StatelessNF: A Disaggregated Architecture for Network Functions

by

Murad Kablan

B.S.C.S, University of Benghazi, 2006

M.S.C.S, Worcester Polytechnic Institute, 2012

A dissertation submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2017

This dissertation entitled:
StatelessNF: A Disaggregated Architecture for Network Functions
written by Murad Kablan
has been approved for the Department of Computer Science

_____
Prof. Eric Keller

_____
Prof. Sangtea Ha

Date _____

The final copy of this dissertation has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Kablan, Murad (Ph.D., Computer Science)

StatelessNF: A Disaggregated Architecture for Network Functions

Dissertation directed by Prof. Eric Keller

The networking space is undergoing a transformation. Today's networks (enterprises, data centers, and service providers) are replacing the expensive hardware network appliances (*e.g.,* firewalls, load balancers, security monitors) with software that can run on commodity servers. This movement significantly reduces capital expenditures, as these devices are an important component of today's networks.

While this general movement shares a similar vision of a dynamic network, existing solutions fall short to adapt in both scale and function to provide true network agility where network services can be instantly deployed, scale in and out on demand, and be resilient to failure. The reason is that these solutions are simply modifying existing network device designs to have the device run in software. This design decision is the root of this problem. Specifically, that the network functions that make up the network infrastructure (*e.g.,* firewalls, load balancers, and routers) despite their variety, share a common monolithic architecture where the control plane, data plane, and most importantly the state (*e.g.,* information about network flows) are maintained internally in a single appliance. This tight coupling of all of these components within network functions hinders the agility. Thus, leading to complex and expensive solutions to manage and configure these network functions on a larger scale.

This dissertation is centered on overcoming these limitations through the introductions of what we call Stateless Network Functions, or StatelessNF. StatelessNF aims to provide self-managed and configured agile network functions that are resilient to failure and able to scale in and out without intervention from network administrators. Thus, enabling the network to be invisible to the applications that run on top of it.

To achieve this goal, we have fundamentally re-designed the architecture of network devices with the flexibility of software in mind, rather than other software approaches that mimic the hardware architecture in software, and therefore inherits a design that was designed around the limitations of hardware, not the flexibility of software. The key idea with StatelessNF is to decouple the processing of a network device

from its state, storing any needed state in a separate data store. In separating the state and the processing, we enable a highly dynamic management of scalable and failure resilient network functions.

StatelessNF's architecture is especially challenging to realize in the context of network processing, where we need to handle millions of packets per second and where there may be one or more reads and writes to the data store for every packet.

In this dissertation, we ask **"Can we redefine and redesign network functions' architecture from the ground up, without sacrificing performance, to achieve true network agility?"**. Overcoming this challenge was one of the main contribution in this dissertation.

As we show in this dissertation, we envision StatelessNF being the core of multiple future research projects in the area of software defined networks (SDN) and Network Function Virsualization (NFV). In addition, StatelessNF has broader impacts that include commercialization opportunity. We validated this through our interaction with over 150 companies in cloud infrastructure and data center industries and two companies that agreed to deploy StatelessNF system in their data center environments.

Dedication

I dedicate this dissertation to my beautiful and amazing wife and best friend, Sabria, I could not do this without your support and love. To my baby girl, Summer. You are my joy in this life even when you drive me crazy with your stubbornness. To my newborn son, Adam. I know I will love you so much, and I know that you will drive me crazy as well.

Acknowledgements

This document could not have been completed without the assistance of a variety of individuals. First, I would like to express my special gratitude to my PhD advisor Professor Eric Keller for his guidance and unlimited support and for keeping me motivated throughout my five years at CU. Professor Keller impressed me with his broad and deep knowledge of the area of computer networks and systems. His influence led me to choose SDN and NFV as my research focus. Second, I would like to thank all of my committee members for their oversight, suggestions, and time. Dr. Hani Jamjoom provided suggestions and assistance on a variety of topics including the original StatelessNF paper. Prof. Richard Han, Prof. Sangtea Ha, and Prof. Shivakant Mishra all provided suggestions and answers a variety of technical questions. Their insight, knowledge, and guidance were invaluable. Third, a big thanks to my systems-lab mates and friends - Oliver Michel, Azzam Alsudais, Matthew Monaco, Anurag Dubey, and Michael Coughlin. It was pleasure working with you all on a variety of different projects.

And finally, to my mom, your prayers, unconditional support, and belief in me helped me to start and finish my PhD.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

1.1 The need for network agility

As evident by their proliferation, network functions are an important component in today's enterprise, cloud, and telecom networks. Recent studies have shown networks contain hundreds to thousands of physical and virtual network functions, like firewalls and load balancers which are as prevalent in networks as routers[116]. Network functions provide datacenter and enterprise network operators with an ability to deploy new network functionality as addon components which can directly inspect, modify, and block or re-direct network traffic. This, in turn, can help increase the security and performance of the network.

With Network Functions Virtualization (NFV), the expensive hardware network appliances are replaced with software that can run on commodity servers. This movement significantly reduces capital and operating expenditures and increases deployment flexibility, as these devices are an important component of todays networks. Ninety-three percent of the IT industry is moving toward software based data networks [2], and the market size for this industry is reaching $1.1 Billion in the United States and $5 Billion worldwide [15].

However, networks still need agile network functions to match the agility (or the ability to quickly and easily move) of the rest of the infrastructure that the cloud architecture has provided. We are now able to launch applications in a matter of seconds and adjust the amount of storage that we have on demand, instantaneously as applications need it. But, for the networks functions that connect and protect these applications and make them run efficiently, such agility does not exist.

For networks to match the definition of agility, network functions need to meet three requirements:

- **Seamless scalability:** The ability to scale in and out without disrupting network traffic.
- **Failure resiliency:** The ability to detect and recover from failure, again, without disrupting network traffic.
- **Instant deployment:** The ability to instantly launch network functions, and configure their scalability and failure resiliency.

However, the network infrastructure (in the form of virtualized network functions, end-host network stacks, or via cloud provider APIs), fails to live up to the same elasticity, ability to handle failures, and ability to adapt to new interconnections with new and modified applications as of the rest of the infrastructure. Simply put, the network fails to adapt in both scale and function to the dynamic changing behavior of cloud applications.

The root of this problem is that, even with software approaches, we are still stuck with the appliance mentality when it comes to networking. Software approaches are simply mimicking the hardware architecture in software, therefore inherit a design that was designed around the limitations of hardware, not the flexibility of software. In other words, we are left with the same core architecture with the same fundamental limits.

1.2 Getting stuck with the appliance mentality

While traditionally deployed as physical appliances, with the introduction of Network Functions Virtualization (NFV) network functions such as firewalls, network address translators, routers, and load balancers no longer run on proprietary hardware, but do so in software, on commodity servers, in a virtualized environment. Moving away from fixed physical appliances holds the promise that the network can achieve the ability to elastically scale the network on demand and quickly recover from failure. Rather than live on underutilized devices (*e.g.,* 5-20% [116]), which in many cases can still become overloaded to the point of failure [116], a NFV-based infrastructure can simply be scaled in and out as needed. Rather than have failure disrupt business (costing lots of money [61]), networks can simply have new instances launched and traffic re-directed to them.

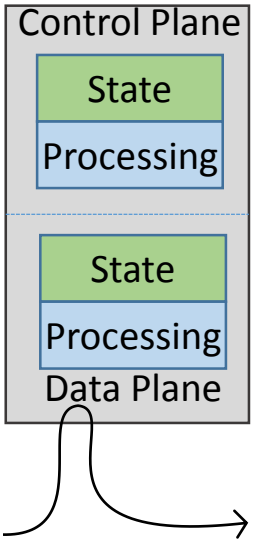


Figure 1.1: Traditional Network Function (Appliance Design)

But, as others have pointed out, it is not that simple [107, 108, 58, 115]. The central issue revolves around the fact that by having an appliance design, a monolithic architecture is created where the process and state are locked into the network functions as shown in Figure 1.1. State such as BGP sessions and MPLS label distribution in which case has scalability limitation. And state such as connection information in a stateful firewall, address mappings in a network address translator, or server selection mappings in a stateful load balancer. In each case, any packet for a given flow of traffic needs to go through the same network function instance every time. This tight coupling limits the elasticity, failure resilience, and ability to handle other challenges such as asymmetric/multi-path routing and network function software updates.

1.3 StatelessNF - Think differently

In order to overcome the limitation in existing network functions to provide agility, we realized that we could not fix this problem using the same old design and approach to networking. Instead, we looked at the architecture of other cloud-scale application that adapted their design principles. Companies like Netflix and Airbnb 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-

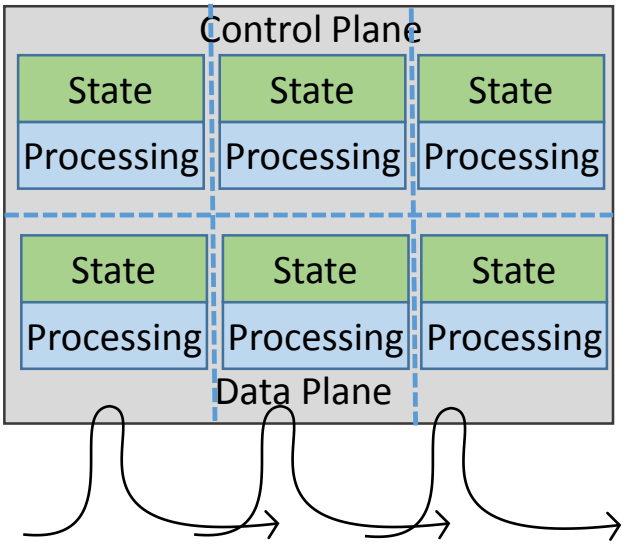


Figure 1.2: EdgePlex Design Principle (Breaking apart the processing)

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 the underlying infrastructure [14].

To achieve such great agility in networking, we need to break the monolithic design of network functions and break apart both the process and state. By doing so, network functions can be scale out as opposed to scaling up, recover quicker from failure events, and can add new features and functionality as opposed to scheduling disruptive maintenance windows to perform such as updates.

Breaking apart the processing:

Our first step towered agile design is EdgePlex [47], a new edge router architecture following SDN and NFV principals. EdgePlex is a distributed system where the process, as shown in Figure 1.3, is partitioned and the functions performed by a traditional edge router are decomposed and assigned to different elements in the system.

The opportunity we leveraged to demonstrate such architecture is that routing protocols and sessions can be partitioned across multiple instances. This unique architecture allows scaling from the core network

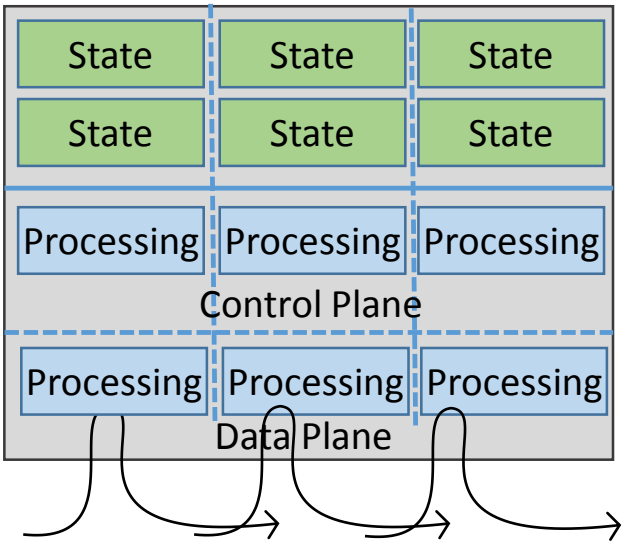


Figure 1.3: StatelessNF Design Principle (Breaking apart the state)

perspective while allowing for routers connections to be migrated with minimal impacts.

However, as we will show in this thesis, EdgePlex network functions were limited as while we partition the process among different instances, the state is still coupled with the process within the instances. Thus, their failure resilience and ability adopt new applications and updates is not optimized.

Thus, our second step is decouple and break apart the state and introduced **stateless network functions**

Breaking apart the state:

To demonstrate the potential of a stateless architecture in network functions, we present 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.

Our StatelessNF architecture is inspired by the design of other applications, such as web services, that are designed to be highly resilient and elastic through leveraging stateless components coupled with a resilient data store.

With stateless network functions, the network application (*e.g.,* a stateful firewall) consists of instances which perform the processing and a remote data store which stores all of the state.

In addition, by breaking the network functions into multiple components that exchange data among them, such as reading and writing to state, one may think that will double the amount of network traffic in the infrastructure. As we show in Chapter 4, there is not a lot of back and forth with a data store, it is simply a look up stage in a processing pipeline.

This fundamental shift in design results in high scalability and great failure resiliency that network operators badly need. In addition, this unique architecture brings with it complete agility as new network applications can be deployed in seconds, as opposed to days or weeks. and software updates are seamless, as opposed to scheduling disruptive maintenance windows to perform updates.

In addition to the overall StatelessNF properties of failure resilience and elasticity, our StatelessNF architecture provides the following properties: (i) We do not rely on affinity to ensure proper network operation, but instead any packet can go to any instance. (ii) We have a zero failover cost, where upon failure we can send traffic to a new instance without any penalty such as causing existing flows to fail, taking time to recover the state, etc. This in fact allows us to speculatively fail over, rather than conservatively only fail over when we are sure of failure. Thus, significantly reducing failure detection time.

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).

1.4 Is it even possible?

A main challenge with adopting a disaggregated architectures such as EdgePlex and StatelessNF is achieving acceptable performance levels. By breaking the network functions into multiple components that run on commodity servers, the exchanged data among them will be increased and the system may become a bottleneck. However, recent advances in high-performance I/O, virtualization, and data stores enabled us to break through the performance challenges by integrating such technologies with our implemented custom software. Here we highlight one some of these technologies we leveraged to achieve our design goals.

Data plane processing: Previous work and library tools for software packet processing form the basis of many virtualized network elements. Click [19] and RouteBricks [12] are among the early research endeavors

to enable software packet processing. PacketShader [14] exploits the parallel processing capability of GPU to accelerate packet processing in software routers. Recently, technologies such as Intels DPDK[66], PFQ [3], PF RING [4], and Netmap[110] have emerged. These provide programming libraries to build fast network I/O applications. Open vSwitch[87], Hyper-Switch [21] and VALE [23] build software switches in a hypervisor to create an Ethernet network among VMs.

