Elastic Scaling of Stateful Network Functions

Shinae Woo, ***KAIST, UC Berkeley;*** **Justine Sherry,** ***CMU;*** **Sangjin Han,** ***UC Berkeley;***
Sue Moon, ***KAIST;*** **Sylvia Ratnasamy,** ***UC Berkeley;*** **Scott Shenker,** ***UC Berkeley, ICSI***

https://www.usenix.org/conference/nsdi18/presentation/woo

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-939133-01-4

Open access to the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

Elastic Scaling of Stateful Network Functions

Shinae Woo[⋆†], *Justine Sherry*[‡], *Sangjin Han*[⋆], *Sue Moon*[†], *Sylvia Ratnasamy*[⋆], *and Scott Shenker*[⋆§]

[⋆]*University of California, Berkeley* [†]*KAIST* [‡]*CMU* [§]*ICSI*

Abstract

Elastic scaling is a central promise of NFV but has been hard to realize in practice. The difficulty arises because most Network Functions (NFs) are *stateful* and this state need to be *shared* across NF instances. Implementing state sharing while meeting the throughput and latency requirements placed on NFs is challenging and, to date, no solution exists that meets NFV's performance goals for the full spectrum of NFs.

S6 is a new framework that supports elastic scaling of NFs without compromising performance. Its design builds on the insight that a distributed shared state abstraction is well-suited to the NFV context. We organize state as a distributed shared object (DSO) space and extend the DSO concept with techniques designed to meet the need for elasticity and high-performance in NFV workloads. S6 simplifies development: NF writers program with no awareness of how state is distributed and shared. Instead, S6 transparently migrates state and handles accesses to shared state. In our evaluation, compared to recent solutions for dynamic scaling of NFs, S6 improves performance by 100x during scaling events [25], and by 2-5x under normal operation [27].

1 Introduction

The Network Function Virtualization (NFV) [13] vision advocates moving middlebox functionality – called Network Functions (NFs) – from dedicated hardware devices to software applications that run in VMs or containers on shared server hardware. An important benefit of the NFV vision is elastic scaling — the ability to increase or decrease the number of VMs/containers currently devoted to a particular NF, in response to changes in offered load. However, realizing such elastic scaling has proven challenging and solutions to date come with a significant cost to performance, functionality, and/or ease of development (§3).

The difficulty arises in that most NFs are *stateful*, with state that may be read or updated very frequently (e.g., per-packet or per-flow). Hence, elastic scaling requires more than simply spinning up another VM/container and updating a load-balancer to send some portion of the traffic to it.

Instead, scaling can involve *migrating* state across NF instances. Migration is important for high performance (as it avoids remote state accesses) but its implementation must be fast (to avoid long "pause times" during scaling events) and should not be burdensome to NF developers. In addition, elastic scaling must ensure *affinity* between packets and their state (*i.e.*, that a packet is directed to the NF instance that holds the state necessary to process that packet), and such affinity must be correctly enforced even in the face of state migrations. A final complication is that some types of state are not partitionable, but *shared* across instances (see §2 for examples). In such cases, elastic scaling must support access to shared state in a manner that ensures the consistency requirements of that state are met, and with minimal disruption to NF throughput and latency.

The core of any elastic scaling solution is how state is organized and abstracted to NF applications. Recent work has explored different options in this regard. Some [33] assume that all state is local, but neither shared or migrated – we call this the *local-only* approach. Others [25,37] support a richer model in which state is exposed to NF developers as either local or remote, and developers can migrate state from remote to local storage, or explicitly access remote state – we call this the *local+remote* approach. Still others [27] assume that *all* state is remote, stored in a centralized store – we call this the *remote-only* approach.

The above were pioneering efforts in exploring the design space for NF state management. But, as we elaborate on in §3, they still fall short of an ideal solution: the *local-only* approach achieves high performance but is limited in the NF functionality that it supports; the *local+remote* approach supports arbitrary NF functionality but complicates NF development and incurs long downtimes from repartitioning state en bloc during scaling events; the *remote-only* approach is elegant but imposes high performance overheads even under normal operation.

In this paper, we propose a new approach to elastic scaling in which state is organized as a *distributed shared object* (DSO) space: objects encapsulate NF state and live in a global namespace, where all NF instances can read/write any object. While DSO is an old idea, it has not to our knowledge been applied to the NFV context. In particular, DSO has not been shown to the meet the elasticity and performance requirements that NFV imposes.

We present S6, a development and runtime framework tailored to NFV. To meet the needs of NFV workloads, S6 extends the DSO concept as follows: (1) for space elasticity, we introduce dynamic reorganization of the DSO keyspace; (2) to minimize the downtime associated with scaling events, we introduce a "smart but lazy" state reorganization; (3) to reduce remote access overheads, we introduce per-packet microthreads and; (4) to optimize

performance without burdening developers, we expose per-object hints via which the developer can inform the DSO framework about appropriate migration or caching policies. S6 hides all those internal complexities of distributed state management under the hood, simplifying NF development.

We present three elastic NFs implemented on top of S6: NAT, PRADS (a network monitoring system [6]), and a subset of the Snort IDS [8]. We show that NFs on S6 elastically scale with minimal performance overhead, and compare them to NFs built using prior approaches. A local-only system like E2 [33] cannot support two of our use-cases (NAT and PRADS) because it does not support shared state. Compared to OpenNF [25], a state-of-the-art framework based on the local+remote approach, S6 achieves 10x - 100x lower latency while sustaining 10x higher throughput during scaling events. Compared to StatelessNF [27], a state-of-the-art framework based on a remote-only approach, S6 achieves 2x - 5x higher throughput under normal operation.

2 Background: NF State Abstractions

The difficulty of elastic scaling arises in how to handle NF state appropriately. NFs keep state about ongoing connections (*e.g.*, TCP connection state, last activity time, number of bytes per user-device, a list of protocols used at a given IP address). NFs read and update a state while processing a packet and reuse the updated state to process subsequent packets. Stateful NF instances should maintain correct NF state collectively to prevent the inconsistent or incorrect behavior. Also, the system must handle general forms of state sharing among instances, as we will present in this Section. Such NFs' behavior requires significantly more care than merely spinning up another NF instance and sending some portion of traffic to it.

We categorize NF state based on whether it is *partitionable*. We say that state is partitionable if it can be distributed across NF instances in a way that state is only locally accessed, assuming a certain traffic load balancing scheme. For example, per-flow state (such as state for individual TCP connections) is partitionable, if traffic is distributed on a flow basis. On the other hand, a counter for the total number of active flows is an example of non-partitionable state, since all NF instances need to update the counter.

Whether state is partitionable or not is important since it determines both the mechanisms needed to manage that state and the achievable performance levels. With partitionable state, we can collocate state with the NF instance that processes it, and hence efficient state migration is key to achieving high performance. If state is not partitionable, high performance requires a different set of techniques: *e.g.*, caching state (when its consistency semantics allows it), placing state to minimize remote accesses, and minimizing the cost of remote state access.

