this design outsources the state embedded in virtual network functions to a dedicated "state storage" layer, facilitating elastic scaling and resiliency. While related work mostly focuses on performance, we in this paper pinpoint all other factors that weigh in when it comes to deploying the stateless design in a carrier-grade operator network. Among those we argue that reliability and flexibility are key, and we propose a system design that can be adapted to any telco use case without the need for complex coordination among the network control, the stateless network functions, and the state storage backend. Then, in extensive evaluations on synthetic use cases we show that the additional flexibility provided by our design does not come at a performance penalty; in fact, in certain cases our design outperforms the state-of-the-art significantly. Finally, we present what to our knowledge is the first product-phase realization of the stateless paradigm, an operational virtualized IP Multimedia Subsystem that can restore the live call records of thousands of mobile subscribers under a couple of seconds with half the resources required by a traditional "stateful" design.**

***Index Terms*—Network Function Virtualization, Telco Industry, Stateless Network Function, Distributed Key-Value Store**

I. Introduction

Virtualization techniques bring abrupt changes to how we design telco systems. Network Function Virtualization (NFV) offers the opportunity to move the software running on traditionally expensive custom physical nodes into the cheap multi-purpose cloud, resulting in fast configuration and development cycles and cost-efficient scalability [1]–[3]. The modern NFV ecosystem is fundamentally *stateless* [4]–[7]; if the virtual Network Functions (NFs) do not maintain persistent state on their own, then scale-in/scale-out, and even fail-over events, are less complex to handle, improving overall elasticity, scalability, resiliency and performance [7]. In this paper, *we take a new look at the stateless NF paradigm and we re-evaluate its applicability in large-scale operational telco networks*.

Taken from cloud-native designs [8], the stateless NF paradigm calls for the complete decoupling of NF processing, responsible for the forwarding, modification, and filtering of network packets, and the associated state, involving all persistent and semi-persistent information maintained in a NF that is necessary to perform the required processing on the network traffic traversing the NF [4], [5]. Such state may include address mappings in a network address translator (NAT), or connection tracking information maintained in a stateful firewall [9]. The main motivation for factoring this state out of NFs is scalability and resiliency. Consider the case of a NAT, for example: if address mappings are scattered in the address space of the NAT NF, then upon a scale-out event the address mappings to be handled by the new NF instance need to be tediously removed from the old instance and conveyed to the new instance. Meanwhile, a seamless flow of live traffic and complete synchrony of the NFs must be guaranteed, which requires complex control and coordination of the components involved, like the NFs themselves, the load-balancer, and the SDN controller [1], [3]. If, however, no persistent state is embedded in an NF but rather state is maintained by a separate storage backend layer, then scale-out is seamless as the state needed by the new NF instance is readily available there.

Of course, eliminating the need for explicit state management within NFs does not come for free; the downside is the complexity faced by all distributed systems: performance and data consistency [4]–[7]. These difficulties are especially compelling in telco deployments, which pose their own requirements [10], [11]. *A carrier-grade telco NF ecosystem must provide extreme steady-state performance and provide strict Service-Level Objectives (SLOs) simultaneously*, at the order of hundreds of millions of packets per second throughput and 1–10 ms delay budget [12]. This requires *extreme scale and elasticity in terms of the sheer size of the deployments, NFs, and state information*; in Section IV we show a virtualized IP Multimedia Subsystem (vIMS) solution handling the live state of possibly millions of mobile subscribers. Carrier-grade NF systems pose *tough resiliency requirements*; staying with the above example the vIMS should never drop calls, lose subscriber information, or miss handover events, not even in the case when one or more NFs, possibly managing tens of thousands of subscriber calls, fails. Finally, typical cloud-based telco deployments [2] are *massively heterogeneous and multi-party* and, correspondingly, must expose simple APIs and a modular system design to third-party developers.

In summary, in order to gain traction in the telco world *the stateless NF paradigm must deliver high performance in normal operation, seamless scalability and ultra-high reliability during scale-in/scale-out and under failures, and finally a modular and use-case-agnostic design.* In this paper, we show that existing stateless NF designs, mostly originating in academia, fall short in some way or another in the face of industry-scale telco requirements.

We argue that the state management problem posed by

the stateless NFV setting is complex enough to warrant the introduction of a separate state-management plane, next to (or below) the forwarding plane. For this purpose, we adopt DAL, a powerful distributed shared-memory system [6] from the literature. Our contributions are, accordingly

(i) the design, construction, implementation and evaluation of an entire telco-grade stateless NF ecosystem on top of the storage layer DAL;
(ii) an extensive evaluation on various use cases to compare the features and performance of our stateless NF design with the state-of-the-art [4]–[7];
(iii) the first production-quality industrial-scale stateless telco NF system, a virtual IP Multimedia Subsystem (vIMS) that provides resiliency against NF instance failures within a similar time budget as the traditional 1+1 protection scheme at half the costs. Our vIMS is expected to go into production in early 2019.

The paper is organized as follows. In Section II we provide a deep-dive into the stateless data plane paradigm by describing the design principles, the necessary data store features driven by the telco-grade requirements, and the shortcomings of the existing solutions. In Section III we present DAL, our proposal for the state plane. Then, in Section IV we evaluate the performance of DAL in a comprehensive set of network functions. First we measure throughput and packet latency in steady-state and during the transitional phases of scale-out/ scale-in events in two applications with read-heavy and write-heavy access patterns. Second, as an illustrative feasibility proof, we present and evaluate a fully fledged telco vIMS, in which our stateless NF design reaches significant resource savings compared to the traditional 1+1 protection scheme. Finally we conclude the paper in Section V.