Virtualization: Virtualization is the enabling technology behind the success of cloud computing and network function virtualization. In popular solutions, such as KVM [18], Xen [9] and VMWare [7], a hypervisor runs on servers and allows for the creation of multiple VMs, each running a separate instance of operating system. Multiple VMs can share the network devices on the physical host with I/O sharing technologies such as SR-IOV[22] and VMDq [6]. To bring virtualization to the network, NetVM [15] builds mechanisms across a KVM hypervisor and VMs to reduce data copies and locking overheads in inter-VM communications. ClickOS[93] provides a platform for building small middlebox VMs using Click in Xen hypervisor. Both Cisco and Juniper have virtualized instances of their physical routers [10, 16] using many of these recent capabilities.

Containerization: Another form of isolation, in addition to virtual machines, is containers. The primary advantage of containers over hypervisor-based virtualization is capacity utilization, much faster deployment, and native performance. This is because unlike VMs, containers do not bundle a full operating system - only libraries and settings required to make the software work are needed. The most leading software container platform is Docker[5]. It automates the repetitive tasks of setting up and configuring development environments. This makes docker an attractive platform for network applications where network services can be deployed instantly while having native performance. VMs are typically heavier and slower than containers, but because they are fully isolated, VMs tend to be more secure than containers' process level isolation. However, recent advances in containerization technologies show that the VMs security advantage over VMs will decrease.

High performance data stores: Be decoupling the state from processors, an overhead is added, so we need to make sure that the state can be retrieved and accessed in the shortest amount of time possible (*e.g.,* usec).

In addition, the data store has to be reliable and resilient in case any of its node fails. There has been much work in data stores, specifically low-latency. For example, FaRM [51], is a shared address space, symmetric cluster which uses RDMA for efficient transfers. Algo-Logic [1] provides an FPGA based key-value store with very low latency (less than one usec) and processing energy. In addition, RAMCloud [96], an open source project from Stanford, provides a resilient and distributed high performance data store with read and write operations that can be completed in a few micro seconds.

With EdgePlex, we used a combination of open source and custom software in our implementation. We used KVM as the hypervisor and ran multiple VMs on a single server. We used OpenBSD in one type of VMs that handles control plane. OpenBSD has in-built support for MPLS and for other routing protocols like OSPF and BGP. We configured OpenBSD appropriately to connect with route reflectors and other core network routers. We used GNU/Linux running Quagga and a custom packet forwarding application (called FastForward) in another type of VMs that handle data plane. Quagga handled individual customer BGP sessions and exchanges control plane between the router customer and the control VMs. FastForward was responsible for forwarding data packets. It uses Intels DPDK [1] for fast packet processing and Cuckoo hash library [26] for route lookup.

For StatelessNF, we specifically focus on the network functions that are in-line middleboxes that process network traffic. Our system to achieve the performance is centered on an efficient packet processing pipeline that leverages recent advances in high-performance I/O (DPDK), and is packaged as a container (Docker) for easy deployment. We introduce an optimized data store client interface to reduce and ultimately mask the added latency a remote lookup adds, and leverage advances in low-latency data stores (RAMCloud).

1.5 Thesis overview and summary

In this thesis, we present a novel disaggregated design for network functions to provide true network agility. We use a distributed architecture composed of commodity servers and switches and takes advantage of advances in virtualization and high speed packet processing to replicate the functionality in today's traditional network appliances. Our results lead us to conclude that our design principles not only promises

significant operational benefits over existing network functions, it is also viable and capable of meeting the performance needs of service providers.

Chapter 2 represents our first step to solve some of the network agility challenges in the router space. We describe EdgePlex, a novel approach for service provider edge router functionality we designed and implemented in collaboration with AT&T research labs. In this collaboration, we received an industry insight into the network function space. Chapter 3 motivates the need for stateless network functions by describing network state and limitations in existing solutions that deal with state. Chapter 4 describes the StatelessNF architecture and how we were able to overcome the challenges with performance when we decoupled the state from processing. To further support our claims of the importance of this work, in Chapter 5 we describe our commercialization effort for StatelessNF and current case studies and deployments. And finally, in Chapter 6 we describe our future directions and work to optimize the management and performance of StatelessNF system.

Chapter 2

EdgePlex: a new approach toward providing agility in routers

The goal of this thesis is to present a novel disaggregated design for network functions to provide true network agility. As a first step, we explore partitioning the processing component of network functions. We look at provider edge routers as an example and as a part of our research project with AT&T research toward virtualizing their network infrastructure.

The service provider edge is responsible for connecting customers using standard protocols such as IP and BGP to the service providers internal network while enforcing service specific policies and service guarantees. Today this function is performed by the Provider Edge Router (PE). The specialized nature of the PE, however, restricts operational flexibility and their monolithic design impacts reliability. In this chapter, we propose a new edge router architecture following SDN and NFV principals called EdgePlex. EdgePlex is a distributed system where the functions performed by a traditional edge router are decomposed and assigned to different elements in the system. A key aspect of our design is the use of a sandboxed environment (through the use of virtual machines) per customer. This gives EdgePlex the ability to isolate customers from one-another and independently move customers within and across EdgePlex platforms. We describe the architecture and a prototype implementation of EdgePlex. We perform detailed experiments using this prototype and show that EdgePlex is able to saturate the server in terms of throughput while having acceptable latency and jitter overheads. Our results lead us to believe that the EdgePlex design not only addresses the limitations of existing routers, but is also viable and can meet performance demands of production networks.

2.1 It is the time to modernize IP networks

IP networks have traditionally been designed and architected such that the core of the network performs simple tasks quickly, while pushing the more complex tasks to the edge. This allows for the core to perform fast packet forwarding (e.g., using Multi-Protocol Label Switching (MPLS)) while the edge aggregates traffic, exchanges control plane information, performs routing, and supports other features demanded by the services (access control, Quality of Service (QoS), etc). To enable this, service providers use specialized, physical devices known as Provider Edge (or PE) routers that allow customers to connect to the service providers network. A single PE router aggregates traffic from hundreds of customers and funnels the traffic into the core network. The specialized and physical nature of PEs, however, introduces challenges in how service providers manage and operate them. Since many customers share a router, they share the same feature sets and are all impacted when changes are applied to a router. There is also generally a static mapping between a PE and a customer. This is because customers can not be moved without prolonged service interruptions or without affecting other customers configured on that router. As a result, customers cannot quickly recover during a network failure. Resources on the PE router cannot be optimized dynamically. For example, todays PE routers can exhaust their capacity in terms of bandwidth, routing state or physical port count. The static nature of PE assignments forces service providers to ensure that none of those resources are exhausted during the entire multi-year life cycle of a device.

In this chapter, we examine how Network Function Virtualization (NFV) and Software Defined Networking (SDN) can be applied to edge routers to alleviate these challenges. To that end, we propose EdgePlex, a novel distributed architecture that uses servers, switches, and advances in virtualization and high speed software packet forwarding to realize PE router functionality. The switches interconnect the servers and connect them to the customers and the core network. The servers host virtual machines (VM), each of which represents a single customer endpoint. Associating each customer with a VM gives independent control over each customer, the ability to sandbox a customer from other customers, and to migrate customers. In addition, we can leverage a rich set of existing VM tools to manage customers.

There are, however, challenges in terms of scale, performance, and fault tolerance. We address these

by intelligently partitioning functions across elements in EdgePlex. For example, we split control plane processing between a single central entity and multiple customer facing VMs. This allows scaling from the core network perspective while allowing for customers to be migrated with minimal impacts. Similarly, we distribute packet forwarding across the customer VMs to take advantage of parallel processing and increase performance. While not a requirement, our design also allows for offloading packet forwarding to switches when capable and possible.

The need to dynamically move customers from one router to another has been recognized even before the use of virtual machines became widespread. Sebos et al. [24] proposed the idea of migrating customers affected by access router failures and demonstrated the feasibility using Linux-based software routers. Agarwal et al. [8] built on the idea in [24] to propose the concept of a router farm which, similar to a server farm, can be used to reduce customer outages during router maintenance. VROOM [25] allows hot migration of customers from one device to another and uses the idea of programmable transport networks as proposed in [24] to move customer connections. Keller et al. [17] proposed the idea of router grafting, where parts of a router are removed from one router and merged into another. In contrast to these approaches, we propose to use the live migration ability of virtual machines to move customers, not only between logical PEs but also across servers within the same logical PE. Moreover, EdgePlex provides separation between and control of customers at a much finer granularity than any of these approaches.

There are several key design aspects of EdgePlex that set it apart from alternate proposals. Our use of virtualization and commodity servers dramatically improves the flexibility and manageability of PE routers. Service providers can quickly and cost-effectively provision customers when and where needed. The isolation of customers from each other opens new options for service providers; from providing customized services to each customer, to alternate operational models. For example, it is now possible for service providers to provide on-demand connectivity to customers, run different versions of protocols for different customers without impacting each other, roll out updates at a finer granularity, or perform maintenance at a specific customers convenience rather than at fixed times. Having the flexibility to run a different protocol version for each customer will be particularly important as service provider networks transition towards a fully software-defined environment. This is because it decouples the speed of SDN adoption by individual

customers and within the providers network, and thus leads to an incremental path for SDN deployment in service providers networks.

In this chapter we describe the architecture, design, and prototype implementation, as well as initial experimental results to demonstrate the efficacy of EdgePlex design. We use a combination of open source and custom-built software in our implementation. Our results demonstrate that we are able to get close to 10Gbps throughput from a single server just using software forwarding. We also show that we are able to leverage existing VM tools to provide fault tolerance. Our implementation and results lead us to conclude that the EdgePlex architecture is not only feasible, but is also scalable. It provides the desired performance characteristics while immensely improving flexibility and manageability.

2.2 The EdgePlex architecture

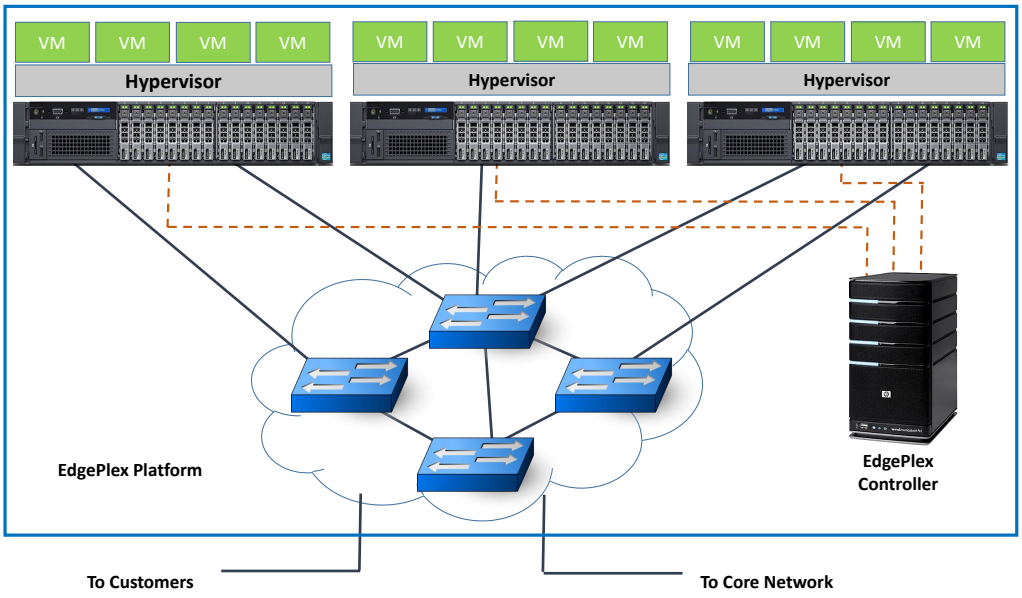


Figure 2.1: Architecture of EdgePlex platform

We discuss the architecture of EdgePlex in this section. Drawing inspiration from the success of cloud computing platforms, the goal of our approach is to leverage advances in servers, switches and virtualization technology to replace appliance-based edge routers. To that end, our design of EdgePlex architecture is presented in Figure 2.1. As shown in the figure, EdgePlex is a distributed platform consisting of commodity

servers and switches. The switches interconnect the servers with each other and provide connectivity to the customers and to the core network. The servers support virtual machines by running hypervisors and hosting one or more virtual machines.

The key novelty in the EdgePlex architecture is that each customer is assigned a virtual machine which we call a PortVM (or PVM) to which they connect. While not a one-to-one map, a PVM can be likened to the physical port on traditional hardware routers. The PVM stores the customer specific configuration like routing, access control, and quality-of-service information. It also stores other control- and data-plane state for the customer like its routing and forwarding tables. Thus, if a customer uses BGP (as is usually the case) for control plane, the TCP session associated with their BGP process terminates in the PVM. Associating each customer with a VM and storing their state in that VM gives us independent control over each customer. It provides operators with the new ability to sandbox and migrate individual customers with limited or no impact to other customers.

While this handles communications on the customer side, we still need to deal with communications with the core of the network. The simplest solution would be to allow each PVM to communicate both control- and data-plane exchanges with the rest of the network. In essence, each PVM would behave like a virtualized PE router with just one customer configured. This, however, has control plane implications and scalability limitations. Since each PVM has to have at least one BGP session back into the network (assuming the existence of BGP route reflectors), we now start scaling the number of BGP sessions from number of PE routers (order of hundreds to a few thousands) to number of customer PVMs (which could be tens to hundreds of thousands). In addition to BGP sessions, it has implications for MPLS label distribution because we need a different MPLS label per PVM and that information has to be distributed to every other PVM.

To address this, we introduce another VM called a ControlVM (or CVM). The controlVM (much like the loopback address in traditional routers) represents the PE from the control plane perspective on the core network. Thus it maintains BGP sessions with the route reflector and converses other control plane protocols (interior gateway protocols like OSPF, label distribution protocol for MPLS, etc.) with the core. The CVM then relays the necessary control plane information to and from each PVM. We assume that there

is a single CVM per EdgePlex instance. Since the failure of a CVM can cause the core to renegotiate labels and redistribute routes, we use an active backup for the CVM. This can be easily achieved using existing VM redundancy techniques (e.g., Micro-Checkpointing in KVM). Our results in Section 4.4 show that this approach is also effective. Thus the use of CVM, while simple, is sufficient to overcome the possible control plane scaling limitations. Moreover, the use of CVM does not restrict or hinder flexibility in managing customer PVMs within an instance of EdgePlex.

