Stateless Network Functions: Breaking the Tight Coupling of State and Processing

Murad Kablan, Azzam Alsudais, and Eric Keller, *University of Colorado Boulder;*
Franck Le, *IBM Research*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Stateless Network Functions: Breaking the Tight Coupling of State and Processing

Murad Kablan, Azzam Alsudais, Eric Keller
University of Colorado, Boulder

Franck Le
IBM Research

Abstract

In this paper we present Stateless Network Functions, a new architecture for network functions virtualization, where we decouple the existing design of network functions into a stateless processing component along with a data store layer. In breaking the tight coupling, we enable a more elastic and resilient network function infrastructure. Our StatelessNF processing instances are architected around efficient pipelines utilizing DPDK for high performance network I/O, packaged as Docker containers for easy deployment, and a data store interface optimized based on the expected request patterns to efficiently access a RAMCloud-based data store. A network-wide orchestrator monitors the instances for load and failure, manages instances to scale and provide resilience, and leverages an OpenFlow-based network to direct traffic to instances. We implemented three example network functions (network address translator, firewall, and load balancer). Our evaluation shows (i) we are able to reach a throughput of 10Gbit/sec, with an added latency overhead of between 100μs and 500μs, (ii) we are able to have a failover which does not disrupt ongoing traffic, and (iii) when scaling out and scaling in we are able to match the ideal performance.

1 Introduction

As evidenced by their proliferation, middleboxes are an important component in today's network infrastructures [50]. Middleboxes provide network operators with an ability to deploy new network functionality as add-on components that can directly inspect, modify, and block or re-direct network traffic. This, in turn, can help increase the security and performance of the network.

While traditionally deployed as physical appliances, with Network Functions Virtualization (NFV), network functions such as firewalls, intrusion detection systems, network address translators, and load balancers no longer have to run on proprietary hardware, but can run in software, on commodity servers, in a virtualized environment, with high throughput [25]. This shift away from physical appliances should bring several benefits including the ability to elastically scale the network functions on demand and quickly recover from failures.

However, as others have reported, achieving those properties is not that simple [44, 45, 23, 49]. The central issue revolves around the state locked into the network functions – state such as connection information in a stateful firewall, substring matches in an intrusion detection system, address mappings in a network address translator, or server mappings in a stateful load balancer. Locking that state into a single instance limits the elasticity, resilience, and ability to handle other challenges such as asymmetric/multi-path routing and software updates.

To overcome this, there have been two lines of research, each focusing on one property[1]. For failure, recent works have proposed either (i) checkpointing the network function state regularly such that upon failure, the network function could be reconstructed [44], or (ii) logging all inputs (*i.e.,* packets) and using deterministic replay in order to rebuild the state upon failure [49]. These solutions offer resilience at the cost of either a substantial increase in per-packet latency (on the order of 10ms), or a large recovery time at failover (e.g., replaying all packets received since the last checkpoint), and neither solves the problem of elasticity. For elasticity, recent works have proposed modifying the network function software to enable the migration of state from one instance to another via an API [29, 45, 23]. State migration, however, takes time, inherently does not solve

[1] A third line, sacrifices the benefits of maintaining state in order to obtain elasticity and resilience [20].

the problem of unplanned failures, and as a central property relies on affinity of flow to instance – each rendering state migration a useful primitive, but limited in practice.

In this paper, we propose *stateless network functions* (or StatelessNF), a new architecture that breaks the tight coupling between the state that network functions need to maintain from the processing that network functions need to perform (illustrated in Figure 1). Doing so simplifies state management, and in turn addresses many of the challenges existing solutions face.

Resilience: With StatelessNF, we can instantaneously spawn a new instance upon failure, as the new instance will have access to all of the state needed. It can immediately handle traffic and it does not disrupt the network. Even more, because there is no penalty with failing over, we can failover much faster – in effect, we do not need to be certain a network function has failed, but instead only speculate that it has failed and later detect that we were wrong, or correct the problem (*e.g.,* reboot).

Elasticity: When scaling out, with StatelessNF, a new network function instance can be launched and traffic immediately directed to it. The network function instance will have access to the state needed through the data store (*e.g.,* a packet that is part of an already established connection that is directed to a new instance in a traditional, virtualized, firewall will be dropped because a lookup will fail, but with StatelessNF, the lookup will provide information about the established connection). Likewise, scaling in simply requires re-directing any traffic away from the instance to be shut down.

Asymmetric / Multi-path routing: In StatelessNF each instance will share all state, so correct operation is not reliant on affinity of traffic to instance. In fact, in our model, we assume any individual packet can be handled by any instance, resulting in an abstraction of a scalable, resilient, network function. As such, packets traversing different paths does not cause a problem.

While the decoupling of state from processing exists in other settings (*e.g.,* a web server with a backend database), the setting of processing network traffic, potentially requiring per packet updates to state, poses a significant challenge. A few key insights and advances have allowed us to bring this new design to a reality. First, there have been recent advances in disaggregated architectures, bringing with it new, low-latency and resilient data stores such as RAMCloud [39]. Second, not all state that is used in network functions needs to be stored in a resilient data store – only dynamic, network state needs to persist across failures and be available to all instances. State such as firewall rules, and intrusion detection system signatures can be replicated to each instance upon boot, as they are static state. Finally, network functions share a common pipeline design where there is typically a lookup operation when the packet is first being processed, and sometimes a write operation after the packet has been processed. This not only means there will be less interaction than one might initially assume, but also allows us to leverage this pattern to optimize the interactions between the data store and the network function instances to provide high performance.

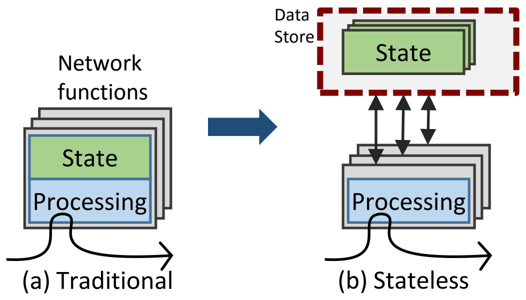


Figure 1: High level overview showing traditional network functions (a), where the state is coupled with the processing to form the network function, and stateless network functions (b), where the state is moved from the network function to a data store – the resulting network functions are now stateless.

