Scalable and Reliable Middlebox Deployment

by

Milad Ghaznavi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Stefan Schmid Professor Department of Computer Science University of Vienna
Supervisor:	Raouf Boutaba Professor David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Bernard Wong Associate Professor David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Ali José Mashtizadeh Assistant Professor David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Wojciech Golab Associate Professor Department of Electrical and Computer Engineering University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This dissertation includes first authored peer reviewed materials that have appeared in conference and journal proceedings published by the Institute of Electrical and Electronics Engineers (IEEE) and Association for Computing Machinery (ACM).

The ACM's policy on reuse of published materials in a dissertation is as follows:

> *"Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included."*

The following list serves as a declaration of the Versions of Record for works included in this dissertation:

Portions of Chapter 2:

Milad Ghaznavi, Nashid Shahriar, Shahin Kamali, Reaz Ahmed, and Raouf Boutaba. Distributed service function chaining. *IEEE Journal on Selected Areas in Communications*, 35(11):2479–2489, November 2017. https://doi.org/10.1109/JSAC.2017.2760178

Portions of Chapter 3: Milad Ghaznavi, Ali Jose Mashtizadeh, Bernard Wong, and Raouf Boutaba. Constellation: A high performance geo-distributed middlebox framework. Technical Report arXiv:2003.05111 [cs.NI], ArXiV, March 2020. https://arxiv.org/abs/2003.05111

Portions of Chapter 4:

Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali Jose Mashtizadeh. Fault tolerant service function chaining. In *Proceedings of the 2020 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '20, New York, NY, USA, August 2020. Association for Computing Machinery (ACM). *To appear*

Abstract

Middleboxes are pervasive in modern computer networks providing functionalities beyond mere packet forwarding. Load balancers, intrusion detection systems, and network address translators are typical examples of middleboxes. Despite their benefits, middleboxes come with several challenges with respect to their scalability and reliability.

The goal of this thesis is to devise middlebox deployment solutions that are cost effective, scalable, and fault tolerant. The thesis includes three main contributions: First, distributed service function chaining with multiple instances of a middlebox deployed on different physical servers to optimize resource usage; Second, Constellation, a geo-distributed middlebox framework enabling a middlebox application to operate with high performance across wide area networks; Third, a fault tolerant service function chaining system.

Acknowledgements

I would like to express my special gratitude to my supervisor, Professor Raouf Boutaba, for his support throughout my doctoral studies. Raouf has been amazingly patient with me allowing me to mature as a researcher; he has been a wonderful source of knowledge and experience guiding me through my graduate career.

I extend my thanks to my examining committee, Professor Stefan Schmid, Professor Wojciech Golab, Professor Bernard Wong, and Professor Ali José Mashtizadeh, for reviewing my thesis and participating in my defense during the COVID 19 pandemic. I have been incredibly fortunate to have closely worked with and learned from Bernard and Ali.

I would thank my friends and colleagues, Benjamin Cassell, Shihab Rahman Chudhury, and Shahin Kamali for their collaboration and friendship, in particular my friend, Ali Abedi. Big thanks to Elaheh Jalalpour, my colleague and good friend, for all joyful and stressful moments we experienced throughout our graduate studies.

My deepest appreciation goes to my parents, Touran and Nosratollah, for everything they have done for me; my father will be forever in my heart. I am immensely grateful to my bothers and sisters for their unconditional love, support, and confidence; special thanks to my brother, Mahmoud, for his support over the past few years in Canada.

To my father

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Middleboxes, such as firewalls, WAN optimizers, and network address translators are pervasive in computer networks. They enable modern networks to perform advanced functionalities that are beyond packet forwarding. For example, they protect against threats by filtering malicious traffic, improve network performance by optimizing resource usage, and mitigate scarcity of IPv4 addresses [131, 142, 147, 167].

Deploying and operating middleboxes have been hindered by employing hardware middlebox appliances. A small number of vendors customize hardware and software of these appliances for specific functionalities. At the beginning of their service life time, resources of these appliances are under utilized, as they are over provisioned for peak loads [142]. On the other hand, fixed hardware resources become obsolete quickly and unable to handle average network loads. Providing fault tolerance, which is essential for middlebox applications, can be also burdensome for middlebox hardware appliances. A backup infrastructure requires purchasing and configuring a duplicate set of hardware appliances [145]

Network Function Virtualization (NFV) [44] aims to accelerate and facilitate innovations for middleboxes by transforming the way they are provided. NFV decouples middlebox functionalities from hardware by creating software instances, for example virtual machines or containers that run on commodity servers. This allows network operators to install and remove middlebox instances on demand in response to traffic loads. Network operators can also devise more general and flexible mechanisms to provide fault tolerance for a diverse set of middlebox applications and deployments.

NFV can revolutionize the networking industry; however, its realization in practice is challenging. It still has to face several theoretical and system design challenges on its path to success.

This dissertation presents three contributions aiming to bridge the gap between NFV promises and practical realization:

- *Cost reduction:* Distributed service function chaining optimizes the cost of deploying middleboxes in cloud infrastructures. Middleboxes are commonly chained together in an ordered sequence, e.g., firewall→IDS→proxy, to meet high level service requirements. This middlebox deployment is called a *service function chain* [130, 131]. Our approach places middlebox instances of a chain distributedly where multiple instances of a single middlebox can be deployed in different servers. Compared to prior approaches that do not allow such a distributed deployment, our approach decouples the throughput of a middlebox from the underlying hardware while utilizing resources more efficiently.

- *Dynamic scaling:* Constellation is a framework for a geo-distributed deployment of middleboxes. Constellation allows network operators to dynamically scale the number of middlebox instances that cooperatively process traffic over a wide area network. Using loosely coupled replication, Constellation enables middlebox instances to be deployed across a geo-distributed sites with high latency links, while prior middlebox frameworks are designed and optimized only for deployments in local area networks with low latency.

- *Fault tolerance:* Fault Tolerant service function Chaining (FTC) is a system that provides fault tolerance to an entire chain of middleboxes. FTC's design takes advantage of the structure of a service function chain to provide fault tolerance. This design allows FTC to support higher performance than can be achieved by prior systems; their design comes with high performance overhead because they consider individual middleboxes as independent fault tolerant units that together form a fault tolerant chain.

These contributions enable network operators to support middlebox deployments that are cost efficient, scalable, and fault tolerant. Next, we briefly describe each contribution.

1.1 Distributed Service Function Chaining

The deployment of a service function chain involves selecting and instantiating a number of middlebox instances, placing these instances, and routing traffic through them. In the current optimization models of a chain deployment, instances of the same function are assumed to be identical, while typical service providers offer middleboxes with heterogeneous throughput and resource configurations. The instances of a middlebox are installed in a single server which limits

a chain to the throughput of a few instances that can be installed in a single physical machine. Furthermore, the selection, placement, and routing problems are solved in isolation.

We present distributed service function chaining that coordinates these operations, places instances of a middlebox distributedly, and selects appropriate instances from typical middlebox offerings. This deployment uses network resources efficiently and decouples a chain's throughput from that of physical servers. Our specific contributions in this work are:

- Providing a mathematical model for the problem of distributed service function chaining and proving its NP hardness.
- Proposing *Kariz*, a local search heuristic that solves this problem for large networks.

We implemented Kariz and evaluated its performance. In accepting requests for deploying middlebox chains, our results show that Kariz achieves an acceptance ratio of 76–100% at an extra cost of at most 24% compared with the optimal solution.

1.2 Constellation: A Geo-Distributed Middlebox Framework

Middleboxes are increasingly deployed across geographically distributed data centers. The instances of a middlebox distributed across different sites maintain and share state information to cooperatively process traffic. In these scenarios, the WAN latency between different sites can significantly impact the middlebox performance. The deployment across such infrastructures can even become impractical due to the high cost of remote state accesses over wide area networks.

Constellation is a framework for the geo-distributed deployment of middleboxes. Constellation uses asynchronous state replication of specialized state objects to achieve high performance and scalability. Our framework has the following key attributes:

- Highly scalable and performant state sharing using asynchronous state replication.
- Providing convergent state objects that can be independently updated across different instances.
- Building an efficient and reliable multicast state replication layer that synchronizes convergent state objects using their own properties.

Our evaluation of Constellation shows that compared to the state of art [173], Constellation improves the throughput by a factor of 96 over wide area networks.

1.3 Fault Tolerant Service Function Chaining

Tolerating failures when they occur along chains is imperative to the availability and reliability of enterprise applications. Service outages due to chain failures severely impact customers and cause significant financial losses. Making a chain fault tolerant is challenging since, in the case of failures, the state of faulty middleboxes must be correctly and quickly recovered while providing high throughput and low latency.

We present *FTC*, a novel system design and protocol for fault tolerant service function chaining. FTC provides strong consistency with up to f middlebox failures for chains of length $f + 1$ or longer without requiring dedicated replica nodes. In FTC, state updates caused by packet processing at a middlebox are collected, piggybacked into the packet, and sent along the chain to be replicated. FTC makes the following specific contributions:

- Extending the replication protocol in [163] to support a chain of middleboxes.
- Improving the usability of multicore middleboxes by introducing transactional packet processing where a middlebox thread processes each packet in a transaction.
- Improving the replication performance using data dependency vectors that track dependencies between packet transactions with higher flexibility compared to prior thread-based techniques.

We implemented and evaluated FTC. Our results for a chain of 2 to 5 middleboxes show that FTC improves throughput by 2× to 3.5× compared with the state-of-the-art [146] with lower latency per middlebox.

1.4 Dissertation Plan

This dissertation proceeds as follows to detail our contributions. In Chapter 2, we present distributed service function chaining. We discuss Constellation in Chapter 3. Next in Chapter 4, we present fault tolerant service function chaining. Finally, we discuss the future of middlebox deployments and conclude this dissertation in Chapter 5.

Chapter 2

Distributed Service Function Chaining

A service-function chain, or simply a *chain*, is an ordered sequence of middleboxes composing a service [131]. For example in a typical data-center network, traffic from a server passes through an IDS, a firewall, and a NAT before reaching to the Internet [167]. Until recently, middleboxes have been vertically integrated in dedicated hardware middleboxes, i.e., a chain of pricey hardware middleboxes are provisioned to provide throughput for peak-load, and traffic must be routed through fixed locations in which these middleboxes are placed [130].

By decoupling network functions from underlying hardware, NFV implements middleboxes as software appliances – also known as *virtual network functions* – that run on commodity servers. In this way, a chain of inexpensive middlebox instances provide the same packet-processing functions at a desired throughput, and we can route traffic through appropriate locations in which these instances are dynamically placed. Such a deployment reduces capital and operational costs and optimizes network operations.

A chain deployment involves *selecting* and instantiating a number of middlebox instances, *placing* these instances, and *routing* traffic through them. An optimal chain deployment *coordinates* the *selection*, *placement*, and *routing* to minimize resources allocated while satisfies the resource capacity and location constraints. Prior middlebox chaining models have several limitations as follows.

Gaps in Selection: Most of the optimization models [6, 11, 171] do not consider the typical middlebox offerings and assume that instances of the same middlebox are identical in their resource consumption and throughput. Service providers offer middlebox instances with different configurations to provide predictable quality of service. For example, HP offers virtual IPSec [73] that provides throughputs of 268, 580, and 926 Mbps assuming respectively 1, 4, and 8 CPU cores.

Similarly, Riverbed offers WAN-optimizers [137] with throughputs of 10 and 50 Mbps given respectively 2 and 4 CPU cores. Note that the correlation between the throughput and resource consumption is not necessarily linear. In practice, predicting the performance of middlebox software instances is not trivial [8, 39, 40].

Gaps in Placement and Routing: To process a traffic flow, some models use a single physical-machine to place the instances of a same middlebox [6, 11, 93, 104, 112] or even all middleboxes of a chain [142]. However, doing so severely limits throughput of a middlebox and a chain to a few instances that can be instantiated in a physical-machine. The throughput of these instances might not be sufficient to process the total traffic routed through them, and this problem is exacerbated by the fact that traffic volume through middleboxes has an increasing trend [75, 170].

Gaps in Coordination: A middlebox instance cannot be selected if there is not sufficient resources to place its instances. Further, it is impractical to place an instance in a given location when adequate bandwidth is not available to route traffic from/to the location. To achieve an optimal deployment of service chains, selection, placement, and routing must be performed in a coordinated manner; otherwise, the deployment results in sub-optimal utilization of network resources and quality of service. Most of existing solutions solve the placement and routing in isolation [47, 56, 130, 175]. There are few solutions [11, 112] that coordinate the placement and routing; however, they treat the selection of middlebox instances separately.

To fill the above gaps, we present distributed service function chaining (DSFC). For each middlebox of a chain, DSFC selects from provided middlebox offerings and determines the appropriate number of instances to be placed. DSFC places these instances in a way that instances of a same middlebox can be installed distributedly in multiple machines. Such a placement decouples a chain's throughput from physical-machines. Further, DSFC, utilizing the global knowledge of the network, routes traffic and distributes the load among the middleboxes instances. DSFC coordinates selection, placement, and routing operations in such a way that network resources are utilized more efficiently.

Specifically, our contributions in this chapter are:

- We model and solve DSFC using mixed integer programming (MIP), and prove its NP-Hardness.
- For larger networks, we propose *Kariz*, a local search heuristic that employs a tuning parameter to balance the speed-accuracy trade-off;

- We perform extensive simulations to evaluate Kariz against the MIP implementation for various chain-lengths and throughput-demands. The results demonstrate that Kariz achieves the competitive acceptance ratio of 76-100% at an extra cost of less than 24%, in comparison to the MIP implementation.

2.1 Challenges

A chain specifies that the traffic originating from a *source*, is processed by an ordered sequence of middleboxes, and finally is delivered to a *target*. To deploy a chain distributedly, several system and optimization challenges have to be addressed.

2.1.1 System Implementation Challenges

Middleboxes often operate on data-packets at a *flow* granularity and maintain *state information* on the flows and sessions they process [151, 166]. State information consists of configuration and statistical data, and differs from one middlebox to another. If a middlebox is replaced with multiple software instances, the functionality should not change, and these instances must act in concert. Further, the traffic processed by a single middlebox, should now be processed by multiple software instances. Thus, *consistent state distribution* and *consistent traffic distribution* among the software instances are essential.

Consistent State Distribution

Deployment of multiple software instances to provide a middlebox requires distribution of the state information. Hence, we need to *model* the state information and *distribute* it among the middlebox instances consistently. The state information can be classified as *internal* or *external*. The internal state is stored and used only by a single instance, while the external state is distributed and shared across multiple instances.

Since the state information is stored in a key-value store [80, 151], data structures like distributed hash-tables and technologies like *remote direct memory access* (RDMA) can fulfill this challenge efficiently. Moreover, it might be required to modify the middleboxes to cope with the defined model. There are abstraction models and system implementations that address this challenge. Rajagopalan et al. [134] introduce a system-level abstraction called Split/Merge that stores the internal state exclusively inside each middlebox instance, while the external state is

distributed and accessible by other instances. As a proof of concept, they implemented FreeFlow as a Split/Merge system, and ported Bro IDS [123] inside it. Further, they analyzed and confirmed the compatibility of two other middleboxes, i.e., application delivery controller and stateful NAT64. In addition, Joseph and Stoica [80] provide a model to describe different middleboxes. As concrete examples, firewall, NAT and layer4 and layer 7 load-balancer are described using the proposed model. Further, Qazi et al. [57] and OpenNF [58] introduce a unified framework to manage state information.

Consistent Traffic Distribution

Replacing a single middlebox with multiple software instances requires *splitting* and *distributing* the traffic load among these instances. Per-flow traffic splitting distributes the traffic in the granularity of flows, and packets of a flow have to be routed along the same path.

Split/Merge [134] utilizes a similar approach. However, this approach does not support accurate load-distribution and is not always applicable. For instance, if the load of a flow is higher than the throughput of an assigned middlebox instance, that instance cannot handle the load and we have to split the traffic into a smaller granularity.

Flowlet switching [2, 148] can be leveraged to split the traffic into a finer granularity. A flowlet is a "burst of packets from the same flow followed by an idle interval" [148]. If the interval between two flowlets is greater than the maximum delay difference between parallel paths, the second flowlet – and consequently following flowlets – can be sent through different paths. Thus, a single flow can be split into multiple paths without packet-reordering.

Furthermore, accurate load balancing is achieved using short flowlet intervals ($[50, 100]ms$) [148]. Specifically, flowlets are abundant in data-center networks since the latency is very low and the traffic is intensively bursty [82]. In addition to these distributed methods, the central schemes leveraging SDN and OpenFlow capabilities [101] can also be used. For instance, *group tables* [51] can be used to split and balance the traffic.

We have shown the feasibility of distributed deployment of middlebox instances to provide a middlebox and distributing traffic among these instances. Next, we state the assumptions that ground our optimization model:

- The state information of a middlebox can be consistently distributed among multiple middlebox instances. This assumption holds for the state information of a single flow.
- The traffic can be consistently distributed into multiple paths among multiple middlebox instances. This assumption holds for a single flow.