II. Concept of stateless: the road to adoption

A. Network Functions and Embedded State

Contrary to traditional packet processing deployments, which typically targeted only a specific protocol layer, like Ethernet or IP, and applied simple processing patterns to packets, like forward on a port or drop, modern telco pipelines, like Broadband Network Gateways or Mobile Gateways [10], [11], proliferate in complex middlebox functionality [2], [9]. Such functionality involves the flexible and efficient filtering, monitoring, modification, or redirection of network traffic, allowing telcos and operators to provide "value-added" services on top of the plain old carrier service, like firewalls and intrusion detection systems for improved security, NATs for IP address sharing and isolation, video and audio transcoders for media-agnostic connectivity, and load balancers for parallelization and resource pooling. Moreover, today's telco middlebox deployments tend to get more and more *complex*, e.g., virtualized telco systems handle millions of signaling events per second and industrial IoT and cloud-controlled collaborative systems run at an even larger scale [2], and pose previously unseen *throughput and latency SLOs*.

The skyrocketing complexity calls for the softwarization of NFs, which would allow to elastically scale, both horizontally and vertically, on demand, to quickly recover from failures, to mix multiple vendors' NFs in a single deployment (vendor-heterogeneity), and to leverage contemporary cloud-native facilities to outsource the NF infrastructure management [2]. Unfortunately, softwarization makes fulfilling SLOs much more difficult, since most software frameworks that NFs run on top of, like the Linux kernel or the OpenStack network and compute virtualization platform, lack the real-time capabilities of hardware appliances. The most difficult problem arises with operational state embedded in NFs, like address mappings in a NAT, connection tracking information in a stateful firewall, or server associations in a stateful load balancer. Such state is typically created in NFs in response to certain data-plane events, like the creation of a new flow by the reception of a TCP syn packet, or after certain signaling events received from the control plane, like a subscriber performing a handover in a mobile core device. Locking state into a single NF instance curtains scaling and elasticity, since state must be tediously migrated through different NF instances using possibly piecemeal NF-specific methods, and harms resilience, in that restoration of a failed NF will need to re-establish all the embedded state of the failed NF instance in a timely manner.

B. Stateless Network Functions

The typical way to tackle embedded state in cloud-native computing is the refactoring of the processing functionality by the externalization of all persistent state [8]. This involves decoupling the processing logic from state storage in network functions and placing the state in a separate layer, typically a distributed key-value store, that is solely responsible for the maintenance of application state and offering it for access and modification to interested NFs. This design pattern, where all embedded state is offloaded to a dedicated state storage backend, is called *Stateless Network Functions* [4]–[7].

Stateless NFs bring lots of operational benefits. First, *resilience/restoration* is simpler with the stateless design since the new NF instance will have instantaneous access to all the state needed; second, *elasticity* is improved, e.g., NFs can be seamlessly scaled out and new instances will be immediately capable to process traffic since, again, all state is available in the storage backend; and finally *resource-pooling/load-balancing* will also benefit since the load balancer will not be reliant on the affinity of traffic flows to NF instances (so that packets will always "meet" the state that was left behind for them by earlier packets), but rather packets/flows can be freely spread across workers. Contrast this with a traditional "stateful" design, e.g., for a new firewall NF instance to handle a packet that is part of an already established connection whose conntrack state has been created earlier at another NF instance, it first needs to access and download the corresponding state from the old NF instance, provided it is still available, or recreate it from logs; otherwise, the packet will be dropped because a lookup will fail. In a stateless NF, said conntrack state is readily available in the state storage backend and the

new instance can immediately process packets after retrieving the state from the backend.

C. Taxonomies for Stateless Network Functions

Below, we categorize existing stateless designs along various important problem dimensions.

1) Deployment Model: Our first categorization dimension considers the way in which worker nodes that process packets and the storage servers that maintain state are deployed and associated with each other.

Local-only model. Early NF frameworks assumed that NF state is per-flow and, correspondingly, it is only accessed locally, of course assuming a certain traffic load balancing scheme that ensures flow–NF affinity [1], [3]. For example, TCP connection state is per-flow. However, a global active-flow counter will need remote access since it may be modified from multiple NF instances. Accordingly, these frameworks do not accommodate NFs with shared state.

Remote-only model. In this model *all* network state is accessed via a remote backing store [4], [5]. While maximally flexible and readily provides shared state, this approach comes at the cost of steady-state performance: accessing remote state in the fast packet-processing path inflates packet latency and consumes network bandwidth for I/O. It has been reported [7] that, relative to an NF that uses local state only, a remote-only approach can lead to a 2–3 times degradation in throughput and an orders-of-magnitude increase in latency. Crucially, the penalty grows with the number of state accesses, which can reach high numbers in typical NFs [9].

Local+remote model. In this model, all NF state is fixed at compile time to be either local or remote and accessed accordingly in operation: per-flow state is accessed locally, while shared state is handled remotely [13], [14]. With adequate hinting on each state's access pattern, this model can be very efficient; however, scaling events that require replication of local state usually result in "stop the world" behavior, leading to long pause times during which both old and new NF instances stop processing packets. This model is complex to implement and to deploy due to tight coordination among the SDN controller, the NF controller, and the load-balancer.

Distributed shared object (DSO) model. In this model, NF state is distributed among NFs and can be accessed by any NF, which removes the long pause times with the "local+remote" model [7]. In addition, all state variables reside in a single shared address space and the state management framework transparently resolves all state accesses, therefore the NF developer does not need to distinguish between local and remote state. Accordingly, the API is much simpler. While extremely flexible and easy to develop against, experience suggests that this model is the most difficult to support efficiently in implementations: we show in Section IV that the only publicly available DSO prototype, S6 [7], does not implement adaptive state migration at this point and it lacks efficient remote state handling; correspondingly, it becomes prohibitively slow in certain carrier-grade telco access patterns.