We describe how four common network functions, can be re-designed in a stateless manner. We present the implementation of a stateful firewall, an intrusion prevention system, a network address translator, and a load balancer. Section 3 discusses the remote memory access. Section 6 discusses our utilization of RAMCloud for the data store, DPDK for the packet processing, and the optimized interface between the network functions and data store. Section 7 presents the evaluation: our experiments demonstrate that we are able to achieve throughput levels that are competitive with other software solutions [49, 45, 23] (4.6 Million packets per second for minimum sized packets), with only a modest penalty on per-packet latency (between 100us and 500us in the 95th percentile, depending on the application and traffic pattern). We further demonstrate the ability to seamlessly fail over, and scale out and scale in without any impact on the network traffic (as opposed to substantial disruption for a traditional design). Of course, the stateless network functions approach may not be suitable for *all* network functions, and there are further optimizations we can make to increase processing rates. This work, however, demonstrates that there is value for the functions we studied and that even with our current prototype, we are able to match processing rates of other systems with similar goals, while providing both scalability and resilience.

2 Motivation

Simply running virtual versions of the physical appliance counterparts provides operational cost efficiencies, but falls short in supporting the vision of a dynamic network infrastructure that elastically scales and is resilient to failure. Here, we illustrate the problems that the tight coupling of state and processing creates today, even with virtualized network functions, and discuss shortcomings of recent proposals.

First, we clarify our definition of the term "state" in this particular context. Although there is a variety of network functions, the state within them can be generally classified into (1) static state (*e.g.,* firewall rules, IPS signature database), and (2) dynamic state, which is continuously updated by the network function's processes [45, 21]. The latter can be further classified into (i) internal instance specific state (*e.g.,* file descriptors, temporary variables), and (ii) network state (*e.g.,* connection tracking state, NAT private to public port mappings). It is the dynamic network state that we are referring to that must persist across failures and be available to instances upon scaling in or out. The static state can be replicated to each instance upon boot, so will be accessed locally.

2.1 Dealing with Failure

For failure, we specifically mean crash (as opposed to byzantine) failures. Studies have shown that failures can happen frequently, and be highly disruptive [43].

The disruption comes mainly from two factors. To illustrate the first factor, consider Figure 2(a). In this scenario, we have a middlebox, say a NAT, which stores the mapping for two flows (F1 and F2). Upon failure, virtualization technology enables the quick launch of a new instance, and software-defined networking (SDN) [36, 14, 10] allows traffic to be redirected to the new instance. However, any packet belonging to flows F1 or F2 will then result in a failed lookup (no entry in the table exists). The NAT would instead create new mappings, which would ultimately not match what the server expects. This causes all existing connections to eventually timeout. Enterprises could employ hot-standby redundancy, but that doubles the cost of the network.

The second factor is due to the high cost of failover of existing solutions (further discussed below). As such, the mechanisms tend to be conservative when determining whether a device has failed [6] – if a device does not respond to one *hello* message, does that mean that the device is down, the network dropped a packet, or that the device is heavily loaded and taking longer to respond? Aggressive thresholds cause unnecessary failovers, resulting in downtime. Conservative thresholds may forward traffic to a device that has failed, resulting in disruption.

Problem with existing solutions

Two approaches to failure resilience have been proposed in the research community recently. First, pico replication [44] is a high availability framework that frequently checkpoints the state in a network function such that upon failure, a new instance can be launched and the state restored. To guarantee consistency, packets are only released once the state that they impact has been checkpointed – leading to substantial per-packet latencies (*e.g.,* 10ms for a system that checkpoints 1000 times per second, under the optimal conditions).

To reduce latency, another work proposes logging all inputs (*i.e.,* packets) coupled with a deterministic replay mechanism for failure recovery [49]. In this case, the per-packet latency is minimized (the time to log a single packet), but the recovery time is high (on the order of the time since last check point). In both cases, there is a substantial penalty – and neither deals with scalability or the asymmetric routing problem (discussed further in Section 2.3).

2.2 Scaling

As with the case of failover, the tight coupling of state and processing causes problems with scaling network functions. This is true even when the state is highly partitionable (*e.g.,* only used for a single flow of traffic, such as connection tracking in a firewall). In Figure 2(b), we show an example of scaling out. Although a new instance has been launched to handle the overloaded condition, existing flows cannot be redirected to the new instance – *e.g.,* if this is a NAT device, packets from flow F2 directed at the new instance will result in a failed lookup, as was the case with failure. Similarly, scaling in (decreasing instances) is a problem, as illustrated in Figure 2(c). As the load is low, one would like to shut down the instance which is currently handling flow F3. However, one has to wait until that instance is completely drained (*i.e.,* all of the flows it is handling complete). While possible, it is something that limits agility, requires special handling by the orchestration, and highly depends on flows being short lived.

Problem with existing solutions

The research community has proposed solutions based on state migration. The basic idea is to instrument the network functions with code that can export state from one instance and import that state into another instance. Router Grafting demonstrated this for routers (moving

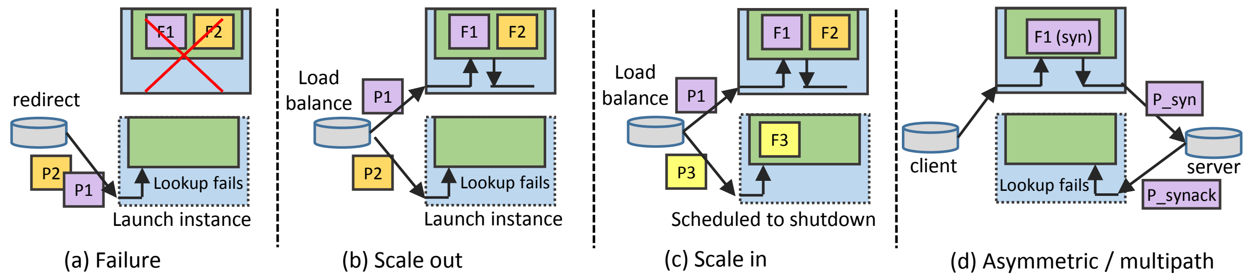


Figure 2: Motivational examples of traditional network functions and the problems that result from the tight coupling of state to the network function instance. State associated with some flow is labeled as F (*e.g.,* F2), and the associated packets within that flow are labeled as P (*e.g.,* P2).

BGP state) [29], and several have since demonstrated this for middleboxes [45, 23, 22] where partitionable state can be migrated between instances. State migration, however, takes time, inherently does not solve the problem of unplanned failures, and as a central property relies on affinity of flow to instance (limiting agility).

2.3 Asymmetric / Multi-path Routing

Asymmetric and multi-path routing can cause further challenges for a dynamic network function infrastructure: Asymmetric and multi-path [41] routing relates to the fact that traffic in a given flow may traverse different paths, and therefore be processed by different instances. For example, in the scenario of Figure 2(d), where a firewall has established state from an internal client connecting to a server (SYN packet), if the return syn-ack goes through a different firewall instance, this packet may result in a failed lookup and get dropped.

Problem with existing solutions

Recent work proposes a new algorithm for intrusion detection that can work across instances [35], but does so by synchronizing processing (directly exchanging state and waiting on other instances to complete processing as needed). Other solutions proposed in industry strive to synchronize state across middleboxes [31] (*e.g.,* HSRP [32]), but generally do not scale well.