Middlebox	Instance type	Throughput	CPU demand	Memory demand
IDS	IDS_1	50 Mbps	1 core	24 GB
	IDS_2	80 Mbps	1 core	32 GB
Firewall	FW_1	100 Mbps	1 core	1.75 GB
	FW_2	200 Mbps	2 core	3.50 GB

Table 2.1: Middleboxes

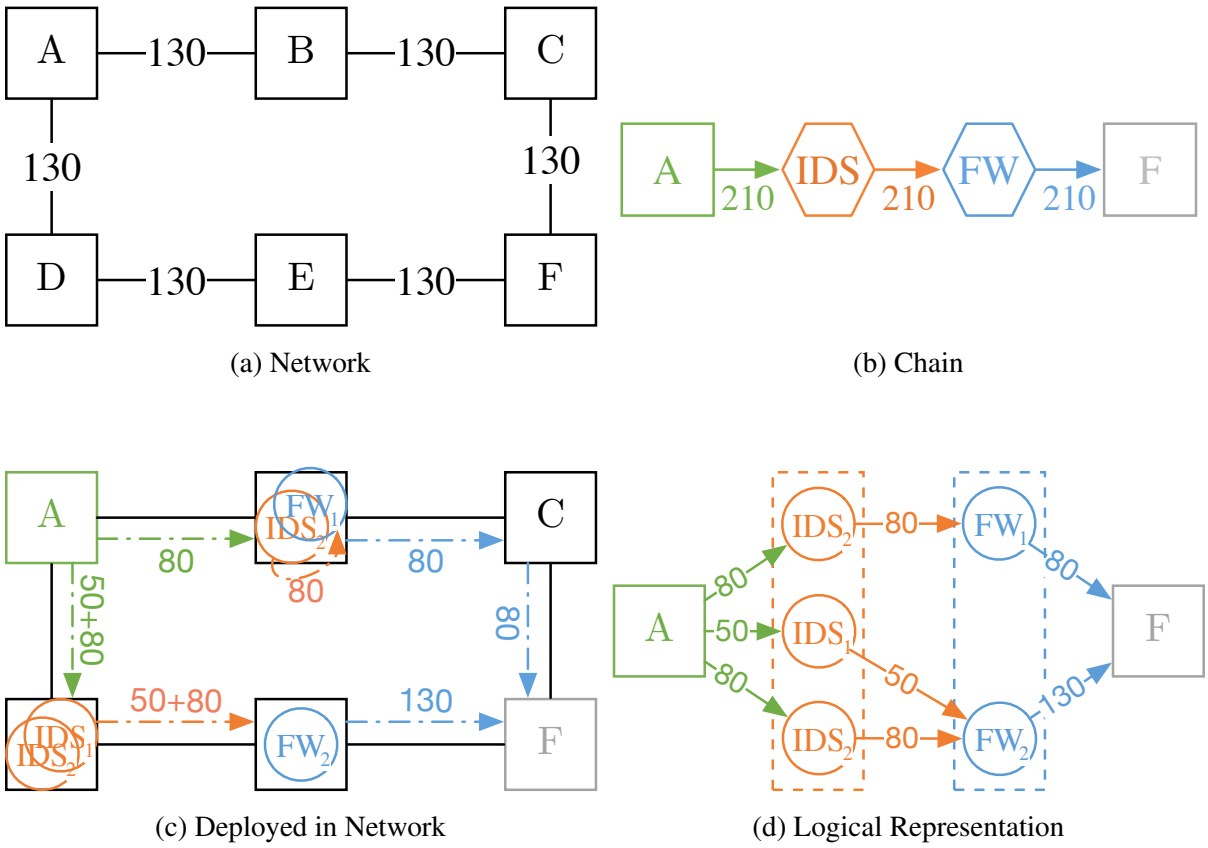
2.1.2 Optimization Challenges

The optimization challenge is in computing an optimal allocation of host and bandwidth resources to a chain. Each middlebox in a chain is replaced with a number of software instances providing the requested throughput. These instances are placed in a set of chosen hosts. In addition, the traffic is split and routed among the instances. Thus, certain decisions have to be made optimally: *number of middlebox instances* (*selection*), *placement* of these instances, and *routing the traffic* through the placed instances. These decisions are inter-dependent and must be made in a coordinated manner.

Figure 2.1 shows a chain deployment. The network of Figure 2.1a includes 6 hosts, each with an 8-core CPU and 64 GB residual memory. For simplicity, switches are not shown, and the presented paths are disjoint in this example. All paths have 130 Mbps available bandwidth.

The chain of Figure 2.1b includes 2 middleboxes with 210 Mbps throughput: an IDS and a firewall (FW). The flow comes from the host A, the source, is processed by IDS and FW, and then sent to host F, the target. As listed in Table 2.1, there are 4 different instances for IDS and FW.

Figure 2.1c depicts the chain deployed in the network, and Figure 2.1d shows the logical representation of this deployment: with 3 instances for IDS ($1 \times IDS_1 + 2 \times IDS_2$) and 2 instances for FW ($1 \times FW_1 + 1 \times FW_2$). The IDS instances are installed in hosts B and D. The flow splits, and 80 Mbps and 130 Mbps are routed from the source to hosts B and D, respectively. FW instances are installed in hosts B and E. In host B, the flow after being processed by IDS_2 is sent to FW_1. IDS_1 and IDS_2 forward the flow to host C in which instance FW_2 is placed. Finally, the flow from the FW instances is sent to the target. Note that it is possible to place the middlebox instances in the source and target if sufficient host resources are available.



(a) Network

(b) Chain

(c) Deployed in Network

(d) Logical Representation

Figure 2.1: Distributed Deployment of a Chain

2.2 Distributed Service Function Chaining

With the assumptions and challenges established, we now introduce the formal definitions and the mathematical model.

2.2.1 Definitions

Physical Resources: $R = \{\text{CPU, memory, storage, } \dots\}$ is a set of available physical resources.

Network: Graph $G = (N, E)$ is the substrate network, where N and E are substrate nodes and links, respectively. $c_{mr} \in \mathbb{R}^+$ is the residual capacity of node m for resource $r \in R$. Set E_m denotes incident links on node m. Moreover, $mn \in E$ is the link between node $m \in N$ and node $n \in N$ and has a residual bandwidth capacity of $c_{mn} \in \mathbb{R}^+$.

Chain: Symbols with over-line are for chain definitions. Forwarding graph $\overline{G} = (\overline{N}, \overline{A})$ denotes a chain. $\overline{N}$ includes middleboxes $\overline{V} \subset \overline{N}$, and two endpoints $\overline{s}$ and $\overline{t}$. Traffic flow coming from $\overline{s} \in \overline{N}$ is processed by middleboxes in the chain, and is forwarded to $\overline{t} \in \overline{N}$. Respectively, $\overline{s}$ and $\overline{t}$ are the *source* and *target* of the traffic. The corresponding substrate nodes for source and target are respectively $s \in N$ and $t \in N$.

Middlebox $\overline{v} = f(\overline{u})$ follows middlebox $\overline{u}$. We define *ring* $\overline{uv} \in \overline{A}$ as 2 consecutive middleboxes $\overline{u}$ and $\overline{v}$, where $\overline{v} = f(\overline{u})$. We assume that $\overline{u}$ generates traffic type $\overline{u}$ and $\overline{v}$ consumes this traffic type. Each ring $\overline{uv}$ has the *throughput demand* $\overline{b}$ denoting integer traffic volume flow generated or consumed by the ring nodes.

Middlebox instances: V denotes a set of middlebox instances. An instance $u \in V$ has throughput $q_u \in \mathbb{R}^+$ showing the maximum traffic that u can process. $d_{ur} \in \mathbb{R}^+$ is the demand of u for resource r. These demands include the overhead of accessing distributed state information. For $\overline{s}$ and $\overline{t}$, we assume that there are instances $u_{\overline{s}}$ and $u_{\overline{t}}$, respectively. These instances have throughput $\overline{b}$ and no demand for any resource. Finally, instances of middlebox type $\overline{u}$ are identified by $V_{\overline{u}}$.

2.2.2 Mathematical Model:

Variable $x^{\overline{u}}_{mn} \in \mathbb{R}$ is the traffic volume of type $\overline{u} \in \overline{N}/\{\overline{t}\}$ on substrate link mn. Target $\overline{t}$ is excluded since it only consumes traffic; thus, no traffic of this type exists in the network. Variable

$y_{mu} \in \mathbb{Z}$ is the number of instances u in substrate node m. Instances of $V_{\overline{u}}$ installed in node m provide throughput of type $\overline{u}$. Variable $z_{m\overline{u}} \in \mathbb{R}$ denotes the allocated throughput of these instances.

A solution for the problem is shown by a tuple of allocation vectors (X, Y, Z), defined as follows. Let vector $X_{\overline{u}} = \{x^{\overline{u}}_{mn} : \forall mn \in E\}$ be allocated bandwidth of links to traffic type $\overline{u}$, and $X = \bigcup_{\overline{u} \in \overline{N}/\{\overline{t}\}} X_{\overline{u}}$. If $Y_{\overline{u}} = \{y_{mu} : \forall m \in N, \forall u \in V_{\overline{u}}\}$ identifies the instances for middlebox type $\overline{u}$, let $Y = \bigcup_{\overline{u} \in \overline{N}} Y_{\overline{u}}$. Finally, $Z_{\overline{u}} = \{z_{m\overline{u}} : \forall m \in N\}$ denotes the allocated throughput of type $\overline{u}$ in every node, and $Z = \bigcup_{\overline{u} \in \overline{N}} Z_{\overline{u}}$.

Node Capacity Constraint: Equation 2.1 ensures that instances are placed with respect to the substrate nodes capacities.

$$\forall m \in N : \forall r \in R : \sum_{u \in V} y_{mu} d_{ur} \leq c_{mr} \tag{2.1}$$

Location Constraint: Equalities in Equation 2.2 ensure that an instance of $u_{\overline{s}}$ and an instance of $u_{\overline{t}}$ are placed only in $s \in N$ and $t \in N$, respectively.

$$\begin{aligned} y_{su_{\overline{s}}} = 1, \quad \sum_{m \in N/\{s\}} y_{mu_{\overline{s}}} = 0 \\ y_{tu_{\overline{t}}} = 1, \quad \sum_{m \in N/\{t\}} y_{mu_{\overline{t}}} = 0 \end{aligned} \tag{2.2}$$

Substrate Link Capacity Constraint: Equation 2.3 makes sure that the capacities of substrate links are not violated.

$$\forall mn \in E, m < n : \sum_{\overline{u} \in \overline{N}} (x^{\overline{u}}_{mn} + x^{\overline{u}}_{nm}) \leq c_{mn} \tag{2.3}$$

Throughput Constraint: Equation 2.4 ensures that the aggregate throughput capacity of instances of type $\overline{u}$ placed in substrate node m is more than allocated throughput $z_{m\overline{u}}$.

$$\forall m \in N : \forall \overline{u} \in \overline{N} : \sum_{u \in V_{\overline{u}}} y_{mu} q_u \geq z_{m\overline{u}} \tag{2.4}$$

Throughput Demand Constraint: Equation 2.5 guarantees that for each middlebox $\overline{u}$, throughput $\overline{b}$ is allocated by instances $V_{\overline{u}}$.

$$\forall \overline{u} \in \overline{N} : \sum_{m \in N} z_{m\overline{u}} = \overline{b} \tag{2.5}$$

Flow Conservation Constraint: Equation 2.6 is a modified version of the flow-conservation constraint [161]. Let us say that in node $m \in N$, instances of middlebox types $\overline{u}$ and $\overline{v} = f(\overline{u})$ are installed. Therefore, instances of $V_{\overline{v}}$ locally process a volume of traffic type $\overline{u}$ generated by instances of $V_{\overline{u}}$. This volume is $z_{m\overline{v}}$. Unprocessed traffic volume should exit the node m. This constraint ensures this phenomenon.

$$\begin{gathered} \forall m \in N : \forall \overline{u} \in \overline{N}/\{\overline{t}\} : \overline{v} = f(\overline{u}) : \\ \sum_{mn \in E_m} \left(x^{\overline{u}}_{mn} - x^{\overline{u}}_{nm}\right) = \left(z_{m\overline{u}} - z_{m\overline{v}}\right) \end{gathered} \tag{2.6}$$

Bandwidth Allocation Cost: Equation 2.7 is the bandwidth allocation cost. Coefficient $\beta \in \mathbb{R}^+$ identifies the relative importance of bandwidth resources. The communication overhead to access the distributed state information is negligible vs. the actual service traffic volume.

$$B(X) = \beta \sum_{\overline{u} \in \overline{N}/\{\overline{t}\}} \sum_{mn \in E} x^{\overline{u}}_{mn} \tag{2.7}$$

Host Resource Allocation Cost: Equation 2.8 is the cost of allocating host resources to place middlebox instances. Coefficient $\alpha_r \in \mathbb{R}^+$ is the relative importance of resource $r \in R$.

$$H(Y) = \sum_{u \in V} \sum_{r \in R} \alpha_r d_{ur} y_{mu} \tag{2.8}$$

Objective Function: Equation 2.9 minimizes the aggregate cost of allocating host and bandwidth resources.

$$\min \Big(B(X) + H(Y) \Big) \tag{2.9}$$

2.3 NP Hardness of Distributed Service Function Chaining

In this section, we prove that the Distributed Service Function Chaining (DSFC) problem is NP hard. We use a *reduction* from the NP hard *minimum dominating set* problem. Our reduction technique is of independent interest and can be potentially applied to analyze the complexity of other problems related to service function chaining.

A dominating set of a graph is a set of nodes so that each node is either a member or adjacent to at least a member of this set. The goal is to find a dominating set with minimum size.

Given a graph $\boldsymbol{G}$ with $\boldsymbol{n}$ nodes as an instance of the dominating set problem, we create an instance of DSFC and prove that there is a dominating set of size $\boldsymbol{k}$ if and only if there is a solution of cost $3\boldsymbol{n} + \boldsymbol{k}$ in the DSFC instance.

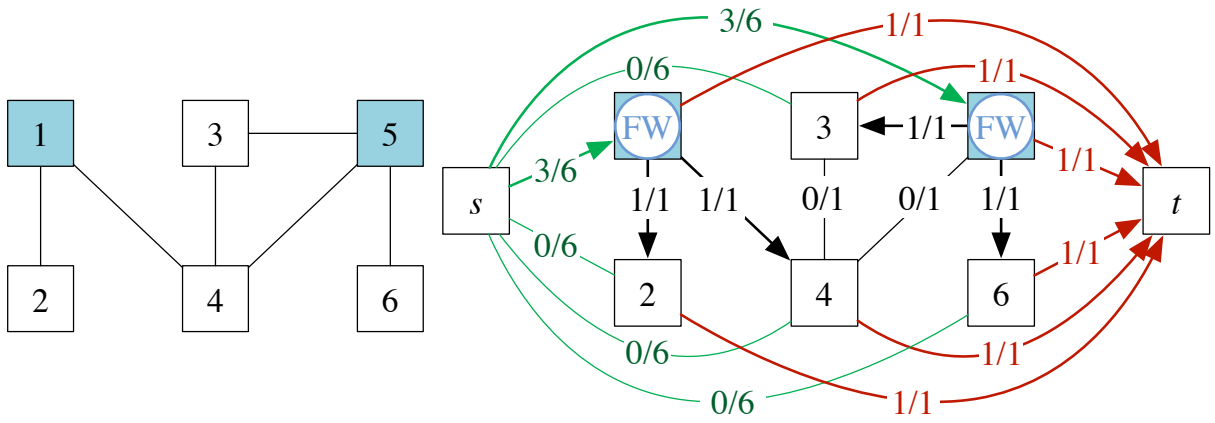
Recall from Section 2.2.1, an instance of DSFC is defined with resources, a network, a service chain, and middleboxes. In our reduced instance, CPU is the only host resource. We define the network to be the same as $\boldsymbol{G}$ with two extra nodes s and t. These two nodes are connected to all nodes of $\boldsymbol{G}$. Nodes s and t have 1 CPU core, and others have 2 CPU cores. Incident links on s have the capacity $\boldsymbol{n}$, and the capacity of other links is 1. The chain is defined as $\overline{s} \rightarrow \text{FW} \rightarrow \overline{t}$. Endpoints $\overline{s}$ and $\overline{t}$ respectively correspond to source s and target t in the network, and FW is a firewall SF. Moreover, there is a single firewall instance demanding 2 CPU cores and providing throughput capacity $\boldsymbol{n}$. On this instance of DSFC, the goal is to install firewall instances to process a *flow* of size $\boldsymbol{n}$ from s to t. Note that these middleboxes require 2 CPU cores and cannot be placed in s or t. Here, by 'flow', we mean the traffic that is to be sent from s to t. We define α_r ($r = \text{CPU}$) and β to be 1, hence the total cost of DSFC is the total number of allocated CPU cores (resource cost) and bandwidth. Figure 2.2 depicts this reduction. Clearly, the DSFC instance can be constructed in polynomial time from the dominating set instance.

To prove the hardness, we show that there is a dominating set of size $\boldsymbol{k}$ if and only if there is a solution for the DSFC instance with cost $3\boldsymbol{n} + \boldsymbol{k}$. We start with the easy direction:

Lemma 1. *If there is a solution of size $\boldsymbol{k}$ for the dominating set problem, then there is a solution of cost $3\boldsymbol{n} + \boldsymbol{k}$ for the DSFC problem.*

Proof. Let $\boldsymbol{v}_1, \ldots, \boldsymbol{v_k}$ denote the nodes in the dominating set, and $\boldsymbol{a_i}$ ($1 \leq \boldsymbol{i} \leq \boldsymbol{k}$) denote the number of nodes $\boldsymbol{v_i}$ dominates. If a node is dominated by more than one node, we count it only once (arbitrary assign it to one node in the dominating set). Note that we have $\sum_{\boldsymbol{i}} \boldsymbol{a_i} = \boldsymbol{n} - \boldsymbol{k}$.

In the DSFC instance, we send a flow of size $\boldsymbol{a_i} + 1$ from s to any $\boldsymbol{v_i}$. Doing so results in bandwidth cost of $\boldsymbol{n}$. We also send a flow of size 1 from $\boldsymbol{v_i}$ to any of the nodes that it dominates; this requires a bandwidth of $\boldsymbol{a_i}$ for $\boldsymbol{v_i}$ and in total bandwidth of $\boldsymbol{n} - \boldsymbol{k}$ for all dominating nodes.



(a) Dominating Set (b) DSFC

Figure 2.2: To send a flow of size $\boldsymbol{n} = 6$ from s to t, a flow of size 3 is sent to nodes 1 and 5. These nodes then send a flow of size 1 to any node that they dominate. Firewalls are placed at dominating nodes 1 and 5.

Finally, we send a flow of size 1 from all nodes (except s) to t. This results in bandwidth cost of $\boldsymbol{n}$ (see Figure 2.2b). We install a FW for the service chain in each node in the dominating set, resulting in resource cost of $2\boldsymbol{k}$. In total, the cost is $\boldsymbol{n} + (\boldsymbol{n} - \boldsymbol{k}) + \boldsymbol{n} + 2\boldsymbol{k} = 3\boldsymbol{n} + \boldsymbol{k}$. □

To prove the other side of the reduction, we start with Lemma 2.

