Stateless Network Functions

Murad Kablan† Blake Caldwell† Richard Han† Hani Jamjoom‡ Eric Keller†
†University of Colorado
Boulder, CO, USA
‡IBM Watson Research Center
Yorktown Heights, NY, USA

ABSTRACT

Newly virtualized network functions (like firewalls, routers, and intrusion detection systems) should be easy to consume. Despite recent efforts to improve their elasticity and high availability, network functions continue to maintain important flow state, requiring traditional development and deployment life cycles. At the same time, many cloud-scale applications are being rearchitected to be stateless by cleanly pushing application state into dedicated caches or backend stores. This state separation is enabling these applications to be more agile and support the so-called continuous deployment model. In this paper, we propose that network functions should be similarly redesigned to be stateless. Drawing insights from different classes of network functions, we describe how stateless network functions can leverage recent advances in low-latency network systems to achieve acceptable performance. Our Click-based prototype integrates with RAMCloud; using NAT as an example network function, we demonstrate that we are able to create stateless network functions that maintain the desired performance.

CCS Concepts

•**Networks** → **Middle boxes / network appliances;** Network experimentation;

Keywords

NFV, RDMA, InfiniBand, NAT, Stateless Architecture

1. INTRODUCTION

Running cloud-scale applications is not easy. Startups like Netflix [4] and Airbnb [1] are demonstrating how applications can be decomposed into micro—dominantly *stateless*—services that rely on backend data stores (and middle-tier caching layers) to provide the needed state on-demand. Their architecture achieves greater agility across three dimensions. First, most services can be easily scaled up or down to match incoming demand. Second, services can be developed, tested, and deployed independently from other services. Third, services can better withstand failures by cloud providers [8]. Not surprisingly, these startups could *not* apply this—stateless—design philosophy across every component in their deployment. Virtualized network functions or middleboxes (e.g., Zuul [6], Ribbon [5], and HAProxy [3]), in particular, remain stateful and pose a risk to their mostly stateless deployment.

In this paper, we reexamine network function virtualization (NFV) using a stateless design lens. Specifically, we focus on one primary question: *can the state of a network function be cleanly separated and persisted in a backend store or cache without loss of performance?* Our goal is to achieve similar agility across the three dimensions above. This would, of course, allow NFV to naturally support stateless micro-service architectures.

Initial work in the space has focused on supporting elasticity and high availability in middleboxes [20, 21]. The work focused on tagging and migrating flow state between middlebox replicas. The work kept legacy implementations and introduced additional support for inter-middlebox state management. This, we believe, introduces additional complexity into already complex (virtualized) appliances, which in turn will hinder their adoption into large microservice-based deployments. Even more, this mode does not handle failures without relying on hot standby replicas.

In this paper, we propose re-designing network functions to be stateless. We call it *StatelessNF*. In decoupling the state from the processing of network functions, we can achieve more seamless elasticity and better tolerate failures for the individual devices. Leveraging recent research on low-latency communication between servers and data stores (such as FaRM [9] and RAMCloud [17]), we show how a StatelessNF can overcome performance limitations.

We have implemented a prototype of StatelessNF in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotMiddlebox'15, August 17-21, 2015, London, United Kingdom
© 2015 ACM. ISBN 978-1-4503-3540-9/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785989.2785993