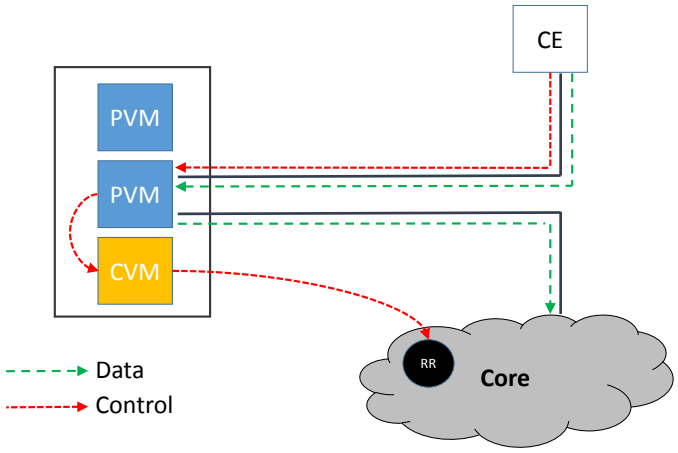


Figure 2.2: Split of control- and data-plane functions in EdgePlex

Figure 2.2 shows the overall split in control- and data-plane functions in EdgePlex. We make the simplifying assumption that customers use Ethernet to connect to the service providers network and that each customers traffic is separated using VLANs. This simplifies routing customer traffic to the corresponding PVM because the switches and hypervisors in the EdgePlex platform can be configured to terminate each customers VLAN in their PVM. As discussed previously, and as shown in Figure 2, customers control plane state terminates at their PVM. The PVM then exchanges relevant information to and from the CVM. The CVM has control plane sessions representing the EdgePlex PE with the core network. The exchange of control plane information between the PVM and CVM can be achieved through iBGP or through other custom protocols. The PVM, by default as shown in Figure 2, is also responsible for the data forwarding for each customer. The PVM identifies the appropriate forwarding rule, makes the required header modifications (e.g., applying the right MPLS labels), and forwards the packets. While not essential, the EdgePlex

architecture allows for alternate forwarding strategies such as offloading the forwarding of packets to the hypervisor or even to the physical switch to take advantage of faster packet forwarding capabilities. Finally, as shown in Figure 2.1, given that EdgePlex is a distributed platform, we make use of a controller that manages this platform. The main role of the controller is to configure customers by creating their PVMs and configuring connectivity in the hypervisor and in the switches. It is also responsible for monitoring the functioning of the platform and take action for example, react to failures, load balance the platform, etc.

2.2.1 A smarter, faster, and flexible design

Having described the EdgePlex architecture, we discuss the strengths of the proposed design. Our proposed design addresses many of the shortcomings of existing edge routers, especially in terms of every-day operation and management, customer isolation, supporting new services, and providing a cost-efficient path for gradually introducing alternate networking technologies.

Intelligent Task Partitioning: We intelligently partition router functions across the elements in EdgePlex. We split control plane processing between the central CVM and all the PVMs. This allows scaling from the core network perspective while allowing for customers to be migrated with minimal impacts. Similarly, we distribute packet forwarding across the PVMs to take advantage of parallel processing and increase performance. We also allocate tasks based on the entitys strengths. For example, control plane processing (e.g., route computation, BGP exchanges) is computationally heavy. Hence, we retain control plane processing on the servers and VMs. Physical switches support high-speed packet switching and forwarding. Thus when capable and possible, switches can be used to offload packet forwarding.

Efficient Fault Tolerance: Since EdgePlex is made up of commodity elements, failure of individual elements is going to be the norm. The design of EdgePlex allows us to minimize the impacts of faults. As discussed previously, we use existing redundancy techniques for CVM. We do not use them for PVMs because of the number of PVMs, and the limited impact of its failure. Only the customer configured on the PVM will lose connectivity when a PVM fails. However, we can quickly create a new VM in its place and restore connectivity. Similarly if a server fails, only customers on that server need to be handled. The

impact of a switch failing depends on how the switches and servers are interconnected; however, redundant connections between servers and switches can help handle switch failures. Most importantly, we can handle failures (or maintenance) locally without having to move customers between remote PEs.

Easier Capacity Planning: Our use of virtualization, and our design in particular, dramatically simplifies capacity planning for PE routers. Since connecting a customer really requires just spinning up a VM for the customer, it allows service providers to quickly and cost-effectively provision customers when and where needed. Providers do not have to carefully plan the specific device that a customer has to connect. The elastic scaling that comes from the use of commodity servers and virtual machines allows service providers to elastically allocate capacity, making it capitally efficient. Thus when extra capacity is needed, more servers can be included into the platform; when capacity is no longer needed, these servers can be used for other applications.

Flexible Management: EdgePlex allows operators significantly more flexibility in managing customer connectivity. Unlike today, they can roll out updates in a finer granularity, by starting off with a few customers at a time and then rolling it out widely after making sure that the update does not have unintended consequences. They can perform maintenance at a specific customers convenience rather than at a time that works for all customers. More importantly, they can leverage the wide array of tools available to manage servers and VMs rather than rely on primitive tools available to manage networks.

Customized/New Services: The isolation of customers in VMs opens new options for service providers to provide customized services to each customer. For example, it is now possible for service providers to provide different versions of protocols for different customers without impacting each other. They can also turn on new services for customers (like DDoS detection, caching, redundancy elimination, etc.) relatively easily. The platform also opens up opportunities to support new technologies like information centric networking which requires the ability to both route traffic and store data within the network.

Simplified and Cost-Efficient Upgrades: EdgePlex supports existing network protocols. But the ability to support customized services for each customer allows service providers to run custom protocol versions for each customer. This allows providers to gradually transition their network to an SDN only environment since

it decouples the speed of SDN adoption within their network and across individual customers. Customers wanting SDN can adopt it as soon as it is available, while others can continue using existing technology. It also eases transition for providers as they can gradually make features available.

2.3 Implementation and Evaluation

We built a prototype implementation of EdgePlex and performed experiments to understand the performance aspects of the architecture. We present details of our implementation and our experimental results in this section.

2.3.1 Prototype Implementation of EdgePlex

To demonstrate the efficacy of the EdgePlex design, we implemented a prototype of the EdgePlex platform. We used a combination of open source and custom software in our implementation. We used KVM as the hypervisor and ran multiple VMs on a single server. We used OpenBSD in the CVM. OpenBSD has in-built support for MPLS and for other routing protocols like OSPF and BGP. We configured OpenBSD appropriately to connect with route reflectors and other core network routers. Since, the CVM is primarily handling control-plane, we did not utilize any high-performance packet processing library in the CVM. We used GNU/Linux running Quagga and a custom packet forwarding application (called FastForward) in our PVMs. Quagga handled individual customer BGP sessions and exchanges control plane between the customer and the CVM. FastForward was responsible for forwarding data packets from and to the customer. FastForward uses Intels DPDK [1] for fast packet processing and Cuckoo hash library [26] for route lookup. When it receives packets from the customer, it has to insert MPLS label(s), update the ethernet headers and forward the packet to the backbone. FastForward gets MPLS label information from the CVM and stores it in the Cuckoo Hash. Based on the destination, FastForward uses the Cuckoo hash to find the right MPLS label(s), and inserts the labels into the packet. When it receives packets destined for the customer, it strips the MPLS labels, updates the ethernet headers, and forwards the packet.

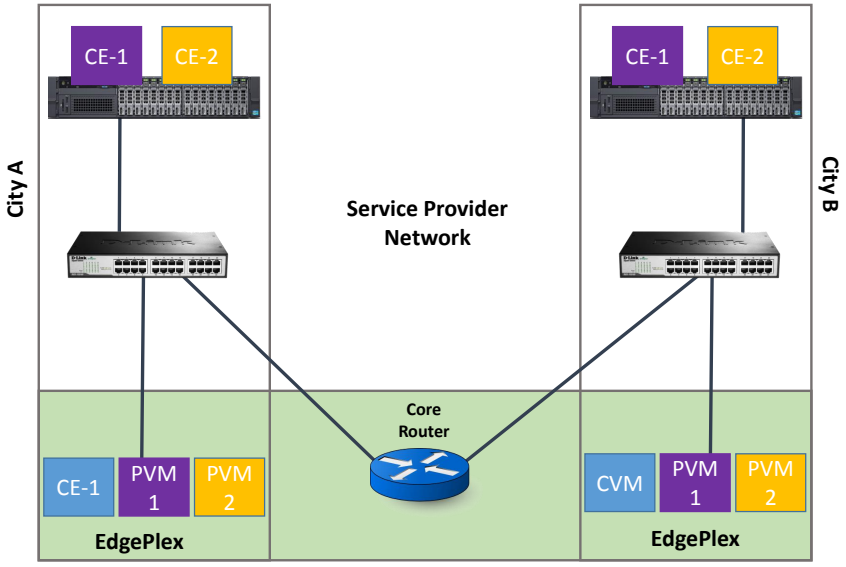


Figure 2.3: Testbed setup of a prototype implementation of EdgePlex

2.3.2 Testbed Setup

We used the prototype implementation in a test bed consisting of x86 servers and OpenFlow capable switches to evaluate the performance of the platform. We interconnected the servers using OpenFlow-enabled switches to emulate a wide-area setup as shown in Figure 2.3. We used Ryu as our OpenFlow controller to talk to and configure the switches. We used the switches not only to interconnect the servers, but also to forward packets to the correct PVMs.

We partitioned the servers such that they host customer edge (CE) routers in two different cities. We then connected these CEs to a corresponding PVM in the EdgePlex platform in each of these cities. The two EdgePlex instances communicated with each other over the core network. We used a commercial, production-grade virtual router to function as our core router. This helps us demonstrate that EdgePlex can in fact work alongside existing infrastructure. We ran traffic end-to-end, including configured as an MPLS VPN with overlapping addresses, to demonstrate the viability of EdgePlex as a replacement for PE.

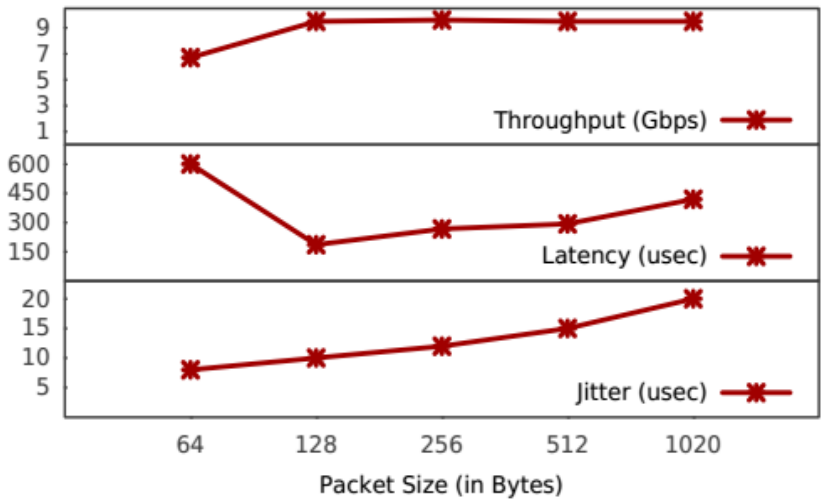


Figure 2.4: Throughput, Latency and Jitter for various packet sizes with 16 PVMs

2.3.3 Data Plane Performance Evaluation

Since our servers had 16 cores, we varied the number of customers configured on EdgePlex between 1 and 16 to see the packet forwarding performance of EdgePlex. We used SR-IOV to virtualize the network interfaces across these PVMs. In the remainder of this section, we present results from performance experiments with 16 PVMs per server and measured at the ingress EdgePlex instance. The cuckoo hash in each PVM stores one million entries of target IP address to MPLS label mappings.

We use a modified version of pktgen (a DPDK-based packet generator) to generate traffic to the 16 PVMs. In addition to throughput, our modified version gives important information like latency of packet processing in a PVM and jitter between packets. To get these latter metrics, however, we had to bypass the end-to-end path and instead return the packets back to the generator.

Figure 2.4 shows the throughput achieved with various packet sizes with 16 PVMs. (Note that, like previous studies, we were able to achieve the full 10Gbps with 1 PVM). The computed value is the average of throughput achieved in 10 runs of the experiment. We see from Figure 4 that EdgePlex is able to achieve close to capacity of 10Gbps with packet sizes greater than 128-bytes. With 64-bytes packets, EdgePlex is only able to get up to 6.7 Gbps beyond which we start seeing packet drops. We attribute this to the overheads

Input Rate (Gbps)	2G	3G	4G	5G	6G	7G
Latency(us)	36	31	50	30	44	602
Jitter(us)	29	21	10	7	9	6

Table 2.1: EdgePlex's Latency and jitter under different input rates with 64-byte packets

in processing the increased number of packets with 64-byte packets.

Throughput, however, tells us a part of the story. Hence we look into how long it takes to process the packets and any batching effects by looking into the latency and jitter experienced by packets and plot them also in Figure 4. We see that jitter is quite acceptable between 8 and 20 usecs for all packet sizes. Latency, however is more interesting. We see that there is a steady increase from about 180 usecs to 420 usecs as we go from 128 bytes to 1020 bytes. 64- byte packets however have higher latency at about 600 usecs. Despite these interesting data points, the main conclusion from our result is that EdgePlex platform is able to easily utilize the capacity of servers. To understand the latency with 64-byte packets we studied latency and jitter under different input rates for 64-byte packets.

Table 2.1 shows that the latency remains relatively stable until the input rate reaches the maximum throughput. At that point latency increases by more than ten fold. Interestingly we see that jitter drops as we increase the input traffic rate. This result leads us to believe that there is some form of queuing/batching in play. We plan to investigate the root cause further as part of our future work.

2.3.4 PVM Migration and CVM Availability

A key feature of EdgePlex is that we can flexibly migrate PVMs from one host machine to another for tasks such as maintenance or load balancing. We evaluated the impact of migrating PVMs on the data forwarding performance using existing VM live migration capabilities. We noticed a connectivity interruption for less than one second when migrating a PVM. A major contribution of the connectivity interruption comes from the modification of switch forwarding rules. Recall that we used an OpenFlow capable switch to forward the packets to the correct PVM. We notify the switch to change its rules after the live migration is done. While this short down time is reasonable, we are investigating alternate mechanisms to avoid this interruption, including strategies proposed in OpenNF [13].

Checkpoint Interval (msec)	100	200	500	1000
Data Transferred (MB))	7.47	7.92	16.6	36.6

Table 2.2: Amount of data transferred with checkpointing the CVM for different time periods

To guarantee the availability of EdgePlex, especially of the CVM, during failures, we used an existing tool in KVM called Micro-Checkpointing (similar to Remus [11] for Xen), which periodically generates a micro-checkpoint of CVM and copies it to a backup VM. We tested that with the help of Micro-Checkpointing, EdgePlex can recover from failures without affecting its running protocols. For example, the MPLS labels for different paths remain the same after the failure recovery. A key parameter of Micro-Checkpointing is the checkpoint frequency. On the one hand, if we generate checkpoints too frequently, the communication overhead to transfer dirty memory may be high. On the other hand, if we do not create enough checkpoints, we may lose some network state changes, such as routing updates. We measured the amount of transferred bytes for different checkpoint frequencies when CVM received 200 BGP updates per second. We tabulate our results in Table 2.