Lemma 2. *Given a solution of the DSFC problem with cost* $\mathbf{c}$*, one can achieve a solution of cost no more than* $\mathbf{c}$ *in polynomial time, for which the following properties for each node other than* s, t *hold: (1) the total inflow received through nodes other than* s *is at most 1; (2) the inflow is through* s *or a node that receives inflow through* s*; moreover, (3) FWs are placed at nodes receiving some flow directly from* s*; (4) each node receives either all or none of its inflow from* s*.*

Proof. We modify the solution to satisfy properties (1)-(4) in the same order without affecting previously satisfied properties. In this process, the cost of the solution is never increased. To satisfy (1), assume there is node $\boldsymbol{u}$ with inflow $\boldsymbol{x} > 1$ from node(s) other than s. This assumption implies a bandwidth cost of at least $2\boldsymbol{x}$ for the flow passing at least another node between s and $\boldsymbol{u}$. In the new solution, we remove this flow and send a flow of size $\boldsymbol{x}$ from s to $\boldsymbol{u}$ and place a FW at $\boldsymbol{u}$. Doing so gives a bandwidth cost of $\boldsymbol{x}$ and resource cost of 2. The increase in cost is no more than $\boldsymbol{x} + 2 - 2\boldsymbol{x} \leq 0$.

Property (2) follows directly from (1). To satisfy (3), note that by (1), the flow between nodes excluding s and t form a forest. Placing FWs only in the roots in the forest does not increase the

cost. For (4), assume a node receives an inflow of $\boldsymbol{x}$ through s and an inflow of 1 through another node. By (3), there is a FW at $\boldsymbol{u}$. In the new solution, we send a flow of $\boldsymbol{x}+1$ from s to $\boldsymbol{u}$ and remove the flow from the other node. □

We use Lemma 2 to prove the other side of the reduction:

Lemma 3. *If there is a solution of cost $3\boldsymbol{n}+\boldsymbol{k}$ for the DSFC problem, then there is a solution of size $\boldsymbol{k}$ for the dominating set problem.*

Proof. First, we apply Lemma 2 to achieve a solution with the desired properties. We refer to the nodes that receive flow through s as *critical nodes*. By property (4), a node receive all or none of its inflow from s. By (3), there is a FW located in critical nodes. By (1), each non-critical node has inflow of 1 and by (2), such node receives this inflow through a critical node. In other words, the graph formed by flows (excluding t), is a tree of diameter 2 rooted at s.

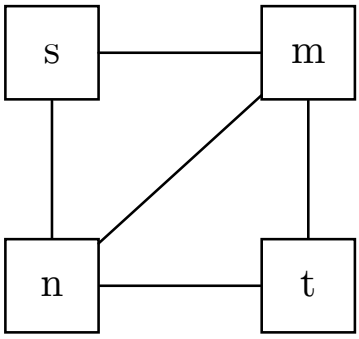
Let $\boldsymbol{m}$ denote the number of critical nodes; the resource cost for FWs would be $2\times\boldsymbol{m}$. The bandwidth cost is $3\boldsymbol{n}-\boldsymbol{m}$: a cost of $\boldsymbol{n}$ for the outflow of s, another cost of $\boldsymbol{n}$ for the inflow of t, and an extra bandwidth cost of $\boldsymbol{n}-\boldsymbol{m}$ for the flow from critical nodes to non-critical ones (recall that the inflow of each non-critical node is 1). In conclusion, the total cost of the solution is $3\boldsymbol{n}+\boldsymbol{m}$, i.e., we have $3\boldsymbol{n}+\boldsymbol{m}=3\boldsymbol{n}+\boldsymbol{k}$. In other words, the number of critical nodes is $\boldsymbol{k}$. On the other hand, by (1) and (2) each non-critical node has an inflow of exactly 1 through a critical node. Hence, critical nodes form a dominating set of size $\boldsymbol{k}$. □

From Lemmas 1 and 3, Theorem 1 is direct.

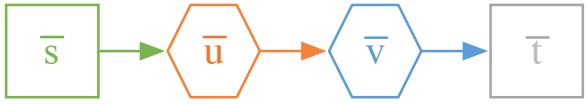
Theorem 1. *Finding the solution with minimum cost for DSFC is NP hard.*

2.4 Kariz: Heuristic Solution

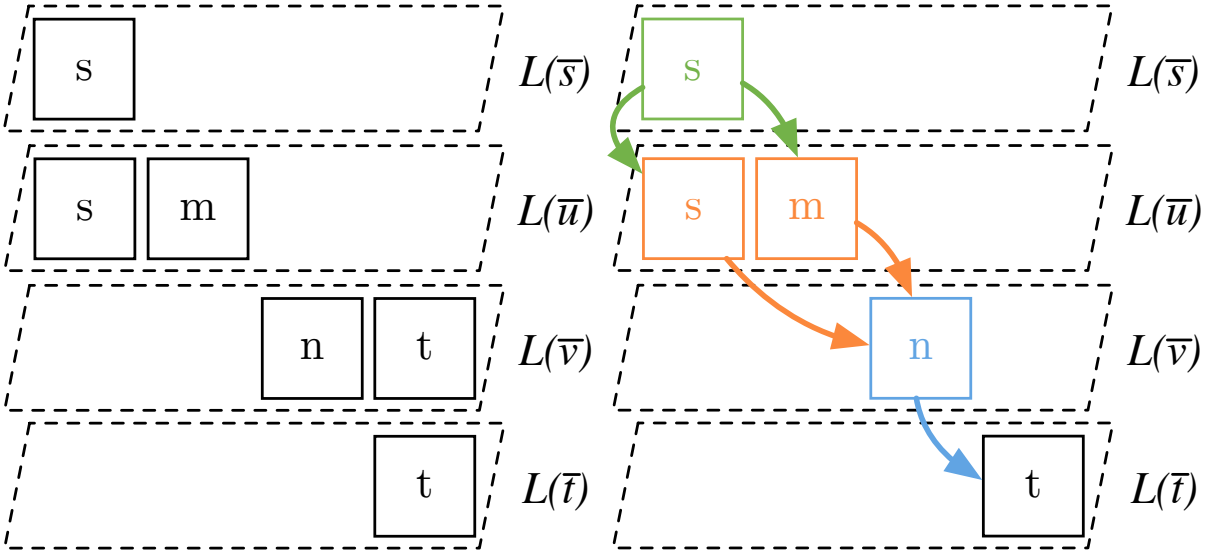
Before explaining our solution, we construct a visualization tool to simplify our description. Let us assume that each $\overline{u}\in\overline{N}$ is deployed in a *layer*. Each layer contains a set of nodes in which instances of a corresponding middlebox type can be installed. In other words, in the layer corresponding to $\overline{u}$, we initially place a subset of nodes in which at least a middlebox instance $v\in V_{\overline{u}}$ can be instantiated. $L(\overline{u})$ denotes this layer. Figure 2.3c depicts the layers for the chain of Figure 2.3b. As shown in Figure 2.3c, s and t are the only nodes present in layers $L(\overline{s})$ and $L(\overline{t})$, respectively. Further, nodes $\{s,m\}$ and $\{n,t\}$ are respectively included in layers $L(\overline{u})$ and $L(\overline{v})$ because these nodes have sufficient resources to host instances of these middleboxes. Figure 2.3d presents a sample solution for the chain of Figure 2.3b.



(a) Network



(b) Chain



(c) Layers

(d) Sample Solution

Figure 2.3: Layers

Inspired by [69, 120], we develop a local search heuristic, *Kariz*, which routes traffic layer by layer. We introduce the process first, and then provide a detailed overview.

Kariz is shown in Algorithm 1 and works as follows. We first initialize layers as described above and set solution as empty (line 1). Starting from layer $L(\overline{s})$ (line 2), iteratively route $\overline{b}$ volume of traffic from layer $S = L(\overline{u})$, the *source-layer*, to the next layer $T = L(\overline{v})$, the *sink-layer* (lines 3-11). After finding the optimal route between two layers (line 5), compute the number of instances of $V_{\overline{v}}$ by considering the allocated throughput (line 6). Add the solution of the sink-layer to the earlier solution (line 7). Improve the current solution (line 8), and update layers (line 9). Now, traffic has reached the sink-layer; consider this layer as new source-layer (line 10). Repeat this procedure if traffic has not reached the last layer yet, and there are nodes in the new source-layer (line 11).

Algorithm 1 Kariz Algorithm

1: $init\text{-}layers()$; $(X, Y, Z) \leftarrow (\varnothing, \varnothing, \varnothing)$;
2: $\overline{u} \leftarrow \overline{s}$; $z_{s\overline{s}} \leftarrow \overline{b}$; $z_{t\overline{t}} \leftarrow \overline{b}$; $S \leftarrow L(\overline{s})$;
3: **do**
4: $\quad \overline{v} \leftarrow f(\overline{u})$; $T \leftarrow L(\overline{v})$;
5: $\quad X_{\overline{v}}, Z_{\overline{v}} \leftarrow route(S, T, \overline{b})$;
6: $\quad Y_{\overline{v}} \leftarrow instances(Z_{\overline{v}})$;
7: $\quad (X, Y, Z) \leftarrow (X \cup X_{\overline{v}}, Y \cup Y_{\overline{v}}, Z \cup Z_{\overline{v}})$;
8: $\quad improve(X, Y, Z)$;
9: $\quad update\text{-}layers()$;
10: $\quad \overline{u} \leftarrow \overline{v}$; $S \leftarrow L(\overline{v})$;
11: **while** $\left(\overline{u} \neq \overline{t} \text{ and } S \neq \varnothing\right)$;

Still to clarify are the traffic routing between two layers and the number of instances in the sink-layer, how the solution is improved, and how the layers are updated.

2.4.1 Route and Middlebox Instances

Procedure $route(.)$ in Algorithm 1 computes the route between two layers by solving the multi-source multi-sink minimum cost flow problem (MCFP) [63]. MCFP is the problem of routing a volume (say $\overline{b}$) of a *commodity* (in our case traffic of type $\overline{u}$) from multiple sources (say a source-layer) to multiple sinks (in our case a sink-layer). Any multi-source multi-sink MCFP can be modeled as a single-source single-sink MCFP that is solvable in polynomial time [63]. For

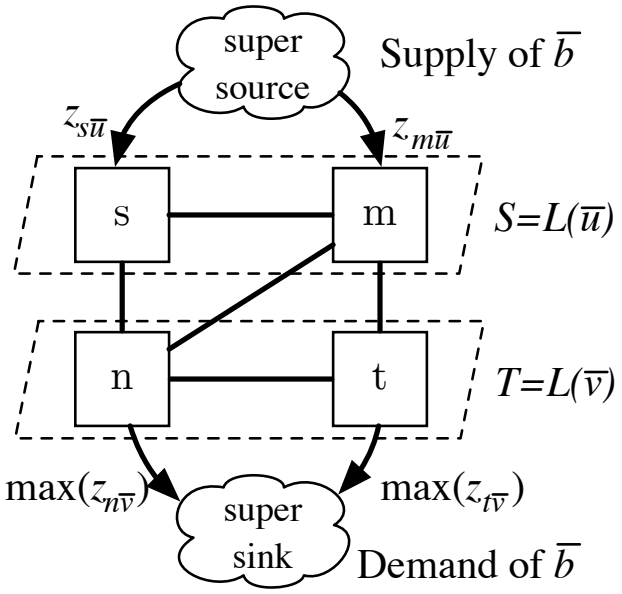


Figure 2.4: Routing as Single-Source Single-Sink MCFP

our problem, this is achieved by representing the source- and sink-layers with imaginary nodes *super-source* and *super-sink*, respectively.

Figure 2.4 depicts this model for layers S and T in Figure 2.3. The procedure is as follows. Add a super-source and connect it to every node $m \in S$ in the source-layer with a directed-link whose capacity is $z_{m\overline{u}}$. For the sink-layer, add a super-sink node and connect every node $n \in T$ using a directed-link. The capacity of the directed-link connecting node n to the super-sink is the maximum throughput $\max(z_{n\overline{v}})$ of the instances that can be installed in node n. There is no cost to sending the traffic via these links. As the result, the minimum cost route of traffic from super-source to super-sink gives the optimal routing between the two layers. If p denotes the super-sink, the throughput allocation in each $n \in L(\overline{v})$ is $z_{n\overline{v}} = x^{\overline{u}}_{np}$.

Finding the capacity of directed-links from the sink-layer to the super-sink is similar to the problem of $instances(.)$. The former is finding the maximum throughput $\max(z_{n\overline{v}})$ out of instances that can be installed in node n. The latter is finding the minimum allocation of resources to instances providing throughput of at least $z_{n\overline{v}}$ in each node $n \in L(\overline{v})$.

In fact, these problems are *dual* and can be modeled as a *multidimensional knapsack problem* [26]. Think of the node as an $|R|$-*dimensional knapsack*, each *dimension* corresponding to a resource $r \in R$. The *items* to be packed are instances with the *profits* of their throughputs and *weights* of their host resource demands.

Though this problem is known to be NP-Hard [26], since the resources of a single machine, especially the number of CPU cores, are limited, the problem size is small. Thus, we can solve it efficiently. Instead, as CPU cores are the most pricey and restricted resources, a feasible solution optimizing the number of allocated cores is a good optimum.

2.4.2 Solution Improvement Rounds

Routing between two layers focuses on the cost of traffic routing and does not consider the cost of host resource allocation. Doing so may lead to high host resource cost. Hence, we need to improve the solution.

Procedure $improve(.)$, as shown in Algorithm 2, facilitates this: repeatedly search for some *actions* to improve the solution (lines 2-8). If no such action is found, report the current solution (line 4-6). Otherwise, perform the action with the greatest drop in cost, the best *admissible* action (line 7), and continue with the adjusted solution. We define actions and *admissibility* in Section 2.4.2 and Section 2.4.2, respectively.

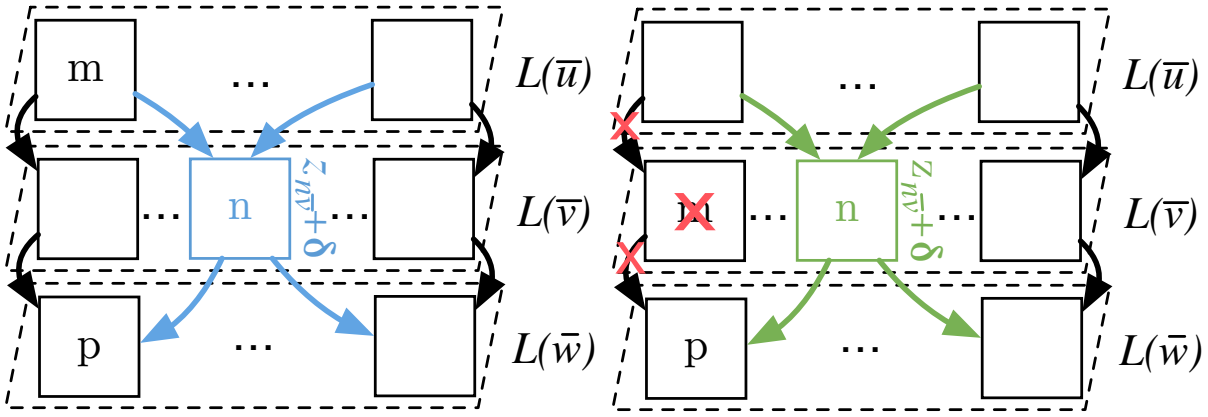
Algorithm 2 Procedure $improve(.)$

```
1: procedure improve(X, Y, Z)
2:     loop
3:         a ← best-action(X, Y, Z);
4:         if not admissible(a) then
5:             return (X, Y, Z);
6:         end if
7:         perform-action(X, Y, Z, a);
8:     end loop
9: end procedure
```

Actions

An action is a *local transformation* intended to reduce the cost of solution. Let (X', Y', Z') be the modified solution after performing an action on a current solution (X, Y, Z). The cost difference before and after performing an action is regarded as the *action cost*, as defined in Equation 2.10. The best action has the lowest cost.

$$\big(B(X') + H(Y')\big) - \big(B(X) + H(Y)\big) \tag{2.10}$$



(a) $add(n, L(\overline{v}), \delta)$ (b) $open(n, \{m\}, L(\overline{v}), \delta)$

Figure 2.5: Actions

We define the below actions variants of actions used by [120]:

- $add(n, L(\overline{v}), \delta)$: Include node $n \in N$ in $L(\overline{v})$ and allocate more $\delta > 0$ units of throughput in this node ($z_{n\overline{v}} \leftarrow z_{n\overline{v}} + \delta$). Then, find the minimum cost routing from layer $L(\overline{v})$ to the next and previous layers in the current solution, given allocated throughputs of $L(\overline{v})/\{n\}$. The next and previous layers are $L(\overline{w})$ and $L(\overline{u})$ if $\overline{w} = f(\overline{v})$ and $\overline{v} = f(\overline{u})$, respectively. Finally, tune the allocated throughput of nodes $L(\overline{v})$. This action is shown in Figure 2.5a.

- $open(n, M, L(\overline{v}), \delta)$: Add node $n \in N$ into layer $L(\overline{v})$, remove nodes $M \subseteq L(\overline{v})$, and allocate more $\delta > 0$ units of throughput in node n ($z_{n\overline{v}} \leftarrow z_{n\overline{v}} + \delta$). Finally, reroute the traffic either received or originated in layer $L(\overline{v})$. This action replaces a set of fragmented middlebox instances installed in different nodes M with instances collocated in one node n. This action makes sense only if $\delta \geq \sum_{m \in M}(z_{m\overline{v}})$. Figure 2.5b depicts an example of this action.

Traffic routing in the above actions is a bit different from routing in $route(.)$. The difference lies in routing two different traffic types. Considering each traffic type as a commodity, still this problem can be modelled it as a multi-commodity MCFP (real flows) that is solvable in polynomial time [45].

We also need to examine actions and select the best in polynomial time and ensure that the number of performed actions is not exponential. Particularly, we need to select the best action with sufficient improvement efficiently. These criteria, *efficient action selection* and *sufficient improvement*, are essential to assure that the algorithm terminates in polynomial time.

Efficient Action Selection

The number of $add(.)$ actions is less than $|N| \times |\overline{V}| \times \bar{b}$ under the assumption of integrality of $\bar{b}$. Thus, it is possible to check all actions and select the best in polynomial time. We can even do better and select the value of δ considering the throughputs of middlebox instances $V_{\overline{v}}$. However, the number of possible $open(., M, L(\overline{v}), .)$ actions is equal to the number of subsets $M \subseteq L(\overline{v})$ which is exponential ($2^{|L(\overline{v})|}$). Thus, we need an efficient procedure to select a good $open(.)$ action.

For a fixed layer $L(\overline{v})$, fixed node $n \in N$ and fixed δ, we find this subset in a greedy procedure working as follows. Starting from empty set M, iteratively remove a node m from $L(\overline{v})$ and add it to M. Removing this node has the minimum cost vs. other nodes $L(\overline{v})/m$. Continue this procedure while such a node $m \in L(\overline{v})$ exists, the removal of m decreases the cost, and m's throughput is less than $\delta - \sum_{p \in M} z_{p\overline{v}}$. This procedure repeatedly removes a node $m \in L(\overline{v})$ whose removal results in the greatest decrease in both bandwidth and host resource allocation costs.