Takeaway 1: Different industrial use cases may require different deployment models and *an industrial-scale stateless NF framework should support all deployment models*; e.g., the "local-only" model is undoubtedly the fastest possible mode in steady state, so in use cases that admit this model, like a "soft" flow packet counter, the local-only mode should be supported out of the box, and similarly for the rest of the models.

2) Access Pattern Hinting: Next, we categorize existing work based on whether the NF programmer needs domain-specific knowledge to "hint" the state-backend regarding the typical access pattern of externalized state instances.

Hinted mode. The "local+remote" model requires the programmer to specify at compile time whether a particular state instance is per-flow, in which case the state storage backend will optimize state placement for local access, or "remote", in which case the state instance may be remotely modified from multiple NF instances. This requires domain-specific knowledge that may not necessarily be available during NF development and restricts NF state to be static in nature, i.e., a state cannot change access pattern during its lifetime.

Unhinted mode. In this mode there is no distinction between local and remote state, no proactive migration is required during scaling events, and state migration no longer needs tight coordination with load-balancing/resource pooling. The outcome is reduced system complexity and simpler NF development. This model trivially works in the "local-only" model (where there is no "remote state" *per se*) and in the "remote mode" (where there is no "local state"); DSOs should also support this model out of the box but, as we show in Section IV, implementations currently do not follow theory.

Takeaway 2: An industrial-scale stateless NF framework may default to an unhinted mode provided it will adaptively exploit the performance optimization opportunities offered by the hinted mode, otherwise it must provide both modes.

In the next section we show how DAL, the state layer we adopt, supports extremely efficient unhinted state management using intelligent adaptive state placement; accordingly, our stateless design will default to the unhinted mode.

3) State Management: Next, we consider whether moving/migrating state from one backend server to the other, e.g., upon scale-in/scale-out event or during restoration, needs a dedicated migration manager or happens spontaneously.

Managed state migration. In this mode, moving state from one backend server to the other in order to improve access locality, i.e., turning remote accesses local, requires an explicit controller to manage the whole process. Unfortunately, the interaction between NFs and a dedicated manager makes the system complex, breaks modularity, and introduces unwanted dependencies inside the workload. [7] adopts this model.

Unmanaged/adaptive state migration. In this mode, moving of state "ownership" between backend servers occurs adaptively, in response to a change in access patterns as detected by the storage backend. Correspondingly, state is always moved as close as possible to the NFs that frequently access it; for instance the S6 framework contains this model [7] but, unfor-

tunately, the current implementation lacks adequate support, as we show in Section IV.

Takeaway 3: For maximal flexibility, *the stateless NF framework must support the unmanaged mode*.

So far, no prior work that would support this mode has appeared; we introduce one in the next section.

4) State Representation: The final categorization dimension we consider is the type and nature of NF-specific persistent state that is offloaded to the storage backend.

Explicit state representation. In this model, state variables, like NAT mappings, counters, etc., are explicitly represented in the state storage backend. Early solutions following this model involve pico-replication [15], a framework for frequently checkpointing the state in a NF such that upon failure a new instance can be launched and the state restored instantaneously; recent realizations all adopted this model due to its simplicity and easy API [4]–[7], [14].

Implicit state representation. This model works through "input logging", whereby input events to the NF instances are logged in a separate component and state is restored by replaying the input to the new instances. In stateless designs targeting data-plane state the typical input to be logged is every packet that spawns a state change (e.g., a TCP syn packet that creates a new mapping in a NAT [16]), while in NFs that interact with the control plane the input to be logged is every control message that was received by the NF instance.

Takeaway 4: An industrial-scale stateless NF framework should support both implicit and explicit representation; the explicit mode is much simpler to code against, while the implicit model may allow the timely restoration of possibly immense sized state.

In Section IV, we show how our stateless NF framework benefits from supporting both modes: using the explicit mode we can seamlessly scale essentially any stateful data-plane function, while the implicit mode will be indispensable when we restore the full call records of thousands of subscribers in a mobile core device, a massive scale that would be impossible to support in the explicit mode (see also [9]).

D. Contributions for Industrial Adoption

We argue that in order to gain industry acceptance, a stateless NF framework must support all deployment, state management and state representation models. In addition, it should provide efficient implementation for the unhinted mode. Adopting the "local-only", the "remote", and the "remote+local" deployment models maximizes performance in steady state under different use cases while the DSO model maximizes deployment flexibility; supporting the unhinted mode makes the stateless NF design use-case agnostic and allows fast and seamless scale-in/scale-out and restoration; making explicit as well as implicit state representation available to developers renders it possible to match the state representation model with particular use cases, resulting vendor-agnostic, heterogeneous deployments; and finally flexible state management removes dependency on the load-balancer and resource-pooling framework, allowing maximal modularization and easy third-party development model. In the rest of the paper, we provide such a design.

III. DAL: DESIGN AND IMPLEMENTATION

We present DAL, the state-storage backend we adopted for our carrier-grade stateless NF framework. First, we summarize the original design from [6] and then we describe how DAL supports our carrier-grade stateless NF framework.

A. Architecture

From the viewpoint of a NF, externalized state can be accessed with the best possible performance if it is co-located. Achieving co-location might be relatively simple with steady configurations, however cloud based deployments must adapt continuously to the changing environment and to load fluctuation. DAL is a distributed shared memory system, designed from the ground up to achieve the lowest possible latency in this dynamic context [6]. One of the *main features of DAL is taking data sharding to the extreme by being able to handle individual data items independently and move them between the instances in the cluster* as dictated by the actual access pattern, thus enabling dynamic and automatic co-location.