Apps \ State	Partitionable	Non-partitionable
NAT	-	Address mapping entry Available address pool
Firewall	Connection context	-
Load balancer [11, 21]	Connection - server mapping	Server pool usage statistics
Traffic Monitoring [6]	Connection context	Per-host context; Statistics for packets, used protocols and host
IDS/IPS [8, 10, 34]	Connection context	A set of certificates, malicious servers, or infected hosts; Per-host port scanning counter
Web proxy [9]	Connection context	Statistics for cached entry
EPC [3]	User state	SLA/Usage per device/plan
IMS [5]	SIP / RTP sessions	Usage accounting per user

Table 1: Examples of state in popular NF types

Most non-partitionable NF state also provide opportunities for efficient sharing. We can categorize state by whether it is updated mostly by a single or multiple instances. From our observation, single-writer state tends to be read-heavy, thus caching or replication can be effective. When the state is updated by multiple writers simultaneously (*e.g.*, global counters), looser consistency is often tolerable so as to trade freshness of data for performance.

Table 1 lists examples of partitionable and non-partitionable state found in some real-world NFs. We see that both forms of state are common in real-world NFs. For example, traffic monitoring systems [6] maintain state at both the connection (partitionable) and the host (non-partitionable[1]) levels. We also note that state variables, whether partitionable or not, typically relate to each other forming complex data structures. For example, traffic monitoring systems manage a global table of hosts, each referencing a list of its active connections.

3 State Management for Elastic Scaling

State management for elastic scaling of stateful NFs involves many design options, such as where to place state and when to initiate migration. They all affect the overall NF performance, in terms of throughput and latency. Existing approaches cause high performance overhead, either during scaling events (*i.e.*, instances join and leave) or under normal operations (*i.e.*, no ongoing scaling events). In this section, we discuss the limitations of their approaches in §3.1 and propose our new approach in §3.2.

3.1 Limitations of existing approaches

Figure 1 shows the typical components of an NFV architecture as assumed by prior research [27, 27, 33, 37] and industry efforts [13]. An NFV controller [24, 33] manages NF instances that run on servers, while an SDN

[1] No load balancing scheme can ensure data locality of state for both source and destination hosts at the same time.

controller manages the network fabric between servers, including how traffic is load-balanced across the different NF instances. Responsibility of NF state management—initiating migration, placement, etc.—resides in the NF controller. We now discuss existing approaches to NF state management.

Local-only state: Early work on NFV management [22, 24, 33, 35, 38] typically assumed that NF state is partitionable and hence they do not address the issue of shared state. Some (*e.g.*, E2 [33]) address elastic scaling but for NFs with per-flow state only. In such cases, state migration is avoided by directing only new flows to new NF instances. Thus, these early systems do not accommodate NFs with shared or more general (*i.e.*, beyond per-flow) state, which is common in practice (Table 1). Hence, we will not consider solutions based on a "local-only" model further in this paper.[2]

Remote-only state: In this model, all NF state is kept in a standalone centralized store. First proposed by Kablan *et al.* [27], the elegance of this model is that support for state sharing, consistency management, and durability (of state) moves to the centralized store, while the NF instances themselves are stateless and hence can be easily added or removed. StatelessNF [27] uses this approach to build various NFs such as a NAT, Firewall and TCP reassembler.

Unfortunately, this approach comes at the cost of performance. In statelessNF, *all* state accesses are remote: not only do these remote accesses inflate packet latency, they also consume extra CPU cycles and network bandwidth for I/O to the remote store. Our results in §7.2 show that, relative to an NF that uses local state, a remote-only approach can lead to a 2-3x degradation in throughput and a 100-fold increase in packet latency. Problematically, these overheads are incurred even under normal operation (*i.e.*, in the absence of scaling events) and grow with the number of state accesses and the size of messages. Recent work reports that such state accesses are frequent in NFs: *e.g.*, StateAnalyzr [28] reports that typical NFs maintain 10-100s of state variables that are per-flow or shared across flows.

Local+Remote In this model, state is distributed across NFs and exposed to NF developers as either local or remote. All NF state is defined (by the developer) to be either local or remote, and is accessed accordingly. OpenNF [25] and SplitMerge [37] adopts this model. For high performance, partitionable state is typically defined as local state (similar to local-only NFs) while non-partitionable state requires explicit push/pull function calls to synchronize with remote state.

[2]We note that systems such as E2 could be augmented with the state management capabilities that we and others [25, 27, 37] propose.

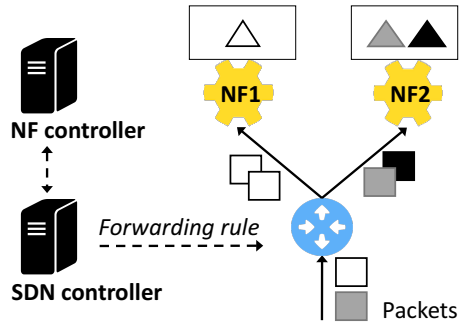


Figure 1: Typical components of an NFV architecture

In this model, state management plays two roles: (1) implementing access to remote state, (2) migrating state upon scaling events so as to not break local memory accesses. This functionality is implemented by a state management framework (such as OpenNF) working in concert with the NFs and SDN controllers. Under normal operation, the local+remote approach has the potential to achieve throughput and latency comparable to the previous models by migrating state to be co-located with the NF instances that access it. Unfortunately, the overhead *during* scaling events is high in this model. The reason stems from the fact that state is explicitly defined and accessed as local or remote. When a new NF instance is launched, all state that may be accessed as local state at the new NF instance must be migrated over to it *before* any access occurs (since otherwise the local access would simply fail).

Scaling events thus result in "stop the world" behavior, which involves the following steps: first, the SDN controller buffers traffic destined for both the old and new instance, by rerouting traffic from the fabric/load-balancer to itself; next, the state management controller coordinates the migration of relevant state from the old to new NF instance; once migration completes, the SDN controller releases buffered traffic and coordinates with the fabric/load-balancer to turn off detouring traffic to the SDN controller. This approach can lead to long pause times during which both old and new NF instances stop processing packets while state is repartitioned. This is also complex to implement due to tight coordination among the SDN controller, NF controller, and the inline switches/load-balancer.[3]

As we show in §7.1, the local+remote approach lead to very long pause times for practical NFs. For example, PRADS implemented on OpenNF incurs a pause time of **490 ms** when migrating only 1,500 flows despite extensive optimization to the process. In practice, the pause time is likely to be even higher considering that a typical 10 Gbps link has tens of thousands of concurrent flows [43].

[3]A subtle additional challenge is that these components often come from different vendors, complicating the adoption of such techniques.

3.2 Our approach: Distributed shared state

The limitations of the aforementioned approaches lead us to consider a new approach, the distributed shared state model, which is familiar from distributed computing. Here, state is distributed among the NFs, and can be accessed by any NF. However, the NF developer makes no distinction between local vs. remote state. Instead, all state variables reside in a shared address space and the state management framework transparently resolves all state access. The framework is also responsible for deciding where state is placed and migrating state across NF instances when appropriate.

Done right, this model can achieve throughput and latency comparable to the local-only model by migrating state to be co-located with the NF instances that access it. This model can also avoid the long pause-times incurred by the local+remote approach. Because there is no distinction between local and remote state, no proactive migration is required during scaling events. The overhead of migration is gradually amortized as packets arrive; *e.g.*, for the same PRADS scenario above, the pause-time can drop to under 1 millisecond (§7). For the same reason, state migration no longer needs tight coordination with traffic load-balancing, hence reducing the system complexity associated with the local+remote model. Finally, the distributed shared state model simplifies NF development. Developers can write NFs on top of a uniform interface for state access, local or remote, outsourcing the underlying details of state lookup, remote access, and migration to the state management framework.