Sufficient Improvement

If we allow performing actions that yield minor improvements, the number of actions can be large. Thus, only actions with sufficient cost improvement are allowed. An action yielding sufficient improvement is said *admissible*. More precisely, we define an action as admissible if it improves the solution no less than $\frac{\epsilon}{5|N|}\big(B(X) + H(Y)\big)$ for some tuning parameter $\epsilon > 0$ [106]. Using ϵ, we can control the trade-off between the accuracy and speed of our solution.

Let (X^*, Y^*, Z^*) be the optimal solution. The number of actions performed will be at most $\frac{5|N|}{\epsilon} \ln \frac{B(X)+H(Y)}{B(X^*)+H(Y^*)}$ because the optimal solution is the lower bound for our solution. Since $\ln \big(B(X) + H(Y)\big)$ is polynomial in the size of the network and chain, the number of actions performed is also polynomial.

2.4.3 Update Layers

As the last piece of the puzzle, procedure $update\text{-}layers()$ updates nodes in layers as shown in Algorithm 3. From a layer $L(\overline{u})$ that traffic has already reached, every node $m \in L(\overline{u})$ is eliminated if this node does not allocate throughput of type $\overline{u}$ (lines 3-4). From other layers, nodes whose resources are allocated and hereafter cannot host corresponding instances are excluded (lines 5-7). Layers $L(\overline{s})$ and $L(\overline{t})$ are not updated.

Algorithm 3 Procedure $update\text{-}layers()$

procedure $update\text{-}layers()$
  for $\overline{u} \in \overline{V}$ **do**
    if traffic has reached $L(\overline{u})$ **then**
      $L(\overline{u}) \leftarrow \{m | m \in L(\overline{u}), z_{m\overline{u}} > 0\}$
    else
      $L(\overline{u}) \leftarrow \{m | m \in L(\overline{u}), \exists v \in V_{\overline{u}}, \forall r : c_{mr} \geq d_{rv}\}$
    end if
  end for
end procedure

Through §2.4.1 to 2.4.2, we show that the running times of all $route(.)$, $instances(.)$, $improve(.)$, and $update\text{-}layers(.)$ are polynomial in the size of the network and chain. Hence, Kariz terminates in a polynomial time. Next, we analyze the time complexity of our algorithm.

2.4.4 Time Complexity Analysis

Kariz routes traffic and installs instances layer by layer. For each layer, Kariz (i) finds a feasible initial solution, (ii) improves this solution, and (iii) updates layers accordingly.

One can verify that the second step dominates the time complexity for the computation performed in each layer. In this step, Kariz performs repeatedly either best $add(.)$ or best $open(.)$. The number of performed actions depends on the quality of the initial solution and is at most $\frac{5|N|}{\epsilon} \ln \frac{B(X)+H(Y)}{B(X^*)+H(Y^*)}$. In the worst case, the initial solution is $O(|N|)$ worse than the optimal solution (placing an 'almost' idle instance in each substrate node). Thus, the number of performed actions is in $O(|N| \log |N|)$.

Finding the best $add(.)$ action is examining at most $\overline{b}|N||\overline{V}|$ actions each of which entails MCFP problem. Let $\Psi(G)$ be time complexity of solving an instance of MCFP problem [14]. The complexity of finding the best $add(.)$ action is $\overline{b}|N||\overline{V}|\Psi(G)$. Each $open(.)$ action also involves solving MCFP problem.

For the best $open(., M, L(\overline{v}), .)$, Kariz finds a subset M in $L(\overline{u})$ that at worst is in $O(|N|^2)$. Thus the complexity of finding the best $open(.)$ action is in $O(|N|^2)\Psi(G)$. In total, the time complexity of running the second step for each layer is $O(|N|^2 \log(|N|))\Psi(G)$.

Middlebox	Instance types	Throughput	CPU demand
firewall [13]	Level 1	100 Mbps	1 core
	Level 5	200 Mbps	2 core
	Level 10	400 Mbps	4 core
IDS	Bro [18]	80 Mbps	1 core
IPSec [73]	VSR1001	268 Mbps	1 core
	VSR1004	580 Mbps	4 core
WAN-opt. [137]	CCX770M	10 Mbps	2 core
	CCX1555M	50 Mbps	4 core

Table 2.2: Off-the-shelf middleboxes

2.5 Evaluation

We start with describing our experimental setup and methodology in Section 2.5.1. We evaluate the performance of Kariz in terms of its acceptance ratio in Section 2.5.2. Then in Section 2.5.3, we evaluate the resource utilization of Kariz. Finally, we measure the operational costs of Kariz in Section 2.5.4.

2.5.1 Experimental Setup and Methodology

Simulated Network: The 6-ary Fat-tree, a common data-center topology, is used as the simulated network, and contains 99 nodes (54 hosts and 45 switches) and 162 links providing full bi-sectional bandwidth. Hosts are equipped with a 20-core CPU and 2 Gbps network-adapter. The link capacities are 2 Gbps. This network is the largest network that we could run the implementation of DSFC model, as explained in Section 2.5.1, in a manageable time. The relative importance of allocating 1 Mbps of bandwidth over one link vs. one core CPU is 1% (i.e., $\frac{\text{number of CPU cores of a host}}{\text{bandwidth capacity of a host}}$).

Middlebox instances: We select the firewall, IDS, IPsec and WAN-opt. as middleboxes. Table 2.2 reveals the middlebox instances used in the simulation. Since the CPU is the most restricted host resource and dominates the cost, we ignore memory and storage requirements.

Chains: Sources and targets are uniformly distributed in the network. Poisson distribution with the average of 1-chain per 100-seconds simulates the arrival rate. Chain lifetimes follow

exponential distribution with an average of 3 hours.

Parameters: We asses Kariz in respect to *throughput-demand* and *length* of chains. In each experiment, the throughput-demand is fixed to one of $\{200, 250, 300, \ldots, 500\}$ Mbps, and one of the following chains is selected. Note that Len-i contains all middleboxes of Len-i-1.

- Len-1: {firewall},
- Len-2: {firewall $\rightarrow$ IDS},
- Len-3: {firewall $\rightarrow$ IDS $\rightarrow$ IPSec}, and
- Len-4: {firewall $\rightarrow$ IDS $\rightarrow$ IPSec $\rightarrow$ WAN-opt.}

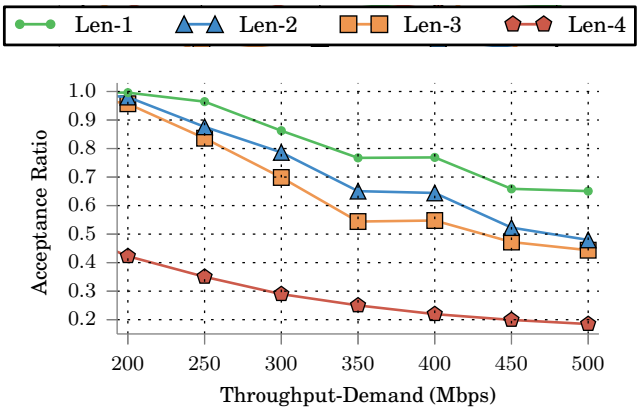
Evaluation Method: We compare Kariz with the model in Section 2.2.2 reffered as *MIP*. We implemented MIP using CPLEX. Note that MIP optimally deploys a single chain. Moreover, the tuning parameter of Kariz is set to $\epsilon = 32$. Thus, an action is performed if it improves the current solution by $\sim 6\%$. With fixed parameters, we repeat each experiment 10 times for every 1000 chains generated, and report the average.

2.5.2 Acceptance Ratio

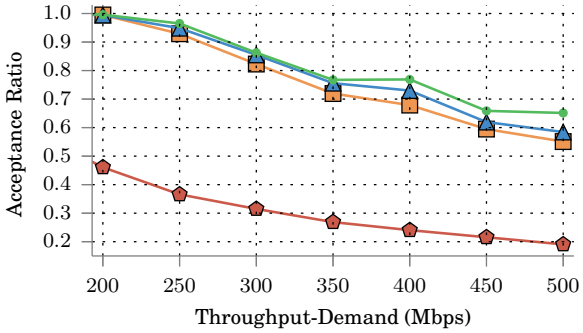
Figure 2.6a and Figure 2.6b depict the acceptance ratios of Kariz and MIP, respectively. The values are the average acceptance ratios from 10 experiments. As expected, the longer chains with higher throughput-demand have less chance to be accepted. The low acceptance ratio for Len-4 is due to the resource hunger of these chains, especially for WAN-opt. instancess.

The range of numbers of chains accepted by Kariz vs. MIP in Figure 2.6c are: 100-100% for Len-1, 82-99% for Len-2, 76-96% for Len-3, and 89-97% for Len-4. Considering the chain length and throughput-demand impacts in Figure 2.6c, Kariz performs closely to MIP.

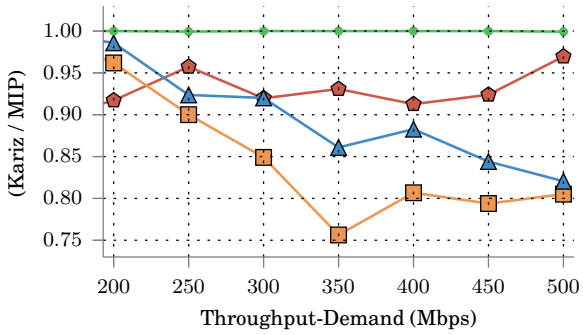
It might be expected that increasing the length of chain and throughput-demand should cause Kariz to have a lower acceptance ratio than MIP. However, Kariz has better results for Len-4 than Len-3 and Len-2, especially for 500 Mbps throughput-demand. Recall from Section 2.4.2 that Kariz attempts to improve the solution after deployment of every middlebox of a chain. Since, Len-4 includes all middleboxes of Len-3 and Len-2 chains (see Section 2.5.1), the expense of more improvement rounds increases the chance of adjusting the earlier solution. All in all, Kariz has a competitive acceptance ratio, within 76-100% of MIP.



(a) Kariz Acceptance Ratio



(b) MIP Acceptance Ratio



(c) Acceptance Ratio Comparison

Figure 2.6: Acceptance Ratio

2.5.3 Resource Utilization

Figure 2.7 compares the resource utilization of Kariz with MIP's. Bandwidth/CPU utilization for Kariz and MIP are the ratio of allocated bandwidth/CPU resources over aggregated bandwidth/CPU capacities in the network. For Middlebox resources, the reports are the average of per-middlebox throughput utilization.

The bandwidth utilization ratios as depicted in Figure 2.7a are 100-101% for Len-1, 86-102% for Len-2, 83-104% for Len-3, and 125-134% for Len-4. Figure 2.7a and Figure 2.6c show that Kariz efficiently utilizes the bandwidth resources for Len-1, Len-2, and Len-3 for various throughput-demands. Regarding Len-4, Kariz's efficiency in utilizing bandwidth resources decreases.

The CPU utilization ratios are in the range of 100-100% for Len-1, 82-98% for Len-2, 76-96% for Len-3, and 98-101% for Len-4, as observed in Figure 2.7b. According to Figure 2.7b and Figure 2.6c, Kariz utilizes CPUs efficiently, close to MIP's.

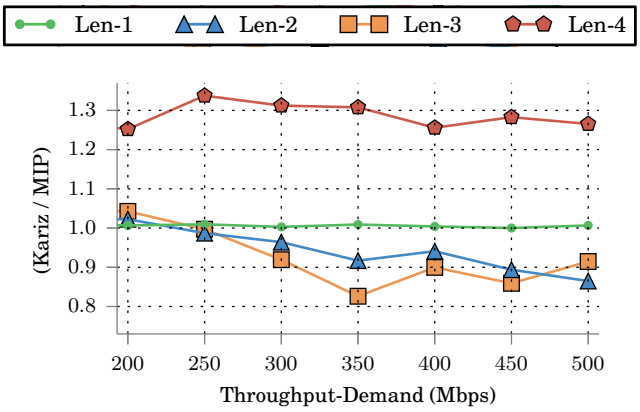
Finally, the instance utilization ratios vs. MIP are shown in Figure 2.7c. The following ranges are reported: 100-100% for Len-1, 99-100% for Len-2, 98-106% for Len-3, and 102-107%. Kariz utilizes middlebox instances with an efficiency close to that of MIP for different lengths and throughput demands.

2.5.4 Operational Costs

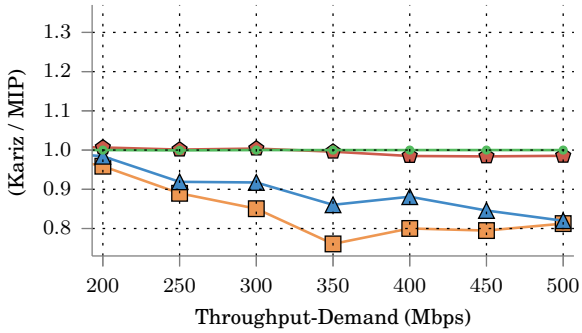
Figure 2.8 shows Kariz's costs vs MIP's. We collect Kariz's and MIP's average of per-chain costs. The reported values are the ratio of Kariz's and MIP's costs.

As shown in Figure 2.8a on average, Kariz allocates bandwidth resource vs. MIP in the range 100-101% for Len-1, 104-108% for Len-2, 108-113% for Len-3, and 132-141% for Len-4. Regarding CPU as presented in Figure 2.8b, on average, the same number of CPU cores is allocated for Len-1, Len-2, and Len-3. For Len-4, Kariz allocates 3-8% more CPU cores.

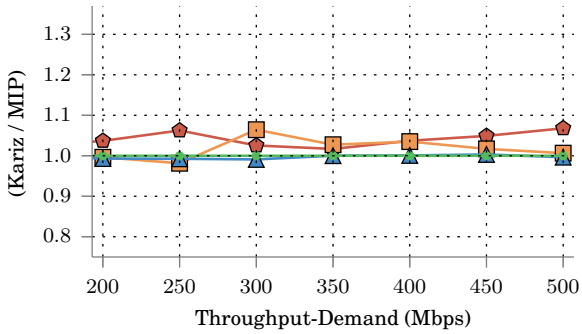
Finally, in respect to the total operational cost in Figure 2.8c, the following cost ratios vs MIP are observed: 100-101% for Len-1, 103-105% for Len-2, 105-108% for Len-3, and 117-124% for Len-4. Kariz is more cautious to allocate CPUs than to allocate bandwidth showing that improvement rounds (see Section 2.4.2) optimize the total cost by releasing CPUs while allocating more bandwidth. In summary, Kariz incurs a competitive per-chain cost that is less than 124% of MIP's.



(a) Bandwidth Utilization

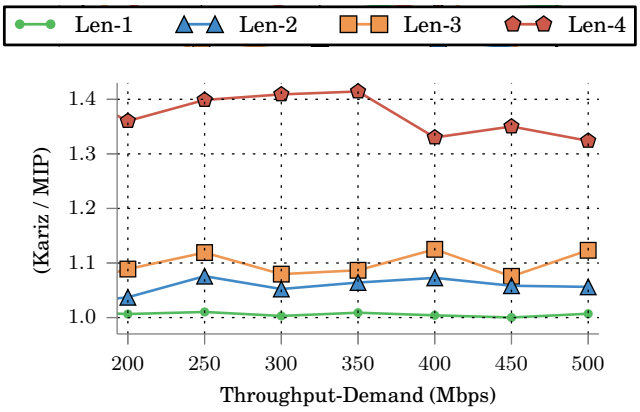


(b) CPU Utilization

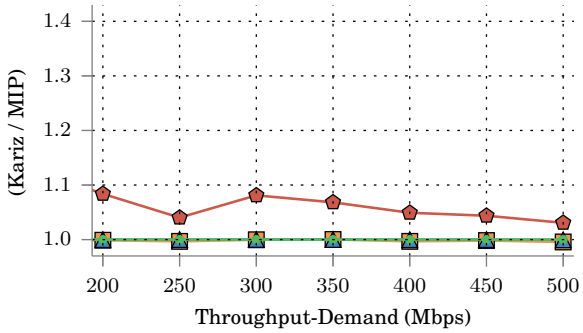


(c) VNF Utilization

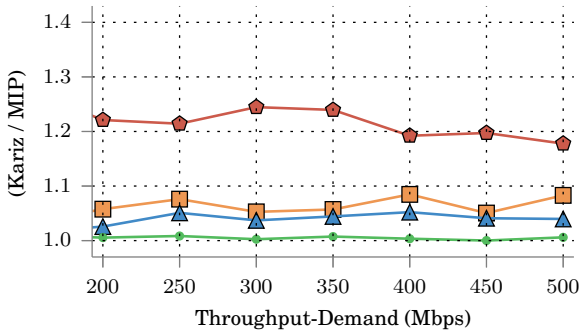
Figure 2.7: Resource Utilization Comparison



(a) Bandwidth Allocation Cost



(b) CPU Resource Allocation Cost



(c) Total Cost

Figure 2.8: Operational Costs

2.6 Related Work

In this section, we discuss our work compare to the related work.

Selection: VNF-P [112] studies a hybrid deployment scenario using hardware-middleboxes and software middleboxes to provide a requested service. VNF-OP [11], [74], JoraNFV [168], and [104] model batch-deployments of multiple chains. [121] is a scheduling framework for deploying middlebox instances. These papers assume that instances of the same middlebox are identical. Slick [6] is a framework that allows users to write fine-grained *elements* to perform custom packet-processing. Predicting the performance of such arbitrary packet-processing element is not trivial. In contrast to these studies, we select appropriate middlebox instances from different typical offerings providing predictable performance.

Placement: Split/Merge [134] and OpenNF [58] redistribute packet-processing across a collection of instances. In contrast to DSFC, they do not focus on placement optimization models. Stratos [56] orchestrates instances on a remote cloud. It uses a rather simple technique that places instances of middleboxes in a chain as closely as possible to each other. JVP [93] considers the relation of bandwidth usage and host resource usage in the deployment of chains. However, JVP instantiates a single VM for each middlebox type. VNF-OP [11] and VNF-P [112] place all instances of a middlebox on a single machine. In contrast to these works, we place multiple instances of each middlebox distributedly. Luizelli et al. [103] only optimize the placement of middleboxes and does not consider the routing. CoMb [142] is an architecture designed to consolidate the chain deployment. In contrast to DSFC, CoMb places all instances of middleboxes in a chain that deal with the same session at a fixed location. Such deployments consolidate middleboxes of a chain on the same server to multiplex substrate resources and reuse processing modules of a server across multiple middleboxes. However, these deployments limit chain performance to the resources of an individual server.