DAL consists of two components, the server and a client library that can be linked to the NFs. The client library exposes a key-value API towards the applications. The API has standard key-value components that allow for writing and reading keys, and it also exposes advanced features, such as messaging and cluster-global announcements. While traditional key-value databases, like Redis [17], organize keys into shards with hash functions, DAL separates the keys and values with an additional abstraction layer. This layer manages the *key map*, a data structure which stores all meta information for the keys, most notably the location of the value (or data block) associated with the key. Multiple DAL servers form a cluster, in which all peers share the same key map. All key map updates, e.g. key creation or data move, are coordinated by a leader DAL node, and replicated to the rest of the cluster.

This abstraction makes it possible to ensure the co-location of data elements with NFs currently working on them, but comes at the cost of a two-phase lookup. In the first step, the client library queries the DAL server it is connected to for the current location of the related data element, then, in a second step, the data itself is addressed directly. The two-phase lookup is done transparently from the NFs' point of view and the API provides an abstraction, a *handle object*, that can be used to store the lookup information, thus subsequent accesses using the handle object will bypass the first lookup step.

Figure 1 shows the high-level architecture of DAL. While many deployment options are possible, to enable fast local data access via Inter Process Communication (IPC), it is recommended to deploy a server and the NF to the same server node as it is shown for `Host 1`. In this deployment the DAL library can directly read the memory pool of the server using the location information from the handle object. DAL employs an opportunistic approach by checking a version

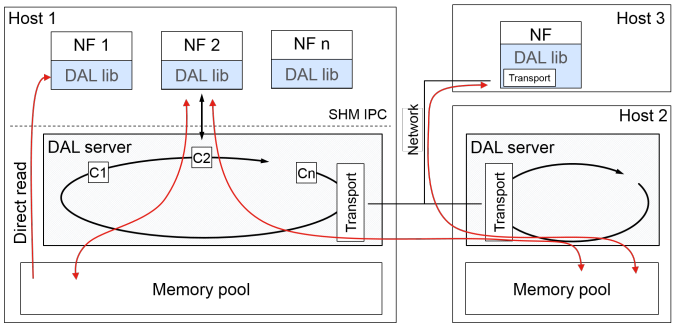

Fig. 1: DAL architecture

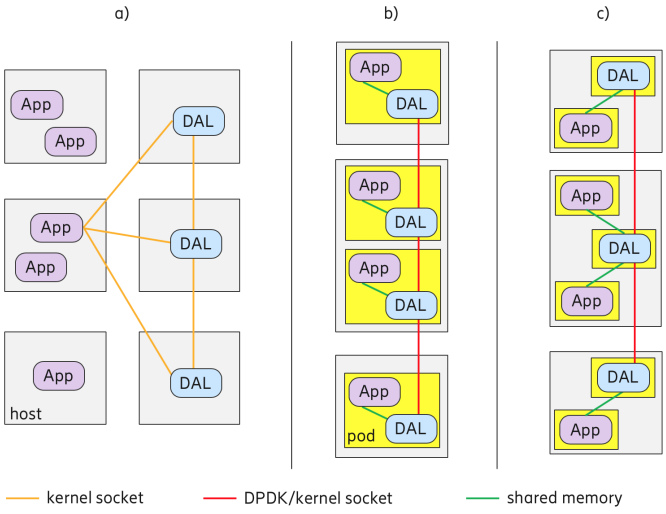

Fig. 2: DAL deployment options

number associated with the data block before and after the read. In case the server updates the block while it is being read, the library retries the operation.

DAL servers are single threaded and are operating in poll mode. In every execution loop the server checks the so-called *control block* of each co-located client. A control block is essentially a shared memory region that is used by the client library and the server for two-way communication, specifically for the local write requests and handling all remote operations. The server also checks for incoming packets on its transport interface in each loop. DAL has two different transport implementations, one using a normal UDP socket and one that uses DPDK [18] to achieve low latency. The advantage of this design is that co-located clients can be lightweight, e.g., they do not need a dedicated interface as in RAMCloud [5]. However, to extend the supported deployment options, the client library can also use a network interface to access the DAL cluster, as `Host 3` in Figure 1 shows.

In a cloud environment any component may fail and as a result data reliability is essential. NFs using DAL can configure their keys to have multiple data replicas, in which case multiple DAL server instances will store a copy of the value associated with the key. When a client issues a write, the command is always routed to the master data replica, which will replicate the new value to the slave data blocks as well.

B. DAL: a Carrier-grade State Storage Backend

Next, we show the features in DAL that we added in order to provide all the carrier-grade requirements laid down in Section II and allow us to adopt DAL as the state storage backend in our stateless NF framework.

1) Deployment Model: The DAL design adopted in our stateless NF framework *provides great flexibility in how clients (NFs) and servers are deployed;* Figure 2 lists three fundamentally different deployment options.

In (a) DAL is deployed as a "remote only" storage system, where NFs use the DAL client library with kernel sockets to access data. This model puts the fewest requirements on the underlying execution and orchestration system, at the cost of increased data access times. Option (b) depicts the "local", or "strongly coupled" deployment model (this model also reflects the current industry-trend to deploy value-added services as "sidecars" at worker nodes [19]), where each NF instance has a dedicated DAL instance (see DDAL in Section IV). Here, each NF instance communicates with its side-DAL server via shared memory, while DAL servers utilize a DPDK-enabled fast data-plane connection to communicate within the cluster. This option provides the lowest possible data access latency in steady state, but it is rigid in terms of scaling: the application and the storage layer are always scaled together. A more flexible DSO-like deployment is depicted in (c), where DAL servers and NFs reside in two distinct scaling domains. Orchestration must ensure that at least one DAL server is deployed on each host. Then, NFs can communicate with a host-local DAL server via shared memory, but multiple NF instances can share the same DAL server when talking to the DAL cluster (see CDAL in Section IV). This way we can fine-tune resource usage without sacrificing the latency performance of the "sidecar" model.