As expected the amount of data transferred increases as we increase the checkpointing interval. However, even in the worst case, the amount of data transferred is very manageable considering that this approach is used only for the CVM. In practice, we expect the routing protocol update frequency to be much lower and hence a lower communication overhead for Micro-Checkpointing.

2.4 Statelessness in EdgePlex

While EdgePlex was designed to address many shortcoming of existing edge routers, we notice that there are some limitations in its current design where StatelssNF principles can play a big role to overcome these limitations and challenges.

For example, while each customer is assigned a virtual machine (PVM) to which they connect, these VMs contain important state (*e.g.,* customer specific configuration like routing, access control, and data-plane state for the customer like its routing and forwarding tables) that are critical in the case of failure of them. In addition, the other proposed types of VMs (CVM) also contain crtitical network state (*e.g.,* BGP

sessions and label distribution protocol for MPLS).

While EdgePlex provides fault tolerance by practicing redundency techniques on these VMs, as we show in Chapter 3, this approach is impractical when the number of instances scales. Thus, a loss of all internal state in these VMs upon failure would require the router to request neighbors to re-announce all routes, and in turn cause the failed (and then recovered) router to announce routes to its neighbors, causing a great deal of convergence and transient disruptions.

While other related solution to EdgePlex tried to overcome this challenge (*e.g.,* Router Grafting [76] migrates BGP session state across routers), a common design theme among these works is that the decoupling of state only occurs during management 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. On 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 prone to having bugs and are hard to deploy in production environments. They also do not support graceful upgradablity as required by microservice-based deployments.

In the following chapters, we discuss in detail the challenges in network state and how we took a further step than just breaking apart the control plane as we did with EdgePlex to breaking apart the state from processing to achieve complete network agility. Thus, breaking apart the state in systems such as EdgePlex will certainly optimize their efficiency in terms of scalability and fault tolerance.

Chapter 3

The challenge with state

To motivate the need to dissaggregate the state from processing, in this section we illustrate the problems that the tight coupling of state and processing creates today, even with the flexibility of virtualized network functions. We also present the shortcomings of recent proposals which attempt to address the challenge with state. Here, we narrow the focus to network functions that are in-line middleboxes that process network traffic, as they clearly highlight the problem, and as we discuss in chapter 4, are the most challenging to solve for.

3.1 Understanding state

Although they serve a variety of purposes, network functions share a common design [108], as illustrated in Figure 3.1. In this packet processing pipeline, a packet is received, the contents of the packet are decoded, the contents (*e.g.,* the 5-tuple of IP source and destination, transport type, and transport source and destination) are then used as the key in a lookup, the lookup result then dictates the processing that should occur.

The states 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 [108, 56]. The latter can be further classified into (i) internal instance specific state (*e.g.,* cache, file descriptors, and CPU and memory statistics), (ii) and network state (*e.g.,* session state, timers, and counters).

The state we care about that is causing all the problems with network function elasticity and resiliency is the dynamic state since this state needs to be tracked and is shared among different flows or different

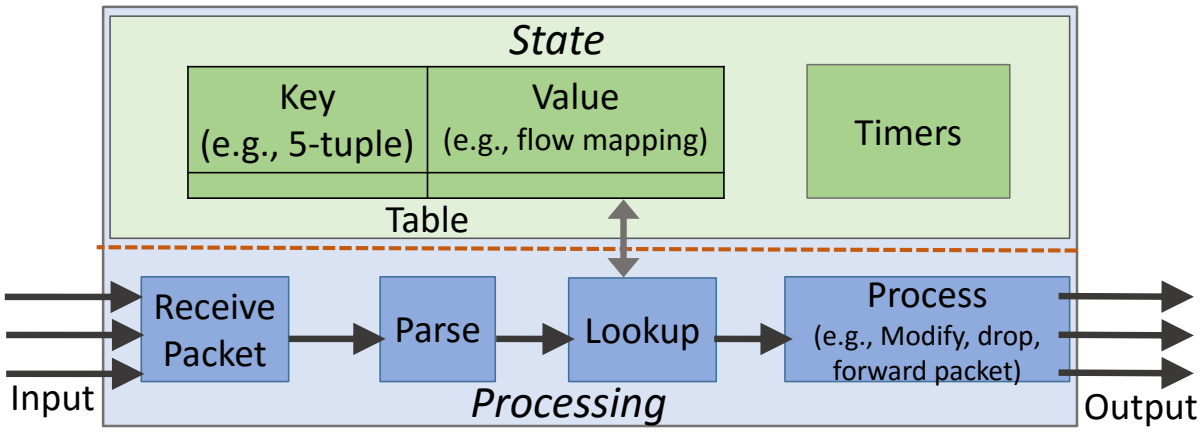


Figure 3.1: Traditional Network Function Design

network functions. Thus, all the state we refer to in the remaining of this thesis is the dynamic state. This state can include, for example, connection establishment information (for a firewall), address mappings and the pool of available ports and IP addresses (for NAT), and a mapping of flow to back-end server (for a load balancer).

3.2 Dealing with Failure

For failure, we specifically mean fail-stop failures (as opposed to byzantine failures). This can be due to the hardware itself failing, or some form of network error which makes the network function unreachable. As recent studies have shown, failures do happen regularly. Even more, they are highly disruptive [105].

The disruption comes from two factors – both related to the tight coupling of processing and state. For the first factor, consider Figure 3.2(a). In this scenario, we have a middlebox, say a NAT, which stores the mapping for two flows (F1 and F2). Upon failure, with virtualization we can quickly launch a new instance, and with technology such as software-defined networking (SDN) [87, 43, 29], we can direct traffic to the new instance. However, any packets within flows F1 or F2 will then result in a failed lookup (no entry in the table exists), which will then cause the NAT to create new mappings, which will ultimately not match what the server expects. This will cause all of the connections to eventually timeout and need to be re-established. Of course, enterprises could employ a hot-standby redundancy, but the expense of having extra devices that are unused is high, and still fall victim to the second factor.

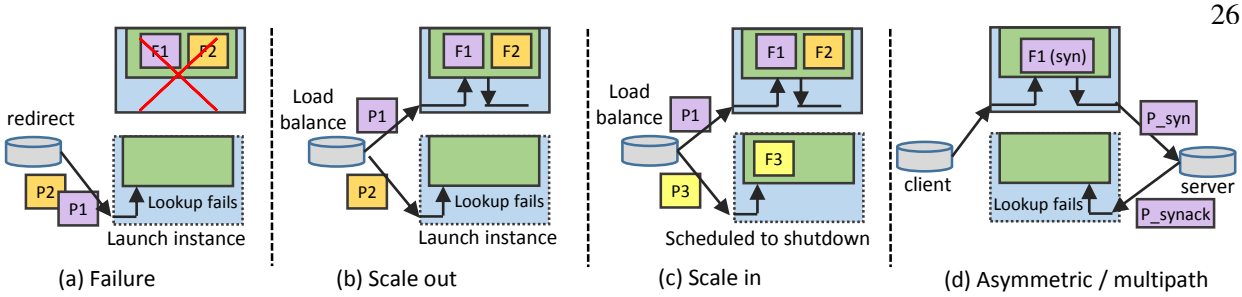


Figure 3.2: Motivational examples of traditional network functions and the problems that result from the tight coupling of state to the network function instance. State associated with some flow is labeled as F (*e.g.,* F2), and the associated packets within that flow are labeled as P (*e.g.,* P2).

The second factor is due to the high cost of failover. In all existing and proposed solutions (discussed below), there is a high cost to fail over. As such, the mechanisms tend to be conservative when determining whether a device has failed' [19] – if a device does not respond to one hello message, does that mean it is down or that the network dropped a packet or that the device is highly loaded and is taking a bit longer than expected to respond? Being too aggressive will cause some unnecessary failovers – resulting in downtime or possibly the case where the backup becomes a master while the original is still alive (but considered dead), leading to problems. Being too conservative, any traffic will still be sent to the device (that has failed), leading to disruption.

Problem with existing solutions

Two approaches to failure resilience have been proposed in the research community recently. First, pico replication [107] is a high availability framework that frequently checkpoints the state in a network function such that upon failure, a new instance can be launched with the state intact. To guarantee consistency, packets are only released once the state that they impacted 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 the latency, another work proposes logging of all inputs (*i.e.,* packets) coupled with a deterministic replay mechanism for failure recovery [115]. 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.

3.3 Scaling

The move to virtualized network functions promises to provide greater agility in comparison to installing physical appliances. This is highly desirable when considering that utilization of middleboxes is typically on the order of 5-20% in enterprises today, and even so, still get overloaded [116]. With virtualization, we can elastically scale in and out to meet demand.

As with the case of failover, the tight coupling of state and processing causes problems. 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). Shared state (*e.g.,* what ports are available to use in a NAT) still requires synchronizing the state across all instances. Referring to just the problem with partitionable state, in Figure 3.2(b) we show an example of scaling out. For this, even though we launch a new instance to handle the overloaded condition, we cannot move any existing flows 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, which in turn will cause the connection to timeout (or require the application to detect, if using UDP). Scaling in (decreasing instances) is also a problem, as illustrated in Figure 3.2(c). Here, we have scaled to multiple instances, but the load has since decreased such that we would like to shut down an instance which is currently handling flow F3. We need 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 flows.

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 BGP state) [76], and several have since demonstrated this for middleboxes [108, 58, 57] where partitionable state can be migrated between instances. This state migration, however, does not address failure and requires affinity of traffic to instance

to work.

3.4 Asymmetric / Multi-path Routing

As a final case which can cause a challenge for a dynamic network function infrastructure, we consider asymmetric and multi-path routing. Asymmetric routing relates to the fact that forwarding of traffic from a client to a server may traverse a different path than the traffic from a server to a client. Traffic in both directions needs to go through the same network function instance. For example, consider Figure 3.2(d), where a firewall has established state from an internal client connecting to a server (SYN packet). The return syn-ack from the server goes through a different firewall instance, which results in a failed lookup and the packet being dropped. Multi-path routing relates to recent pushes, *e.g.,* with multi-path TCP [98] to be able to split a flow to capitalize on the availability of path diversity (*e.g.,* Wi-Fi and LTE). This traffic may still go through a single provider's network function infrastructure (either at the client side or the server side), and as such, need the traffic to go through a single instance.

Problem with existing solutions

A recent work proposes a new algorithm for intrusion detection that can work across instances [84], but this is not generalizable. Other solutions proposed in industry make use of, for example, HSRP [81] to synchronize state across middleboxes [78] – which doesn't particularly scale too well.

Chapter 4

StatelessNF - Breaking the Tight Coupling of State and Processing

4.1 Stateless Network Functions

The key idea with StatelessNF is to decouple the processing from the state in network functions – placing the state in a data store, as illustrated in Figure 4.1. 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.

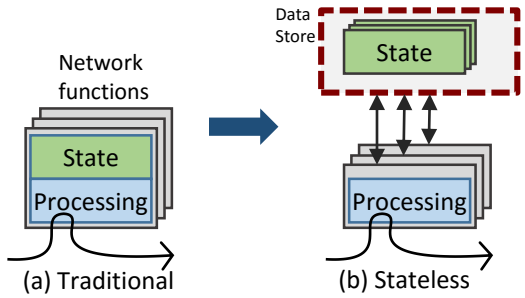


Figure 4.1: High level overview showing traditional network functions (a), where the state is coupled with the processing to form the network function, and stateless network functions (b), where the state is moved from the network function to a data store – the resulting network functions are now stateless.

This idea is inspired by the architecture of multi-tiered web applications, which decouples processing (*e.g.,* web server) from state (*e.g.,* data base and caching layers) to enable a dynamic and resilient architecture along with opening up opportunities for interacting with the state (*e.g.,* data mining in the background). In this section we focus on the benefits toward a dynamic and resilient network infrastructure, and leave exploration of additional opportunities that may be opened up as future work. Below we briefly highlight.

Failure: 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 instantaneously 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. The network function instances which the traffic is directed to will have access to the state needed to process.

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 an scalable, resilient, network function. As such, packets traversing different paths does not cause a problem.

The main challenge in achieving this architecture is performance – the types of network function middleboxes that we target typically handle millions of packets per second – which is because interacting with the data store is (in many cases) in the critical path of network processing. The common structure is that the incoming packet is parsed, information is used to perform a lookup, and the result of the lookup is then used to make a decision on what to do with the packet and sometimes to update the state. To overcome this challenge, we exploit this common packet processing structure to minimize interactions with the data store, and leverage recent advances in high-performance network I/O [66], and low-latency data stores [96]. In doing so, we achieve a comparable throughput to traditional network functions (discussed further in Section 4.6). Note that we do not necessarily aim to retain an identical code structure for existing code, where state can be integrated with the code (*e.g.,* as variables in many different places). We instead

Network Function	State	Key	Value	Access Pattern
Load Balancer	Pool of backend servers	Cluster ID	List of IP addresses	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 IP and Port	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 one read/write for every out-of-order packet
IPS	Automata State	5-Tuple	Int	1 read/write for every packet

Table 4.1: Network Function Decoupled States

assume we can restructure with identical functionality but better grouping of state (*e.g.,* in tables, rather than variables in a number of places). We elaborate on our StatelessNF architecture in the next two sections.

4.2 How Network Functions Access State

The key idea in this work 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 4.1, 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 4.5, we only decouple network state.

We demonstrate how the decoupled state is accessed with pseudo-code of multiple network function

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)
```

algorithms, and summarize the needed reads and writes to the data store in Table 4.1. 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 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 ← processByte(b, automataState)
12:         writeRC(5-tuple, automataState)
13:         sendPacket(P)
```

Algorithm 3 TCP Re-assembly

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      extract 5-tuple from incoming packet
3:      if (P is a TCP SYN) then
4:          record ← (getNextExpectedSeq(P), createEmptyBufferPointerList())
5:          writeRC(5-tuple, record)
6:          sendPacket(P)
7:      else
8:          record ← readRC(5-tuple)
9:          if (record == NULL) then
10:             dropPacket(P);
11:         if (isNextExpectedSeq(P)) then
12:             record.expected ← getNextExpectedSeq(P)
13:             sendPacket(P)
14:             // check if we can send any packet in buffer
15:             while (bufferHasNextExpectedSeq(record.buffPtr, record.expected)) do
16:                 P ← readRC(pop(record.buffPtr).pktBuffKey)
17:                 record.expected ← getNextExpectedSeq(p)
18:                 sendPacket(P)
19:             writeRC(5-tuple, record)
20:         else
21:             // buffer packet
22:             pktBuffKey ← getPacketHash(P.header)
23:             writeRC(pktBuffKey, P)
24:             record.buffPtr ← insert(record.buffPtr, p.seq, pktBuffKey)
25:             writeRC(5-tuple, record)
```

Algorithm 4 Firewall

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      key ← getDirectional5tuple(P, i)
3:      sessionState ← readRC(key)
4:      newState ← updateState(sessionState)
5:      if (stateChanged(newState, sessionState)) then
6:          writeRC(key, newState)
7:      if (rule-check-state(sessionState) == ALLOW) then
8:          sendPacket(P)
9:      else
10:         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')
```

Algorithm 2 presents the pseudo-code for a signature-based intrusion prevention systems (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 [32] (as used in Snort [24]). 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 automate 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 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 [127]. Similarly to the Snort TCP reassembly preprocessor [24], we rely on a TCP re-assembly module to deliver the bytes to the IPS in the proper order.