3 How Network Functions Access State

The key idea in this paper is to decouple the processing from the state in network functions – placing the state in a data store. We call this stateless network functions (or StatelessNF), as the network functions themselves become stateless, and the statefulness of the applications (*e.g.,* a stateful firewall) is maintained by storing the state in a separate data store.

To understand the intuition as to why this is feasible, even at the rates network traffic needs to be processed, here we discuss examples of state that would be decoupled in common network functions, and what the access patterns are.

Table 1 shows the network state to be decoupled and stored in a remote storage for four network functions (TCP re-assembly is shown separate from IPS for clarity, but we would expect them to be integrated and reads/writes combined). As shown in the table, and discussed in Section 2, we only decouple network state.

We demonstrate how the decoupled state is accessed with pseudo-code of multiple network function algorithms, and summarize the needed reads and writes to the data store in Table 1. In all algorithms, we present updating or writing state to the data store as *writeRC* and reads as *readRC* (where RC relates to our chosen data store, RAMCloud). Below we describe Algorithms 1 (load balancer) and 2 (IPS). The pseudo-code of a stateful firewall, TCP re-assembly, and NAT are provided in Appendix for reference.

For the load balancer, upon receiving a TCP connection request, the network function retrieves the list of backend servers from the remote storage (line 4), and then assigns a server to the new flow (line 5). The load for the backend servers is subsequently updated (line 6), and the revised list of backend servers is written into remote storage (line 7). The assigned server for the flow is also stored into remote storage (line 8), before the packet is forwarded to the selected server. For a data packet, the network function retrieves the assigned server for that flow, and forwards the packet to the server.

Algorithm 2 presents the pseudo-code for a signature-based intrusion prevention system (IPS), which monitors network traffic, and compares packets against a database of signatures from known malicious threats using an algorithm such as Aho-Corasick algorithm [11] (as used

Network Function	State	Key	Value	Access Pattern
Load Balancer	Pool of Backend Servers	Cluster ID	IP List	1 read/write at start/end of conn.
	Assigned Server	5-Tuple	IP Address	1 read/write at start/end of conn. 1 read for every other packet
Firewall	Flow	5-Tuple	TCP Flag	5 read/write at start/end of conn. 1 read for every other packet
NAT	Pool of IPs and Ports	Cluster ID	IP and Port List	1 read/write at start/end of conn.
	Mapping	5-Tuple	(IP, Port)	1 read/write at start/end of conn. 1 read for every other packet
TCP Re-assembly	Expected Seq Record	5-Tuple	(Next Expected Seq, Keys for Buffered Pkts)	1 read/write for every packet
	Buffered Packets	Buffer Pointer	Packet	1 read/write for every out-of-order packet
IPS	Automata State	5-Tuple	Int	1 write for first packet of flow, 1 read/write for every other packet

Table 1: Network Function Decoupled States

Algorithm 1 Load Balancer

```
 1: procedure PROCESSPACKET(P: TCPPacket)
 2:     extract 5-tuple from incoming packet
 3:     if (P is a TCP SYN) then
 4:         backendList ← readRC(Cluster ID)
 5:         server ← nextServer(backendList, 5-tuple)
 6:         updateLoad(backendList, server)
 7:         writeRC(Cluster ID, backendList)
 8:         writeRC(5-tuple, server)
 9:         sendPacket(P, server)
10:     else
11:         server ← readRC(5-tuple)
12:         if (server is NULL) then
13:             dropPacket(P)
14:         else
15:             sendPacket(P, server)
```

Algorithm 2 IPS

```
 1: procedure PROCESSPACKET(P: TCPPacket)
 2:     extract 5-tuple, and TCP sequence number from P
 3:     if (P is a TCP SYN) then
 4:         automataState ← initAutomataState()
 5:         writeRC(5-tuple, automataState)
 6:     else
 7:         automataState ← readRC(5-tuple)
 8:         while (b ← popNextByte(P.payload)) do
 9:             // alert if found match
10:             // else, returns updated automata
11:             automataState ← process(b, automataState)
12:         writeRC(5-tuple, automataState)
13:         sendPacket(P)
```

in Snort [9]). At a high-level, a single deterministic automaton can be computed offline from the set of signatures (stored as static state in each instance). As packets arrive, scanning each character in the stream of bytes triggers one state transition in the deterministic automaton, and reaching an output state indicates the presence of a signature.

The 5-Tuple of the flow forms the key, and the state (to be stored remotely) simply consists of the state in the deterministic automaton (e.g., an integer value representing the node reached so far in the deterministic automaton). Upon receiving a new flow, the automata state is initialized (line 4). For a data packet, the state in the deterministic automaton for that flow is retrieved from remote storage (line 7). The bytes from the payload are then scanned (line 8). In the absence of a malicious signature, the updated state is written into remote storage (line 12), and the packet forwarded (line 13). Out-of-order packets are often considered a problem for Intrusion Prevention Systems [52]. Similar to the Snort TCP reassembly preprocessor [9], we rely on a TCP re-assembly module to deliver the bytes to the IPS in the proper order.

For the load balancer, we observe that we require one read for each data packet, and at most one additional read and write to the remote storage at the start and end of each connection. For the IPS, we observe that we require one write to the remote storage to initialize the automata state at the start of each connection, and one read and one write to remote storage for each subsequent data packet of the connection. Table 1 shows similar patterns for other network functions, and Section 7 analyzes the performance impact of such access patterns, and demonstrates that we can achieve multi Gbps rates.

4 Overall StatelessNF Architecture

At a high level, StatelessNF consists of a network-wide architecture where, for each network function application (*e.g.,* a firewall), we effectively have the abstraction of a single network function that reliably provides the necessary throughput at any given time. To achieve this, as illustrated in Figure 3, the StatelessNF architecture consists of three main components – the data store, the hosts to host the instances of the network function, and an orchestration component to handle the dynamics of the network function infrastructure. The network function hosts are simply commodity servers. We discuss the internal architecture of network function instances in Section 5. In this section, we elaborate on the data store and network function orchestration within StatelessNF.