2) Access Pattern Hinting: In Section II we concluded that supporting the unhinted mode, where the access pattern on state is not fixed at compile time, is key to industry-scale use cases. A key design objective of DAL was to support this feature through *implementing an intelligent adaptive state placement mechanism.*

In order to optimize the locality of individual data elements, the DAL servers continuously record and analyze access statistics on a per data item basis. If an item is in majority accessed through a single remote server, it gets moved there. The length of the access history that the algorithm takes into account is configurable as different NFs may have different requirements. E.g., in case of a counter that is kept in DAL for a given flow, when the flow is re-located to another NF instance, the data should be relocated at the first remote access.

3) State Management: For maximal flexibility, a carrier-grade stateless NF framework must support the unmanaged mode where state migration from one node, or one DAL server, to the other does not need central orchestration and additional management infrastructure. DAL supports this feature through

the so-called *auth* mechanism.

In DAL, data can be moved between DAL servers without notice. Correspondingly, the cached handle objects may occasionally store obsolete location information. DAL uses an opportunistic approach to tackle this problem: DAL-handles store an additional *auth* value which is used to validate location information. In case of an auth mismatch, the client library gets an internal error and executes the full two-phase lookup, again transparently to the NF. Hence, *state migration is fully adaptive and occurs without central control in DAL.*

4) State Representation: Finally, we show how *DAL supports both explicit and implicit state representations*.

As mentioned earlier, the DAL client library exposes a key-value API towards the applications, but the values stored into DAL are completely opaque to the DAL cluster. so it is up to the NF developer to choose which model to use to represent NF state. In most use cases we found the explicit mode to be much easier to work with thanks to its simple API; however, in the particular case of the vIMS where the explicit representation of the call records of millions of subscribers amounts to a prohibitive quantity of state, we rather built our solution based on the implicit mode, as shown in Section IV.

IV. Performance and Feature Evaluation

In our proposed stateless NF framework we deploy NFs with custom Python scripts in Docker containers. We build pipelines from standard BESS (Berkeley Extensible Software Switch) [20] modules to implement load balancing, and to provide connectivity between components. The state plane is provided by DAL. To generate traffic and to measure throughput and packet latency we also use BESS. Here we show the performance of our design in illustrative corner cases, and as a proof of feasibility, we present a telco use-case soon to be deployed in an operational network.

A. Synthetic Measurements

For these measurements we used a 28-core Intel Xeon CPU E5-2680 v4 @ 2.40GHz - 56 hyperthread server.

Performance of key-value stores. Since there is no stateless NF system without storage components, first we benchmark different key-value stores. Table I sheds light on their raw performance.

TABLE I: Access time for key-value stores [μs]

	Redis	S6	DDAL[1]	CDAL
Local read	69.84	0.16	0.12	0.12
Local write	94.65	0.17	2.30	2.30
Remote read	73.88	18.41	21.20	-
Remote write	94.04	18.23	22.74	-

Key-value stores distribute the NF states among the cluster's instances. Based on the proximity of the NF and an instance that can serve the NF's read/write request, operations are categorized as local or remote. In S6, for example, if the NF has the ownership of a state, then read/write operations are significantly faster than the remote ones.

[1]DDAL used kernel socket for remote operations, instead of DPDK

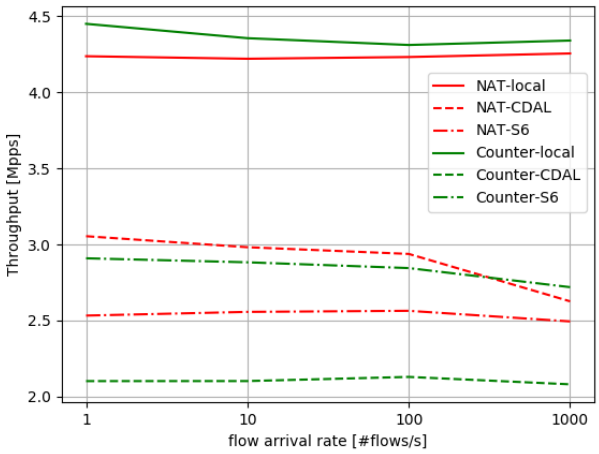

Fig. 3: Steady-state throughput of different NFs

Redis is a popular example of a TCP-based, general purpose key-value store[2] [17], which explains why the time it takes for an application to write or read a single piece of information to the store ($70-90\mu s$) is orders of magnitude worse than the access times of the other technologies. Therefore, we exclude Redis from the subsequent evaluations.

DAL is evaluated in a single node setup, but with two extremes: (i) DDAL corresponds to option (b) of Figure 2, where in each container runs a DAL instance alongside with a NF, similarly to the S6 architecture; and (ii) CDAL corresponds to option (c), where there is a single DAL instance per host running in its own dedicated container. In CDAL there is shared memory between the NFs and the DAL instance, thus inside a single node there is no remote operation.

Steady-state performance. Although S6, DDAL, and CDAL exhibit similar performance on the atomic state-layer operations, the overall performance of NFs greatly depends on the complexity of the packet processing task and the frequency of such atomic operations. We examine two NFs that belong to the extremes in terms of state access: *(i)* a read-heavy NAT, and *(ii)* a write-heavy source IP address Counter NF. For each incoming packet, NAT reads the state store to get the translated IP address and port number in order to overwrite the original packet header. When a new flow arrives to the NF, it generates new address mappings (one for the outgoing and one for the reverse direction) and writes the new state to the state store. The Counter NF is a simplified statistical function, which samples the number of incoming packets for each flow. Flows are defined by their source IP address. The NF writes packet counts into the state layer for every tenth local update in order to reduce the load on the state layer.