Routing: Unlike our work, [47, 130, 175] optimize only bandwidth usage. In processing a network flow, Slick [6] uses a single instance for each middlebox. On the contrary, DSFC routes traffic among multiple instances for each middlebox. Stratos [56] solves the routing separately after placing instances. LightChain [74] optimizes the number of switches between ingress and egress points of chains. The authors of [62] solve the joint placement and routing problem using a dynamic programming algorithm. E2 [121] instantiates instances in certain servers to optimize the inter-server communication. Although [11, 62, 112, 121] coordinate the placement and routing,

they still treat the selection separately. We jointly optimize routing, placement, and selection that was not the focus of these studies.

Virtual network embedding: Our problem has similarities to *virtual network embedding* (VNE) problem that deals with mapping virtual nodes and links onto a substrate network [49, 138]. However, our problem is fundamentally different; virtual links are part of the input to the VNE problem, while communication links between middlebox instances are unknown in our problem, and the solution must find these links.

2.7 Conclusion and Future Work

The limitations of current optimization models restrict a chain's throughput to resources of a physical-machine and result in sub-optimal resource utilization and service performance. In this chapter, we presented distributed service function chaining (DSFC) to overcome these limitations. DSFC decouples a chain's throughput from physical servers by placing instances of a middlebox in multiple servers. Furthermore, DSFC optimizes network utilization by coordinating the deployment operations.

We formulated DSFC using a mixed integer programming (MIP) model and proved its NP-Hardness for substrate networks with general graph topologies. An interesting future research direction is to analyze the complexity of our problem for specific datacenter topologies, such as fat-tree and VL2 [66].

For larger scales, we proposed and evaluated a heuristic called Kariz. The experimental results for various chain lengths and throughput demands demonstrate that Kariz achieves competitive cost and acceptance ratio compared to the MIP model. Kariz can be also adapted to deploy *reorder compatible chains* [9], i.e., their middleboxes can be reordered without affecting the semantics of the chains. The number of valid chains generated using reordering are small because service function chains are short, commonly with less than six middleboxes [92], and many reorderings are invalid because they result in changing the chain semantics [9]. Thus, we can generate possible valid chains, compute their deployment costs using Kariz, and select a chain with the lowest cost.

Chapter 3

Constellation: A Geo-Distributed Middlebox Framework

Middleboxes, such as firewalls, load balancers, and intrusion detection systems are pervasive in computer networks [3, 41, 135, 147]. The network function virtualization vision enables middleboxes to be flexibly deployed across a network and provisions new instances on demand. Middleboxes may have multiple instances to satisfy traffic demand and share state across instances to cooperatively process traffic.

Although the instances of a middlebox are typically deployed in the same data center, there has been an increasing demand for deploying middlebox instances across wide area networks [5, 12, 44, 121]. This growth stems from the trend towards building multi-data center applications, which necessitates global scale network management.

In many cases, even though the instances are connected by high latency wide area links, it is still necessary for them to share state [7, 33, 105, 111, 132, 174]. Examples include distributed rate limiters that share traffic information to monitor and limit the traffic of multi-data center applications [33, 132], intrusion detection systems with instances across an ISP network that share statistics to detect attacks [7, 105, 174], and proxies in a content delivery network that actively share their health statuses [169].

Existing middlebox frameworks that support state sharing have focused on optimizing for local area network deployments [58, 85, 134, 173]. Full control over routing allows these frameworks to maintain affinity between traffic flows to middlebox instances. This results in fewer remote accesses since per-flow state remains mostly local to an instance [85, 134, 173]. However, in wide area networks, traffic can span multiple administrative domains, giving a middlebox framework much less control over routing. Asymmetric routing and multipath protocols [119, 156] compound

this issue because a single flow may traverse multiple instances, thus requiring state sharing to process such flows. The result is more remote accesses to shared state across wide area links, which increases packet latency and reduces middlebox throughput. These frameworks use synchronous state access for correctness, which is only practical within a local area network as it can add a network roundtrip delay to each packet.

In this chapter, we introduce Constellation, a framework for geo-distributed middlebox deployments. Constellation provides a state management system that is highly scalable and performant even when middlebox instances are deployed across wide area networks. It separates the middlebox state from its application logic and abstracts shared state using *convergent state objects*, which can be independently updated yet still converge. Transparent to the middlebox application logic, Constellation asynchronously replicates state objects to other middlebox instances. Replication makes the state local to each instance, and convergence allows a middlebox instance to mutate state with only lightweight coordination.

Asynchronous replication of convergent state enables more flexible load balancing. Replication subsumes the need for flow-instance affinity, and enables any middlebox instance to process any packet with high performance, as the instance already has the required state. Replication also provides seamless dynamic scaling since traffic load can be rebalanced among instances without waiting for state migration.

Standard conflict free replicated data types (CRDTs) [143, 144] are convergent objects that can be used to represent the shared state for a class of common middleboxes. However, to support middleboxes such as intrusion detection systems and network monitors [19, 34, 46, 100, 102], we also develop new CRDTs including a counting bloom filter and count-min sketch. These CRDTs rely on an *ordering* property that is provided by our framework. Moreover, state updates in some middleboxes may necessitate violating CRDT properties. To address this limitation, Constellation support *multi-object updates* for packet processing where the same set of objects are always updated together. This is commonly used in network address translators and load balancers.

The properties of convergent state objects offer unique opportunities for us to build an efficient and reliable multicast state replication layer. Using the idempotence and commutativity properties of state objects, this layer coalesces state updates for more efficient utilization of wide area network bandwidth. It also provides higher tolerance for straggler instances, as they can receive and apply batched updates to reduce bandwidth and processing resources compared with executing uncoalesced operations.

We implemented Constellation using Click [87], and evaluated our framework by comparing its performance with S6 [173], the state-of-the-art middlebox framework for local area networks. Our results show that Constellation scales linearly with throughput and experiences no overhead due to network end-to-end latency. Over wide area networks, Constellation can process $96\times$ the

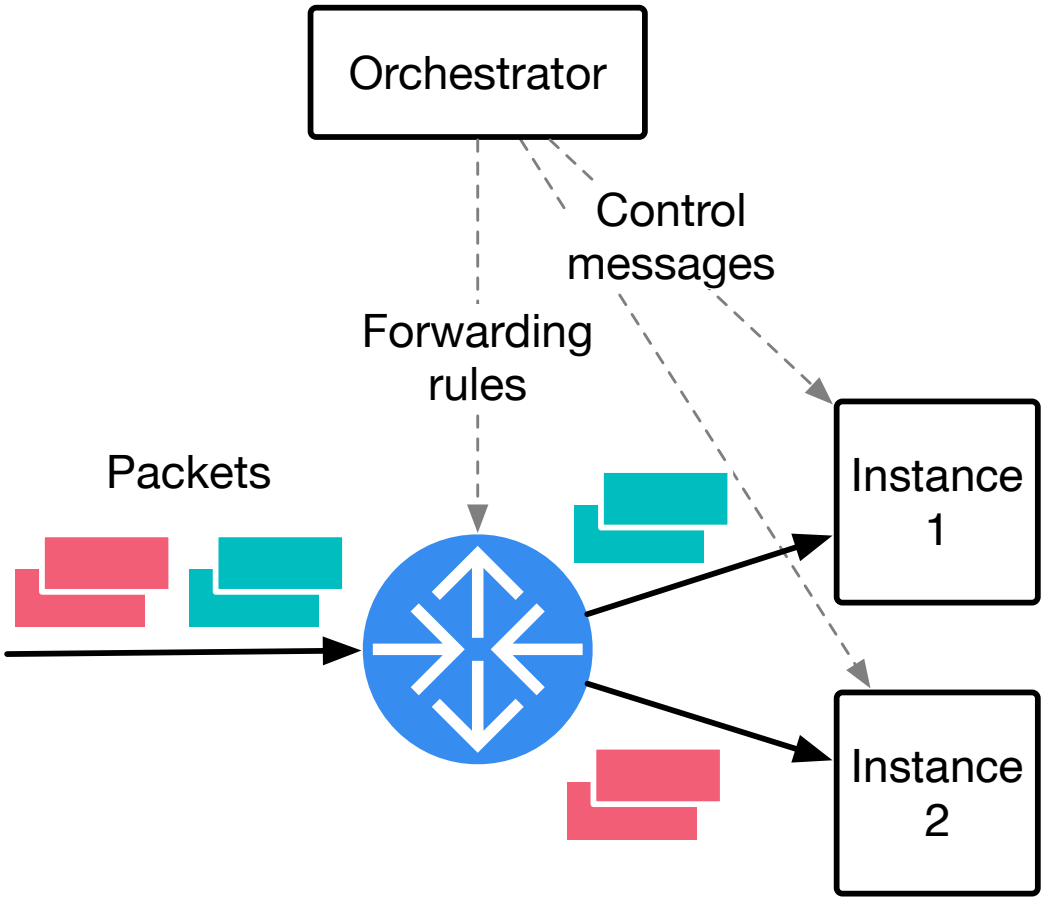


Figure 3.1: The architecture of an NFV environment

bandwidth of S6, which was not designed to tolerate latency. In local area networks, Constellation can process up to 11× the bandwidth of S6, which comes from eliminating the heavyweight mechanisms that S6 uses to hide remote state accesses (see Section 3.5.2). Finally, we show that the complexity of our middleboxes is similar to synchronous approaches when compared to S6.

3.1 Background and Motivation

Figure 3.1 shows a typical *network function virtualization* (NFV) environment, where an orchestrator manages middlebox instances deployed on servers and the network connecting these servers. In response to traffic load, the orchestrator dynamically adds or removes middlebox instances,

and installs forwarding rules in the network to redistribute traffic.

The above operations are sufficient to scale *stateless* middleboxes, e.g., firewalls that pass or block individual packets based on static rules. However, scaling *stateful* middleboxes becomes challenging, since in addition to the aforementioned operations, the middlebox state must be migrated simultaneously with the new workload distribution [121, 134, 173].

3.1.1 Middlebox State

Stateful middlebox instances maintain *dynamic state* regarding traffic flows, which changes how they process packets [67, 133, 151]. For example, stateful firewalls filter packets based on information collected about flows [10].

The middlebox state consists of *partitionable* and *shared* state. The partitionable state is accessed by a single instance, for example the cache in a web proxy [152]. Shared state can be for a single flow or a collection of flows processed across middlebox instances, and multiple instances query and mutate them. For example, IDS instances read and update port-counts per external host to detect attacks.

3.1.2 Recent Work: State Management for LAN

Existing frameworks that manage shared state are optimized for local area networks and support synchronous accesses to state [58, 81, 85, 134, 173]. State sharing using this model leads to remote accesses that incur performance cost relative to the network latency.

Two main approaches exist. One approach separates middlebox state into a remote data store [81, 85]. Remote state accesses increase packet latency and can reduce throughput by ∼60% [81], since extra CPU and bandwidth resources are consumed for remote I/Os. Another approach [58, 134, 173] distributes state across middlebox instances. For non-local state, instances query remote instances [173]. Frequent remote accesses can significantly degrade performance. Full control over routing allows them to reduce remote accesses by consistently routing a traffic flow to the same instance so that the flow state remains local to that instance.

Synchronous remote accesses of shared state lead to decreased performance for both approaches. These frameworks introduce several optimizations to maintain performance of their middlebox implementations. The state-of-the-art framework, S6 [173], masks the overhead of remote accesses using concurrency. An instance creates a microthread per packet to enable context switching to other packets while waiting on synchronous requests. However, as we will discuss

in Section 3.5.2, our results show that the overhead of using a microthread per packet halves the maximum throughput of the framework.

Another optimization is to trade consistency for performance. An instance caches state and performs reads and writes locally [85, 173]. To avoid permanent divergence, cached state copies must be merged periodically. Doing so naïvely can result in the consistency anomalies, such as lost updates.

These optimizations complicate the middlebox design. To achieve acceptable levels of performance and scalability, developers may need to use a combination of these mechanisms. Reasoning about their correctness is complicated; an incorrect usage can be a source of subtle bugs in middlebox applications.

3.1.3 Geo-distributed Middleboxes

Middleboxes are increasingly being deployed across wide area networks, e.g., ISP networks, multiple data centers, and content delivery networks. In such deployments, middlebox instances share state to cooperatively process traffic.

For example, rate limiter instances monitor and limit traffic loads from multiple locations in a content delivery network [33, 132, 169]. They share their state to be able to limit the global traffic load of multi-data center applications [132], limit the load at the edge [33], and control the impact of traffic spreaders [169].

IDS instances deployed across an ISP network [7] share their local statistics to detect intrusions [7, 105, 174]. Using network-wide statistics collected from different network locations is essential to detect attacks, such as port scanning and denial of service [105, 149, 165, 174]. Moreover, multiple NAT instances share the same flow table to translate network addresses across an ISP network [89].

In the CoDeeN peer-to-peer content delivery network [169], distributed proxies share their health status. To handle a cache miss, a proxy redirects a content request to another peer that is healthy based on this information.

These middlebox deployments face two challenges. The first challenge is due to characteristics of wide area network traffic. Traffic can span multiple administrative domains with less control over routing. This increases shared state hindering scalability and performance. Moreover, asymmetric routing and multipath routing [119, 156, 160] undermine the flow-instance affinity. In asymmetric and multipath routing, a traffic flow, e.g., sub-flows of a MPTCP session [119], may traverse different paths and consequently different middlebox instances. For a correct operation, the instances must share even per-flow state.

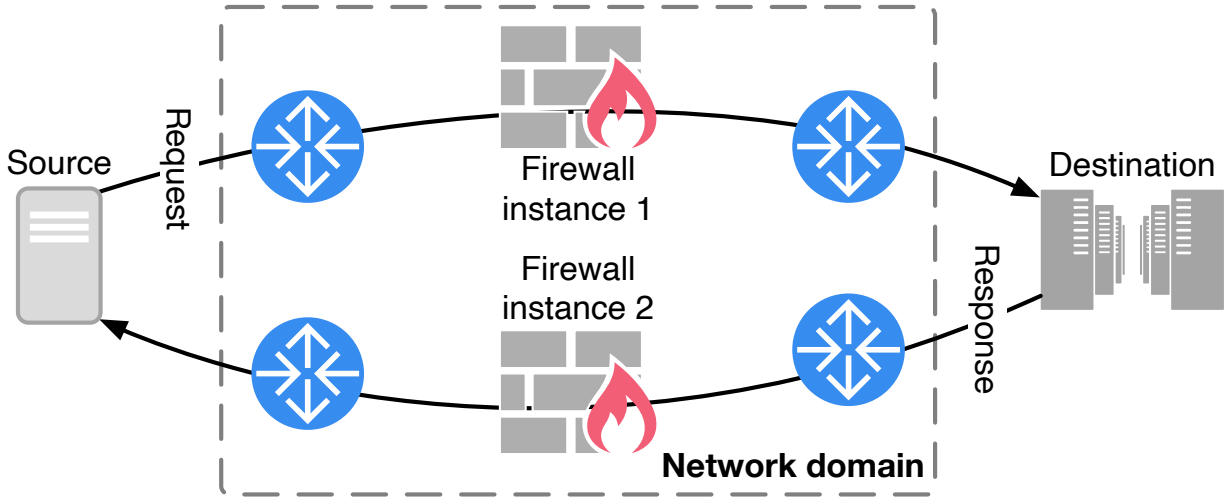


Figure 3.2: Firewall and asymmetric routing. The second accepts the response traffic because the instances share state.

State Location	Latency	Throughput
Local Machine	1–100× ns [17]	10 M–1 G
Remote LAN	10–100× μs [50, 127]	10 k–100 k
Remote WAN	10–100× ms [28, 88]	10–100

Table 3.1: Time to access state in different locations: The throughput of remote state across a wide area network can be as low as 10 to 100 accesses per second.

Figure 3.2 shows an example asymmetric routing scenario where the traffic of a connection passes through two firewall instances. A firewall commonly allows connections to initiate only from protected zones, e.g., "Source" in Figure 3.2. State sharing among firewall instances is essential, since the second instance allows the response stream only if the first instance shares that it has observed the request stream [64, 111].

The second challenge is wide area network latency making state sharing extremely costly. Table 3.1 lists the access times to shared state when it is stored locally, or remotely over a local and wide area networks. Existing frameworks [81, 85, 173], designed for infrequent state sharing in local area networks, cannot tolerate frequent state sharing over networks with higher latency [23, 28, 88, 107].

Middlebox	State	Purpose	Shared	Abstract data type	Size (B)
IDS/IPS [18, 34, 91, 100]	Session context	Session inspection	✓	Map	$96\times c$
	Connection context	Connection inspection	✓	Map	$\sim 250\times c$
	Bloom filter	String rule matching	×	Bloom filter	6250
	Flow size distribution	Traffic summaries	✓	Count-min sketch	20 k–200 k
	Port scanning counter	Port scan detection	✓	Counters	$112\times n$
Firewall [10, 118, 136, 164]	Flow table	Dynamic rules	✓	Map	$32\times c$
	Connection context	Connection inspection	✓	Map	$264\times c$
Network monitor [68, 76, 77, 149]	Traffic dispersion graph	Anomaly detection	✓	Graph	$256\times n$
	Heavy hitters	Tracking top heavy flows	✓	Count-min sketch	-
Load balancer [41, 72, 83]	Flow table	Connection persistence	✓	Map	$28\times c$
	Server pool	Available backend servers	×	-	$20\times n$
	Server pool usage	Usage of backend servers	✓	Vector	$28\times n$
NAT [5, 118]	Flow table	Address mapping	✓	Map	$74\times c$
	Available address pool	Tracking available addresses	✓	Set	$80\times c$
Web proxy [46, 153]	Stored entries	Metadata of cached contents	×	Map	$104\times c$
	Cache contents	In memory or storage cache	×	-	Dynamic
	Cache digests	Compact cache summary	✓	Counting bloom filter	$20\times c$

Table 3.2: Examples of common middleboxes: A list of common middlebox applications are shown. For "Size (B)", c and n are respectively the number of connections/sessions and hosts/servers. Note that we provide a representative set of state for each middlebox, and the they are not exhaustive. Moreover, for each middlebox application, we list the state of multiple implementations, and an implementation does not necessarily include all the presented state.