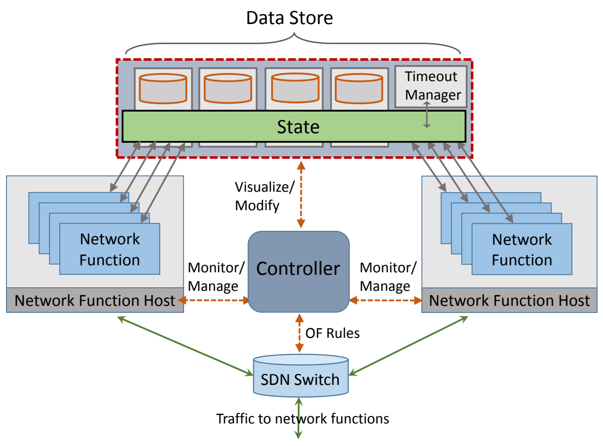


Figure 3: StatelessNF System Architecture

4.1 Resilient, Low-latency Data Store

A central idea in StatelessNF, as well as in other uses of remote data stores, is the concept of separation of concerns. That is, in separating the state and processing, each component can concentrate on a more specific functionality. In StatelessNF, a network function only needs to process network traffic, and does not need to worry about state replication, etc. A data store provides the resilience of state. Because of this separation, and because it resides on the critical path of packet processing, the data store must also provide low-latency access. For our purposes, we assume a data store that does not need support for transactions, but we anticipate exploring the impact of network functions that may require transactions as future work. In this paper, we choose RAMCloud [39] as our data store. RAMCloud is a distributed key-value storage system that provides low-latency access to data, and supports a high degree of scalability.

Resilient: For a resilient network function infrastructure, the data store needs to reliably store the data with high availability.

This property is common in available data stores (key value stores) through replication. For an in-memory data store, such as RAMCloud [39], the cost of replication would be high (uses a lot of RAM). Because of this, RAMCloud only stores a single copy of each object in DRAM, with redundant copies on secondary storage such as disk (on replica machines). To overcome the performance cost of full replication, RAMCloud uses a log approach where write requests are logged, and the log entry is what is sent to replicas, where the replicas fill an in-memory buffer, and then store on disk. To recover from a RAMCloud server crash, its memory contents must be reconstructed by replaying the log file.

Low-Latency: Each data store will differ, but RAMCloud in particular was designed with low-latency access in mind. RAMCloud is based primarily in DRAM and provides low-latency access (6μs reads, 15μs durable writes for 100 bytes data) at large-scale (*e.g.,* 10,000 servers). This is achieved both by leveraging low-latency networks (such as Infiniband and RDMA), being entirely in memory, and through optimized request handling. While Infiniband is not considered commodity, we believe it has growing acceptance (*e.g.,* Microsoft Azure provides options which include Infiniband [5]), and our architecture does not fundamentally rely on Infiniband – RAMCloud developers are working on other interfaces (*e.g.,* RoCE [47] and Ethernet with DPDK), which we will integrate and evaluate as they become available.

Going beyond a key-value store: The focus of data stores is traditionally the key-value interface. That is, clients can read values by providing a key (which returns the value), or write values by providing both the key and value. We leverage this key-value interface for much of the state in network functions.

The challenge in StatelessNF is that a common type of state in network functions, namely timers, do not effectively conform to a key-value interface. To implement with a key-value interface, we would need to continuously poll the data store – an inefficient solution. Instead, we extend the data store interface to allow for the creation and update of timers. The timer alert notifies one, and only one, network function instance, for which the handler on that instance processes the timer expiration.

We believe there may be further opportunities to optimize StatelessNF through customization of the data store. While our focus in this paper is more on the network-wide capabilities, and single instance design, as a future direction, we intend to further understand how a data store can be adapted to further suit the needs of network functions.

4.2 Network Function Orchestration

The basic needs for orchestration involve monitoring the network function instances for load and failure, and adjusting the number of instances accordingly.

Resource Monitoring and Failure Detection: A key property of orchestration is being able to maintain the abstraction of a single, reliable, network function which can handle infinite load, but under the hood maintain as efficient of an infrastructure as possible. This means that the StatelessNF orchestration must monitor resource usage as well as be able to detect failure, and adjust accordingly – *i.e.,* launch or kill instances.

StatelessNF is not tied to a single solution, but instead we leverage existing monitoring solutions to monitor the health of network functions to detect failure as well as traffic and resource overload conditions. Each system hosting network functions can provide its own solution – *e.g.,* Docker monitoring, VMWare vcenter health status monitoring, IBM Systems Director for server and storage monitoring. Since we are using Docker containers as a method to deploy our network functions, our system consists of an interface that interacts with the Docker engines remotely to monitor, launch, and destroy the container-based network functions. In addition, our monitoring interface, through ssh calls, monitors the network function resources (cores, memory, and SR-IOV cards) to make sure they have enough capacity to launch and host network functions.

Important to note is that failure detection is different in StatelessNF than in traditional network function solutions. With StatelessNF, we have an effectively zero-cost to failing over – upon failure, any traffic that would go through the failed instance can be re-directed to any other instance. With this, we can significantly reduce the detection time, and speculatively failover. This is in contrast to traditional solutions that rely on timeouts to ensure the device is indeed failed.

Programmable Network: StatelessNF's orchestration relies on the ability to manage traffic. That is, when a new instance is launched, traffic should be directed to the instance; and when a failure occurs or when we are scaling-in, traffic should be redirected to a different instance. With emerging programmable networks, or software-defined networks (SDN), such as OpenFlow [36] and P4 [14], we can achieve this. Further, as existing SDN controllers (*e.g.,* ONOS [13], Floodlight [4], OpenDaylight [10]) provide REST APIs, we can integrate the control into our overall orchestration.

5 StatelessNF Instance Architecture

Whereas the StatelessNF overall architecture provides the ability to manage a collection of instances, providing the elasticity and resilience benefits of StatelessNF, the architecture of the StatelessNF instances are architected to achieve the deployability and performance needed. As shown in Figure 4, the StatelessNF instance architecture consists of three main components – (i) a packet processing pipeline that can be deployed on demand, (ii) high-performance network I/O, and (iii) an efficient interface to the data store. In this section, we elaborate on each of these.

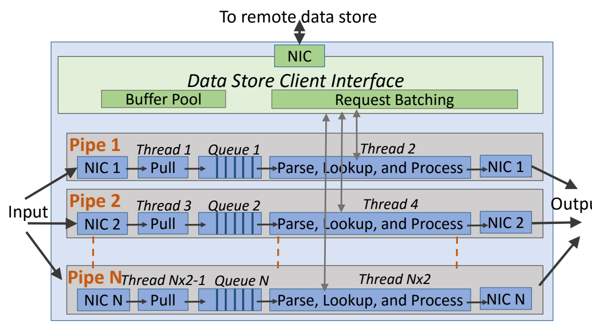


Figure 4: Stateless Network Function Architecture

5.1 Deployable Packet Processing Pipeline

To increase the performance and deployability of stateless network function instances, each network function is structured with a number of packet processing pipes. The number of pipes can be adaptive based on the traffic load, thus enabling a network function with a better resource utilization. Each pipe consists of two threads and a single lockless queue. The first thread is responsible for polling the network interface for packets and storing them in the queue. The second thread performs the main processing by dequeuing the packet, performing a lookup by calling the remote state interface to read, applying packet processing based on returned state and network function logic, updating state in the data store, and outputting the resulting packet(s) (if any).