For 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 IPS, we observe that we require one read and one write to the remote storage for each data packet, and, similar to load balancer, at most one additional read and write to the remote storage at the start and end of each connection. Table 4.1 shows similar patterns for other network functions, and Section 4.6 analyzes the performance impact of such access patterns, and demonstrates that we can achieve multi Gbps rates.

4.3 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 4.2, the StatelessNF architecture consists of three main components – the data store, the hosts to host 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 4.4. In this section, we elaborate on the data store and network function orchestration within StatelessNF.

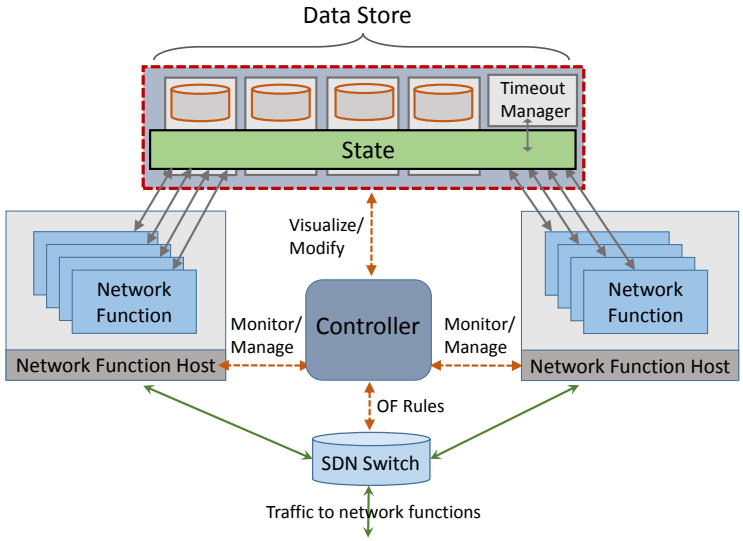


Figure 4.2: StatelessNF System Architecture

4.3.1 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 need to process network traffic, and not worry about state replication, etc. A data store provides the resience of state. Because of this separation and because it resides in the critical path of packet processing, the data store must also provide low-latency access. In our current work, we decided to choose RAMCloud [96] as our data store. RAMCloud is a distributed key-value storage system that is based entirely in DRAM and 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 [96], 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 entirely 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 feel it has growing acceptance (*e.g.,* Microsoft Azure provides options which include Infiniband [11]), and our architecture does not fundamentally rely on Infiniband – RAMCloud developers are working on other interfaces (*e.g.,* RoCE [111]), 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 event.

We believe there may be further opportunity to optimize StatelessNF through customization of the data store. While our focus in this work 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.

4.3.2 Network Function Orchestration

The basic needs for orchestration involve monitoring the network function instances for load and failure and adjusting 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 it has 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 which 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, when failure occurs or we are scaling-in, traffic should be redirected to a different instance. With emerging programmable networks, or software-defined networks (SDN), such as OpenFlow [87] and P4 [43], we can achieve this. Further, as existing SDN

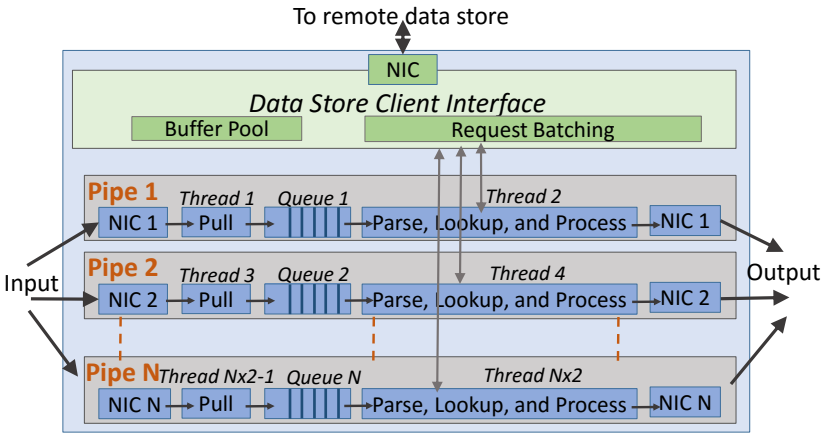


Figure 4.3: Stateless Network Function Architecture

controllers (*e.g.,* ONOS [42], Floodlight [8], OpenDaylight [29]), provide REST APIs we can integrate the control into our overall orchestration.

4.4 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 4.3, 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.

4.4.1 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[22] 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[67] 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[88], 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.

4.4.2 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 Data Plane Development Kit (DPDK) architecture [66]. DPDK provides a simple, complete framework for fast packet processing in data plane applications.

One challenge that arises with the use of DPDK in the context of containers is that huge page support is required for the large 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 huge

page filename and specified socket memory amount to ensure isolation[1] . We used The DPDK Environment Abstraction Layer (EAL) interface for system memory allocation/de-allocation and core affinity/assignment procedures among the network functions.

4.4.3 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, for example.

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 only takes 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.

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

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.

4.5 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 4.6 for elasticity and failure experiments). The controller interacts with the Floodlight [8] 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 [66] 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 [96]. The packet processing pipes are implemented in C/C++ in a sequence of pipeline functions which 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.

4.6 Evaluation

In this section, we claim that StatelessNF will add elasticity and failure handling, with the main challenge of whether we can achieve good performance. In this section, we evaluate the achievable performance as well as the failure-handling and elasticity benefits.

4.6.1 Experimental Setup

Our experimental setup consists of a total of 6 servers and two switches – with two servers hosting network function instances, those are connected via Infiniband to two servers for hosting RAMCloud, and via Ethernet to a server which serves as the traffic generator and sink. An additional server hosts the StatelessNF controller which orchestrates the entire management. Specifically, we use the following equipment:

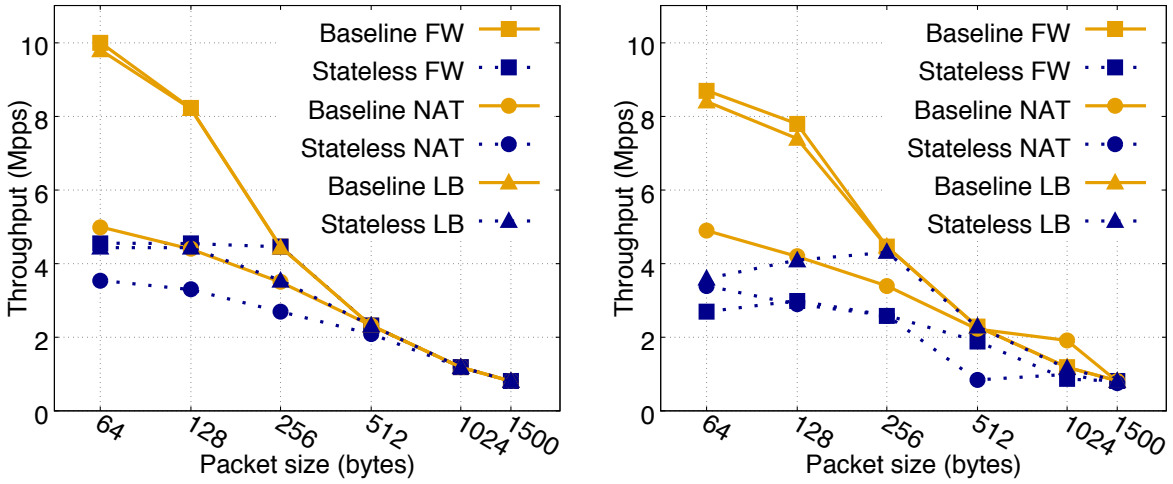
- Network Function hosts: 2 Dell R630 Servers[49]: each with 32GB RAM, 12 cores (2.4GHz), 1 Intel 10G Server Adapter with SR-IOV support[67], 1 10G Mellanox InfiniBand Adapter Card[88].
- RAMCloud: 2 Dell R720 Servers[50], each with 48GB RAM, 12 cores (2.0GHz), 1 Intel 10G Server Adapter[67], 1 10G Mellanox InfiniBand Adapter Card[88].
- Traffic generator/sink: 1 Dell R520 Servers[48]: 4GB RAM, 4 cores (2.0GHz), 2 Intel 10G Server Adapters[67] .
- Control: 1 Dell R520 Servers[48]: 4GB RAM, 4 cores (2.0GHz) to run StatelessNF and Floodlight controllers.
- SDN Switch: OpenFlow enabled 10GbE Edge-Core [52].
- Infiniband Switch: 10Gbit Mellanox Infiniband switch between RAMCloud nodes and the network function hosts[89].

4.6.2 StatelessNF Single Server Performance

Here, we evaluate the packet processing performance of StatelessNF as compared to non-stateless (traditional) network functions (which we refer to as baseline).

4.6.2.1 RAMCloud Client Limits

Before we measure the throughput and latency, it is important to understand the performance of the RAMCloud servers as they may serve as a fundamental limit of the rates we are able to attain. We focus on lookup operations since we perform them for every packet. Our benchmark tests show that a single server



(a) Best case

(b) Worst case

Figure 4.4: 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.

in RAMCloud can handle up to 1.8 Million lookups/sec for a single client and up to 4.7 Million lookup/sec for three and more clients. This means that a network function needs three packet processing pipes so it can handle 4.7 Million packets per second[2] . For write operations, a single server can handle 0.55 Million write per second with a single client and up to 0.7 Million write per second for two and more clients.

We integrated the interface to a basic network function that only forwards packets after the lookup operation. We found out that a single processing pipeline can handle 1.6Mpps and 4.7Mpps with three pipelines[3] .

4.6.2.2 Impact of needing remote reads/writes

The throughput of a network function is highly tied to the ability to generate requests to the data store fast enough. A main factor that influences this is the application access pattern itself. While each of our example network functions perform a lookup operation for every packet, they differ in writes – the load balancer, for example, performs three write operations per flow. On the other hand, a firewall for instance, performs five write operations per flow.

[2] Each pipe has its own client instance within the data store interface mentioned in Section 4.4

[3] The packet size did not change these limits since the network function does not inspect packets.

From this, we can see that the performance will be dependent on the traffic processed. Therefore, we define three cases: best, worst, and average in regards to the size and number of flows. We consider the best case as there are few long flows. The worst case, on the other hand, is when there are many short flows (which would require many more writes for the same amount of traffic). For the average case, we used real world enterprise traces [4], where traffic is based not on best or worst case, but real traces. In each case we varied the packet sizes to understand the impact of having different packet-processing rates, as larger packets mean fewer packets per second.

We used Tcpreplay with Netmap [120] to stream the three types of traces. The first, representing the best case, has 3 thousand large TCP flows of 10K packets each. The second trace, representing the worst case, has 100,000 short TCP flows of 100 packets each. And finally for the average case, we replayed a real captured enterprise trace with 17,000 flows that range in size from 10 to 1000 packets.

Figure 4.4 shows the throughput of StatelessNF middleboxes compared to their baseline counterparts with long and short flows of different packet sizes – achieving 4.6Mpps for minimum sized packets. The gap between stateless and non-stateless in throughput with small packets (less than 128 bytes) is due to a single RAMCloud server being able to handle up to around 4.7 Million lookups/sec, while in the baseline all read and write operations are local. As packets get larger (greater than 128 bytes), the rates of stateless and baseline network functions converges. This indicates that, although StatelessNF imposes an overhead of reading and writing from a remote data store, the impact of such overhead can be masked, but the rate limit exists.

To test how stateless middleboxes perform with real traces, we show how it can handle the traffic even when increasing its rate to more than the original rate at which it was captured[4] . It is important to mention that since the packet sizes considerably vary (80 to 1500 bytes), we report the throughput in terms of traffic rate (Gbit/sec) rather than packets/sec as we show in the artificial traces earlier. As we see in Figure 4.5, the StatelessNF firewall and loadbalancer have a comparable performance while NAT has about a limit that is 1Gbps lower than the non-stateless version.[5]

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

[5] The reason we see the NAT more limited than the firewall and laod balancer is due to the overhead of IP header checksum after modifying the packet IP addresses and port numbers.