Steady-state operation is free of state migrations, however NFs may cerate per-flow states when new flows arrive to the system; Figure 3 depicts the effect of flow arrival rate on the

[2]Currently there is no official C++ client for Redis, so we used *acl-redis* client: https://github.com/acl-dev/acl/tree/master/lib_acl_cpp/samples/redis

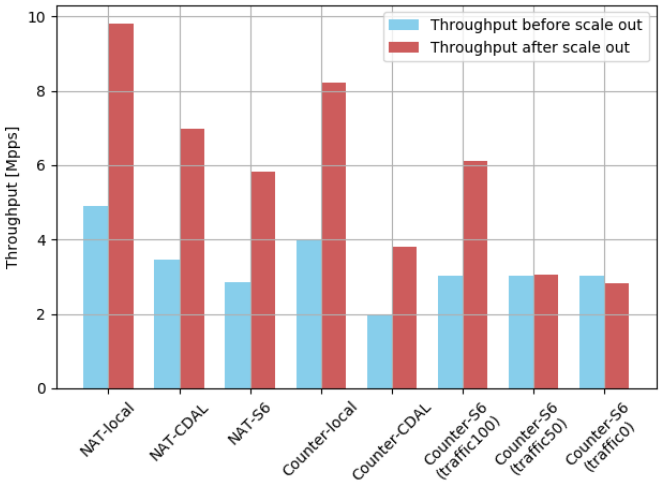

Fig. 4: Total throughput before and after a scale-out event

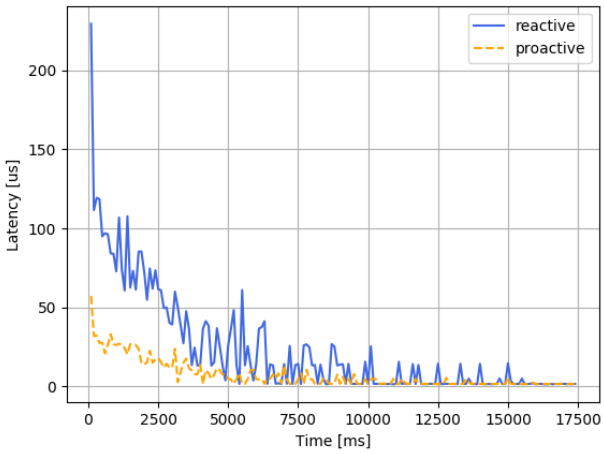

Fig. 5: Packet latencies after a scale-out event of a NAT

throughput in the 2 NFs reading/writing states locally, i.e., without using any external state storage component, into S6 and into CDAL. During the test, the total number of flows is kept constant at 1000, but flow arrival rate is varied. We observe that extensive flow arrival does not significantly impact NFs using local-only state variables and that write-heavy NFs show lower performance than their read-heavy counterparts. S6-based NFs work in hinted mode while DAL works in unhinted mode, still the NAT-CDAL outperforms NAT-S6. On the other hand, Counter-CDAL falls behind Counter-S6; this is because S6 local writes are significantly faster than CDAL local writes (see Table I). Additionally, state creation impacts NAT-CDAL the most.

Elasticity - scale factor. Figure 4 shows saturated, steady-state throughput of a system before and after scaling out from one NF to two NFs. The NFs following local-only deployment (NAT-local, Counter-local) exhibit an ideal scale factor, i.e., they double the throughput after scale-out: these provide an idealistic upper bound as they do not have shared states. NAT-CDAL and Counter-CDAL work in unhinted mode; their adaptive state migration results in local-only operations, hence their scale factor is also around 2. The NFs in local-only mode have significantly larger throughput than the corresponding DAL NFs, but as we argued before, failover is not possible and scaling requires managed state migration.

S6 in DSO-mode does not implement adaptive state migration, so NAT-S6 and Counter-S6 work in remote-mode. If the load-balancer steers a flow to a NF having the ownership of the flow's state, then the remote read/write operations are fast. On the other hand, state access operations are slower when operations are in fact remote. Counter-S6 (traffic*N*) depicts the case when the load balancer steers *N* percent of the flows to the correct NF. Traffic100 requires an explicit state controller and coordination with the load-balancer, which is not feasible in every deployment. Due to the lack of adaptive state migration and intelligent state placement, scaling out does not provide additional throughput in case of traffic50 and of traffic0.

We can observe in Figure 3 and 4 that write-heavy Counter NFs have lower throughput than their ready-heavy NAT counterparts, even though the Counter NF has simpler packet processing algorithm. Note that, however, the state access pattern becomes less significant as complexity of the packet processing algorithm increases.

Performance impact of state migration. State migration happens because of scaling or restoration. These events are usually rare, but their impact on performance should still be minimal to maintain strict industrial-scale delay-SLOs. Figure 5 and 6 show the latencies of flows steered to a newly started NF after a scale-out event has happened (at time 0). As opposed to Figure 4, here the offered load is moderate, less than 1 Mpps, which is well below the NFs' maximal capacity. This choice of input traffic rate deliberately avoids buffering of packets, as we are interested in the performance of the stateless design, without weighing in any potential buffertime.

In Figure 5 the blue solid line denotes NAT's packet latency when applying DDAL with adaptive state migration, i.e., the NF tries to access state information only after encountering an unknown flow. The orange dashed line belongs to a DDAL NAT with *proactive* state migration; here a state management controller makes sure that required state is available in the DAL instance collocated with the NF by synchronizing the state migration with the load balancer before time 0. The proactive method causes latencies to grow around $50\mu s$, because the NF needs to acquire DAL-handles for the new state variables. However, steady-state is reached considerably faster than in the reactive case: the transient phase lasts $5s$ and $10s$ for the proactive and the reactive case, respectively. In addition to the prolonged convergence time, adaptive, i.e., reactive, state migration results in much higher maximal delays ($200\mu s$), as the system first relies on remote reads until DAL notices the change in the access pattern, transfers the affected state to the closest DAL instance, creates the new handles and