To the best of our knowledge, we are the first to apply distributed shared state to NFV. We highlight two challenges distinct from other application domains. NFs have distinct performance requirements from traditional cloud applications [18, 31]. Furthermore, elasticity makes it more difficult, dynamically reorganizing the structure of state space into the new set of NF instances.

Achieving high performance: NFs have I/O intensive workload, requiring very high throughput on the order of millions of packets/s with sub-millisecond latency. Given these requirements, only a few hundreds or thousands of CPU cycles are available for every packet.

The key to achieving high performance is twofold. Firstly, we should reduce the number of remote accesses by leveraging the state-instance affinity and supporting efficient sharing. As each type of NF state has different access patterns and consistency requirements [28], the question is how to leverage the information while minimizing developer's burden. Secondly, we need to minimize the cost of remote access when it is unavoidable. While we can hide the latency by processing other packets in the meantime, it must be done so without increasing programming complexity. The framework should be able to handle data dependency detection and context stashing [15, 42].

Supporting elastic scaling: As explained above, scaling events at runtime must not incur significant performance degradation. Membership change in NF instance group involves two potential sources of service disruption. First, as input traffic is distributed across the new set of NF instances, a subset of state variables must migrate to maintain locality. Second, in addition to the state variables themselves, their location metadata must be reorganized as well, for scalability of the shared state space. The challenge is how to perform these operations in a distributed fashion, in order to avoid a single point of performance bottleneck. Furthermore, the framework must ensure consistent state access during the process, while minimizing delay in packet processing.

4 S6 Design

S6 is a development and runtime framework for elastic scaling of NFs. S6 makes the following assumptions, which are general enough to apply to a wide variety of deployment scenarios and environments. First, an NF runs as a cluster of virtualized instances, such as VMs and containers. Second, the network somehow distributes input traffic across instances. Lastly, an external NFV controller/orchestrator triggers scaling events to adapt to load change.

S6 does not demand any particular network load balancing mechanism or NFV controller behavior for correctness. Therefore they are out of the scope of this paper. One desirable property is that input traffic be distributed across instances on a flow basis as like most of load-balancers and switches already are doing, so that S6 can leverage the state-instance affinity for high performance. S6 differs from the existing NF state management solutions [25, 37], all of which require sophisticated runtime coordination across NFV controller, SDN controller, and NF instances. S6's decoupling from the load-balancer and SDN controller reduces system complexity.

We summarize the main design components: 1) S6 provides the global DSO shared by all NF instances. We choose 'object' as a basic unit of state. An object encapsulates a set of data and its associated operations, allowing access control and integrity protection of state. All objects in the space are accessible with a uniform API, regardless of where the objects physically reside. 2) Our object abstractions provide NF developers with knobs to specify object access patterns. The S6 framework uses this information to improve performance by reducing the number of remote state accesses. 3) When remote state access is inevitable, S6 mitigates its cost by hiding latency with microthreads; NF worker instances can keep processing other flows only if they have no data dependency on outstanding accesses to remote objects.

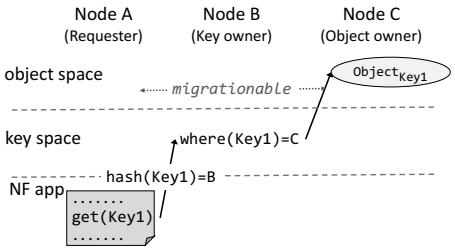


Figure 2: DHT-based Distributed Shared Object Space

4) Upon scaling events, S6 reorganizes the space, while keeping the workers processing traffic. S6 minimizes service disruption with the smart but lazy migration of objects and their metadata. We explain each component in greater detail below.

4.1 DHT-based DSO space architecture

In the DSO space, state objects are uniquely identified with a *key*. Keys can be of any type, such as 5-tuples, host names, and URLs as necessary. When an NF instance requests an object with some key (*e.g.*, extracted from packets), S6 returns a *reference* to the object, rather than the object binary itself. With the reference, the instance can read or update the state by invoking object methods. S6 constructs the DSO space as a DHT-based two-layer structure (Figure 2) of *a key layer* and *an object layer* [40]. Both layers are distributed over NF instances. The key layer keeps track of the current location of every object, rather than directly storing objects. This layer of indirection offers great flexibility in object migration; no broadcast is necessary to locate an object, although it may reside (if it exists) on any instance at the moment. The object layer stores the actual binary of objects. A reference to an object guarantees accessibility, no matter where the object currently is.

When an instance accesses an object for the first time, it hashes the key to identify the *key owner*, the instance who knows the current object location, the *object owner*. The object access request is sent to the key owner first, then the key owner forwards the request to the object owner. Once the location of the object is resolved, the instance caches it so that subsequent object requests can go directly to the object owner. When an object migrates to another instance, the key owner must be notified. The key owner updates the location of the object and invalidates the cached location in workers.

The key owner takes charge of object creation, deletion, and its reference creation; Those requests are serialized and sequentially processed at the key owner. Once the key owner receives object deletion request, it rejects subsequence object access requests until new object creation request comes.

Note that this two-layer structure is only internally managed. S6 hides the complexity of placing and locating objects from NF developers, so that they can focus on the application logic itself.

4.2 Object abstractions: Per-object optimization

We provide an object abstraction that allows developers to hint to the framework about what caching, migration, and optimization strategies are appropriate for each object. Different objects hence have different consistency guarantees depending on their usage. While state management is a generic problem in distributed systems, we focus on NFs' distinct state characteristics and access patterns that we can leverage to achieve good performance.

We first categorize object types into two types depending on whether the object permits updates from multiple flows (thus multiple instances): partitionable objects and non-partitionable objects. Based on the different characteristics for each state type in §2, we introduce appropriate optimization strategies for each object type. APIs in detail and usage examples are in covered in §5.

Partitionable: leveraging state-instance affinity Partitionable objects are primarily used for state that is updated by a single flow; up to one writable reference to the object is allowed. If an instance is holding the writable reference, other instances have to wait for their turn to acquire a writable reference. This is enforced at the key owner since it is a natural serialization point for all reference requests to the object.

In NF contexts, while partitionable objects have high affinity on a single instance, but occasionally, its state affinity may move to other instances. For example, per-flow objects' affinity is decided based on the traffic load-balancing policy of the network, which is not controlled by S6. As NFs frequently update partitionable state, often on a per-packet basis, keeping high state-instance affinity is the key to achieving high performance for partitionable objects.

Partitionable objects are gradually migrated between instances when affinity changes. S6 uses a new request for writeable access from other instances as an affinity change indicator. When a key owner for a partitionable object gets an object access request other than the current object owner, it initiates the object migration process. The current object owner voluntarily releases the reference when the local reference count for the object reaches to zero, then the object is transferred, and the instance becomes a new object owner. Now in the new object owner, all accesses to the object locally happens as the reference points to the object binary in the memory.