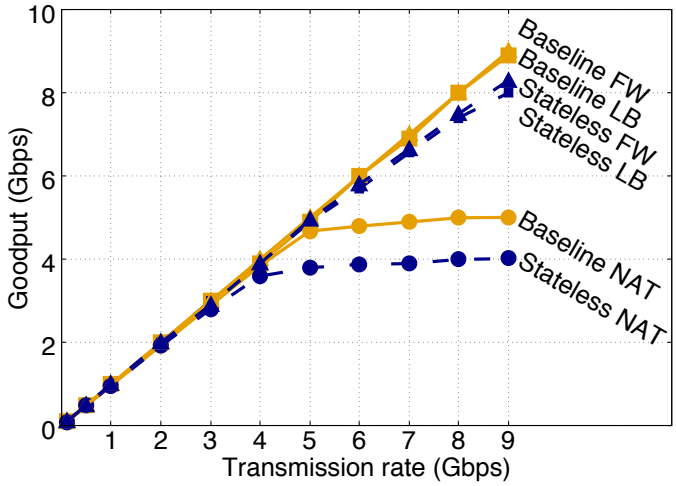


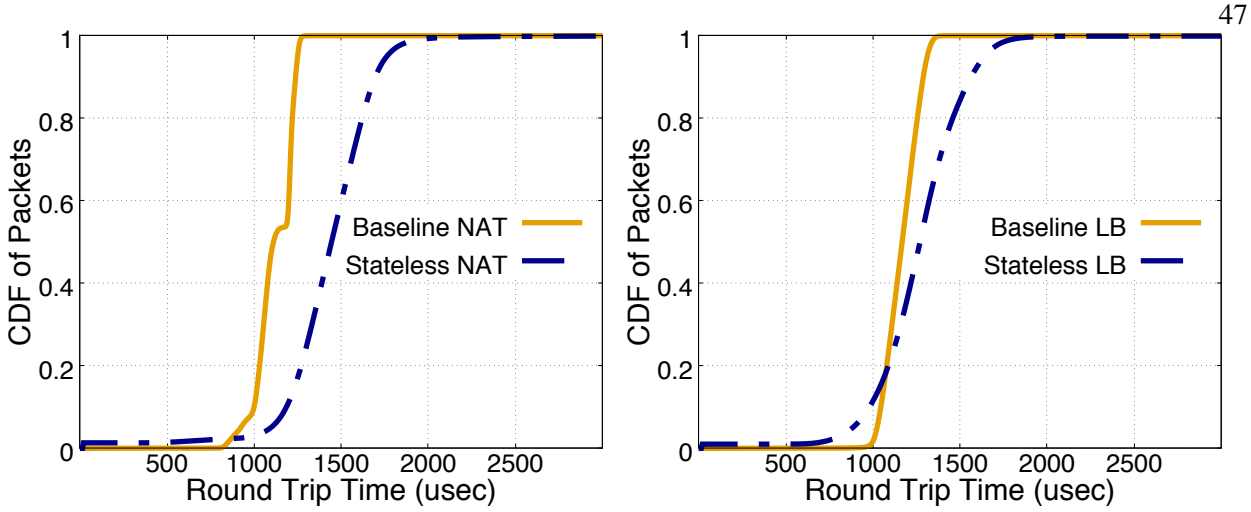
Figure 4.5: Measured goodput (Gbps) for enterprise traces.

4.6.2.3 Latency

The interaction with the data store as part of the packet processing will obviously impact the latency of each packet since each one will be buffered until its lookup operation is completed. To evaluate this impact, we measured round-trip time (RTT) for each packet in stateless and baseline network functions to observe the added delay. We generate and stream and timestamp packets, send the traffic through the network function, which the output is then directed (by the SDN switch) back to the packet generator. We calculate the round trip time RTT by comparing the timestamp in the packet to the current time (and since the sending and receiving are on the same machine, we do not worry about clock synchronization).

Figure 4.6 shows the cumulative distribution function (CDF) for the RTT of packets[6] . In the 50th percentile, the RTT of StatelessNF packets is only 100μs longer than the baseline's for the load balancer and firewall, and in the 95th percentile the RTT is only 300μs longer. The added delay we see in StatelessNF is a combination of miss reads (which can reach 100μs), preparing objects for read requests from RAMCloud, and casting returned data), and the actual latency of the request. Although 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 [115]), we believe we can greatly reduce our added latency – further discussed in

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



(a) NAT

(b) Load Balancer

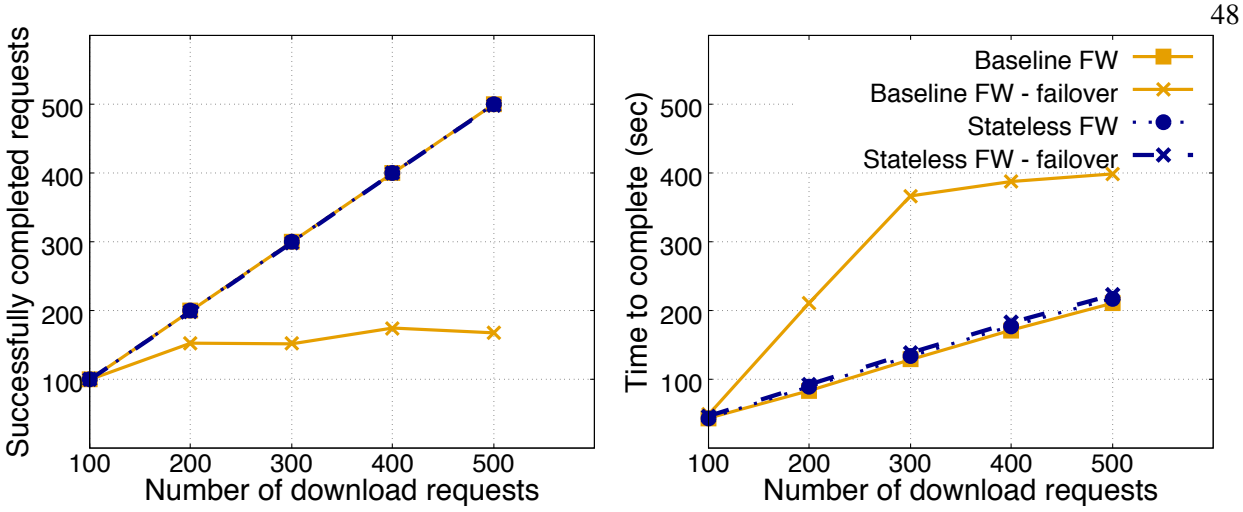
Figure 4.6: Round-trip time (RTT) of packets for baseline and stateless middleboxes.

Section 4.7.

4.6.3 Failure

As we discussed in Section 4.2, 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 measure 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 4.7 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 the client re-initiating the connections after a TCP connection timeout.[7]



(a) Completed requests

(b) Time to complete requests

Figure 4.7: (a) shows the total number of successfully completed requests, and (b) shows the time taken to satisfy completed requests.

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.

4.6.4 Elasticity

With StatelessNF, we claim that decoupling state from processing in network functions provides elasticity, where scaling in/out (*i.e.,* removing/adding middlebox instances) 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 millions of TCP packets while gradually increasing the traffic rate every 5 seconds (as shown in Figure 4.8), keep it steady for 5 seconds, and then start decreasing the traffic rate every 5 seconds. The three lines in Figure 4.8 represent: the ideal throughput (Ideal) which matches the send rate, the baseline firewall, and 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

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

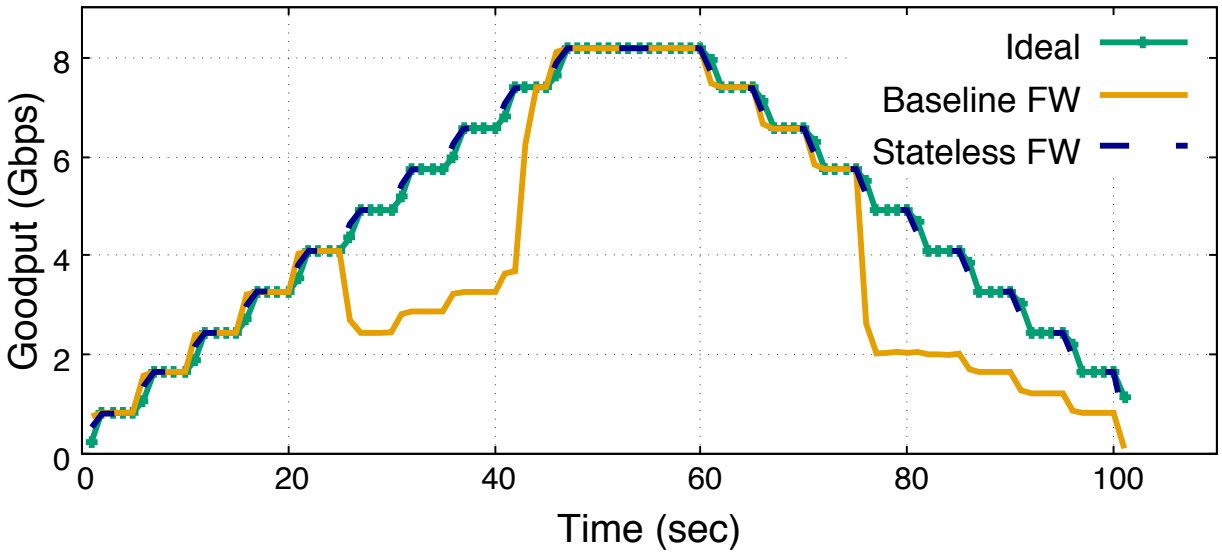


Figure 4.8: Goodput (Gbps) for stateless and baseline firewalls while scaling out (t=25s) and in (t=75s).

instance.

As Figure 4.8 shows, the stateless firewall matches the base goodput. That is due to the fact that 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.,* the SYN packets already sent to and stored at the the first firewall's flow table). By the time we scale in (at second 75), the baseline firewall has a bad time because it does not recognize the merged traffic that was assigned to the other firewall after scaling out. This shows that when state is decoupled from processing, as the case in our StatelessNF, packet loss becomes negligible when scaling in and out.

4.7 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, which we specifically avoided to explore an extreme design point, for certain types of state (read mostly), to enable lookups to be serviced locally, 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).

Use of cache: Caching in stateless network functions can enhance their performance, but the consistency can be violated. In the simplest case, we can leverage techniques like consistent replication and cache invalidation to eliminate such consistency issues (at the cost of higher update cost).

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 [51] (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.

4.8 Related work

Apart from the most related work, which we highlight in Chapter 3, below we elaborate on additional related work along two categories.

Disaggregation: The concept of decoupling processing from state follows a line of research in disaggregated architectures. [83], [82], and [97] all make the case for disaggregating memory into a pool of RAM. [60] 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. In contrast to a previous attempt at a similar decoupled architecture [70], in this work we provide a complete implementation of both the entire system as well as more network functions, a complete evaluation demon-

strating scalability and resilience, and and optimized design that achieves significantly higher throughput and lower latency.

Micro network functions: The consolidated middlebox [113] 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 [99] provides network operators with a coherent system for managing network functions while enabling developers to focus on implementing new network functions. In effect, each are re-thinking the architecture of network functions, and complementary.

Chapter 5

StatelessNF in the real world

We have always believed that academia should have an impact beyond research publication. One of the main outcomes of this thesis is that we were able to commercialize StatelessNF technology and receive strategic investments for it. This provides some indication that the problem we have identified and solved in this thesis is real and that there is a need for fundamentally solving it. In this chapter, we highlight on the data we collected from the market and our deployment of statelssNF in two data centers to illustrate how StatelessNF can integrate into real data centers.

5.1 Is there a need for such solution?

Market Research: To verify the benefits of StatelessNF outside of the academic world, we conducted intense market development where we performed over 150 interviews with data center operators (such as Sendgrid[21] and Twitter), managed service providers (such as OnlineTech[18], Zayo Cloud[30]), and even network vendors (such as Cisco and Juniper Networks) as part of the NSF I-Corps[17] award we received in November 2016. The majority of these interviews were in person in California, Colorado, Michigan, and Texas. We attended two cloud and data center conferences, NANOG-68 [13] in Dallas, Texas and Structure Conference [27] in San Francisco, California.

We found out that while the general assumption is that the problems with network agility can be solved through public cloud providers (Amazon EC2, Google Cloud, Microsoft Azure), that is not entirely true as there are millions of data centers around the world and only hundreds of them belong to public cloud companies. This indicates to us the market size and opportunity as there are thousands of data centers that

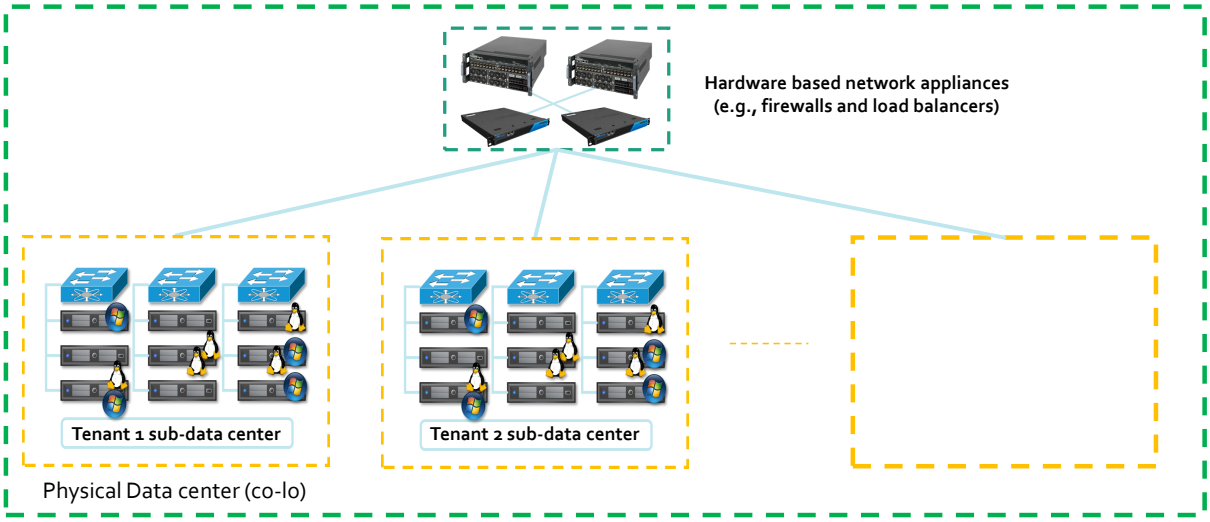


Figure 5.1: Traditional Network Function (Appliance Design)

do not have the resources to build their own agile network function solutions.

Today's network function deployment: To examine how current network services are deployed and offered today, we looked into different data center infrastructures. In one case, as shown in Figure 5.1, the physical network appliances are managed by co-location data center operator. One problem with such architecture is that the tenant has poor or no control over the network services that are provided to him. Thus, the service provider has to manage and deploy all network services. A complex problem with high operational cost that gets worse when the number of tenant and their traffic increase. Another problem, is that we have static slicing of resources within the network appliance. In this case there would be no elasticity in terms of resources used.

In another case, shown in Figure 5.2, the managed service provider in the co-location data center buys a pool of virtual network functions and offers them as managed services to tenant. For example, a firewall is instantly provisioned per tenant. The problem with this model is that the tenant also has poor and no control over the services provided to him. In addition, the service provider has to buy a large pool of licenses that are most of the time not used, thus they are not fully utilized. This leads to static assignment of licenses to tenant, over purchasing, and services disruption and downtime to change or update licenses.

The Stateless platform on the other hand is designed with agility and ease of use as a primary goal.