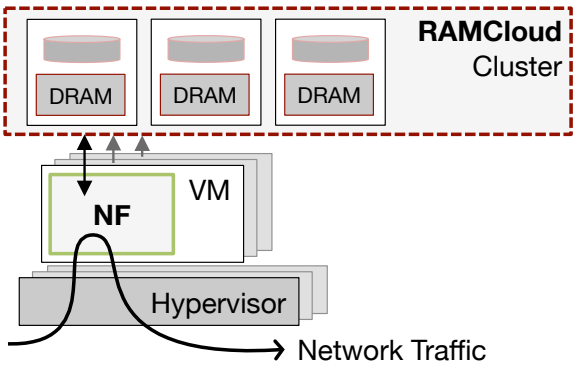

Figure 1: StatelessNF architecture

Click [14] that integrates with RAMCloud. We used NAT as an example network function and changed the default Click implementation in a way that the NAT translation table is maintained in a RAMCloud cluster. We have implemented three modes of operation. In the most restrictive (blocking) mode, Click elements are only allowed to store configuration state. Additionally, they will block on any lookup in RAMCloud. We have also implemented an asynchronous mode where reads and writes are non-blocking. This allows each element to maintain better throughput. Finally, we also implemented a caching element that allows fast reads; any writes are immediately passed through to RAMCloud. Using InfiniBand as a low-latency transport medium between Click and RAMCloud, we have evaluated the performance of StatelessNF across all three modes of operation. Our experiments demonstrate that StatelessNF (with caching) can match the native Click implementation for a broad spectrum of configurations. Even without caching, both throughput and latency were less than 10% below the native Click implementation.

2. ENVIRONMENT & EXAMPLE

Without loss of generality, we assume an environment that consists of a cluster of virtualized network functions (or middleboxes). We also assume that flows are externally assigned to an NF instance by an SDN controller. In the common case, flows use the same NF instance throughout the lifetime of each flow. They, however, can be reassigned to different instances (for example, on failure or to improve load balancing of flows across replicas).

As mentioned earlier, we are interested in designing stateless network functions that can match the performance of existing implementations, but are simpler to manage and scale. In designing our system, we consider the performance requirements of three different classes of network functions: routing software, intrusion detection systems (IDSs), and inline processors (such as firewalls and NATs). We use NAT as a running example of a network function throughout the paper for three reasons. First, NATs have a simple design that allow us to reason about the various design decisions. Second, unlike routing software and IDSs, NATs are both delay and throughput sensitive. We wanted to capture the effect of our design on both performance dimensions. Third, NATs have a robust implementation in Click, which simplifies our implementation prototype.

For simplicity, we assume that the NAT maintains a translation between a packet's public and private address. This is captured in the traditional 5-tuple entry: *Protocol, Private Address, Private Port, Public Address, Public Port*. The NAT also maintains a unique flow ID. In going stateless, this mapping table is maintained externally by all NF instances. When a new flow arrives, an NF instance must first lookup to see if a table entry exists in the global table. If one exists, then it will use the information from the existing entry. If it does not, it will create a new entry for the flow and persist the information back to the global table.

3. ARCHITECTURE

Figure 1 shows the architecture of StatelessNF. It consists of a network processing tier and a data store tier.[1] Network functions (as is the premise of NFV) run within isolated and deployable units such as virtual machines or containers. Each of the network functions is able to communicate with the data store through a data network. There are two key challenges to our design:

- *Decoupling state:* Traditional network functions have been designed with state directly embedded into their code. StatelessNF requires redesigning the network functions to decouple the state.
- *Achieving performance:* Many network functions will sit in the data path, and be required to process a large number of packets per second. We have to ensure that efforts to improve performance do not compromise the consistency of the state or cause important state to be lost upon failure.

We elaborate on these two challenges and how StatelessNF overcome them in the following subsections.

3.1 Decoupling State in Network Functions

Decoupling state from computation is integral to many operations in cloud systems; it spans the entire system and network stacks. For example, techniques like Checkpoint/Restore In Userspace (CRIU) [18] are used to extract out kernel and process state for process/container migration. At a much finer granularity, Split/Merge [21] allows movement of per flow state across middlebox replicas. Similarly, Router Grafting [13] migrates BGP session state across routers.

A common design theme among these works is that the decoupling of state only occurs during management

[1]Note, the current design consists of two tiers; as security requirements grow, we anticipate the creation of other tiers that process information in the background.

operations. The decoupled (extracted) state, for example, can be migrated to another replica or periodically persisted to a backend store (to implement failure recovery). The rest of the time, state remains internal to running systems. By keeping the state internal to the network function, these designs optimize for the common case: processing performance. At the surface, this would appear as the right design point. Practically, because state extraction is hard, especially when the state spans the network and kernel stacks, such systems are bug prone and hard to deploy in production environments. They also do not support graceful upgradablity as required by microservice-based deployments.

In StatelessNF, we argue that state separation should be a first order design principle. We separate network function state into two types.[2] The first type is *cache* state. It is any state that is reproducible by a single network function instance. The second is *shared* state. It includes state that all network function replicas can access. It can be thought of a collection of key-values without any prescribed structure or schema to each value. In the case of a NAT, for example, the key is the flow ID and value is the address mapping.