Non-partitionable: consistency/performance trading Non-partitionable objects are concurrently accessed from multiple flows simultaneously; multiple writable references to an object may exist. Supporting shared state with high performance in distributed systems is generally difficult or impossible to achieve—if an object is very frequently updated by multiple flows in a strongly-consistent manner, it does not scale and S6 cannot help it. Fortu-

nately, we found that the majority of non-partitionable state in NF applications does not require frequent updates (*e.g.*, one update per flow) or allow trading consistency for performance, as shown in Table 1. S6 provides three optimization mechanisms for non-partitionable objects that NF developers can leverage.

First, S6 supports object access via method call shipping, rather than migrating objects. This prevents objects from bouncing between instances, to avoid wasting in-flight time. The method calls are applied serially in the order of request arrival at the object owner, preserving the internal object consistency. For such non-partitionable objects, NF developers need to design the objects to be commutative, *i.e.*, any order of methods calls should produce an acceptable result. Additionally, S6 supports *untethered update*, which allows remote object update without blocking if there is no need to wait for its completion.

Second, S6 supports abstractions to design objects to enable trade-off between consistency and performance. If a read method on an object class is tolerant of *stale* results, instances can cache the results of the method locally. NF developers can bound the staleness for each read method, so that S6 can periodically refresh the cached results with newer ones.

Third, S6 supports object replication, so that multiple instances can update their local replica. Those replicas are regularly *merged* to the main object at the object owner. NF developers are expected to provide this merge function since it is very object-specific. Shared counters are a prime use case of this local update and merge. Frequent updates are done locally, while (infrequent) reads on the counter cause local numbers to be merged globally.

4.3 Microthreads: Hiding the cost of remote access

Even with above optimizations for objects, blocking remote access is necessary when waiting for migrating objects, refreshing the cache, or dealing with objects with strong consistency. The cost of remote state access is high. Suppose that an NF instance issues an RPC request and waits for its response, in order to process a packet. Assuming 10us round-trip time between NF instances, the latency translates to 30,000 cycles on a 3 GHz processor. We can hide this latency with concurrency; the NF can process other pending packets to keep the CPU busy, as long as they do not have data dependency on the RPC or introduce packet reordering in a flow. Once its response arrives, the NF continues processing the packet(s) that were blocked on it.

We adopt a multi-threaded architecture in favor of ease of NF development to maintain execution contexts of blocked flows. The other option was an event-driven architecture, but it hurts programmability since developers must manually manage to save and restore contexts [15, 42] for every state access. Another issue is that whether a method call would block or not must be visible to the NF developers, which adds additional complexity to the application logic. In contrast, with multi-threaded architecture, developers can program packet processing easily while all thread scheduling is automatically done by the S6 runtime.

To minimize the performance overhead of multi-threading, S6 utilizes cooperative, user-space "microthreads". User-level microthreads are much more lightweight than kernel threads, since non-preemptive scheduling is significantly simpler, and context switching does not involve kernel/user boundary crossing. It also scales up to millions (not thousands) of microthreads thanks for their small footprint.

S6 manages a pool of microthreads to avoid thread construction/destruction cost. A microthread runs for each received packet. Whenever the thread is about to block (*e.g.*, an object is remote and/or in migration, cache entry is being refreshed, data dependency is detected as another microthreads is holding a reference, etc.), the microthread yields to other pending threads and wait to be rescheduled after the blocking condition has been resolved. This non-preemptive scheduling is automatically done by S6 and transparent to the NF developer. When multiple microthreads are ready to resume, S6 schedules one with the longest wait time to avoid packet reordering within a flow and to minimize latency jitter.

4.4 Smart but lazy DSO keyspace reorganization

When the membership of NF instances changes—due to scaling events or node failures—S6 must reorganize the DSO space for the new set of instances. This reorganization involves both object space and key space. As we illustrated in §4.2, the object space is repartitioned automatically and gradually for new state-instance affinity, as NF instances access state objects. Assuming reference locality—most state access is done to a small number of objects—frequently accessed objects are quickly migrated to new object owners, incurring minimal performance impact.

On the other hand, like the object space, S6 ensures that the key space reorganization is also done gradually so as to minimize performance impact. Suppose that we reorganize the DSO key space from S_i to S_{i+1}, which use $h_i(key)$ and $h_{i+1}(key)$ as lookup hashes for finding key owner respectively. Reorganization must not break the coherency of the keyspace, such that any key record is neither lost nor owned by multiple key owners. At the same time, we do not want to pause the entire system for coherency; instead NF instances lazily migrate key ownership from $h_i(k)$ to $h_{i+i}(k)$ in the background as necessary. Our keyspace reorganization algorithm ensures coherency even in the middle of scaling process.

When the scaling process starts, new key access requests go to $h_{i+i}(k)$. The new owner $h_{i+1}(k)$ check if the previous owner $h_i(k)$ has a record for k, and if so, the new key owner pulls the record. During the scaling process, every new key lookup requires two-hop routing. After the keyspace converges to S_{i+1} (all key record migration is completed), key lookups can be done with the normal one-hop routing again.

Dealing with race conditions: One challenge comes from the fact that we cannot assume that all nodes start and finish the scaling process exactly at the same time. For example, if two nodes A (previous owner of k, $h_i(k)$) have not been notified the scaling process and run in 'normal' operation but B (next owner of k, $h_{i+1}(k)$) start 'scaling' operation, both two nodes would claim the ownership for k. This corner case may result in two key records for k created in both node A and B.

To prevent such conflict, S6 performs scaling in two stages: *pre-scaling* and *scaling* stages. The workers transition to scaling stage only if the controller has confirmed that all workers are in pre-scaling state. This barrier ensures that nodes in the 'normal' and nodes in the 'scaling' stage do not coexist. Nodes in pre-scaling stage do not actively transfer key ownership yet, while being aware that other nodes may be in scaling process. Ensuring that there is always a single key owner exists, but not two or none, is done with the following rules:

R1 Preventing double ownership *Suppose that node A (prev owner, $h_i(k)$) is in 'pre-scaling' and node B (next owner, $h_{i+1}(k)$) is in 'scaling'. In this case, $h_i(k)$ should be the single owner for k.*

Since pre-scaling nodes can coexist with nodes still in 'normal' stage, node A should serve $h_i(k)$. Meanwhile, node B $h_{i+i}(k)$ have more contexts about scaling process than A. Until node A goes into 'scaling' stage, it defers claiming the ownership for k, but keeps forwarding requests to A.

R2 Preventing lost ownership *Suppose that node A (prev owner, $h_i(k)$) is in 'scaling' and node B (next owner, $h_{i+1}(k)$) is in 'pre-scaling'. If k is for a new object, $h_{i+1}(k)$ should be the single owner. If k is for an existing object, $h_i(k)$ should be the single owner.*

In this case, no one claim the ownership of k, and the two node forward requests on k to each other. We need to prevent such loop. Let's assume that A receives a request on k. If k is for existing objects and A owns the key since B hasn't claimed the ownership. A keeps serving the requests on k, until B claims ownership of k. If k is for new objects, then B doesn't have any information of k. Therefore B would forward the request to A. A potential loop is prevented by attaching version number to the forwarded requests.