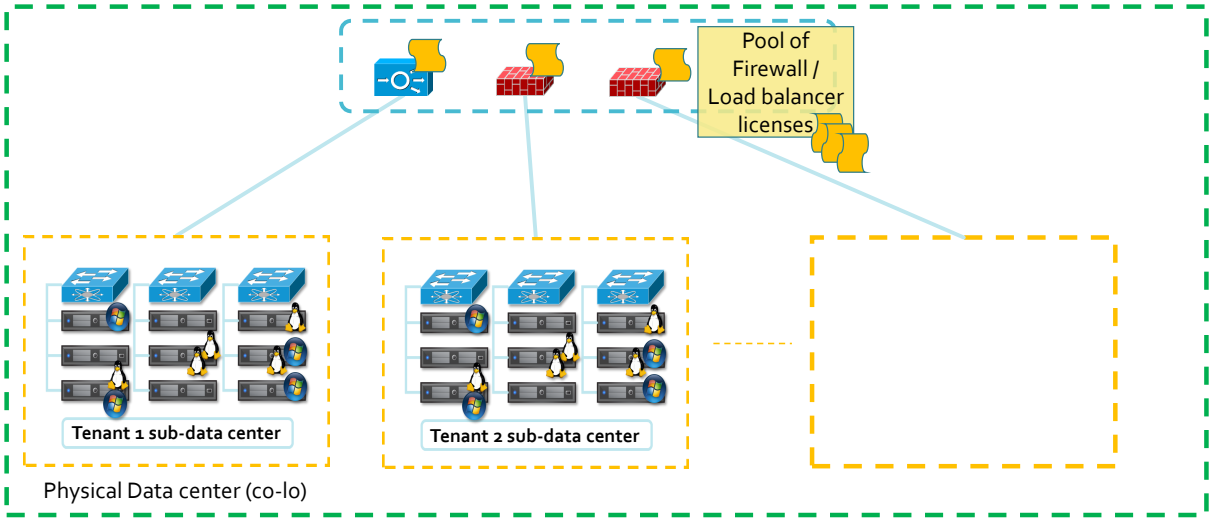


Figure 5.2: Physical network appliances sliced by Co-lo and offered as managed service to tenant

In addition to its performance advantages when it comes to scalability and resiliency, Stateless provides two dash boards for both data center operators and tenants. This allow tenants to have easy and simple control of their network functions and traffic, while at the same time, reduces the burden on data center operators as they don't need to constantly manage tenants' network functions. In addition, as we offer pay-as-you-go model where tenants are charged based on their usage (*e.g.,* traffic), this offers the data center operators and tenants a more cost and operational efficient model as the network functions can scale elasticly in and out without any intervention from network operators.

Complexity and cost of network functions: In addition to the problem in the above infrastructure design mentioned, the data centers and network operators greatly suffer from the existing solutions that are structured with the appliance design. For example Managed Service Providers like Rackspace[20] offer network services such as firewalls to their tenants. Since they are stuck with appliances model, the process to deploy, configure, and deploy new network functions, as shown in Figure 5.3, takes from 1 to 4 weeks. This greatly increase the operational cost for such data centers as 50%-75% of the network engineer time is spent on fixing and configuring network functions.

In addition, enterprise companies like Sendgrid[21] and Sovrn[25], who manage their own data centers, suffer from the managing their network functions especially when it comes to handle increase of traffic

as they have to replace existing appliances with bigger ones. The process is usually disruptive and costly.

With StatelessNF, the whole process of deploying, configuring and connecting network functions takes less than 2 minutes. And since the process is automated and network engineers don not need to handle scalability or failure, we expect network engineers to spend less than 5% of their time managing and connecting network functions. With StatelessNF, since we decoupled the state from processing, we can update either of them seamlessly, as opposed to scheduling disruptive maintenance windows to perform updates. And since it is software that runs on X86 servers and network operators are only charged for what they actually used, this reduces their costs by more than 50%.

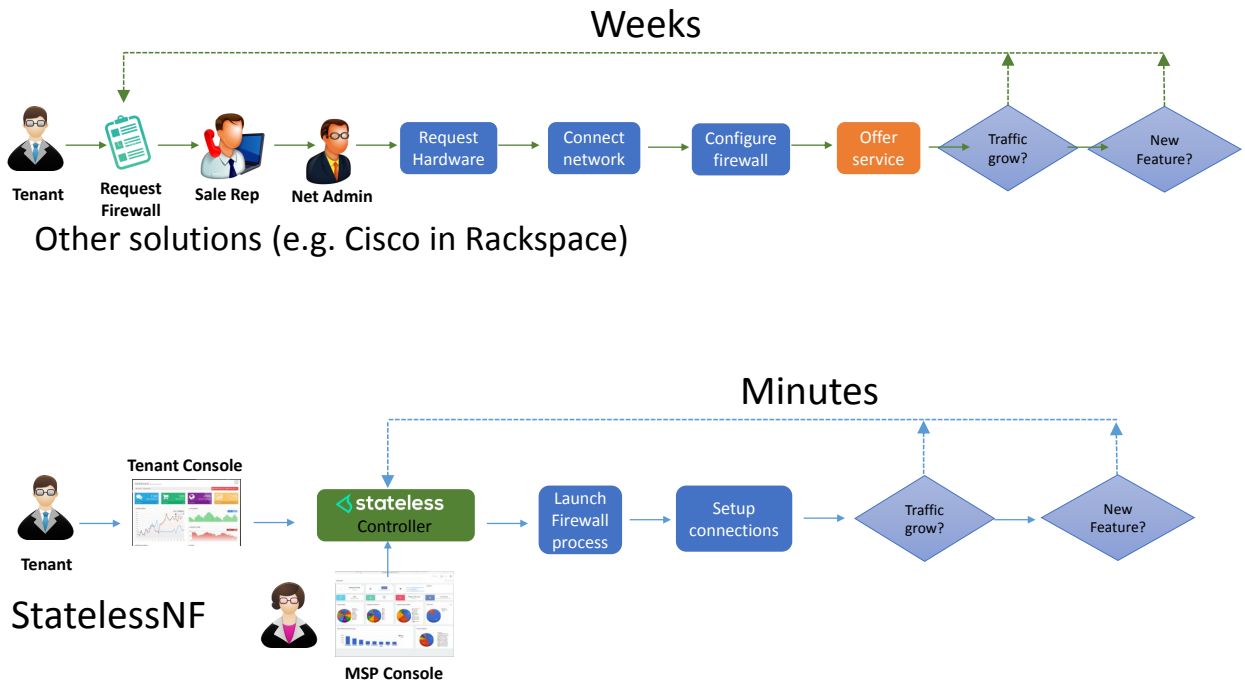


Figure 5.3: Network Services Deployment Model in current solutions and StatelessNF

5.2 StatelessNF deployment

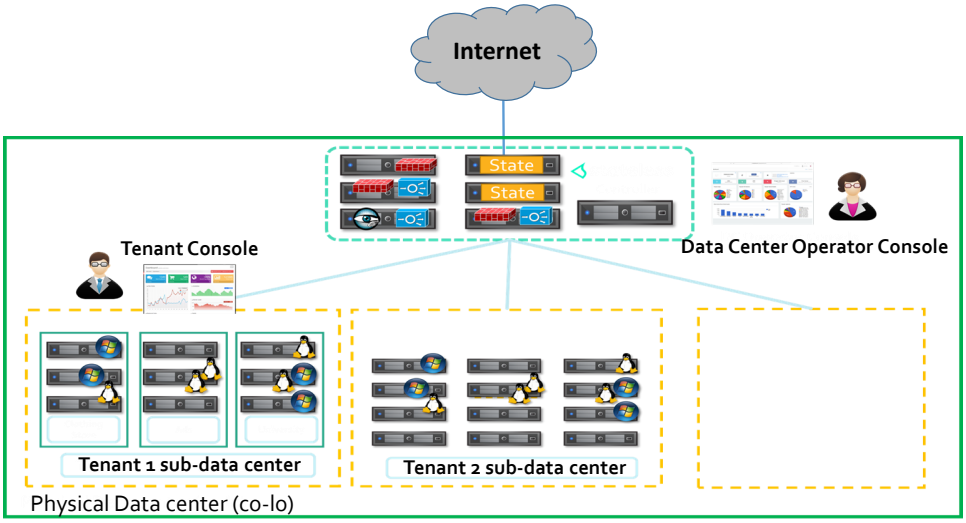


Figure 5.4: StatelessNF deployment within EarthNet data center

We are currently working with two companies to deploy StatelessNF. The first one is EarthNet Data Center[6]. A co-location data center facility that also provides managed services to its tenants. As shown in Figure 5.4, by having StatelessNF system, EarthNet will be able to have a new revenue stream as it can offer network services such as firewall and load balancer to its tenants through StatelessNF platform. The second company is TetherView Cloud Service Provider [28], a managed service provider that offers virtual desktops to other businesses such as accounting firms, clinics, and universities. As shown in Figure 5.5, TetherView's private data center is co-located within a larger data center. Unlike EarthNet, where StatelessNF platform will be shared among different tenants, with TetherView, StatelessNF will be deployed within its private data center. With StatelessNF, TetherView no longer needs to purchase any network appliances or licensees as it will be only be charged for the actual traffic that it uses through StatelessNF platform.

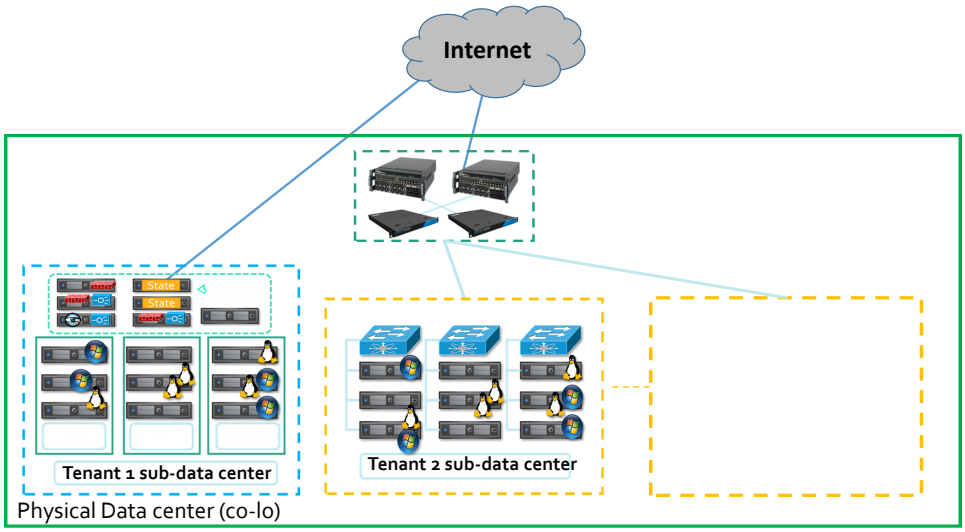


Figure 5.5: StatelessNF deployment within TetherView data center

Chapter 6

Conclusion and future work

In this thesis we proposed a novel disaggregated architecture for network fuctions, called StatelessNF. In doing so, we enabled networks to be highly agile without sacraficing performance. In this chapter we summarize the contributions, discuss future research directions, and provide concluding thoughts.

6.1 Contributions

Our main innovation is breaking apart the processing and state in network functions. This architecture achieves greater agility across multiple dimensions. First, network services can be easily scaled in or out to match incoming demand. Second, network services can be developed, tested, and deployed independently from other services. Third, network services became failure resilient. And fourth, network services became more secure as we can manage and monitor all of their state from a centralized location. We showed that the main stated potential drawback, performance, is actually not a problem. Our evaluation with a complete implementation demonstrates these all of the above 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.

6.2 Future Directions

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 as we highlight in this section. We instead focused on demonstrating the viability of a novel architecture which, we believe, fundamentally gets at the root of the important problem. Below, we highlight some projects that we intend to work on as part of our future work.

6.2.1 Testing in an uncontrolled environment

So far we have tested Stateless technology in a controlled environment with defined sets of traffic loads. Putting the system in a production environment with real world traffic will clearly test its stability and efficiency. Through our commercialization efforts in founding Stateless, Inc. we will have great opportunity to perform these tests. We have been approached by multiple companies to deploy and test our system in their data centers. We believe such environment will provide us with ideal use cases to test our system. for example, focusing on:

- **Monitoring and Reaction Time:** Stateless technology was designed and developed with two important requirements: scalability (*e.g.,* the system can add/reduce network functions based on traffic load), and failure resiliency (*e.g.,* when a network functions fails the system will launch a new one and direct traffic to it. In order to to maintain these two requirements, Stateless system has to monitor its different components and reacts to certain events. The amount of time the system takes to monitor and react is very critical as we are handling millions of network packets per seconds.

- **Isolating tenant connections:** The data centers where we are deploying Stateless technology provide network services to their hosted tenant. Thus, these data centers require our system to handle hundreds of tenant connections concurrently while ensuring the privacy, security, and promised performance of these tenants. The challenge is that these connections will share Stateless system's resources (CPU, memory, etc). However, there are multiple techniques we can leverage to classify and separate traffic, and isolate each tenant's resources. For example, 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.

6.2.2 Integrating the Data store into the Network Function Nodes

Traditionally, we consider the data store as a remote node. We propose exploring an integration of the data store nodes and the network function nodes, or more generally, a rethinking of the concept that the data store is separate from and independent of the nodes accessing the data store. This has two main

advantages that must be considered, along with the potential for negative impacts such as reduced fault tolerance. First, similar to a cache within each node, it can reduce latency and increase bandwidth to access data. Unlike a cache, the data store nodes functionality is to replicate data for resilience, but not provide consistency across all accessing nodes. That is, with a caching architecture, each node accesses data and caches it locally. This, in turn, requires mechanism to maintain cache coherency. With an integrated data store, access to data goes to the nodes actually storing that data (which may be replicated among a few nodes, and coherency needs to be maintained between that small subset of nodes). This subtle difference makes this more scalable. Second, if we do not use replication for fault tolerence and have a 1-1 mapping , this effectively reproduces the architectures which used migration of data from within the network functions to other instances (e.g., Split/Merge [46], OpenNF [19], Router Grafting [33]). It does so, however, with a general data store, moving the burden from every network function implementation to a common data store (which, of course, would require the data store to include the ability to control data placement).

6.2.3 Going stateless with other network functions

While we experimented with three common middleboxes (Firewall, NAT , and Loadbalancer), we expect stateless design should be equally applicable to other open source and commercial applications with different performance requirements (*e.g.,* HAProxy[10], F5 Firewalls[7], Snort[23], Router Grafting[76]). While these systems have different and wider space of network states, we can leverage recent solution to define and extract the needed state[77]. To meet the different performance requirements, routing software is control plane software where a small added latency will not be noticeable. Intrusion detection systems are off path, so as long as they can achieve similar throughput, the extra latency can be tolerated. That said, and with software development in which the programmer designs the network function to be stateless (rather than force it into legacy software), we believe that stateless network functions can achieve the needed performance.