When a new instance is added or removed, data consistency needs to be handled properly. In the NAT example, all address mappings from a failed instance need to be persisted so that the new instance can ensure correct fail over. Each network function instance loosely acts as a master for the flows that are assigned to it. The only programming practice that is required is for each instance to wait for any update to the flow entry, or creation of a new entry, to be written to the backend store (RAMCloud in our case). However the network function instance does not need to wait for the backend store to replicate the new data.

3.2 Achieving Performance

In going stateless, an added latency is naturally introduced: reading from remote memory versus local will always be slower. The questions then become (1) whether it actually impacts performance to a noticeable degree, (2) whether the latency can be tolerated even if it is noticeable, and (3) whether the latency can be masked.

As first step, we reduce part of the expected overhead by leveraging low latency distributed systems and networking technologies. Systems such as FaRM (fast remote memory) [9] and RAMCloud have shown the ability to create systems for which the memory is on a remote machine. Next, we introduce a caching layer.

Caching introduces another layer where consistency can be violated. In the simplest case, techniques like consistent replication and cache invalidation can eliminate such consistency issues (at the cost of higher update cost). We believe that there are domain specific solutions. For example, we can exploit the situation where a given packet may traverse a sequence of network functions. So long as the packet does not reach the end destination before the affected state is committed to the data store, we have opportunity to rollback upon failure if needed.

[2]Split/Merge separates middlebox state into three types: internal, coherent, and partitioned state. In our design, we eliminate the need to differentiate between coherent and partitioned state.

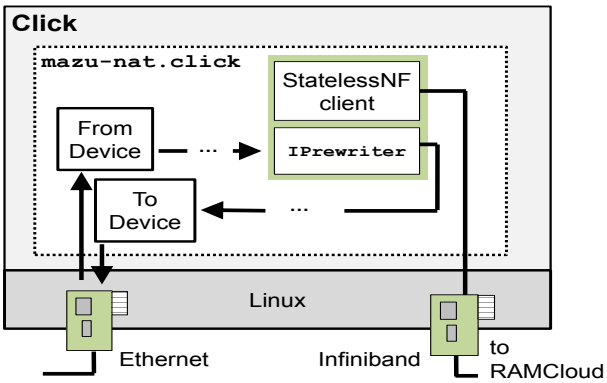

Figure 2: StatelessNF Initial Prototype

4. PROTOTYPE IMPLEMENTATION

Our initial prototype, illustrated in Figure 2 is built using two key technologies: (i) Click, as a platform to build new data plane network functions, and (ii) RAMCloud over InfiniBand, as a platform which provides a low-latency access to a data store.

The central component to our implementation is the StatelessNF client. This StatelessNF client is a class implemented in C++ that carries out data storage and retrieval operations to RAMCloud. It also provides a general data store interface that can be integrated into Click elements and linked at compile time. We implemented different versions of this interface to explore different designs, detailed in the following subsections.

In order to take advantage of RDMA transactions for one-sided writes to RAMCloud and minimal read latency, we used the *infrc* transport mechanism in RAMCloud in preference over the *tcp* transport mechanism that is compatible with an Ethernet fabric. This transport implementation uses the reliable connected (RC) InfiniBand transport. The initial connection handshake between a client and a RAMCloud server is done through a socket interface using IP over InfiniBand, which encapsulates an IP packet in InfiniBand frames. Once a connection has been established between client and server, memory regions are pinned by the InfiniBand device driver for DMA transfers bypassing the OS, and further communication is transitioned to the InfiniBand verbs interface.

4.1 Blocking Read/Write

The StatelessNF client provides a read/write interface to a datastore. To minimize changes in Click elements, our first implementation of this functionality is a synchronous (blocking) interface. Typical elements in

Click might have a lookup in a data structure (such as a Hashmap). These lookups are performed as part of the sequentially executed code to process the packet. As a local data structure, this is perfectly reasonable. For our first implementation, we simply replace these reads and writes with blocking reads and writes to RAMCloud. That is, the Click element submits a lookup to the storage client interface and does not output the packet until the RAMCloud operation completes, with read data or confirmation of a successful write.