Network function instances can be deployed and hosted with a variety of approaches – virtual machines, containers, or even as physical boxes. We focus on containers as our central deployable unit. This is due to their fast deployment, low performance overhead, and high reusability. Each network function instance is implemented as a single process Docker instance with independent cores and memory space/region. In doing so, we ensure that network functions don't affect each other.

For network connectivity, we need to share the physical interface among each of the containers (pipelines). For this, we use SR-IOV[7] to provide virtual interfaces to each network function instance. Modern network cards have hardware support for classifying traffic and presenting to the system as multiple devices – each of the virtual devices can then be assigned to a network function instance. For example, our system uses Intel x520 server adapters[27] that can provide up to 126 virtual cards with each capable of reaching maximum traffic rate (individually). For connectivity to the data store, as our implementation focuses on RAMCloud, each network function host is equipped with a single Infiniband card that is built on the Mellanox RDMA library package[37], which allows the Infiniband NIC to be accessed directly from multiple network function user-space applications (bypassing the kernel). As new interfaces for RAMCloud are released, we can simply leverage them.

5.2 High-performance Network I/O

As with any software-based network processing application, we need high performance I/O in order to meet the packet processing rates that are expected. For this, we leverage the recent series of work to provide this – *e.g.,* through zero copy techniques. We specifically structured our network functions on top of the Data Plane Development Kit (DPDK) architecture [26]. DPDK provides a simple, complete framework for fast packet processing.

One challenge that arises with the use of DPDK in the context of containers is that large page support is required for the memory pool allocation used for packet buffers and that multiple packet processing pipes (containers) may run simultaneously on a single server. In our case, each pipe is assigned a unique page filename and specified socket memory amount to ensure isolation[2]. We used the DPDK Environment Abstraction Layer (EAL) interface for system memory allocation/de-allocation and core affinity/assignment procedures among the network functions.

[2]After several tests, we settled on 2GB socket memory for best performance.

5.3 Optimized Data Store Client Interface

Perhaps the most important addition in StatelessNF is the data store client interface. The importance stems from the fact that it is through this interface, and out to a remote data store, that lookups in packet processing occur. That is, it sits in the critical path of processing a packet and is the main difference between stateless network functions and traditional network functions.

Each data store will come with an API to read and write data. In the case of RAMCloud, for example, it is a key-value interface which performs requests via an RPC interface, and that leverages Infiniband (currently). RAMCloud also provides a client interface which abstracts away the Infiniband interfacing.

To optimize this interface to match the common structure of network processing, we make use of three common techniques:

Batching: In RAMCloud, a single read/write has low-latency, but each request has overhead. When packets are arriving at a high rate, we can aggregate multiple requests into a single request. For example, in RAMCloud, a single read takes 6μs, whereas a multi-read of 100 objects takes only 51μs (or, effectively 0.51μs per request). The balance here, for StatelessNF, is that if the the batch size is too small, we may be losing opportunity for efficiency gains, and too long (even with a timeout), we can induce higher latency than necessary waiting for more packets. Currently, we have a fixed batch size to match our experimental setup (100 objects), but we ultimately envision an adaptive scheme which increases or decreases the batch size based on the current traffic rates.

Pre-allocating a pool of buffers: When submitting requests to the data store, the client must allocate memory for the request (create a new RPC request). As this interface is in the critical path, we reduce the overhead for allocating memory by having the client reuse a preallocated pool of object buffers.

Eliminating a copy: When the data from a read request is returned from the data store to the client interface, that data needs to be passed to the packet processing pipeline. To increase the efficiency, we eliminate a copy of the data by providing a pointer to the buffer to the pipeline which issued the read request.

6 Implementation

The StatelessNF orchestration controller is implemented in Java with an admin API that realizes the implementation of elastic policies in order to determine when to create or destroy network functions. At present, the policies are trivial to handle the minimal needs of handling failure and elasticity, simply to allow us to demonstrate the feasibility of the StatelessNF concept (see Section 7 for elasticity and failure experiments). The controller interacts with the Floodlight [4] SDN controller to steer the flows of traffic to the correct network function instances by inserting the appropriate OpenFlow rules. The controller keeps track of all the hosts and their resources, and network function instances deployed on top of them. Finally, the controller provides an interface to access and monitor the state in the data store, allowing the operator to have a global view of the network status.

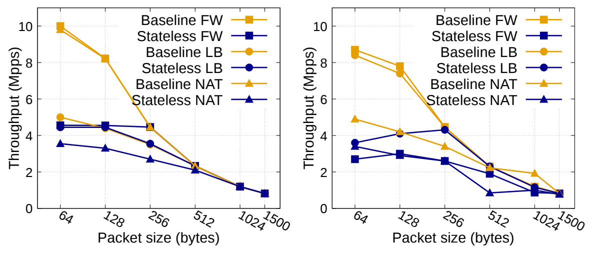
We implemented three network functions (firewall, NAT, load balancer) as DPDK [26] applications, and packaged as a Docker container. For each, we implemented in a traditional (non-stateless) and stateless fashion. In each case, the only difference is that the non-stateless version will access its state locally while the stateless version from the remote data store. The client interface to the data store is implemented in C++ and carries retrieval operations to RAMCloud [39]. The packet processing pipes are implemented in C/C++ in a sequence of pipeline functions that packets travel through, and only requires developers to write the application-specific logic – thus, making modifying the code and adding new network function relatively simple. The data store client interface and the packet processing pipes are linked at compile time.

7 Evaluation

This section evaluates the network functions performance, the recovery times in failure events, and the performance impact when scaling in/out with the proposed stateless architecture.

7.1 Experimental Setup

Our experimental setup is similar to the one depicted in Figure 3. It consists of six servers and two switches. Two servers are dedicated to hosting the network function instances. These two servers are connected via Infiniband to two other servers hosting RAMCloud (one acting as the RAMCloud coordinator, and the other server storing state), and are connected via Ethernet to a server acting as the traffic generator and sink (not shown in Figure 3). The last server hosts the StatelessNF controller which orchestrates the entire management. Specifically, we use the following equipment:



(a) Long flow case (b) Short flow case

Figure 5: Throughput of different packet sizes for long (a) and short (b) flows (*i.e.,* flow sizes >1000 and <100, respectively) measured in the number of packets per second.