6.2.4 Network functions in service chaining

While the notion of service chaining exists [10] and research efforts have studied how to make them possible [18, 44], these efforts relate to the traditional (coarse grained) notion of a network function. Rather than viewing network functions as all-encompassing functions, we seek to investigate the notion of micro network functions, where finer grained functionality can be composed to create a variety of functions. Much like the trend towards microservices in cloud computing [36], we envision that micro network functions will emerge to match the computing environments as they become much more dynamic (in scale and function). For example, we might not need a full featured IDS between two services, but a simple bandwidth monitor coupled with a latency optimizer (capitalizing on multipath) between two services. As mentioned, the commonly provided examples of network functions are coarse grained functions such as firewalls. The consolidated middlebox [50] work observed that many of these middleboxes duplicate the same functionality (e.g., parsing http messages) (also mentioned in [23]). As such, they consolidated multiple middleboxes into a single device, sharing functionality where possible. From this, they created a much more efficient architecture. With stateless network functions, we have decoupled the state from the processing. As such, we propose that a new form of network function micro network functions become possible, without being constrained to a shared device. The key challenge (as with many aspects of this proposal) is the tradeoff between overhead and functionality. With a consolidated middlebox, there is the presumption that there will be an interface to exchange data from one module that services multiple other processing modules. Within a consolidated middlebox, this communication is local, whereas with StatelessNF, it would be implicit through the data store. We will investigate, first, whether a direct partitioning as used in other modular works [50] is practical and beneficial, and second, explore micro network functions as a ground-up exercise to support micro services.

6.2.5 Automating the management and configuration of network functions

Of course, re-designing the network functions' architecture only solves one aspect of their challenges. Another paradigm we started investigating [71], is how network function are configured and managed. As

we show in our extended project, Seit, network functions are traditionally managed in a mostly static manner (*e.g.,* manually configure a firewall). This makes them prone to configuration errors and unable to adapt to the dynamic behavior of today's network application. In our preliminary work, we present frameworks that define and manage the interactions between cloud services and network functions in order to dynamically and automatically configure, optimize, and secure their interactions via a continuous feedback loop. We introduce new metrics to capture the quality of cloud service and its consumption by its users. We show how our metrics can be used by cloud service consumers and providers to improve the security and management of their network functions (*e.g.,* setting rate limits, protecting against DDoS, etc.).

6.3 Concluding Thoughts

The networking space is under big transformation with all the changes we see with cloud, Internet of Things, and mobile communications. So in order for the infrastructure to be agile and adapts in both scale and function to the dynamic changing behavior, a fundamental shift in its architecture design is needed. In contrast to existing approaches, which built on top of the appliance architecture, in this thesis, we designed from the ground up, and ended with a design that fulfills all the requirements to have agile networks. We defined and analyzed current deployment challenges in modern network functions, and presented real use cases of our systems- EdgePlex at AT&T and StatelessNF at Stateless, Inc.

Bibliography

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

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

[3] D-ITG, Distributed Internet Traffic Generator. http://traffic.comics.unina.it/software/ITG/.

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

[5] Docker containers. http://docker.com.

[6] Earthnet data centers. http://earthnet.com.

[7] F5 Solutions. https://f5.com/products/platforms/appliances.

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

[9] Github: PlatformLab/RAMCloud. https://github.com/PlatformLab/RAMCloud.

[10] HAProxy. http://www.haproxy.org/.

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

[12] Nagios. http://www.nagios.org/.

[13] Nanog Conference. https://nanog.org/meetings/nanog68/home.

[14] Netflix Outage. https://www.theregister.co.uk/2015/09/20/aws_database_outage/.

[15] NFV Market Size. https://www.statista.com/statistics/461573/sdn-and-nfv-markets-worldwide-by-region/.

[16] NOX SDN Controller. http://www.noxrepo.org/.

[17] NSF I-Corps. https://www.nsf.gov/i-corps.

[18] Onlinetech Data Centers. www.onlinetech.com.

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

[20] Rackspace Managed Dedicated and Cloud Computing Services . https://rackspace.com.

[21] Sendgrid Email Delivery Service. https://www.sendgrid.com.

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

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

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

[25] Sovrn Advertising and Tools for Publishers. https://www.sovrn.com.

[26] Stack overflow. http://stackoverflow.com/.

[27] Structure Conference. www.structureconf.com.

[28] Tetherview cloud service provider. http://tetherview.com.

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

[30] Zayo Cloud Group. www.zayo.com/services/cloud-infrastructure/.

[31] Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. http://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[32] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. Commun. ACM, 1975.

[33] Amazon. Amazon ec2. http://aws.amazon.com/ec2/.

[34] Amazon. Amazon ec2 security groups. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html.

[35] Amazon. Amazon virtual private cloud. http://aws.amazon.com/vpc/.

[36] Amazon. Possible Insecure Elasticsearch Configuration. http://aws.amazon.com/security/security-bulletins/possible-insecure-elasticsearch-configuration/, May 2014.

[37] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In Proc. SIGCOMM, 2016.

[38] Ramnath Balasubramanyan, Bryan R Routledge, and Noah A Smith. From tweets to polls: Linking text sentiment to public opinion time series. In Proc. AAAI Conference on Weblogs and Social Media, 2010.

[39] Hitesh Ballani, K Jang, and Thomas Karagiannis. Chatty Tenants and the Cloud Network Sharing Problem. Proc. of NSDI, 2013.

[40] Tom Barbette, Cyril Soldani, and Laurent 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.

[41] Yair Bartal. Firmato: A novel firewall management toolkit the hebrew university of jerusalem. Proceedings of the 1999 IEEE Symposium on Security and Privacy, 22(4):381–420, 2004.

[42] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru 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.

[43] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev., 44(3):87–95, July 2014.

[44] Carreira, Joao. [ramcloud-dev] Replicating Ramcloud results (latencies). `https://mailman.stanford.edu/pipermail/ramcloud-dev/2014-December/001033.html`, December 2014.

[45] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. ACM SIGCOMM Computer Communication Review, 37(4):1–12, 2007.

[46] Martìn Casado, MJ Freedman, and Justin Pettit. Ethane: Taking control of the enterprise. ACM SIGCOMM Computer Communication Review, 37(4), 2007.

[47] Angela Chiu, Vijay Gopalakrishnan, Bo Han, Murad Kablan, Oliver Spatscheck, Chengwei Wang, and Yang Xu. Edgeplex: Decomposing the provider edge for flexibilty and reliability. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, pages 15:1–15:6, New York, NY, USA, 2015. ACM.

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

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

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

[51] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion 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.

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

[53] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In USENIX Symposium on Networked Systems Design and Implementation (NSDI), Mar. 2016.

[54] Gregory Frazier, Quang Duong, M Wellman, and Edward Petersen. Incentivizing responsible networking via introduction-based routing. Trust and Trustworthy Computing, 6740, 2011.

[55] Simson Garfinkel. The criminal cloud. http://www.technologyreview.com/news/425770/the-criminal-cloud/, Oct. 2011.

[56] Aaron Gember, Prathmesh Prabhu, Zainab Ghadiyali, and Aditya 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.

[57] Aaron Gember-Jacobson and Aditya 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.

[58] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In Proc. SIGCOMM, 2014.

[59] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In Proc. CoNEXT, 2010.

[60] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott 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.

[61] Chandler Harris. Data center outages generate big losses, May 2011.

[62] Doug Henschen. 10 tips: Tap consumer sentiment on social networks. http://www.informationweek.com/software/information-management/10-tips-tap-consumer-sentiment-on-social\\-networks/d/d-id/1105234, July 2012.

[63] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[64] Jinho Hwang, K. K. Ramakrishnan, and Timothy 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.

[65] IBM. Vpn scenario: Basic business to business connection. http://tinyurl.com/ibm-vpn-b2b. Access March 4th, 2013.

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

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

[68] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. Softflow: A middlebox architecture for open vswitch. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 15–28, Denver, CO, June 2016. USENIX Association.

[69] JNRPE. Java Nagios Remote Plugin Executor. http://www.jnrpe.it/cms/index.php.

[70] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless network functions. In Proc. ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox), 2015.

[71] Murad Kablan, Carlee Joe-Wong, Sangtae Ha, Hani Jamjoom, and Eric Keller. The cloud needs a reputation system. CoRR, abs/1509.09057, 2015.

[72] Murad Kablan, Eric Keller, and Hani Jamjoom. QoX: Quality of Service and Consumption in the Cloud. In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), Denver, CO, June 2016.

[73] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. SIGCOMM Comput. Commun. Rev., 44(4):295–306, August 2014.

[74] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The Eigentrust Algorithm for Reputation Management in P2P Networks. In Proc. International Conference on World Wide Web (WWW), 2003.

[75] Ari Kaplan. Defending Data: An Inside Look at How Corporate Security Officials Are Navigating a Constantly Shifting Information Landscape. http://www.nuix.com/analyst-briefs/defending-data, 2014.

[76] Eric Keller, Jennifer Rexford, and Jacobus 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.

[77] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the way for nfv: Simplifying middlebox modifications using statealyzr. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pages 239–253, Santa Clara, CA, USA, March 2016. USENIX Association.

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

[79] R. Landa, D. Griffin, R. Clegg, E. Mykoniati, and M. Rio. A sybilproof indirect reciprocity mechanism for peer-to-peer networks. In INFOCOM 2009, IEEE, pages 343–351, April 2009.

[80] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. ACM Trans. Comput. Syst., 7(4), Nov. 1989.

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

[82] 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.

[83] 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.

[84] Jiefei Ma, Franck Le, Alessandra Russo, and Jorge Lobo. Detecting distributed signature-based intrusion: The case of multi-path routing attacks. In IEEE INFOCOM. IEEE, 2015.

[85] Yuqing Mao, H Shen, and C Sun. From credit and risk to trust: towards a credit flow based trust model for social networks. Proceedings of the 17th ACM international conference on Supporting group work, pages 209–218, 2012.

[86] N McKeown and Tom Anderson. Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2), 2008.

[87] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69–74, Aug 2008.

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

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

[90] Rene Millman. Hackers target Elasticsearch to set up DDoS botnet on AWS. `http://www.cloudpro.co.uk/cloud-essentials/cloud-security/4353/hackers-target-elasticsearch-to-set-up\\-ddos-botnet-on-aws`, Aug. 2014.

[91] Alan Mislove, A Post, P Druschel, and KP Gummadi. Ostra: Leveraging trust to thwart unwanted communication. Proc. of NSDI, 2008.

[92] Alan Mislove, Ansley Post, Peter Druschel, and P Krishna Gummadi. Ostra: Leveraging trust to thwart unwanted communication. In Proc. of USENIX NSDI, San Francisco, CA, April 2008.

[93] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. SIGOPS Oper. Syst. Rev., 33(5):217–231, December 1999.

[94] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In USENIX Annual Technical Conference (USENIX ATC), Jul. 2015.

[95] Marc Norton. Optimizing pattern matching for intrusion detection. Sourcefire, Inc., Columbia, MD, 2004.

[96] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), pages 29–41. ACM, Oct 2011.

[97] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. SIGOPS Oper. Syst. Rev., 43(4), Jan. 2010.

[98] Christoph Paasch and Olivier Bonaventure. Multipath TCP. Communications of the ACM, 57(4):51–57, April 2014.

[99] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP). ACM, Oct 2015.

[100] Vern Paxson. Bro: a system for detecting network intruders in real-time. Computer networks, 31(23-24), 1999.

[101] PCI SIG. Single Root I/O Virtualization. `https://www.pcisig.com/specifications/iov/single_root/`.

[102] Michael Piatek, Tomas Isdal, Arvind Krishnamurthy, and Thomas Anderson. One hop reputations for peer to peer file sharing workloads. In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, NSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[103] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In Proc. SIGCOMM, 2012.

[104] Lucian Popa, Minlan Yu, Steven Y. Ko, Sylvia Ratnasamy, and Ion Stoica. CloudPolice: Taking Access Control out of the Network. In Proc. Workshop on Hot Topics in Networks (HotNets), 2010.

[105] Rahul Potharaju and Navendu 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.

[106] Dongyu Qiu and Rayadurgam Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. ACM SIGCOMM Computer Communication Review, 34(4):367–378, 2004.

[107] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication: A high availability framework for middleboxes. In Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC). ACM, Oct 2013.

[108] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew 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.

[109] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In Proc. Conference on Computer and Communications Security (CCS), 2009.

[110] Luigi 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.

[111] RoCE. Rdma over converged ethernet. `https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet`.

[112] Andy. Sayler, Eric. Keller, and Dirk. Grunwald. Jobber: Automating inter-tenant trust in the cloud. In Proc. Workshop on Hot Topics in Cloud Computing (HotCloud), 2013.

[113] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu 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.

[114] Alex Sherman, Jason Nieh, and Clifford Stein. FairTorrent: A Deficit-based Distributed Algorithm to Ensure Fairness in Peer-to-peer Systems. IEEE/ACM Transactions on Networking, 20(5):1361–1374, Oct 2012.

[115] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-recovery for middleboxes. SIGCOMM Comput. Commun. Rev., 45(4):227–240, August 2015.

[116] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In Proceedings of the ACM SIGCOMM. ACM, Aug 2012.

[117] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In Proc. USENIX Conference on Networked Systems Design and Implementation (NSDI), 2011.

[118] Michael Sirivianos, Jong Han Park, Xiaowei Yang, and Stanislaw Jarecki. Dandelion: Cooperative content distribution with robust incentives. In 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference, ATC'07, pages 12:1–12:14, Berkeley, CA, USA, 2007. USENIX Association.

[119] SWATCH. Simple log watcher. `http://sourceforge.net/projects/swatch/`.

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

[121] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. IEEE Communications Magazine, 35:80–86, 1997.

[122] JJ Thompson. Why Perimeter Defenses Are No Longer Enough. `http://www.darkreading.com/why-perimeter-defenses-are-no-longer-enough/a/d-id/1235005`, 2014.

[123] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based Defenses against Cross-VM Side-channels. In 23rd USENIX Security Symposium (USENIX Security), San Diego, CA, Aug. 2014.

[124] Verizon. 2014 data breach investigations report. `http://www.verizonenterprise.com/DBIR/`, 2014.

[125] A Wool. A quantitative study of firewall configuration errors. Computer, 37:62–67, 2004.

[126] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham D. Flaxman. SybilGuard: Defending Against Sybil Attacks via Social Networks. IEEE/ACM Transactions on Networks, 16(3):576–589, June 2008.

[127] Xiaodong Yu, Wu-chun Feng, Danfeng (Daphne) Yao, and Michela 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.

[128] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In Proc. Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox), 2016.

[129] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In Proc. Conference on Computer and Communications Security (CCS), 2012.

[130] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In Proc. Conference on Computer and Communications Security (CCS), 2014.