4.2 Async Read/Write—Buffer Packets

As an improvement, we also implemented an asynchronous (non-blocking) read and write interface to the StatelessNF client (using the asynchronous interface to RAMCloud). To utilize this, the Click element needs to break the packet processing into pre-lookup and post-lookup functionality (or more generally, into processing stages if there are multiple reads, but for simplicity we focus on the single read case), along with integration of the Click scheduler. Consider an example element which has a *push* interface, where the `push()` function extracts some fields in the packet, performs a lookup, and based on the results of the lookup modifies the packet and outputs on a given port of the element.

Replacing the lookup with a read to our StatelessNF client requires breaking apart the function. Instead in its `push()` function, the element extracts fields in the packet to be used for a lookup, submits the read request to the StatelessNF client, and then returns from the push function. We periodically check inside `run_task()`, invoked by the Click scheduler, whether any results have completed. If they have, we call the element's post-processing function, which in turn outputs the packet.

4.3 Caching

Finally, we extended the StatelessNF client to support a simple least recently used cache. By simple, we mean that for our initial exploration, we assume state that can be cached will not be shared.[3]

The first change in the StatelessNF client is that the element looks in the cache for a matching data, and if found, returns the value immediately. Then for writes, it will update the cache and perform a write through to RAMCloud. If the data is not there, a read to RAMCloud, and then possibly a write to RAMCloud is performed, and execution returns to the calling element.

4.4 NAT Example

We modified a NAT network function to be stateless. In particular, we used the `mazu-nat.click` configuration that is in the Click source tree. In it, the `IPRewriter` element is the element that stores the state for a NAT. In particular, it uses a `flowID` built from the received packet's 5-tuple as a key into a table, and stores the address-translation mapping as the value.

This follows the processing model described in the previous section: extract lookup key, perform a lookup, based on the result modify the packet (*e.g.,* change the 5-tuple), and output the packet. One added step is that if the lookup fails, signifying a new flow, a new mapping must be added. We assume (for simplicity) that each instance has a partition of the external mapping space. The element then writes to the StatelessNF client this new entry in the table and outputs the packet.

5. EVALUATION

In this section, we evaluate the performance of stateless NAT with integration to RAMCloud in three modes: (1) Blocking Read/Write (Sync), (2) Non-blocking Read/Write (Async), and (3) Cache. We compare these three modes and a Click Native implementation of NAT on the basis of three metrics: throughput, round trip time, and packet processing overhead.

Setup. Our evaluation consists of one machine running StatelessNF implemented in Click, a two node RAMCloud cluster, a traffic generator, and a traffic sink. The generator and sink each had a single 1 Gbit/s interface. The NFV machine had two 1 Gbits/s interfaces. A 10 Gbit/s SDR InfiniBand network connected the server running StatelessNF to the RAMCloud cluster. The node running StatelessNF runs Click in userlevel mode on the host OS with each NFV application having direct access to resources on the InfiniBand device.[4]

We configured RAMCloud such that one replica is sent to a backup process on a server different from where the master process resides, with the primary copy in DRAM. The backup process will return a write completion when it has stored the data in an 8MB buffer in DRAM. When a backup's buffer becomes full, the backup will flush its entire contents to disk in a large sequential write operation. Thus, by the time the client is notified of a write RPC completion, a copy has been written to DRAM by the master and a second copy has been written to the backup's buffer on a different server. In this RAMCloud configuration the crash of a single server is recoverable without data loss to StatelessNF.

Benchmarking InfiniBand. Our RAMCloud cluster made use of early generation SDR InfiniBand running at PCIe gen. 1 speeds (2.5 GT/s). Table 1 shows its performance measured by clusterperf, a benchmark provided with RAMCloud. While SDR made for a functionally complete StatelessNF implementation, significant performance improvements are possible should the RAMCloud servers and NFV servers utilize recent FDR

[3]We fully acknowledge the more difficult aspect of caching is support for consistency and cache invalidation, which we intend to implement as a next step.

[4]Alternatively, techniques utilizing an system's IOMMU such as PCIe passthrough and SR-IOV [19] can share these resources with one or more virtual machines. While the possibility of virtualizing each StatelessNF could be useful, our InfiniBand cards were not capable of PCIe passthrough or SR-IOV.