- Network Function hosts: 2 Dell R630 Servers [16]: each has 32GB RAM, 12 cores (2.4GHz), one Intel 10G Server Adapter with SR-IOV support [27], and one 10G Mellanox InfiniBand Adapter Card [37].
- RAMCloud: 2 Dell R720 Servers [17], each with 48GB RAM, 12 cores (2.0GHz), one Intel 10G Server Adapter [27], one 10G Mellanox InfiniBand Adapter Card [37].
- Traffic generator/sink: 1 Dell R520 Servers [15]: 4GB RAM, 4 cores (2.0GHz), 2 Intel 10G Server Adapters [27] .
- Control: 1 Dell R520 Servers [15]: 4GB RAM, 4 cores (2.0GHz) to run StatelessNF and Floodlight controllers.
- SDN Switch: OpenFlow-enabled 10GbE Edge-Core [19].
- Infiniband Switch: 10Gbit Mellanox Infiniband switch between RAMCloud nodes and the network function hosts [38].

7.2 StatelessNF Performance

7.2.1 Impact of needing remote reads/writes

It is first critical to understand the performance of the RAMCloud servers as they may be a performance bottleneck, and limit the rates we can attain. Our benchmark tests reveal that a single server in RAMCloud can handle up to 4.7 Million lookup/sec. For write operations, a single server can handle up to 0.7 Million write/second.

The performance of a network function therefore heavily depends on the packets sizes, the network function's access patterns to the remote storage, and the processed traffic characteristics: For example, while a load balancer requires three write operations per flow, a firewall requires five write operations per flow. As such, whether traffic consists of short flows (e.g., consisting of only hundreds of packets), or long flows (e.g., comprising tens of thousands of packets), these differences in access patterns can have a significant impact on the network function performance. In particular, short flows require many more writes for the same amount of traffic. We consequently distinguish three cases for the processed traffic: long, short, and average in regards to the size and number of flows. The long case consists of a trace of 3,000 large TCP flows of 10K packets each. The short case consists of a trace of 100,000 TCP flows of 100 packets each. Finally for the average case, we replayed a real captured enterprise trace [3] with 17,000 flows that range in size from 10 to 1,000 packets. In each case, we also varied the packet sizes to understand their impact on the performance. We used Tcpreplay with Netmap [51] to stream the three types of traces.

Figure 5 shows the throughput of StatelessNF middleboxes compared to their non-stateless counterparts (which we refer to as baseline) with long and short flows of different packet sizes. For minimum sized packets, we obtain throughputs of 4.6Mpps. For small sized packets (less than 128 bytes), the gap between stateless and non-stateless in throughput is due to a single RAMCloud server being able to handle around 4.7 Million lookups/sec. In contrast, in the baseline, all read and write operations are local. We highlight that such sizes are used to test the upper bound limits of our system.

As packets get larger in size (greater than 128 bytes), the rates of stateless and baseline network functions converge. The obtained throughputs are competitive with those of existing elastic and fail resilience software solutions [49, 45, 23]. To understand the performance of stateless network functions with real traces, we increase the rate of the real trace to more than the original rate at which it was captured[3], and analyze the achievable throughput. Since the packet sizes vary considerably (80 to 1500 bytes), we report the throughput in terms of traffic rate (Gbit/sec) rather than packets/sec. Figure 6 shows that the statelessNF firewall and loadbalancer have comparable performance than their baseline counterpart. The stateless NAT reaches a limit that is 1Gbps lower than the non-stateless version. Finally, we also observe that the performance of the NAT are several Gbps lower than the firewall and load balancer. This is due to the overhead of IP header checksum after modifying the packet IP addresses and port numbers.

[3]The rates of enterprise traces we found vary from 0.1 to 1 Gbit/sec

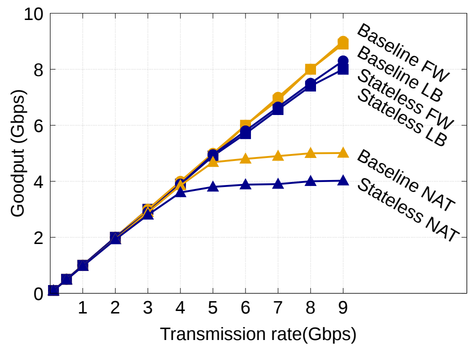
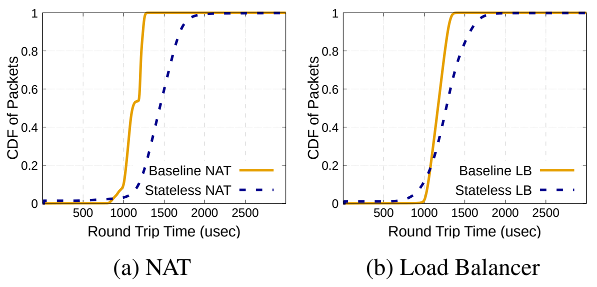


Figure 6: Measured goodput (Gbps) for enterprise traces.



(a) NAT (b) Load Balancer

Figure 7: Round-trip time (RTT) of packets.

7.2.2 Latency

The interaction with the remote storage can increase the latency of each packet, as every incoming packet must be buffered until its lookup operation is completed. To evaluate the delay increase, we compared the round-trip time (RTT) of each packet in the stateless and baseline network functions. We timestamp packets, send the traffic through the network function which resends the packets back to the initial host.

Figure 7 shows the cumulative distribution function (CDF) for the RTT of packets traversing the NAT and load balancer[4]. In the 50th percentile, the RTT of StatelessNF packets is only 100μs larger than the baseline's for the load balancer and NAT, and in the 95th percentile the RTT is only 300μs larger. The added delay we see in StatelessNF is a combination of read misses (which can reach 100μs), preparing objects for read requests from RAMCloud, casting returned data, and the actual latency of the request. These numbers are in the range of other comparable systems (*e.g.,* the low-latency rollback recovery system exhibited about a 300μs higher latency than the baseline [49]). Further, while the reported latency when using Ethernet (with DPDK) to communi-

[4]The RTT for firewall (both stateless and baseline) showed similar trend to load balancer with a better average delay (67μs less).

cate with RAMCloud is higher than Infiniband (31.1μs, read 100B, and 77μs write 100B), it is still less than the average packet delays reported with StatelessNF system (65us, 100us, and 300us for firewall, load balancer, and NAT respectively). Given the actual network traversals of requests can occur in parallel as the other aspects of the request, we believe that the difference in latency between Ethernet with DPDK and Infiniband can be largely masked. We intend to validate as future work.

7.2.3 IPS Analysis

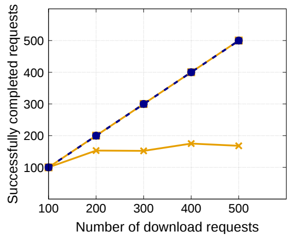
This section analyzes the impact of decoupling the state for an IPS. The overall processing of an IPS is more complex (Section 3) than the three network functions we previously analyzed. However, its access patterns to the remote storage is only incrementally more complex.