Category	API	Description
Object	`SingleWriter`	Exclusive writeable
	`MultiWriter`	Concurrent writeable
Method	`const stale`	Cached read
	`untethered`	Untethered update
	`merge(Object&)`	Merge two objects
Data Structure	`S6Map<Key, Object>`	Define a map in DSO
	`S6Ref<Object>`	Reference to an object
	`S6Iter<Object>`	Iterator of collections
S6Map (DSO)	`create(Key&,Flag&)`	Create an object
	`get(Key&)`	Retrieve an object
	`remove(S6Ref<>&)`	Remove an object

Table 2: S6 Programming API

State Type	Examples	Object Annotation	Method Annotation
Partitioned	UDP/TCP connection state	`SingleWriter`	-
Non-partitioned freq update	Performance statistics	`MultiWriter`	`untethered` `stale` `merge`
Non-partitioned read-heavy	NAT mapping entry	`SingleWriter`	`stale`
Collection of multi-type state objects	linked-list hashtable	Non-intrusive data structures (§8.1)	

Table 3: Common types of NF state and their annotations

5 Using S6

We introduce our programming model (§5.1) and provide some examples of various NFs (§ 5.2).

5.1 S6 Programming model

Table 2 summarizes the S6 API. From a user's perspective, S6's core components are the *shared object space* and *tasks*.

We provide two types of objects depending on whether the object permits update from multiple writers (NF instances). `SingleWriter` allows exclusive writes from a single instance. `MultiWriter` allows concurrent writes from multiple instances simultaneously. Methods on objects can be annotated appropriately to allows more optimization such as cached read (`const stale`), update-and-forget (`untethered`), or regularly pushing merged local updates (`merge`) into the object owner. Then, S6 supports appropriate optimization on behind based on object type as explained in previous section §4.2. Figure 3 shows an example implementation of an object class used in PRADS [6]. It is exactly same as normal object oriented design only except the additional annotations we introduce. In fact, from our survey of popular NFs in Table 1, we found that most of NF state falls into one of four types shown in Table 3.

S6 provides two types of tasks: data-plane and control-plane. *Data-plane tasks* perform packet processing on input network traffic. *Control-plane tasks* perform out-of-band operations, such as updating configurations

```
class HostAsset: public MultiWriter {
  public:
    void update_service(Service s) untether;
    void update_os(OS os) untether;
    uint64_t first_detect_time() const stale;
    uint64_t last_detect_time() const stale;
    void merge(HostAsset local);
  private:
    addr_t ip;
    uint64_t first_detect_time;
    uint64_t last_detect_time;
    List<Service> service_list;
    List<OS> os_list;
};
```

Figure 3: A sample S6 object definition of PRADS's per-host network asset object

```
S6Map<IPKey, Asset> g_asset;
S6Map<FlowKey, Connection> g_conn;

// data-plane task
FlowKey fkey(sip, dip, sport, dport);
S6Ref<Connection> c = g_conn.create(fkey);
    ...
if (new_os_asset) {
    S6Ref<Asset> asset = g_asset.get(sip);
    asset->update_os_asset(new_os_asset);
}
...

// control-plane task
S6Iter<Asset> *it = g_asset.get_iterator();
while (it->next())
    log_asset(it->key,  it->value);
```

Figure 4: A sample implementation of PRADS tasks

or processing user queries. Both types of tasks have access to the shared object space with a uniform interface. Figure 4 shows an application implementation including one data-plane task for packet processing and one control-plane task for logging the per-host assets.

5.2 Programming NFs

5.2.1 Sample applications

We have chosen various applications to implement or port. Table 4 lists the state objects in those NFs.

Network Monitoring System (PRADS) PRADS [6] is a *Passive Real-time Asset Detection System* in Linux. It allows network administrators to access real-time data on types of protocols, services, and devices on their network.

Intrusion Detection System (Snort-rule) We implement IDS which monitors packets using Snort [8] rules. We borrow the rule compilation and detection code from the original Snort code base.

NAT We implement NAT (Network Address Translator) by following the algorithm described in statelessNF [27], so that have the same per-packet/per-flow access patterns with their implementation.

NF	State	Size (B)*	Update	Access Frequency
PRADS	Flow	160	Exclusive	Per-packet RW
	Statistics	208	Concurrent	Per-packet RW
	Asset	$112+64n$	Concurrent	Rarely R Per-packet W
	Hashtable of flows	$40n$	Concurrent	Per-flow RW
	Hashtable of assets	$32n$	Concurrent	Per-flow RW
IDS	Flow context	$160 \sim 32$k	Exclusive	Per-packet RW
	Whitelisted host	16	Exclusive	Per-packet RW
	Malicious server	$12+28n$	Concurrent	Per-flow RW
	Hash table of Malicious server	$32n$	Concurrent	Per-flow RW
	Hash table of whitelisted host	$32n$	Concurrent	Per-flow RW
NAT	Address Pool	8k per IP	Exclusive	Per-flow RW
	NAT entry	8	Exclusive	Per-packet R Per-flow W

* n is the number of elements in the structure.

Table 4: States, update patterns, and access frequencies of NF applications we use.

5.2.2 Experiences of porting NF applications

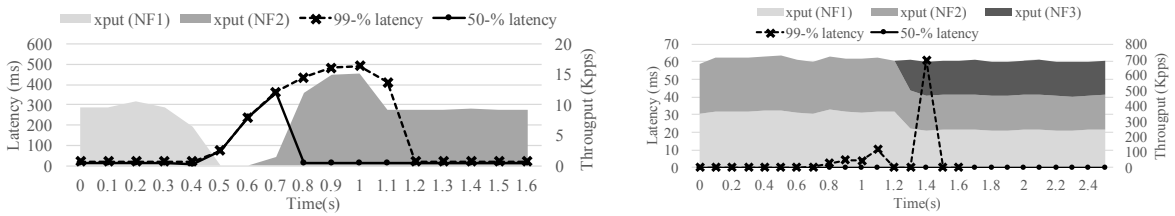
We begin with the assumption that the NF application to port is in an OOP (Object-Oriented Programming) model. Since the baseline code of PRADS is in C, a non-OOP language, our first step is to convert `structs` to C++ objects. Then we start porting these objects in our S6 programming interface.

Porting States Objects: To convert the existing object classes to S6-compatible object classes, we need to (1) identify globally accessible objects, (2) analyze their update patterns, and (3) check the applicability of loose consistency.

In Table 4 we list the states we have identified to be globally accessible and their update patterns: four simple objects (Flow, Statistics, Asset, and Configuration) and two collections of objects (Flow hashtable and Asset hashtable). After identifying the simple objects, we decide their types as `SingleWriter` or `MultiWriter` according to the update pattern. We use `S6Map` to support hash tables for flows and assets, and `S6Iter` to iterate through the list of assets. In case of more complex applications, StateAlyzr [28] can help identifying state variables which need to be shared and their update patterns.

Now the application is compatible with S6 and should run correctly. Next, we turn to performance improvement by loosening the consistency level on objects. We design the Asset and Statistics objects to be commutative, and all their reads as cached reads and all updates as untethered.

Porting Tasks: PRADS has a simple loop processing packets using `libpcap` which is straightforward to port to S6's data plane task. PRADS has other out-of-band tasks from network administrators like generating a log of current assets. We implement these out-of-band operations as control-plane tasks.