3.2 Design Overview

Table 3.2 shows a list of popular middleboxes and a representative selection of their state. This list is not exhaustive but provides a representative set of abstract data types used by middleboxes. To better understand the needs of middleboxes in wide area networks, we start by examining the needs of these popular middleboxes.

3.2.1 Study of Common Middleboxes

Our study reveals two key observations. First, most middleboxes maintain shared state for collecting traffic statistics, resource ownership, and resource usage. Second, middleboxes mostly operate on relativity small shared state and trade off precision for higher scalability and performance [7].

A corollary is that most operations on shared state are simple even when the middleboxes are not [81, 85].

Purpose of sharing state: Middleboxes collect statistics about traffic for detection and mitigation purposes. As shown in Table 3.2, an IDS/IPS track statistics of traffic connections and sessions to detect abnormal or malicious communications. The instances of a signature based IDS need to share their statistics to detect advanced attacks that can exploit multi-path routing in a wide area network. These attacks split their signatures across multiple paths to circumvent traditional signature based detection approaches [105]. A stateful firewall inspects the collected statistics to block malicious connections and maintain dynamic rules for outgoing connections. As mentioned before, the firewall instances may be required to share their state to handle asymmetric flows [111]. A network monitor maintains a traffic dispersion graph that embodies the communications between network nodes. A network wide representation can monitor thousands of hosts to detect large scale attacks [77, 155]

Middleboxes also track resource ownership and resource usage for resource management purposes. For example, a load balancer manages backend servers, and distributes the load among them based on their usage. A NAT manages a set of available public addresses and allocates these addresses among network flows. NAT instances in an ISP network share their state to correctly route asymmetric flows [89].

Shared state implementation: Shared state tends to be small, which reduces communication overheads between instances [105] and per-packet processing costs. For the middleboxes shown in Table 3.2, to serve millions of flows, the shared state requires only a few 100 MB of the memory. For example, the most memory intensive middleboxes are web proxies that keep a large cache local to each instance [152]. Advanced proxies share a compact summary of their cached contents [46] to allow redirecting content requests to nearby instances. This improves the quality of service in serving content requests in a content delivery network.

Although many middleboxes make complex decisions based on shared state, their operations on shared state are simple. Others have also observed that middleboxes operate on shared state with a simple set of operations [81, 85]. For example, an IDS collects lightweight packet summaries, but performs complex detection operations locally. Rate limiter instances share their observed flow rates [132] and use probabilistic analysis to shape traffic of multiple data centers.

Middleboxes collect approximate statistics when collecting precise statistics is too costly in terms of memory or processing overheads. They sometimes use compact and approximated statistics for a faster request serving. For example, IDSes and network monitors often use count-

min sketches or bloom filters, which are probabilistic data structures, to track top heavy hitters. Web proxy uses cache digests to quickly check local contents when serving requests.

3.2.2 Constellation Design Choices

Our observations lead us to design Constellation. Constellation is a geo-replicated middlebox framework that deploy a cluster of middlebox instances distributed across a wide area network. Constellation provides for the management of shared state across the entire deployment.

Constellation separates the design of middleboxes into middlebox logic and middlebox state to hide the complexity of state sharing from the middlebox logic. Transparent to the logic, Constellation asynchronously replicates updates to shared state from each middlebox instance. Using convergent state, Constellation eliminates most of the complexity of managing asynchrony. Our key design choices are as follows.

Asynchronous state replication: Constellation replicates middlebox state to all instances asynchronously. Each middlebox instance collects and sends its updates of shared state to all other instances in near real time, and they apply these updates to their state locally. All instances access shared state locally without querying remote instances.

Asynchronous access to local state allows middlebox instances to share state over high latency links of a wide area network. Replication also supports flexible load distribution and seamless dynamic scaling by subsuming the need for flow-instance affinity. If a middlebox instance is overwhelmed, traffic can be immediately rerouted to an existing instance that is under utilized. In removing an excess instance, the orchestrator can reassign traffic load from this instance to another without waiting for state migration.

Storing a replica of shared state requires more memory than that of existing frameworks, but the overhead is not substantial. As we observed in Section 3.2.1, many middleboxes operate on lightweight shared state with small memory requirement. Even larger memory usage does not change the cost of running middleboxes in the cloud, where computation to memory ratios are fixed. Middlebox applications already require substantial compute resources that usually goes hand-in-hand with more than enough memory.

Asynchronous state replication also trades the *consistency* of state across instances for performance. For many cases, our state model framework automatically resolves inconsistencies using *convergent state objects*.

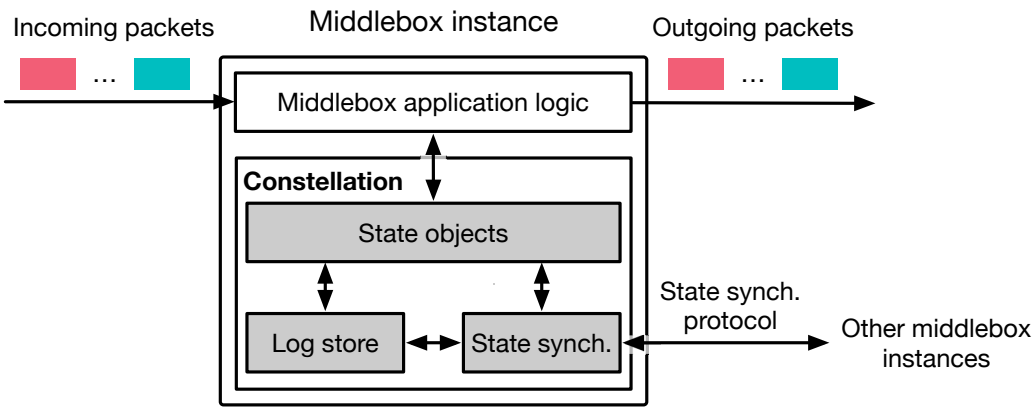


Figure 3.3: Constellation's design components

Convergent state objects: Constellation provides a set of state objects to develop the middlebox state. These objects are guaranteed to be *convergent*, i.e., middlebox instances will observe the same local value for a state object after they receive and perform the same set of state updates applied in other instances. Convergence eliminates complexities that may arise due to asynchronous state replication assuring developers about the correctness of shared state.

As we discussed in Section 3.2.1, most middleboxes perform simple operations on shared state. For these middleboxes, Constellation's builtin convergent state objects can be used to implement their shared state. For more complex cases, Constellation provides mechanisms for developers to customize state objects to reconcile conflicting state updates.

Asynchronous state replication even when using convergent state objects may introduce some artifacts. We discuss these artifacts and their impact on middleboxes in Section 3.4.2.

Centralized orchestration: A logically centralized orchestrator [24, 54, 113, 126] monitors traffic load, scales the number of instances according to load variations, and distributes the load among the instances. Constellation does not involve the orchestrator in state replication to avoid creating a potential performance bottleneck.

3.3 Constellation Middlebox Framework

Constellation is designed to allow developers to create geo-distributed middlebox applications with seamless scalability and network latency tolerance. Figure 3.3 shows a middlebox instance and the main components of the Constellation framework. Our framework works as follows.

The *middlebox logic* registers a set of *state objects* that model the middlebox state. The state objects provide APIs to access and mutate state, which are used by the middlebox logic during its packet processing (discussed in Section 3.3.1).

Constellation internally tracks local state updates by recording them into its *log store*. The *state synchronization* component replicates the recorded state updates to other instances concurrent with packet processing. All other instances will apply state updates received by the state synchronization component to their local state objects (discussed in Section 3.3.2).

During adding or removing of middlebox instances, Constellation adjusts the membership of the middlebox cluster while keeping other instances to process traffic. Constellation selects only one existing instance as a source of the state in bringing a new instance up-to-speed; other instances experience almost no performance disruption by this membership change (discussed in Section 3.3.3).

3.3.1 State Objects

A state object encapsulates a set of variables and operations to access and update their values. Operations are designed to guarantee that state objects remain convergent, which simplifies reasoning about asynchronous state replication.

To ensure convergence, Constellation uses data structures based on CRDTs [143, 144] for its state objects. Two general types of CRDTs exist [143]: i) *state-based* CRDTs where a state update is performed completely local, and the *entire* state object is disseminated for state synchronization; ii) an operation-based CRDTs where operations are propagated. Due to the high bandwidth overhead of state-based CRDTs, we opt for operation-based CRDT.

For convergence, each instance sends its local update operations to other instances, and each downstream instance applies the received operations. All operations are *idempotent* and *commutative* so instances can safely apply the operations out-of-order and still converge [143].

We create a library of convergent state objects based on our study of various middleboxes shown in Table 3.2. The library consists of a collection of abstract data types based on operation-based CRDTs, for instance different flavors of `map`, `set`, `register`, and `counter` objects. These convergent objects can implement the basic state of common middleboxes.

Constellation addresses two limitations of existing CRDTs in implementing the state of middleboxes. First, Constellation provides atomic updates across multiple objects to address which are not supported by CRDTs. Second, Constellation develops a *convergent* counting bloom filter and count-min sketch which are common in intrusion detection systems, network monitors,

and proxies (see Table 3.2). These objects have not been proposed in prior work to the best of our knowledge.

Multi object updates: Middleboxes need to mutate multiple objects simultaneously, but this may violate the properties of CRDTs. This limitation is because convergence is guaranteed for operations on individual CRDT objects [143]. Reasoning about the convergence of general multi-object operations is complicated, because the objects may diverge when mutations on multiple objects do not commute.

In operating on multiple state objects, there are middleboxes that always update the objects together. This is common in managing resources where middleboxes update resource usage or ownership when they allocate or release the resources. For example, a NAT updates an address pool and a flow table object together in processing a new flow, and a load balancer updates the server pool usage and its flow table together in assigning a new flow to a backend server.

Operations on multiple state objects do not violate convergence when two conditions hold. First, such a multi-object operation commutes with other operations defined for the objects. Second, the operation is idempotent.

Constellation supports multi-object operations by defining a derivative object that contains multiple state objects, and performs operations on its state objects simultaneously. Constellation *atomically* replicates this derivative object in a downstream instance to ensure convergence. We discuss using derivative state objects to develop a NAT in Section 3.4.1.

Counting bloom filter: A counting bloom filter (CBF) is a memory efficient object for approximate counting. CBF is used in packet classification, deep packet inspection, and network monitoring [19, 34, 46, 102] where keeping accurate statistics with fine granularity does not scale to traffic load.

A CBF represents a large set of n counters using a smaller vector of m counters. It uses k hash functions to update the counters. A CBF exposes `count` and `value` operations. The `count(x)` operation computes k hash values for an operand `x` (e.g., `x` can be a five tuple flow identifier). Each hash value provides an index $0 \leq i < m$, and the CBF increments the counter at k computed indices. On `value(x)` operation, the CBF computes the same set of k hashes and returns the minimum value among the relevant counters. The returned value is an approximation, since the exact value of `x`'s counter is less than or equal to this value.

For convergence, `value` and `count` operations must be commutative and idempotent [143]. As `value` does not mutate any counter, we focus on `count` operation. Addition commutes, thus `count` also commutes; however, addition is not idempotent. Idempotence can be provided using

Constellation's ordering feature. This feature prevents applying a duplicate `count` operation, thus a local `count` operation performed in a middlebox instance is only applied once at any other instance across the entire middlebox cluster.

Count-min sketch: A count-min sketch (CMS) is a probabilistic data structure similar to counting bloom filter and has application in packet classification and deep packet inspections [100, 141].

A CMS has k arrays of n counters. A CMS uses k hash functions to update the counters, one hash function per array. A CMS provides the same set of operations as a counting bloom filter. To provide convergence, we use the same technique as that of a counting bloom filter. The idempotence of `CMS` is provided using Constellation's ordering feature.

3.3.2 Asynchronous State Replication

An efficient replication system for convergent state objects has two requirements. First, we must deliver and apply all operations to all other instances quickly to achieve fast convergence. Convergent state objects require a one-to-all dissemination of updates and allow commutativity of updates. Constellation uses multicast to build a replication system that allows unordered delivery of state updates. Multicast uses bandwidth efficiently compared to having $O(n^2)$ point to point transports and reduces latency compared to other topologies, e.g., forwarding state updates through middlebox instances one by one.

Second, we must mitigate straggler instances that can slow down the replication for the entire middlebox cluster. Instances may fall behind because of insufficient bandwidth or processing power. We use the idempotence and commutativity properties of CRDTs to build a congestion control scheme that coalesces state updates to adaptively meet bandwidth and processing constraints of stragglers. This congestion control can bound how far behind any given instance is from others. Increasing coalescing can reduce bandwidth requirements, but state updates fall further behind.

Multicast replication: During a middlebox operation, Constellation records state updates into a log store. For each state object, the log store allocates a queue of *log record*s and maintains a sequence number to track these records. A log record tracks an operation by recording the local order at which it is performed. Specifically, a log record denotes that an operation was performed on a set of operands at a particular sequence number. For example, for an IDS's port counter,

log record `(inc,{22},7)` shows that a counter was incremented for SSH port 22 at sequence number 7.

The state synchronization component allocates a multicast group per state object. Via this group, middlebox instances share and exchange their local log records of the object. For convergence, all log records must be delivered and applied to other instances. Log records are released once delivered.

Constellation runs two threads at the sender and receiver sides to transmit log records. Send threads use multicast to send local log records to other instances. Receive threads receive and apply log records, send acknowledgements, and prunes log records delivered to all other instances.

For each state object, the send thread continuously retrieves outstanding log records, i.e., log records that have not yet replicated in other instances, from the queue of this object, creates a *state message* from the records, and sends the message to the associated multicast group. The send thread round-robins between queues belonging to different objects.

A state message carries a list of log records, and an *acknowledgement* vector. Each instance maintains the acknowledgement vector to track log records received from other instances. This vector contains the highest sequence numbers seen for each instance.

The receive thread uses the acknowledgement vector to track external log records. Upon receiving a state message, the receive thread applies the operations from the message, and updates the instance's acknowledgement vector. The receive thread increments a sequence number of the vector when all log records up and including this sequence number have been received.

Since operations are idempotent and commutative, state objects converge even when applying duplicate or out of order log records [143]. The receive thread can be also configured to apply log records in order to provide idempotence for an easier implementation of state objects that are not intrinsically idempotent. We used this property to provide idempotence for `count` operation of counting bloom filter and count-min sketch in Section 3.3.1.

The receive thread also prunes the log store according to received state messages. For each object, this thread records the last acknowledgement that it has received from other instances. The smallest acknowledgement shows the latest log record that has been replicated in all instances. Accordingly, the receive thread prunes all log records up to the smallest acknowledgement.

We provide reliable multicast by retransmitting lost log records due to packet drops. If a log record has not been acknowledged, the send thread retransmits the record after a timeout based on the maximum round trip time of any instance. If a multicast channel is idle, the send thread periodically transmits keep-alive messages containing the latest acknowledgement vector.

Adaptive bandwidth optimization: There are cases when middlebox instances may fall behind in replication due to transient events, such as temporary congestion in the network. In these cases, Constellation uses *coalesced* log records instead of sending individual log records. Coalescing can significantly reduce the bandwidth usage of state replication.

The idempotence and commutativity properties of state objects allows Constellation to coalesce related log records, i.e., the records of state operations modifying the same object. For example, a series of increments and decrements to a single counter can be represented as adding the sum of the operations. To coalesce related operations, the send thread calls back into to the associated state object.

Constellation detects instances that are falling behind by monitoring the round trip time (RTT) of each instance in the multicast group. The receive thread measures the minimum and average RTTs for each instance using acknowledgements. When the average RTT is higher than the minimum RTT by a set threshold, the instance is marked as congested.

Upon detection, the send thread starts to send coalesced log records using an adaptive *lookahead window* based on RTT. The instance continuously monitors the average RTT to increase or decrease the lookahead window. The larger the lookahead window, the more coalescing opportunity.

Constellation adjusts the lookahead window to trade-off the bandwidth reduction from coalescing and the increased synchronization latency from delaying the transmission of log records. The lookahead window size is set based on the throughput. An instance coalesces log records up to a maximum lookahead or when an acknowledgement is received. If the acknowledgement arrives early, the instance immediately transmits any already coalesced updates, which effectively reduces the lookahead size.

Coalescing can handle the majority of cases when middlebox instances fall behind. In rare events, some middlebox instances may become stragglers due to significant bandwidth or processing limitations and can slow down Constellation's replication system.

Constellation detects stragglers using the same RTT based mechanism. Upon detection of such straggler, the instance starts to transmit its log records over a TCP connection for this instance, rather than multicast. First, the send thread transmits any outstanding log records over the TCP connection to assure that the straggler has received all log records up to a particular sequence number. Then, the send thread can transmit coalesced updates from this sequence number. If the RTT to the straggler becomes small and has no coalescing benefit, the instance must no longer be straggling and can rejoin the multicast group. Before switching back to multicast, the instance drains its TCP buffers to ensure that no log record transmission is missed.

3.3.3 Dynamic Scaling

A scaling event changes the members of multicast groups and consequently impacts state replication. For a correct state replication during and after this membership change, Constellation ensures three properties: i) *unique identification*: with a new set of members, each active instance remains uniquely identifiable; ii) *membership agreement*: instances agree upon active members so that their send threads and receive threads can work in harmony; and iii) *convergence*: the new set of members still remain convergent for all state objects.

We assume that the orchestrator is fault tolerant. If a failure occurs during scale-out or scale-in, the orchestrator reliably detects and notifies all instances of the change.

Scale-out event: A new instance joins replication groups and copies a snapshot of middlebox state from an existing instance before starting to process traffic. Adding a new instance is broken down into four steps.

First, the orchestrator deploys a new instance with unique identifier. To ensure uniqueness, it is sufficient that the orchestrator generates a new identifier or reuses the identifier of a removed instance.

Second, the new instance joins the replication groups and starts to record state messages. It sends a `join` message containing its identifier to the multicast groups to announce that its joining. Existing instances confirm receiving a `join` message by adding the instance to the acknowledgement vector of its state messages. Upon receiving this confirmation, the new instance starts acknowledging state messages received from existing instances. It does so by sending empty state messages with an acknowledgement vector. The new instance retransmits the `join` message until all existing instances confirm receiving the message. This ensures the membership agreement property.