Table 1: RAMCloud RPC performance (100 byte data)

	SDR IB 1 replica	SDR IB 0 replicas	FDR IB 0 replicas [7]
Read lat. (μs)	29.7	29.8	3.0
Write lat. (μs)	56.5	30.4	4.3
Read BW (RPC/s)	33,974	32,506	318,767
Write BW (RPC/s)	17,616	32,086	230,686

Table 2: Click TCP & UDP Throughput for 10 parallel connections

	Click Native	Stateless Cache	Stateless Async	Stateless Sync
TCP (Mbit/s)	924.8	925	894	335.2
UDP (Mbit/s)	955.6	954.8	955.4	519.6

InfiniBand hardware and PCIe gen 3 (8 GT/s) buses. The results reported in [7] indicated that read latencies can be reduced by 25 μs with FDR InfiniBand.

Throughput. Using Iperf [22] we measured the maximum UDP and TCP throughput of each implementation and that of Click native code. We used a packet size of 1500 bytes, and averaged the results of 5 iterations. The number of parallel streams was varied from 1 to 100. Each stream was generated with a different source port and therefore resulted in a unique NAT ranslation.

As shown in Table 2, the Cache implementation is on par with the performance of Click Native. There is a slight drop in performance with the Asynchronous case, but much less than with the blocking alternative. We noticed a drop in performance only with the Asynchronous implementation as the number of parallel connections increased, which we believe is the result our implementation's inefficiency in handling outstanding RPC's belonging to different flows. Improving on this is an opportunity for future work.

Round Trip Time. We measured round trip time (RTT) with D-ITG [2] where we streamed 10,000 minimum size UDP packets through the NAT. We can see in Table 3 that Cache gives us the lowest latency among all designs. The reason all of our designs have lower RTTs than Click Native is that our implementations each utilized the Click scheduler, which was necessary for asynchronous processing. The scheduler sets a timer that waits 10 μs between each poll cycle. In the Asynchronous case the scheduler's `run_task()` function will check the status of a RAMCloud RPC.

Packet Processing. We measured packet processing overhead for different packet sizes. In order to generate a high packet rate, we ran a Click UDP generator in kernel space. We streamed one million packets and increased the rate until the traffic sink saw packet drops. As shown in Figure 3, the Cache implementation meets the performance of Click Native at a packet size 500 bytes while the Asynchronous implementation joins the

Table 3: Round Trip Time

	Click Native	Stateless Cache	Stateless Async	Stateless Sync
RTT (μs)	371	317.2	356	332.1

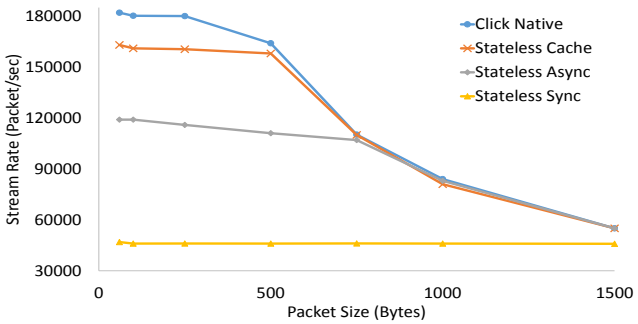

Figure 3: UDP Max Transmission Rate Before Loss

performance of Click Native at 750 bytes.

To investigate reduced Asynchronous performance as compared to Cache, we profiled our modified Click code at a frequency of 25,000 Hz during a UDP Iperf test. The Async implementation spent 20.2% of the sampled cycles in the storage client library's `read()` function compared to 4.6% with the Cache implementation. The `read()` function will either create a RAMCloud read RPC or lookup the flow in the cache, in Async and Cache cases respectively. As a result, the Async implementation is only spending 2.7% of its time in the `poll()` system call waiting for incoming packets, where Cache is able to spend 17.6% of sampled cycles in `poll()`, meaning faster packet processing.

6. DISCUSSION

We performed our initial exploration with Click running on bare metal in user space due to implementation limitations with InfiniBand and the RAMCloud interface. Ideally, we want to run in a fully-virtualized environment and use a lighter weight mechanism like ClickOS [15] or NetVM [11]. Our goal is to match the performance of Click in kernel mode. The main use case of InfiniBand comes from user space reads over the network from the backend store. As such, with the development of high performance (*e.g.,* zero copy) user space I/O technologies over the past few years, our focus will be on getting Click in user space to run faster. In this way we can continue to use InfiniBand in userspace for easy use of backend stores.