(a) OpenNF, 10 kpps, 1.5k flows for migration

(b) S6, 700 kpps, 8k flows for migration

Figure 5: Performance comparison with PRADS scale-out events

6 Implementation

Our implementation of S6 has three main components: the S6 runtime, the S6 compiler, and the NFV controller.

Runtime The S6 runtime plays three roles. First, it manages the DSO space distributed across nodes. It tracks objects location and controls object accesses from multiple instances to support exclusive or concurrent accesses per object type. Second, it manages S6-compatible object references to provide the S6 programming interface. The S6 runtime intermediates every access to objects and performs remote operations or initiate object migration if necessary. Third, it schedules microthreads for data-plane and control-plane tasks. Whenever a microthread is about to block due to remote access, the runtime schedules another pending microthread and continues to process packets without blocking. We use the boost co-routine library [12] to implement non-preemptive, user-level multi-threads.

S6 Compiler While S6 requires a custom programming interface in the source code, there is no convenient way to extend C++ syntax. Instead, we implemented a source-to-source compiler, which translates S6-extended C++ code into plain C++. The generated code abstracts away implementation details of DSO implementation under the hood. For example, when a method is invoked, the generated code checks if the state object is local or remote to call appropriate functions. We implement the compiler on top of clang-3.6 library [14] to perform syntax analysis of a developer's code.

NFV Controller We built a simple NFV controller to manage the S6 instances. It runs an NF cluster by launching S6 instances and initiates scaling-in/out events based on network workloads. The controller also relays out-of-band tasks such as queries or updating configuration from the operators to NF instances.

7 Evaluation

We start our evaluation of S6 with its application-level performance with the scale-out NFs we ported in §5.2. We examine how scaling events impact S6 performance during scaling events in §7.1, and under normal operation in §7.2. Then we show the effectiveness of design choices in S6 with a series of micro-benchmarks in §7.3

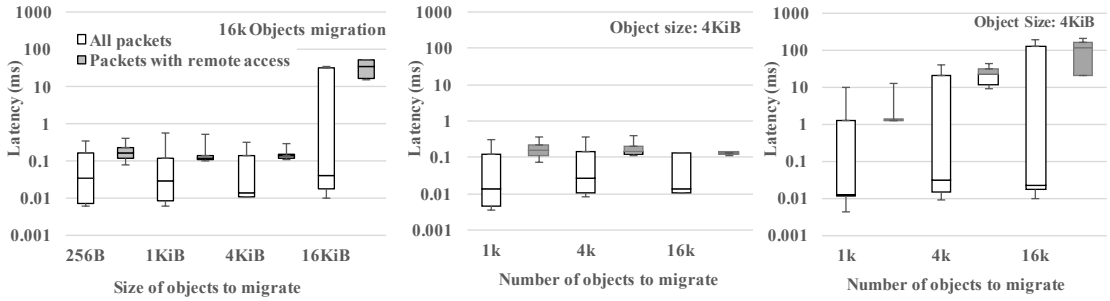
Evaluation setup We use Amazon EC2 c4.xlarge instances (4 cores @ 2.90GHz) for experiments. NF instances run as a Docker container, across the virtual machines in the cluster. Our workload is synthetic TCP traffic based on empirical flow distributions in size and arrival rates. For all experiments shown, we measure the overall throughput and latency measured at input/output ports of each NF. For micro-benchmarks, we use a dedicated Intel Xeon E5-2670 (2×8 cores @ 2.30GHz) server, with a 10 G link for data channel and another 10 G for state channel for inter-instance communication.

7.1 Elastic Scaling

How well S6 performs during scaling events? We compare S6's scaling-out performance with OpenNF using PRADS on each framework. Figure 5(a) shows the throughput and latency of migrating 1.5k flows at 10 kpps workloads using OpenNF. Even with the highest optimization level OpenNF supports, the throughput drops and the latency increases up to hundreds of milliseconds. Not shown, we tested the exact same workload with S6. S6 shows no visible throughput fluctuation and only a few hundreds of *microseconds* increase in latency.

Figure 5(b) shows PRADS scale-out performance on S6 *with a higher input workload*, 700 kpps with 8k concurrent flows. There is not any noticeable throughput degradation (and also zero packet loss). The state channel becomes temporarily congested from object migration, key-space re-partitioning in addition to the shared variable accesses. Still, the peak latency around tens of milliseconds during scaling is transient—within a 0.1 second window—and 10x lower while sustaining 10x higher throughput than OpenNF.

How does workload affect performance during scaling events? We now consider S6 scaling for a synthetic NF, in which we can configure the number of state objects and their size. We send 1 Mpps network load to a single NF instance. Then we initiate a scale-out event, launching another instance and split the traffic. As a result, half of the objects in the original instance move to the new instance as packets arrive. We vary the number of objects and the object size and measure the end-to-end packet processing latency.



(a) Empirical traffic / object size (b) Empirical traffic / object space size (c) Worst-case traffic

Figure 6: Latency (1-25-50-75-99%-iles) during object re-partitioning

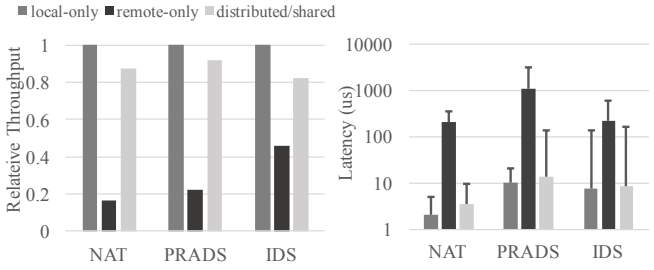


Figure 7: Performance of NFs implemented on different NF state management abstractions

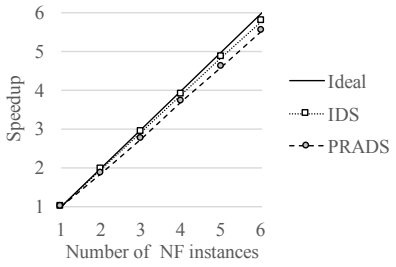


Figure 8: S6 throughput scalability

Figures 6 (a) and (b) show latency during the scaling process. The scaling process takes 100s-1,000s of milliseconds depending on workloads. As shown in the graph, the peak latency is sub-millisecond except for 16 KiB (a) objects, as the state channel to transfer objects among instances gets congested. However, even with the state channel being a bottleneck the peak latency is temporary (re-partitioning ends within a second), and median latency remains under a millisecond. The tail latency comes from packets accessing objects remotely due to gradual migration, and subsequent accesses become local without incurring network round-trips.