To analyze the impact of decoupling automaton state into remote storage, we implemented an in-line stateless network function that emulates a typical IPS in terms of accessing the automaton state and performs a read and write operation for every packet. For comparison, we run Snort [8] as an in-line IPS, and streamed real world enterprise traces through both instances: Snort and our stateless emulated IPS. The stateless emulated IPS was able to reach a throughput of 2.5Gbit/sec while the maximum performance for the Snort instance was only 2Gbit/sec. These results show that for an IPS, the performance bottleneck is the internal processing (*e.g.,* signature search), and not the read/write operations to the remote storage.

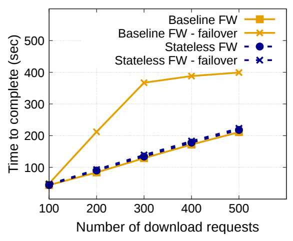
7.3 Failure

As we discussed in Section 3, in the case of failover, the instance we failover to can seamlessly handle the redirected traffic from the failed instance without causing any disruption for the traffic. To illustrate this effect, and compare to the traditional approach, we performed a number of file downloads that go through a firewall, and measured the number of successful file downloads and the time require to complete all of the downloads in the following cases: 1) baseline and stateless firewalls with no failure; 2) baseline and stateless firewall with failure where we redirect traffic to an alternate instance. In this case, we are only measuring the effect of the disruption of failover, as we assume a perfect failure detection, and simulate this by programming the SDN switch to redirect all traffic at some specific time. If we instrumented failure detection, the results would be more pronounced.

Figure 8 shows our results where we downloaded up to 500 20MB files in a loop of 100 concurrent http downloads through the firewall. As we can see, the baseline firewall is significantly affected by the sudden failure because the backup instance will not recognize the redirected traffic, hence will drop the connections, which in turn results in the client re-initiating the connections after a TCP connection timeout[5]. Not only was the stateless firewall able to successfully complete all downloads, but the performance was unaffected due to failure, and matched the download time of the baseline firewall when it did not experience failure.



(a) Completed requests



(b) Time to complete requests

Figure 8: (a) shows the total number of successfully completed requests, and (b) shows the time taken to satisfy completed requests.

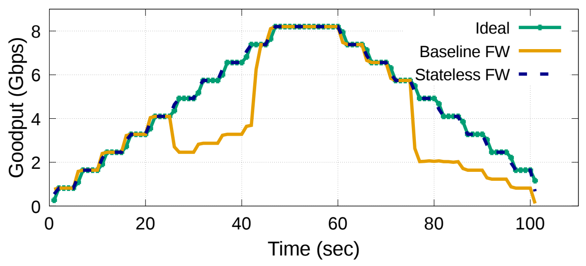


Figure 9: Goodput (Gbps) for stateless and baseline firewalls while scaling out (t=25s) and in (t=75s).

7.4 Elasticity

In this paper, we claim that decoupling state from processing in network functions provides elasticity, where scaling in/out can be done with no disruption to the traffic. To evaluate StatelessNF's capability of scaling in and out, we performed the following experiment: we streamed continuous traffic of tcp packets while gradually increasing the traffic rate every 5 seconds (as shown in Figure 9), keep it steady for 5 seconds, and then start decreasing the traffic rate every 5 seconds. The three lines in Figure 9 represent: the ideal throughput (Ideal) which matches the send rate, the baseline firewall, and

[5]We significantly reduced the TCP connection timeout in Linux to 20 seconds, from the default of 7200 seconds.

the stateless firewall. The experiment starts with all traffic going through a single firewall. After 25 seconds, when the traffic transmitted reaches 4Gbit/sec, we split it in half and redirect it to a second firewall instance. Then after 25 seconds of decreasing the sending rate, we merge the traffic back to the first firewall instance.

As Figure 9 shows, the stateless firewall matches the base goodput. That is because the newly added firewall already has the state it needs to process the redirected packets, and therefore does not get affected by traffic redirection. On the other hand, with the baseline firewall, once the traffic is split, the second firewall starts dropping packets because it does not recognize them (*i.e.,* doesn't have state for those flows). Similarly, upon scaling in, the firewall instance does not have the state needed for the merged traffic and thus breaks the connections.

8 Discussion

The performance of our current prototype is not a fundamental limit of our approach. Here we discuss two aspects which can further enhance performance.

Reducing interactions with a remote data store: Fundamentally, if we can even further reduce the interactions with a remote data store, we can improve performance. Some steps in this direction that we intend to pursue as future work include: (i) reducing the penalty of read misses by integrating a set membership structure (*e.g.,* a bloom filter [2]) into the RAMCloud system so that we do not have to do a read if the data is not there, (ii) explore the use of caching for certain types of state (read mostly), and (iii) exploring placement of data store instances, perhaps even co-located with network function instances, in order to maintain the decoupled architecture, but allowing more operations to be serviced by the local instance and avoiding the consistency issues with cache (remote reads will still occur when the data isn't local, providing the persistent and global access to state).

Date store scalability We acknowledge that we will ultimately be limited by the scalability of the data store, but generally view data stores as scalable and an active area of research. In addition, while we chose RAMCloud for its low latency and resiliency, other systems such as FaRM [18] (from Microsoft) and a commercially available data store from Algo-Logic [1] report better throughput and lower latency, so we would see an immediate improvement if they become freely available.

9 Related Work

Beyond the most directly related work in Section 2, here we expand along three additional categories.

Disaggregation: The concept of decoupling processing from state follows a line of research in disaggregated architectures. [34], [33], and [40] all make the case for disaggregating memory into a pool of RAM. [24] explores the network requirements for an entirely disaggregated datacenter. In the case of StatelessNF, we demonstrate a disaggregated architecture suitable for the extreme use case of packet processing. Finally, this paper significantly expands on our previous workshop paper [28] with a complete and optimized implementation of the entire system and three network functions, and a complete evaluation demonstrating scalability and resilience.

Data plane processing: In addition to DPDK, frameworks like netmap [46] and Click [30] (particularly Click integrated with netmap and DPDK [12]) also provide efficient software packet processing frameworks, and therefore might be suitable for StatelessNF.

Micro network functions: The consolidated middlebox [48] work observed that course grained network functions often duplicate functionality as other network functions (*e.g.,* parsing http messages), and proposed to consolidate multiple network functions into a single device. In addition, e2 [42] provides a coherent system for managing network functions while enabling developers to focus on implementing new network functions. Each are re-thinking the architecture and complementary.

10 Conclusions and Future Work

In this paper, we presented stateless network functions, a novel design and architecture for network functions where we break the tight coupling of state and processing in network functions in order to achieve greater elasticity and failure resiliency. Our evaluation with a complete implementation demonstrates these capabilities, as well as demonstrates that we are able to process millions of packets per second, with only a few hundred microsecond added latency per packet. We do imagine there are further ways to optimize the performance and a desire for more network functions, and we leave that as future work. We instead focused on demonstrating the viability of a novel architecture which, we believe, fundamentally gets at the root of the important problem.