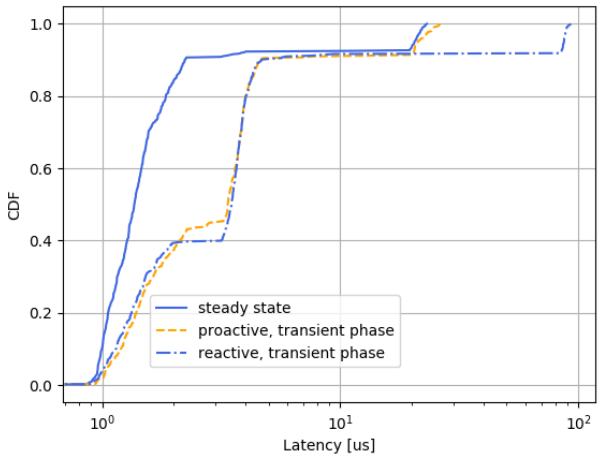

Fig. 6: The empirical distribution of packet latencies after a scale-out event of Counter NFs

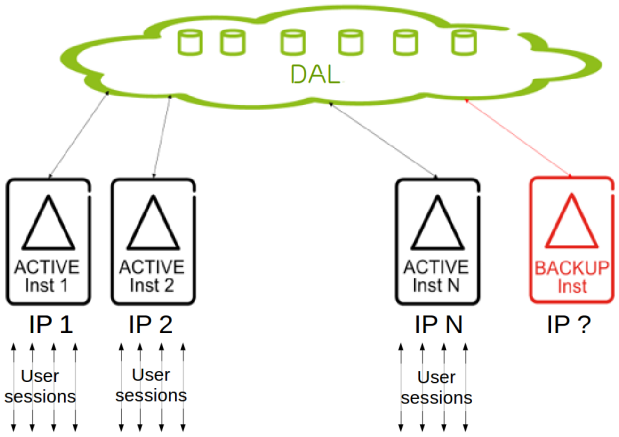

Fig. 7: Cloud deployment of the Border Gateway Function: Active virtual machine instances handle the sessions of a number of users, and a BACKUP instance is in standby for failover, building on the state stored in DAL

eventually carries on with local reads.

Figure 6 shows similar effects for the Counter NF through the displayed empirical distribution of measured packet latencies. Note that packet latencies, due to the remote writes, in the reactive case reach close to $100\mu s$ for 10% of the packets, and latencies due to acquiring state handles in case of proactive state migration all fall in the range of $[1\mu s, 30\mu s]$. These values are larger than individual access times of atomic operations in Table I, because batch arrival of new flows results in queued DAL operations.

Note, however, that a high packet rate would cause more state migration events, and those high latencies for the early packets could start and build up a significant buffer queue, ultimately prolonging the overall transient phase and increasing the packet latency values. This is again an important reason for why the flexibility of state plane scalability is crucial, and for why the elasticity of NFs should be maximally supported.

B. Telco Application: vIMS Restoration from DAL

The IP Multimedia Subsystem (IMS) enables various types of media services to be provided to end-users using common, IP-based protocols. To protect and hide vulnerable details of the operator's core network, the Border Gateway Function (BGF) is placed between the access and core networks as a pinhole firewall and NAT/NAPT functionality. As such, it is responsible for filtering and transferring the RTP based media streams exchanged by mobile subscribers.

Traditional telco nodes couple the states of the user sessions with the physical executors. Accordingly, if a physical entity fails, then the handled user sessions get lost. On the other hand, it is also common that each functionality is implemented on top of a dedicated hardware resource (e.g. board, DSP chip) that overall makes the system distributed and inherently more robust against hardware failures. In case of a failure, only those calls are affected that shared the same resources, which is an insignificant amount of sessions.

However, this does not apply to the cloud anymore where a VM can serve tens of thousands of sessions, relying on a single hardware infrastructure. In a cloud deployment of BGF each user session is tied to a particular instance, i.e., connected to a particular IP address / port of the BGF VM instance. In such a system if a VM fails, all the sessions get lost, which impacts a large number of subscribers and this cannot be tolerated.

Alongside this application we use DAL for storing the state and retrieving it, when necessary, and we compare the DAL-based solution with the traditional (not virtualized) resiliency scheme: 1+1 hot standby for the user plane dedicated hardware. These latter ones work in mated pairs, i.e., if the primary fails then the secondary takes over the traffic.

The evaluation setup is the following. We distinguish two roles of VMs in the cloud deployment of BGF: Active and Standby. In the cloud solution we give N+1 protection: 1 Standby that can take over the calls from any Active VM (see Figure 7 for illustration). We used 4 Active and 1 Standby VMs in each evaluation setup, and we made 10 runs the average results of which are plotted in Figure 8. The VMs were run in an OpenStack environment, and all VMs were placed on different compute nodes. The input traffic were collected from IPv6 access and IPv4 core traffic.

We were interested in seeing how fast the Standby reconstitutes the states and takes over the traffic when we kill an Active VM. The results are shown in Figure 8. The failover time is given on the *y*-axis as the number of flows (representing the load) increases on the *x*-axis. In total we measured an average of 1.6 sec outage on the media plane, that is what all end users handled by the failed VM experience. As the media plane restoration time results are below 2 sec, including the time necessary for failure detection, the design is acceptable as the industry standard threshold of maximum tolerable outage on the media plane is 2-3 seconds, otherwise a regular user drops the call. This tolerable range of service outage duration

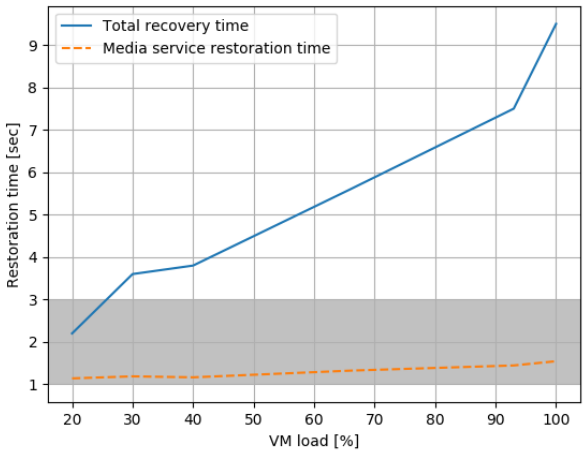

Fig. 8: The dependency of service restoration, and total recovery time in the function of processing load on the BGF VM instances

is represented by the shaded gray area in Figure 8. The media service restoration time is the sum of 1 sec for failure detection, 0.5 sec for media plane recovery, and there is a short time interval that is needed for interface reconfiguration and to have the GARP propagation (having the underlying switches update their MAC tables).