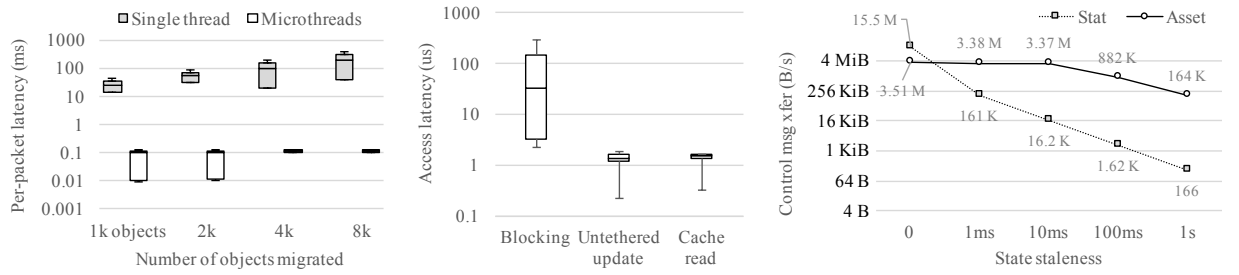
We also evaluate a worst-case scenario with bursty object migration. We generate traffic in a round-robin fashion (*i.e.*, packets are generated sequentially from the flow pool) so that all per-flow state objects migrate back-to-back. Figure 6(c) shows that the peak latency increases as more objects migrate and the state channel becomes more congested. Similarly to the previous graphs, the peak latency lasts only for 500 ms, and the median latency stays under a millisecond.

S6 creates user-level microthreads for non-blocking object migration and key space re-partitioning. The overhead of microthreads was very lightweight for all cases; while S6 can manage up to millions of microthreads, much less is necessary in practice. The maximum number of concurrent microthreads (not shown in the graphs) during migration was about 30k for the 16 Kib case, or a few thousands for other cases.

7.2 Normal Operation

How does S6 compare to existing approaches? We compare S6's performance against the remote-only and local-only options discussed in §3. The local-only model serves as an idealized scenario, as the absence of remote access overhead represents a performance overhead. Since it does not support shared state, we instead replicate non-partitionable state across all instances, thus resulting in incorrect NF behavior. For the remote-only design, we consider StatelessNF [27] as state-of-the-art. While its source code is not publicly available, we implement the algorithms as presented in the paper, including the performance optimization techniques. Our remote-only test uses one remote store and an NF instance; our "distributed/shared" test (S6) uses two NF instances but measures the throughput of only one instance.

Figure 7 shows the throughput and latency of each implementation. The remote-only shows 2-5x lower throughput and 10-100x times higher latency than the ideal (local-only) case since it requires multiple remote state accesses per packet. Another overhead we observed is that depending on workloads and NF types, the state channel (for communication between NF instances and the remote storage) may become more congested than the data channel itself, even with the applications described in the StatelessNF paper. In S6 the only remote access is the first access of a migrating flow context, and all subsequent accesses are local. S6 shows 82-92% of the throughput and comparable latency to the local-only, an ideal case.



(a) Benefit of microthreads (b) Loose consistency (c) Communication overhead vs. Staleness

Figure 9: (a) Per-packet latency during scale-out with and without microthreading, (b) Read/Write latency for remote access (1s maximum staleness), (c) Tradeoff between communication channel overhead vs. accuracy of state

How scalable is S6 with the number of instances? Figure 8 plots the speedup of PRADS and IDS on S6, relative to the baseline throughput of a single instance. The actual aggregated throughput is within 2-8% of ideal linear speedup. We observe little impact on packet processing latency with increasing number of instances.

7.3 Micro-benchmarks

How much do microthreads improve latency relative to a single-threaded approach? In this experiment, we quantify the benefit of microthreading that masks the cost of remote state access on Figure 9(a). To start, we run a single instance which owns all per-flow objects with 1 Mpps input load. Then we split the traffic between two instances, triggering state migration of half of the objects to the second instance. Here we see that the microthreaded architecture improves latency by over three orders of magnitude. Microthreads efficiently pipeline processing packets as they are blocked for object migration. Also, since most packets are processed locally, the number of outstanding microthreads remains small: 130-160 during our experiments.

How much does annotation-based optimization improve state access latency? We compare the performance of different remote access mechanisms in Figure 9(b): 1) blocking RPC, 2) untethered update, and 3) cached read. We run two S6 instances, and 16k shared objects are evenly distributed between two instances. Each instance randomly accesses one of the 16k objects. Thus half of the accesses are local, and the other half are remote. The results show that the latency of untethered updates and cached read is only a few microsecond, since state access can be done with local memory reads/writes; actual synchronization happens in the background. However, in the case of blocking RPC, remote access adds one network round-trip latency for remote objects.

How much does caching reduce communication channel overhead? As we discusses in §4.2, with commutative updates of shared state, we can lower communication channel overhead by allowing bounded staleness. Figure 9(b) quantifies the trade-off, with two different types of shared objects in PRADS. In the case of Stat(istics), a single object is shared by all instances, and every packet triggers at least three updates on it. As we allow more staleness, the required communication channel bandwidth decreases proportionally.

On the other hand, commutativity is not always effective when compared with per-update RPCs. In the case of Assets, State objects—per-host assets—are only shared among flows originated from or destined to the same host. Since updates to an object are not very frequent, periodic synchronization performs no better than individual updates. As shown in the graph, staleness less than 100 ms does not lower the communication channel overhead.

8 Discussion

8.1 State beyond objects

Collections Many NF applications include collection data structures (e.g., linked list, tree, or hash tables). In S6, one can build such collections as non-intrusive containers of references of objects like C++ STL [39]. In non-intrusive data structures, objects do not need to have a special pointer for the container to be a member of it (*e.g.*, a pointer for the next element in list), but the container organizes data structures using references of the objects. One can also specially design a collection structure for efficient concurrent accesses (e.g., RCU [30]).

Framework-level supports on collections will have more opportunities to exploit better locality on its elements. We implements hashtable and read-only iterator on it and leave more framework-level support for data structures as future work.

Multi-object transactions S6 natively supports linearizability – ordering amongst writes to a single object, but not support serializability – ordering with regard to multiple objects. To support multi-object transactions, NF developers can implement a custom lock with exclusively update-able objects. Only a single instance is allowed to have a reference to the lock object; the other instances need to wait until the reference is released from the previous instance. The key owner serializes accesses to the lock object.

8.2 Fault-tolerance

Fault tolerance for middleboxes and network functions has been addressed by prior systems like Pico [36] and FTMB [39]. These systems promise that when an NF fail in a non-scaled out environment, a new NF quickly come back online – with all of the state of the failed NF – and resume processing data.

The most straightforward remediation is to adopt Pico's (checkpoint-based, per-state snapshot) or FTMB's (checkpoint and replay-based, VM snapshot) algorithms at on per node basis. Both systems interpose on accesses to middlebox state during packet processing; these systems also have the ability to interpose on accesses made by S6's RPC calls from other NF instances. Both Pico and FTMB have efficient backup strategies, in that one 'backup' instance can serve as a standby for multiple 'hot' NF instances. S6's knowledge about object access patterns and consistency gives more opportunity to optimize per-object snapshot, balancing between snapshot frequency and amount of logging operations on it.

An alternative approach to fault tolerance could be to extend S6's state management with classic DHT-based failover recovery. Key ownership—and perhaps even data itself—could be replicated thrice across multiple DHT nodes. Hence, if any individual node failed, the rest of the cluster could immediately continue processing incoming flows, accessing the remaining replicated state. Nonetheless, this approach triples intra-cluster traffic, and likely increases read/write latencies. We leave exploration of this approach, its design details, and trade-offs, to future work.

9 Related Work

In §3, we have discussed E2 [33], Split/Merge [37], StatelessNF [27], and OpenNF [25]. We do not revisit them here. There are some specific (not general) NF implementations that internally support horizontal scaling including Maglev (load-balancer) [21], Protego (IPSec gateway) [41], and Bro Cluster [34]. These systems leverage each NF-specific techniques, but cannot be generalized for other types of NFs.