7. RELATED WORK

StatelessNF rearchitects network functions in a way that maintains their internal state in a separate low-latency storage tier. The structure of middleboxes has been characterized in [12]. The closest work to StatelessNF are the works seeking to create more elastic network functions through state migration. In partic-

ular, Split/Merge [21], OpenNF [10], and Pico Replication [20] for middleboxes and Router Grafting [13] for router software. Olteanu and Raiciu [16] similarly attempt to migrate per-flow state between VM replicas without application modifications. These works migrate or replicate state, whereas we seek an alternative architecture which decouples the state from the processing.

Going stateless allows for easier scalability and high availability. There are many ways in which different types of applications are dynamically scaled in the cloud [24]. Clustering techniques have traditionally been used to scale-out middleboxes. The NIDS Cluster [23] is capable of performing coordinated analysis of traffic, at large scale. Similar to other clustering techniques, NIDS is a specialized implementation. StatelessNF is proposing a generic architecture that matches those in microservice-based applications.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented StatelessNF, an architecture that uses recent advances in low-latency systems to decompose network functions into a packet processing tier and a data storage tier. We believe that our work is a good first step towards demonstrating the potential of a stateless architecture in network functions. As for future research, we plan to improve implementation efficiency and test our implementation under higher throughput (*e.g.,* 10-40Gbit/s). We also plan to explore more network functions, including off-path intrusion detection systems and control plane routing software.

9. ACKNOWLEDGEMENTS

This work was funded in part by the following grants: NSF NeTS 1320389 and NSF XPS 1337399.

10. REFERENCES

[1] airbib SmartStack. http://nerds.airbnb.com/smartstack-service-discovery-cloud/.
[2] D-ITG, Distributed Internet Traffic Generator. http://traffic.comics.unina.it/software/ITG/.
[3] HAProxy. http://www.haproxy.org.
[4] Netflix Open Source Software. http://netflix.github.io/#repo.
[5] Netflix Ribbon. https://github.com/Netflix/ribbon.
[6] Zuul. https://github.com/Netflix/zuul/wiki.
[7] Carreira, Joao. [ramcloud-dev] Replicating Ramcloud results (latencies). https://mailman.stanford.edu/pipermail/ramcloud-dev/2014-December/001033.html, December 2014.
[8] A. Cockroft, C. Hicks, and G. Orzell. Lessons Netflix Learned from the AWS Outage. http://techblog.netflix.com/2011/04/lessons-netflix-learned-from-aws-outage.html, April 2011.
[9] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2014.
[10] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proc. of ACM SIGCOMM*, Chicago, IL, Aug. 2014.
[11] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of USENIX NSDI*, Seattle, WA, Apr. 2014.
[12] D. A. Joseph and I. Stoica. Modeling Middleboxes. *IEEE Network*, 22(5), 2008.
[13] E. Keller, J. Rexford, and J. Van Der Merwe. Seamless BGP Migration with Router Grafting. In *Proc. NSDI*, 2010.
[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. M. Kaashoek. The Click Modular Router. *ACM Transactions on Compuer Systems*, 18:263–297, August 2000.
[15] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of USENIX NSDI*, Seattle, WA, Apr. 2014.
[16] V. A. Olteanu and C. Raiciu. Efficiently Migrating Stateful Middleboxes. In *ACM SIGCOMM - Demo*, 2012.
[17] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc.*, 2011.
[18] OpenVZ. Checkpoint/Restore in Userspace. http://www.criu.org. Accessed: 2014-03-14.
[19] PCI SIG. Single Root I/O Virtualization. https://www.pcisig.com/specifications/iov/single_root/.
[20] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. of ACM SoCC*, Santa Clara, California, Oct. 2013.
[21] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of USENIX NSDI*, Lombard, IL, Apr. 2013.
[22] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf, May 2005.
[23] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. of International Conference on Recent Advances in Intrusion Detection*, 2007.
[24] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically Scaling Applications in the Cloud. *ACM SIGCOMM CCR*, 41(1).