References

[1] Algo-logic systems. http://algo-logic.com/.

[2] Bloom filter. https://en.wikipedia.org/wiki/Bloom_filter.

[3] Digital corpora. http://digitalcorpora.org/corp/nps/scenarios/2009-m57-patents/net/.

[4] Floodlight. http://floodlight.openflowhub.org/.

[5] Microsoft Azure Virtual Machines. https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-a8-a9\-a10-a11-specs/.

[6] Palo Alto Networks: HA Concepts. https://www.paloaltonetworks.com/documentation/70/pan-os/pan-os/high-availability/ha-concepts.html.

[7] Single-root IOV. https://en.wikipedia.org/wiki/Single-root_IOV.

[8] Snort IDS. https://www.snort.org.

[9] Snort Users Manual 2.9.8.3. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/.

[10] The OpenDaylight Platform. https://www.opendaylight.org/.

[11] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 1975.

[12] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[13] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 1–6. ACM, Aug 2014.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[15] Dell. Poweredge r520 rack server. http://www.dell.com/us/business/p/poweredge-r520/pd.

[16] Dell. Poweredge r630 rack server. http://www.dell.com/us/business/p/poweredge-r630/pd.

[17] Dell. Poweredge r720 rack server. http://www.dell.com/us/business/p/poweredge-7520/pd.

[18] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414. USENIX Association, Apr 2014.

[19] Edge-Core. 10gbe data center switch. http://www.edge-core.com/ProdDtl.asp?sno=436&AS5610-52X.

[20] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.

[21] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 7–12, New York, NY, USA, 2012. ACM.

[22] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *Proc. 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, Aug 2015.

[23] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella.

OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the ACM SIGCOMM*. ACM, Aug 2014.

[24] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, Nov 2013.

[25] J. Hwang, K. K. Ramakrishnan, and T. Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 445–458, Berkeley, CA, USA, 2014. USENIX Association.

[26] Intel. Data plane development kit. `http://dpdk.org`.

[27] Intel. Ethernet converged network adapter. `http://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-x520-server-adapters-brief.html`.

[28] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[29] E. Keller, J. Rexford, and J. Van Der Merwe. Seamless BGP Migration with Router Grafting. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 235–248. USENIX Association, Apr 2010.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, Aug. 2000.

[31] J. Kronlage. Stateful NAT with Asymmetric Routing. `http://brbccie.blogspot.com/2013/03/stateful-nat-with-asymmetric-routing.html`, March 2013.

[32] T. Li, B. Cole, P. Morton, and D. Li. RFC 2281: Cisco Hot Standby Router Protocol (HSRP), March 1998.

[33] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, Jun 2009.

[34] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of 18th IEEE International Symposium High Performance Computer Architecture (HPCA)*. IEEE, Feb 2012.

[35] J. Ma, F. Le, A. Russo, and J. Lobo. Detecting distributed signature-based intrusion: The case of multi-path routing attacks. In *IEEE INFOCOM*. IEEE, 2015.

[36] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Aug 2008.

[37] Mellanox. Infiniband single/dual-port adapter. `http://www.mellanox.com/page/products_dyn?product_family=161&mtag=connectx_3_pro_vpi_card`.

[38] Mellanox. Infiniband switch. `http://www.mellanox.com/related-docs/prod_eth_switches/PB_SX1710.pdf`.

[39] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41. ACM, Oct 2011.

[40] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4), Jan. 2010.

[41] C. Paasch and O. Bonaventure. Multipath TCP. *Communications of the ACM*, 57(4):51–57, April 2014.

[42] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Oct 2015.

[43] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC)*, Oct 2013.

[44] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. ACM, Oct 2013.

[45] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Network System Design and Implementation (NSDI)*, pages 227–240. USENIX Association, April 2013.

[46] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[47] RoCE. RDMA over Converged Ethernet. `https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet`.

[48] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[49] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-recovery for middleboxes. *SIGCOMM Comput. Commun. Rev.*, 45(4):227–240, Aug. 2015.

[50] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM*. ACM, Aug 2012.

[51] tcpreplay. Tcpreplay with netmap. `http://tcpreplay.appneta.com/wiki/howto.html`.

[52] X. Yu, W.-c. Feng, D. D. Yao, and M. Becchi. O3fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems (ANCS)*, Mar 2016.

A Appendix

Algorithm 3 TCP Re-assembly

```
procedure ProcessPacket(P: TCPPacket)
    extract 5-tuple from incoming packet
    if (P is a TCP SYN) then
        record ← (getNextExpectedSeq(P), createEmptyBufferPointerList())
        writeRC(5-tuple, record)
        sendPacket(P)
    else
        record ← readRC(5-tuple)
        if (record == NULL) then
            dropPacket(P);
        if (isNextExpectedSeq(P)) then
            record.expected ← getNextExpectedSeq(P)
            sendPacket(P)
            // check if we can send any packet in buffer
            while (bufferHasNextExpectedSeq(record.buffPtr, record.expected)) do
                P ← readRC(pop(record.buffPtr).pktBuffKey)
                record.expected ← getNextExpectedSeq(p)
                sendPacket(P)
            writeRC(5-tuple, record)
        else
            // buffer packet
            pktBuffKey ← getPacketHash(P.header)
            writeRC(pktBuffKey, P)
            record.buffPtr ← insert(record.buffPtr, p.seq, pktBuffKey)
            writeRC(5-tuple, record)
```

Algorithm 4 Firewall

```
procedure ProcessPacket(P: TCPPacket)
    key ← getDirectional5tuple(P, i)
    sessionState ← readRC(key)
    newState ← updateState(sessionState)
    if (stateChanged(newState, sessionState)) then
        writeRC(key, newState)
    if (rule-check-state(sessionState) == ALLOW) then
        sendPacket(P)
    else
        dropPacket(P)
```

Algorithm 5 NAT

```
1: procedure PROCESSPACKET(P: Packet)
2:     extract 5-tuple from incoming packet
3:     (IP, port) ← readRC(5-tuple)
4:     if ((IP, Port) is NULL) then
5:         list-IPs-Ports ← readRC(Cluster ID)
6:         (IP, Port) ← select-IP-Port(list-IPs-Ports, 5-tuple)
7:         update(list-IPs-Ports, (IP, Port))
8:         writeRC(Cluster ID, list-IPs-Ports)
9:         writeRC(5-tuple, (IP, Port))
10:        extract reverse-5-tuple from incoming packet plus new IP-port
11:        writeRC(reverse-5-tuple, (P.IP, P.Port))
12:    P' ← updatePacketHeader(P, (IP, Port))
13:    sendPacket(P')
```