The design and implementation of S6 are heavily inspired by previous work. There are many systems adopting the concepts of DSO to build distributed systems. RPC frameworks such as (CORBA [1], DCOM [2], and RMI [4] provide state access in a uniform manner across heterogeneous languages and software. Thor [17] is a distributed database system that takes care of object distribution, sharing, and caching. Fabric [29] is a distributed application building framework, which focuses on guaranteeing information security among distrust users. All of above systems provide uniform access to objects distributed across nodes, guarantee object consistency, and provide high availability. Yet, none of the above focuses on supporting high-performance requirements such for NFs and elastically adjusting the number of instances on the cluster with minimal interrupts. S6 extends the DSO to support elastic scaling and optimal performance both for under normal operations and during scaling events. We also acknowledge that use of lightweight multi-threading for masking remote access latency can be found in other application domains, *e.g.*, distributed graph processing [31].

Distributed shared state can exist at different levels of abstraction from low-level memory to a higher-level object-oriented model. Distributed key-value stores provide a wider range of state abstractions such as blobs [19, 23, 32] and abstracted data types [7], with properties from ACID to eventual consistency [19]. Partitioned Global Address Space (PGAS) allows multiple machines to share the same virtual address space for their physical memory [16, 20, 26]. This abstraction is useful in supporting machine-level optimizations (*e.g.*, dirty page tracking [16], RDMA [20]) but is too low-level for our context. A single page may contain multiple state variables each with different affinity or consistency semantics, making it impossible to migrate state for optimal state-operation affinity. We choose objects to abstract state, because it allows easy to program various requirements of objects; it is easy to program integrity and control accesses to its encapsulated set of data.

10 Conclusion

We presented S6, a framework for building elastic scaling of NF. S6 extends the DSO model to support elastic scaling of NFs without compromising performance, while the object abstraction transparently hides the complex details of data locality, consistency, and marshaling. S6 introduces a various mean to meet the performance requirements of NFs: "smart but lazy" reorganization of DSO space to minimize the performance overhead during scaling events; micro-threaded architecture to mitigate remote access latency; and programming model to trade performance with freshness per object requirements. Compared to previous work, S6 shows minimal performance overhead during scaling events (10-100x than OpenNF [25]) as well as during normal operations (2-5x than StatelessNF [27]. Our code is available at `https://github.com/NetSys/S6`.

11 Acknowledgments

We thank our shepherd Timothy Roscoe and the anonymous reviewers for their invaluable comments. We also thank Aurojit Panda for the enjoyable discussions and feedback, Keunhong Lee and Junmin Choe for their help on evaluations. This work was funded in part by NRF-2014R1A2A1A01007580, NSF-1553747, NSF-1704941 and Intel corporation.

References

[1] CORBA. http://www.corba.org/.

[2] DCOM. https://msdn.microsoft.com/en-us/library/cc226801.aspx.

[3] Evolved Core Network Implementation of OpenAirInterface. https://gitlab.eurecom.fr/oai/openair-cn.

[4] JavaRMI. http://www.oracle.com/technetwork/articles/javaee/index-jsp-136424.html.

[5] openIMS. http://www.openimscore.org/.

[6] PRADS. http://manpages.ubuntu.com/manpages/wily/man1/prads.1.html.

[7] Redis. https://redis.io/.

[8] Snort++. https://www.snort.org/snort3.

[9] Squid. http://www.squid-cache.org/.

[10] Suricata. https://suricata-ids.org/l.

[11] The Software Load Balancer and Dynamic ADC. http://inlab.de/load-balancer/index.html.

[12] BOOST Coroutine. http://www.boost.org/doc/libs/1_60_0/libs/coroutine/doc/html/index.html, accessed 8 May, 2016.

[13] ETSI NFV. http://www.etsi.org/technologies-clusters/technologies/nfv, accessed April 28, 2016.

[14] ETSI NFV. http://clang.llvm.org/, accessed April 28, 2016.

[15] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W. J., AND DOUCEUR, J. R. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track* (2002), pp. 289–302.

[16] CHAPMAN, B., CURTIS, T., POPHALE, S., POOLE, S., KUEHN, J., KOELBEL, C., AND SMITH, L. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (2010), ACM, p. 2.

[17] DAY, M., LISKOV, B., MAHESHWARI, U., AND MYERS, A. C. References to remote mobile objects in Thor. *ACM Letters on Programming Languages and Systems (LOPLAS) 2*, 1-4 (1993), 115–126.

[18] DEAN, J., AND GHEMAWAT, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[19] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review 41*, 6 (2007), 205–220.

[20] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 401–414.

[21] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), pp. 523–535.

[22] FAYAZBAKHSH, S. K., SEKAR, V., YU, M., AND MOGUL, J. C. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013), ACM, pp. 19–24.

[23] FITZPATRICK, B. Distributed caching with memcached. *Linux journal 2004*, 124 (2004), 5.

[24] GEMBER, A., KRISHNAMURTHY, A., JOHN, S. S., GRANDL, R., GAO, X., ANAND, A., BENSON, T., AKELLA, A., AND SEKAR, V. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR abs/1305.0209* (2013).

[25] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 163–174.

[26] HOEFLER, T., DINAN, J., THAKUR, R., BARRETT, B., BALAJI, P., GROPP, W., AND UNDERWOOD, K. Remote memory access programming in mpi-3. *ACM Transactions on Parallel Computing 2*, 2 (2015), 9.

[27] KABLAN, M., ALSUDAIS, A., KELLER, E., AND LE, F. 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, 2017), USENIX Association.

[28] KHALID, J., GEMBER-JACOBSON, A., MICHAEL, R., ABHASHKUMAR, A., AND AKELLA, A. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 239–253.

[29] LIU, J., GEORGE, M. D., VIKRAM, K., QI, X., WAYE, L., AND MYERS, A. C. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 321–334.

[30] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[31] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Confercorence (USENIX ATC 15)* (2015), pp. 291–305.

[32] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., ET AL. The case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[33] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 121–136.

[34] PAXSON, V. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks 31*, 23-24 (1999), 2435–2463.

[35] QAZI, Z. A., TU, C.-C., CHIANG, L., MIAO, R., SEKAR, V., AND YU, M. Simple-fying middlebox policy enforcement using sdn. *ACM SIGCOMM computer communication review 43*, 4 (2013), 27–38.

[36] RAJAGOPALAN, S., WILLIAMS, D., AND JAMJOOM, H. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 1:1–1:15.

[37] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 227–240.

[38] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., AND SHI, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 24–24.

[39] SHERRY, J., GAO, P. X., BASU, S., PANDA, A., KRISHNAMURTHY, A., MACIOCCO, C., MANESH, M., MARTINS, J. A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 227–240.

[40] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet Applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[41] TAN, K., WANG, P., GAN, Z., AND MOON, S. Protego: Cloud-scale multitenant ipsec gateway.

[42] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), HOTOS'03, USENIX Association, pp. 4–4.

[43] WOO, S., JEONG, E., PARK, S., LEE, J., IHM, S., AND PARK, K. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2013), MobiSys '13, ACM, pp. 319–332.