Third, the new instance requests an existing instance for a snapshot of the state and metadata (includes the acknowledgement vector of each state object). The existing instance executes a `fork` system call [97] to duplicate its process to take a state snapshot and transmit it to the new instance. For the snapshot consistency, `fork` is synchronized with packet processing and state synchronization.

Finally, upon receiving the state snapshot, the new instance applies relevant log records from state messages recorded since the second step. Lastly, the new instance notifies the orchestrator to redistribute traffic load to it.

Taking snapshots using `fork` is fast, since memory is not immediately copied. The memory is marked as *copy-on-write*; the operating system copies memory after the original or child process

modifies it. Constellation further reduces this overhead by using `madvise` [98]. Since the duplicated process does not process incoming packets, Constellation tells the operating system to excludes memory pages reserved for receiving incoming packets. This significantly reduces the pause time of `fork` as this is typically a large savings.

Scale-in event: Another benefit of Constellation is that it can scale-in with no state loss and virtually zero packet loss. Removing an excess instance takes four steps.

First, the orchestrator reroutes the traffic load of the excess instance to other instances. Due to state replication, other instances have the state necessary to process this load. Second, the orchestrator notifies the excess instance and the instance waits for remaining inflight traffic to arrive. After some time, the instance drains its outstanding log records to ensure convergence. Third, the excess instance sends a `leave` message to a multicasting group to announce that it is leaving. The instance will retry until all instances acknowledge receiving this message, which ensures membership agreement. Finally, once all other instances have acknowledged the `leave` message, the excess instance notifies the orchestrator to reclaim all resources.

3.4 Implementation and Experience

We built Constellation using the Click modular router. It consists of 6141 SLOC for the runtime and 2155 SLOC for the middlebox implementations. We discuss our development experience in using our system compared to exiting frameworks that provide synchronous middlebox state management. We dive into the implementation of a flow table and a NAT. Lastly, we discuss the artifacts caused by the use of asynchronous replication.

Convergent flow table: A *flow table* is used to track network flows and has application in several middleboxes as shown in Table 3.2. A flow table is a mapping keyed on a hashing of packet headers with values that can be network addresses or some attributes regarding the flows.

A flow table supports `add` and `value` operations. The `add(k,v)` operation either inserts flow `k` and value `v`, or updates the value of flow `k` with `v`. The `value(k)` operation returns a value associated with key `k`.

For convergence, we focus on `add` operation, since `value` does not mutate the state. The `add` operation is idempotent but not commutative. Enforcing a deterministic ordering on concurrent `add` operations "artificially" makes `add` commutative. Specifically, a global ordering across the

middlebox cluster determines the winner in a race between two `add` operations modifying the same key `k`.

The object exposes a callback that allows developers to customize this ordering. By default, the object uses a numerical comparison, where `(k,v)` wins against `(k,v′)` only if the binary value of `v` is greater than that of `v′`.

3.4.1 Network Address Translator

A NAT bridges two address spaces [5, 70, 154]. NAT instances share two state objects. Using a flow table object, a NAT instance maps traffic flows coming from one address space to another address space. A NAT instance identifies each flow by a unique port number from an available port pool object [154]. Both flow table and port pool are updated together, thus we can use Constellation's derivative state object feature to support multi object operations on these two objects.

In rare incidents, due to asynchronous local accesses, NAT instances may concurrently allocate an identical port number for different flows. This violates the NAT's *unique port assignment* invariant, and flow translations may collide.

Constellation's convergent flow table enable us to resolve this inconsistency. We use its callback so that among two collided flows, a flow with larger numerical value of its five tuple wins the race enabling all instances to converge.

3.4.2 Artifacts of Asynchronous Replication

Asynchronous replication may cause middleboxes to experience temporary inconsistencies until instances converge. We study a number of middleboxes including the ones shown in Table 3.2 for their possible artifacts. There are three categories of artifacts: lag or reduced precision; packet loss; and duplicates and collisions. In practice, these artifacts are non-issues, as they are rare and are already mitigated by existing protocols or end user applications. Our design makes the tradeoff of dealing with small artifacts for substantial performance gains on both local area networks and wide area networks.

Lag or reduced precision: The most common problem for most middleboxes is that asynchronous replication induces a lag in measurement or reduces the measurement precision. For example, an IDPS may set a threshold for when it blocks traffic and may lag by approximately the

round-trip time between instances. A distributed rate limiter may be imprecise in its ability to set a limit, but for longer flows it can still maintain a tight bounded error.

Packet loss: Packet loss issues can arise for stateful firewalls, NATs and load balancers. This may occur when traffic passes through a different instance while a connection is being established but before state is synchronized between instances. For example in Figure 3.2, the second firewall instance may receive the response traffic before its state is synchronized with the first instance. Since most protocols will retry dropped packets, this artifact will only result in a small increase in latency.

Duplicates and collisions: NATs and load balancers may also suffer from collisions or duplicate mappings, if two instances simultaneously generate conflicting mappings. For example, two NAT instances might reuse the same public IP and port for two connections to the same destination IP and port. In this scenario we can terminate one connection and have the client to reconnect. For even very large networks this is exceedingly rare and disconnects from other issue sources would be orders of magnitude more common.

An alternative approach is to design a NAT with an extra table to allow instances to acquire leases on regions of the public IP and port space. When a connection arrives the instance would allocate out of one of these pools, thus preventing collisions. This would require the middlebox to eagerly reserve new ranges when it is running low on the current pool.

3.5 Evaluation

We start with a description of our setup and methodology in Section 3.5.1, and then we measure the overhead of Constellation framework in Section 3.5.2. We measure Constellation's performance during its normal operation and dynamic scaling in Section 3.5.3 and Section 3.5.4, respectively. We measure the benefits of coalescing in Section 3.5.5. Then in Section 3.5.6, we measure the impact of Constellation's artifacts in our IDPS example. Finally, we discuss the implementation complexity in Section 3.5.7.

3.5.1 Experimental Setup and Methodology

We compare Constellation with S6 [173] and Sharded. *S6* is the-state-of-art in elastic scaling of middleboxes. We use the publicly available implementation of S6 [172]. An S6's middlebox

	Toolkit	No Operation	Reference	Read	Write	Read + Write
S6	11.80	5.96	3.66	2.52	2.08	1.38
Constellation	10.00	9.28	N/A	9.26	9.20	9.20

Table 3.3: Throughput of a pass-through middlebox in Mpps. S6 is built on DPDK, while Constellation uses DPDK+Click that adds overhead to the toolkit baseline. Reference adds the overhead of finding which node owns a state object. We measure the read and write costs separately and together.

application runs as a process that uses DPDK toolkit [78]. Sharded is a baseline system used to measure the performance upper bound, as middlebox traffic is sharded with no shared state. Moreover, we use two middleboxes, a NAT and an IDPS. Our implementation of IDPS includes only the port-scan detection/mitigation functionality.

We use a server cluster each equipped with a single Intel D-1540 Xeon with 8 cores and 64 GiB of memory. The servers are connected with an Intel Ethernet Connection X557 10 Gbps NIC to a Supermicro SSE-X3348T switch. A separate 10 Gbps Mellanox ConnectX-3 NIC connected to a Mellanox switch is used as the *state channel* for state synchronization. All servers run Ubuntu 18.04.

We use MoonGen [42] to generate traffic and measure performance. Traffic from a generator server is sent through a middlebox instance then back to the generator. We measure latency and total throughput at the traffic generators. The packet size in our experiments is 64 B. MoonGen measures end-to-end latency by sending timestamped 128 B packets while it is simultaneously sending load of 64 B packets. We also developed a tool that can accurately timestamp received packets at microsecond granularity, which allows us to accurately measure throughput changes. In Section 3.5.4 using this tool, we capture the impact of Constellation's dynamic scaling.

Unless stated otherwise, we run 5 second experiments and repeat each experiment 10 times. The confidence intervals of our results are all within 5%. As a result, we do not report them in our plots.

3.5.2 Performance Breakdown

Table 3.3 shows a performance breakdown for a pass through middlebox operating on a counter object. Using this middlebox, we benchmark S6 and Constellation to breakdown the performance cost of common middlebox operations. We configure the middlebox to either perform no operation, or perform a read, a write, or a read and write per packet.

S6 runs directly on DPDK, while Constellation is built using DPDK+Click which reduces baseline throughput by $\sim 15\%$. The no-operation measurement shows the cost from the S6 and Constellation frameworks. S6 processes each packet in a separate microthread, using Boost coroutines [16], allowing an independent microthread to process a packet while another microthread is blocked on a remote state access. Context switching between microthreads results in 49% loss of S6's performance.

The remaining columns measure the cost of reading and writing shared state. The reference column measures the time required for S6 to discover which instance owns the key of a flow. The read and write costs are measured separately and together. S6 slows down by a further 76% percent over the no-operation column, excluding the cost of microthreads.

3.5.3 Performance in Normal Operation

We measure the maximum aggregated throughput and the end-to-end latency of NAT and IDPS. For wide-area experiments, we deploy our NAT instances in a simulated WAN. Using `tc` [99], we configure the servers running instances to artificially add WAN latency [23, 88, 107] to the state channel. For both our LAN and WAN, we use a traffic load where each NAT instance receives 2000 new flows per second.

Throughput in LAN. Figure 3.4 shows the maximum aggregated throughput of the NAT and IDPS instances deployed in our LAN. For IDPS, we configure S6 in two modes. In the first mode, labeled "S6," the state updates are immediately synchronized. In the second mode, labeled "S6 W-behind," the remote counters are updated by a 10 ms delay.

As shown for both middleboxes, Constellation's throughput scales linearly with increasing the number of instances (within 2–4% of the ideal scaling for NAT and within 1–5% for the IDPS). For S6, per instance throughput of the NAT drops up to 21% due to the overhead of state synchronization. The throughput of IDPS drops for S6 and flattens for "S6 write-behind" going from 2 to 3 instances. In the S6 system, each instance has to query other instances to retrieve the values of their state objects. IDPS instances pay this overhead once every few packets (i.e., 50% and 66% of packets for 2 and 3 IDPS instances), while NAT instances incur this cost once every few flows (i.e., 50% and 66% of flows for 2 and 3 IDPS instances).

Compared to S6 for NAT, Constellation improves throughput by 2.5–3.2$\times$ and is within 2–4% of Sharded's aggregated throughput. For IDPS, Constellation achieves a 3.4–6.3$\times$ and 3.4–11.2$\times$ higher throughput compared to that of "S6 write-behind" and S6, respectively.

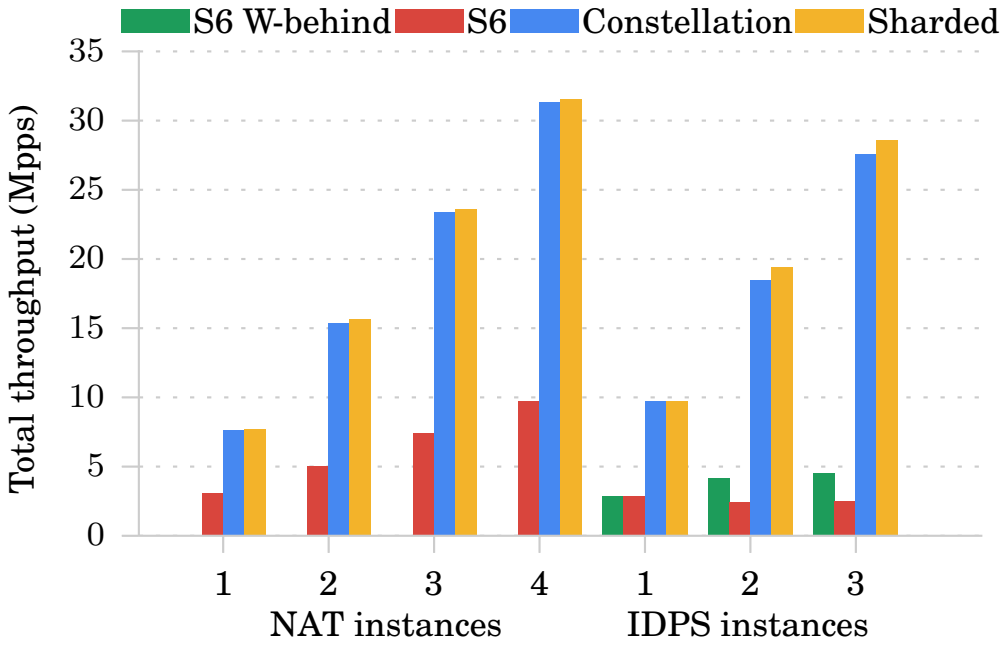


Figure 3.4: Total throughput of middleboxes: Compared to linear scaling, Constellation is within 2–4% for NAT and 1–5% for IDPS.

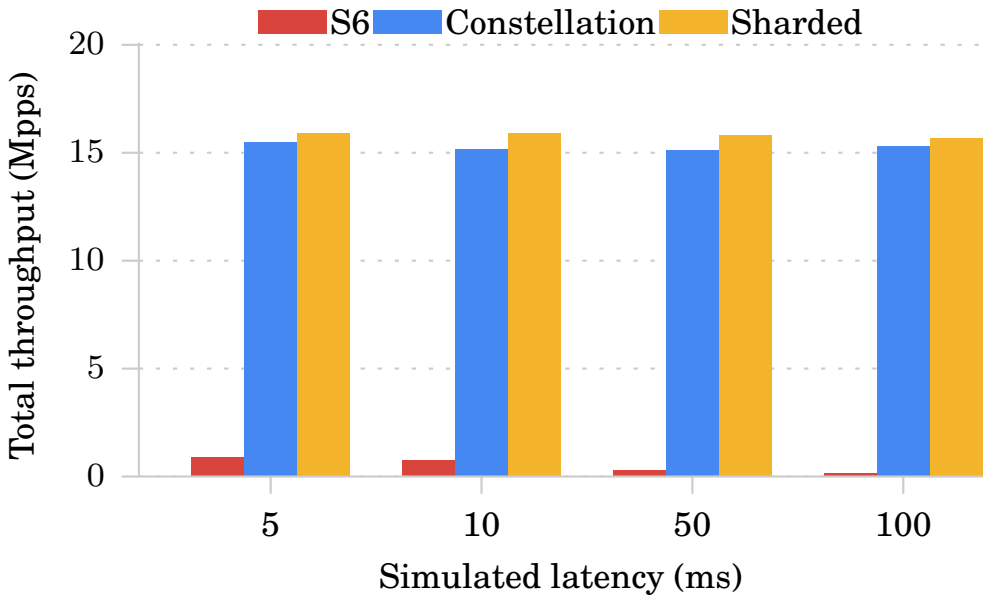


Figure 3.5: Total throughput of two NAT instances in WAN. Constellation's throughput is largely independent of WAN latency, but synchronous accesses to remote state slow down S6's throughput by $6\times$ to $32\times$ going from 5 to 100 ms latency.

	1 instance	2 instances	3 instances
S6	$21 \pm 1\mu$s	$25 \pm 1\mu$s	$26 \pm 1\mu$s
Constellation	$31 \pm 1\mu$s	$44 \pm 3\mu$s	$46 \pm 2\mu$s
Sharded	$31 \pm 1\mu$s	$32 \pm 1\mu$s	$34 \pm 2\mu$s

Table 3.4: NAT average latency. Constellation's latency remains constant going from 2 to 3 NAT instances. Its latency increase going from 1 to 2 is due to the scheduling overhead of Click.

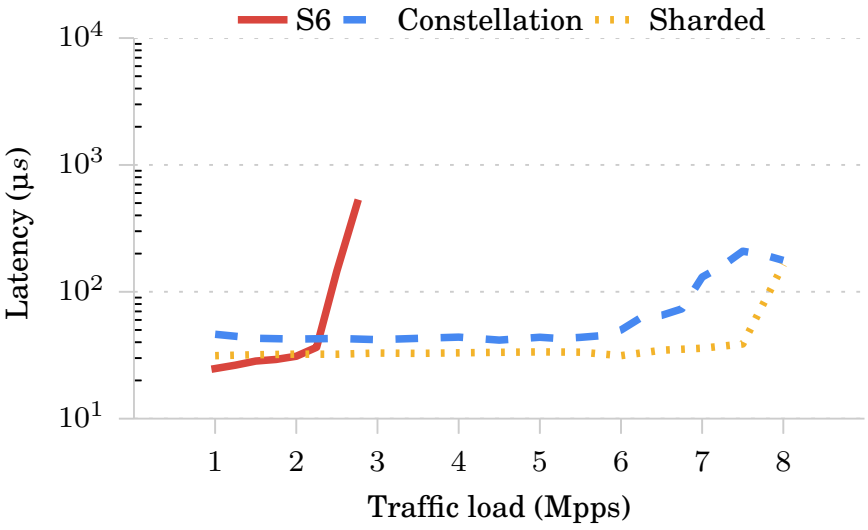
Throughput in WAN. We evaluate the impact of the state channel with WAN delay on the NAT throughput. As shown in Figure 3.5, the aggregated throughput of Constellation's NAT is independent of the WAN delay of the state channel. However, S6's throughput drops significantly. Compared to our LAN measurements, Constellation's WAN throughput is within 2–3% of its LAN throughput, while S6 becomes 6 to 32× slower. Constellation's throughput is 17 to 96× of S6's and is within 2–5% of Sharded's.

Constellation accesses the state locally and does not perform immediate state synchronization when a NAT instance writes or queries the state of flows. This asynchrony allows Constellation's NAT instances to operate at the same throughput level over the WAN as its local area network. On the other hand, S6's middlebox instances access state stored in a distributed hash table. Due to state distribution in this hash table, an instance owns a half of the state and must remotely query the other instance to operate on the other half. The overhead of this synchronous remote access is the root cause of S6's performance drop.