The total recovery time under maximum VM load (also shown in Figure 8) is the sum of the media and the control plane's restoration periods. We measured an average of 7.7 secs needed for control plane recovery in cases where the traffic load on the VMs were at their maximum. The relatively long duration of the control plane restoration is due to its complexity. After the overall recovery, the service level of the call is exactly the same as before the failover, e.g., if an affected user changes its location, the session will not get disconnected. During both the media and control planes' restoration, 33% of the time is spent in DAL reads, and 66% of the time is taken for state restoration. These activities run in an intertwined fashion, alternating between each other.

The benefit of relying on N+1 redundancy and DAL-based retrieval of states is the great cost saving. When introducing the DAL-based redundancy, we need 1 vCPU and 1 vNIC per VM compared to the original flavor in order to be able to cover the resource consumption of the DAL clients co-located with the BGF functions. The additional memory usage is around 200MB in each VM. On the other hand, we need only 1 backup instance for every N vBGF instances, instead of N.

The DAL-based solution will go into production in 2019.

V. Conclusion

We present a novel stateless NFV architecture where the main focus is on flexibility in terms of adaptation to very different applications and usage patterns. State-of-the-art technology in this field is highly optimized for either the "sidecar" solution [19], such as S6 [7], which uses a dedicated database instance for each worker and this way is ideal for workers with heavy packet processing workload, or for a centralized solution such as RAMCloud [5] or Redis [17] which are ideal for latency insensitive, low intensity workloads, such as signaling or control applications where database consistency is essential. Our solution, DAL, allows for scaling the packet processing functions and the database instances independently, this way leading to a fully flexible architecture without any deployment constraints. Moreover, DAL offers automatic state management with the aim of yielding maximal performance.

Other competitive features of our proposed design, which place DAL on top of all the alternatives, are reliability, modularity, and the possibility of deploying it on commodity hardware. The state plane provided by DAL offers industry-scale performance for any use case adaptively, i.e., without explicit burden of managing states, packet processing functions and network control by an additional layer of coordination. We prove the benefits of our design across corner cases in synthetic setups. Furthermore, we also present a telco use case for which the DAL-based stateless design is planned to be deployed shortly in an operational telco network.

References

[1] Z. A. Qazi *et al.*, "SIMPLE-fying middlebox policy enforcement using SDN," in *ACM SIGCOMM*, 2013.

[2] J. Sherry *et al.*, "Making middleboxes someone else's problem: Network processing as a cloud service," in *ACM SIGCOMM*, 2012.

[3] S. Palkar *et al.*, "E2: a framework for NFV applications," in *ACM SOSP*, 2015.

[4] M. Kablan *et al.*, "Stateless network functions," in *ACM HotMiddlebox*, 2015.

[5] ——, "Stateless network functions: Breaking the tight coupling of state and processing," in *USENIX NSDI*, 2017.

[6] G. Németh *et al.*, "DAL: A locality-optimizing distributed shared memory system," in *USENIX HotCloud*, 2017.

[7] S. Woo *et al.*, "Elastic scaling of stateful network functions," in *USENIX NSDI*, 2018.

[8] M. Stine, "Cloud native architecture patterns tutorial," in *O'Reilly Software Architecture Conference*, 2017, [Online: https://www.slideshare.net/mstine/cloud-native-architecture-patterns-tutorial].

[9] J. Khalid *et al.*, "Paving the way for NFV: Simplifying middlebox modifications using statealyzr," in *USENIX NSDI*, 2016.

[10] Intel, "Network function virtualization: Quality of Service in Broadband Remote Access Servers with Linux and Intel architecture."

[11] ——, "Network function virtualization: Virtualized BRAS with Linux and Intel architecture."

[12] N. Mahmud *et al.*, "Evaluating industrial applicability of virtualization on a distributed multicore platform," in *IEEE ETFA*, 2014.

[13] A. Gember-Jacobson *et al.*, "OpenNF: enabling innovation in network function control," in *ACM SIGCOMM*, 2014.

[14] S. Rajagopalan *et al.*, "Split/Merge: system support for elastic execution in virtual middleboxes," in *USENIX NSDI*, 2013.

[15] ——, "Pico Replication: a high availability framework for middleboxes," in *ACM SOCC*, 2013.

[16] J. Sherry *et al.*, "Rollback-recovery for middleboxes," in *ACM SIGCOMM*, 2015.

[17] M. D. Da Silva and H. L. Tavares, *Redis Essentials*. Packt Publishing, 2015.

[18] Intel, "Guide: Data plane development kit for linux," Guide, April 2015.

[19] S. Behara, "Sidecar design pattern in your microservices ecosystem," dotnetvibes, 2018. [Online]. Available: https://dotnetvibes.com/2018/07/23/sidecar-design-pattern-in-your-microservices-ecosystem

[20] BESS Comitters, "BESS (Berkeley Extensible Software Switch)," 2018. [Online]. Available: https://github.com/NetSys/bess