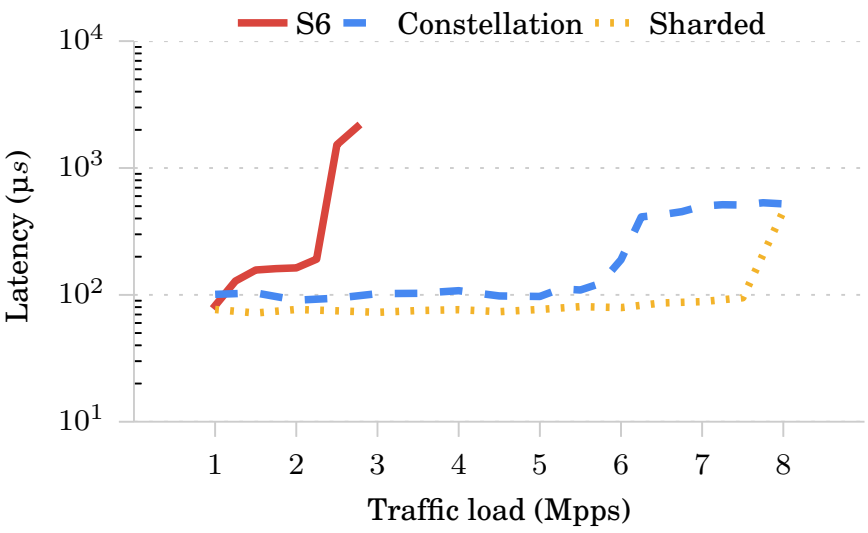
Latency in LAN. Table 3.4 presents the average end-to-end latency of the NAT in our LAN. For a fair comparison, the NAT instances are under S6's sustainable load of 1 Mpps with 2 k new flows per second. Going from one middlebox instance to two or more instances, both S6 and Constellation enable their state synchronization mechanisms between instances.

As shown in Table 3.4, going from 2 to 3 instances, Constellation's latency overhead does not increase. Compared to Sharded, Constellation adds 12 μs overhead. In our implementation, the receive thread of the state synchronization and the middlebox logic run on the same processor core, and we use Click's scheduler [87] to schedule them. The latency increase from 1 to 2 instances is due the overhead of Click scheduler [87]. S6's latency slightly increases going from 1 to 2 and 3 instances. Its latency is lower than Sharded, since S6's middleboxes run on DPDK, while Sharded's middleboxes uses DPDK+Click which adds Click's overhead to the baseline DPDK (recall from Section 3.5.2).

To investigate the latency overhead in more details we report the latency of the first instance of the 2 NAT instances in Figure 3.6 under different traffic loads. Figure 3.6a shows that average



(a) Average end-to-end latency



(b) 99 percentile end-to-end latency

Figure 3.6: End-to-end Latency of the first instance of 2 NAT instances deployed in our LAN. Constellation's average and 99 percentile latency remain steady under sustainable loads. Constellation's latency increases by approaching to its saturation point.

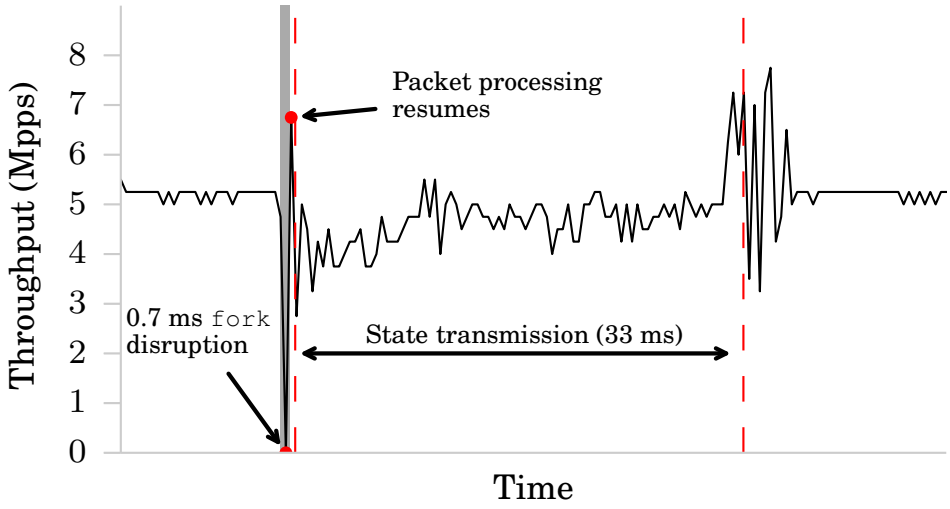


Figure 3.7: Throughput of the first instance of 2 NAT instances in a scale-out event. This instance experiences a sub-millisecond throughput disruption during `fork`. The throughput becomes unsteady for few milliseconds during state transmission.

latency remains steady for all systems as the traffic load increases until they approach their respective saturation points. Near these points, packets start to be queued, and latency rapidly spikes. The average latency of Constellation and Sharded remain under 209 μs, while S6's average latency spikes up to 500 μs. As shown in Figure 3.6b, 99-percentile latency has a trend similar to that of the average latency. Constellation's and Sharded's peak latency values are 451 μs and 539 μs, and S6 exhibits a peak latency of up to 2.1 ms.

3.5.4 Dynamic Scaling

We use our cluster of two NAT instances to quantify the performance of Constellation during dynamic scaling. Each instance is under a 5 Mpps load with 2 k new flows per second. We use our tool to measure throughput at microsecond scale resolution. We discuss only the performance of the instance that is involved in state transmission to the new instance. This instance reside in the same local area network as the new instance.

As shown in Figure 3.7, the first instance does not experience notable throughput degradation.

Packet drop is also zero. Excluding unnecessary memory pages from copy-on-write protection allows `fork` to complete in only a fraction of a millisecond. In a separate experiment, not shown here, we measured that `fork` lasts for 10 ms without this optimization.

Once packet processing resumes, we observe a throughput burst for packets queued during `fork` pause time. During state transmission, the throughput temporarily fluctuates. This is due to state updates in processing the first packets of new flows, since they dirty copy-on-write memory pages containing the state and incur memory copying overheads.

3.5.5 Coalescing Benefits

We evaluate coalescing benefits by measuring bandwidth savings achieved using coalescing state updates of a counting bloom filter. This state object counts the number of packets per network flow. Using network flows as keys is common in middleboxes.

Figure 3.8 shows coalescing benefits for two traffic traces, IXP-1 and IXP2, with different packet rates from an internet exchange point (IXP) [108, 109]. These traces represent traffic loads that a middlebox instance may receive in a wide area network. IXP-1 and IXP-2 are 15 minutes long with 4.2 M and 38.9 M unidirectional flows.

Figure 3.8 reports compression percentages achieved for different look ahead windows based on the number of packets. As shown, coalescing yields a saving of 45–96% for IXP-1 and 24–50% for IXP-2. IXP-1 has a higher saving rate compared with IXP-2 due to its flow distribution; packets are distributed across a less number of flows providing more coalescing opportunities. A larger lookahead window size makes coalescing more effective with the cost of staler state synchronization. Not shown here, we measure a 72% saving rate with a look ahead window of 1 M for IXP-2.

3.5.6 Inconsistency Artifacts

We measure the impact of Constellation's asynchronous replication in mitigating flooding a port number. We deploy two IDPS instances in a simulated WAN with 5 ms delay and configure them with a mitigation policy as follows. An instance blocks traffic destined to a port number if the traffic volume passes the threshold of 1024 Mbits. Two traffic generators flood the instances at 1 Mpps. For each instance, we measure the number of its *leaked packets*, i.e., the number of packets that pass through an instance in the distributed deployment compared with a theoretical centralized IDPS with infinite packet processing that receives the aggregated traffic and filters packets after crossing a given threshold.

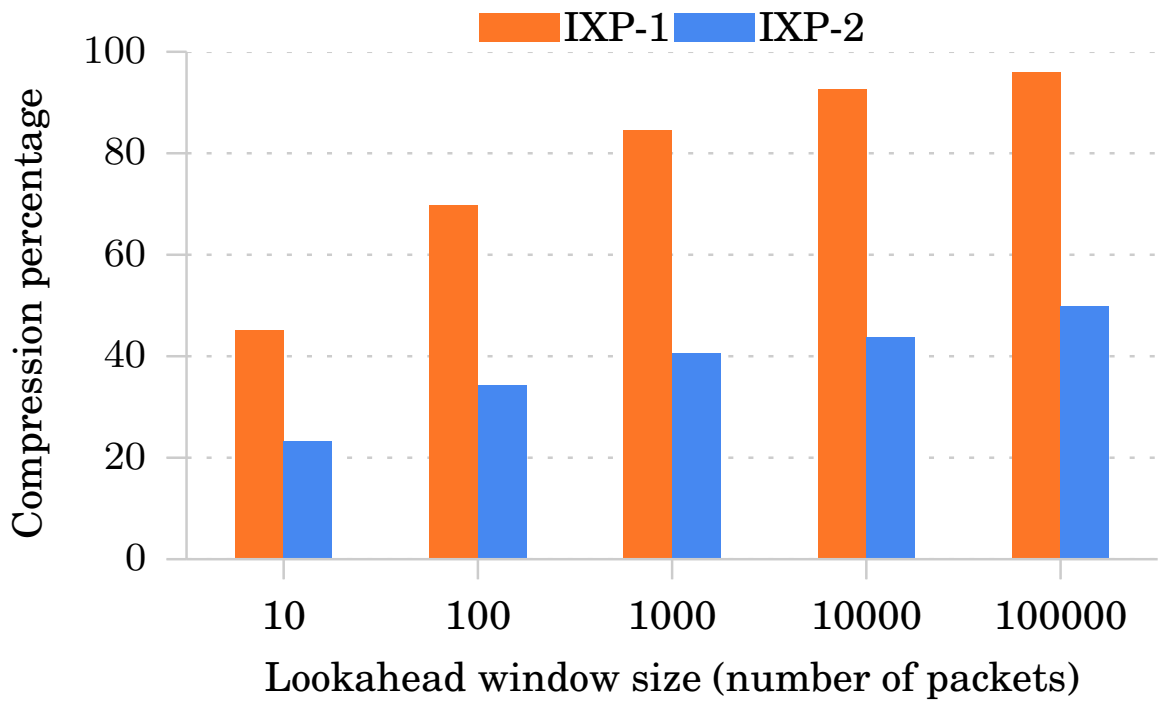


Figure 3.8: Coalescing benefits. IXP-1 and IXP-2 are two traces of an internet exchange point [108, 109]. We observe more compression for IXP-1 because IXP-1 has fewer distinct flows compared to IXP-2. Because the counting bloom filter operates on a smaller flow key space, more state updates can be coalesced for IXP-1.

Figure 3.9 shows a histogram of the number of leaked packets. Asynchronous replication delays blocking the attack. Constellation IDPS reacts to the flood within 5 ms and leaks on average 3.2 k packets.

Delaying an IDPS response by a few milliseconds is a good trade-off as it allows the system to keep up with the throughput demands of high speed networks. Previous work has shown that IDSes unable to keep up with the traffic can be bypassed to successfully launch attacks [30, 129, 150].

3.5.7 Development Complexity

Comparing the complexity of different middlebox frameworks is extremely difficult. Using the lines of code, we provide a rough estimation of the complexity.

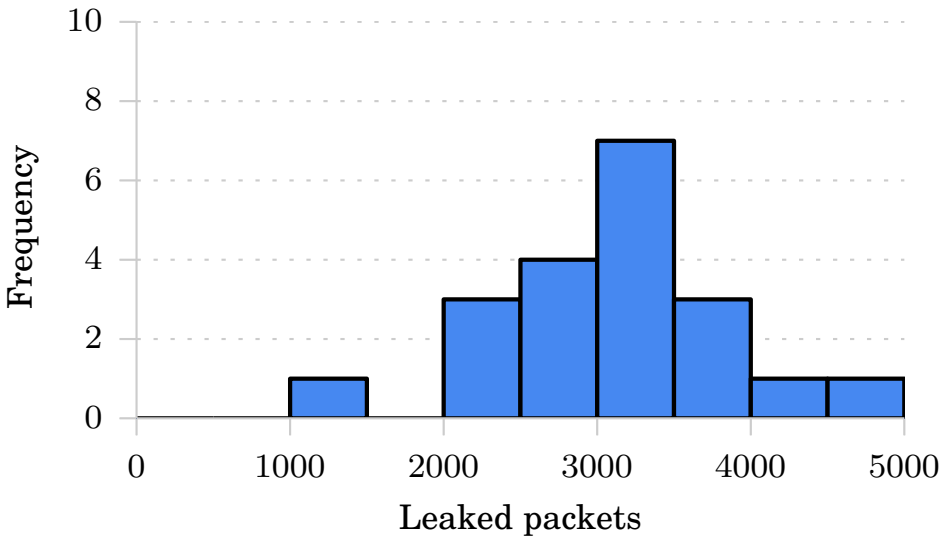


Figure 3.9: Histogram shows the number of packets leaked beyond the target threshold for Constellation's IDPS.

We compared the code of the NAT implemented in S6 to the one implemented in Constellation as described in Section 3.4. Both NATs are roughly the same size, with 361 lines of code for Constellation and 283 lines for S6. We measured S6's code before the source-to-source translator that provides syntactic sugar to simplify the system implementation. This result illustrates that it is not significantly difficult to build middleboxes with asynchronous replication.

3.6 Related Work

We have discussed existing frameworks that support state sharing for general middleboxes in Section 3.1.2. As mentioned, they are not optimized for wide area networks. Next, we compare Constellation with two other lines of related work.

Middlebox specific frameworks: Some systems are highly specialized for particular middleboxes. Most of them deploy middlebox instances as shards with no shared state [5,41,55,116]. Unlike others, vNIDS [95], a microservice based network intrusion detection system, and Yoda [55], an application layer load balancer, share their state in a central data store.

Database replication protocols: Two phase commit is a common protocol to replicate transactions in distributed databases [29]. This protocol supports general transactions using synchronous coordination. However, it does not scale for geo-distributed transactions as a transaction involves multiple rounds of message passing between a coordinator and replicas residing in different sites.

Multi datacenter consistency (MDCC) [88] optimizes performance by involving a coordinator only when transactions may conflict or violate constraints. For example, MDCC allows concurrent deposits and withdraws from a bank account only when the account balance is far from becoming zero.

Highly available databases relax generality of transactions or consistency guarantees for higher scalability and performance. Some systems split data into shards and restrict updates to only a single shard. Eventually consistent databases [32] allow asynchronous state replication with complex resolution mechanisms to resolve conflicts; however, these mechanisms can cause consistency anomalies.

3.7 Conclusion and Future Work

WAN latency can significantly impact the performance of a stateful middlebox whose instances are deployed across dispersed network infrastructures. In this chapter, we introduce Constellation, a framework for middlebox geo-distributed deployment. Using asynchronous state replication of convergent state objects, Constellation achieves high performance and scalability. Our results show that Constellation can improve middlebox performance by almost two orders of magnitude compared to state of the art.

Constellation can also be augmented to tolerate the failures of middlebox instances. Constellation can make each instance fault tolerant using existing systems, such as FTMB [146] and pico [133] that have addressed middlebox fault tolerance. These systems synchronously take state snapshots and replicate these snapshots into standby replicas. To recover from a failure, one of the replicas replaces the failed instance and joins the multicast channels of the middlebox cluster. The replica can reuse the identifier of the failed instance to hide the failure from other alive instances.

Chapter 4

Fault Tolerant Service Function Chaining

Middleboxes are widely deployed in enterprise networks, with each providing a specific dataplane function. These functions can be composed to meet high-level service requirements by passing traffic through an ordered sequence of middleboxes, forming a service function chain [130, 131]. For instance, data center traffic commonly passes through an intrusion detection system, a firewall, and a network address translator before reaching the Internet [167].

Providing fault tolerance for middleboxes is critical as their failures have led to large network outages, significant financial losses, and left networks vulnerable to attacks [27, 128, 158, 159]. Existing middlebox frameworks [81, 85, 90, 133, 146] have focused on providing fault tolerance for individual middleboxes. For a chain, they consider individual middleboxes as fault tolerant units that together form a fault tolerant chain. This design introduces redundancies and overheads that can limit a chain's performance.

Independently replicating the state of each middlebox in a chain requires a large number replica servers, which can increase cost. Part of that cost can be mitigated by having middleboxes share the same replica servers, although oversharing can affect performance. More importantly, replication causes packets to experience more than twice its normal delay, since each middlebox synchronously replicates state updates before releasing a packet to the next middlebox [81, 85, 90, 133].

Current state-of-the-art middlebox frameworks also stall as they capture a consistent snapshot of their state leading to lower throughput and higher latency [90, 133, 146]. These stalls significantly increase latency with packets experiencing latencies from 400 μs to 9 ms per middlebox compared to 10–100 μs without fault tolerance [90, 133]. When these frameworks are used in a chain, the stalls cause processing delays across the entire chain, similar to a *pipeline stall* in a

processor. As a result, we observed a ∼40% drop in throughput for a chain of five middleboxes as compared to a single middlebox (see Section 4.6.4).

In this chapter, we introduce a system called *fault tolerant chaining* (FTC) that provides fault tolerance to an entire chain. FTC is inspired by *chain replication* [163] to efficiently provide fault tolerance. At each middlebox, FTC collects state updates due to packet processing and piggybacks them onto the packet. As the packet passes through the chain, FTC replicates piggybacked state updates in servers hosting middleboxes. This allows each server hosting a middlebox to act as a replica for its predecessor middleboxes. If a middlebox fails, FTC can recover the lost state from its successor servers. For middleboxes at the end of the chain, FTC transfers and replicates their state updates in servers hosting middleboxes at the beginning of the chain. FTC does not need any dedicated replica servers.

We extend chain replication [163] to address challenges unique to a service function chain. Unlike the original protocol where all nodes run an identical process, FTC must support a chain comprised of different middleboxes processing traffic in the service function chain order. Accordingly, FTC allows all servers to process traffic and replicate state. Moreover, FTC's failure recovery instantiates a new middlebox at the failure position to maintain the service function chain order, rather than the traditional protocol that appends a new node at the end of a chain.

Furthermore, FTC improves the usability and performance of multicore middleboxes. We introduce *packet transactions* to provide a simple programming model to develop multithreaded middleboxes that can effectively make use of multiple cores. Concurrent state updates to middlebox state result in non-deterministic behavior that is hard to restore. A transactional model for state updates allows serializing concurrent state accesses that simplifies reasoning about both middlebox and FTC correctness. The state of the art [146] relies on complex static analysis that supports unmodified applications, but can have worse performance when its analysis falls short.

FTC also tracks dependencies among transactions using *data dependency vectors* that define a partial ordering of transactions. The partial ordering allows a replica to concurrently apply state updates from non-dependent transactions to improve replication performance. This approach has two major benefits compared to *thread-based* approaches that allow concurrent state replication by replaying the operations of threads [146]. First, FTC can support *vertical scaling* by replacing a running middlebox with a new instance with more CPU cores or failing over to a server with fewer CPU cores when resources are scarce during a major outage. Second, it enables a middlebox and its replicas to run with a different number of threads.

FTC is implemented on Click [87] and uses the ONOS SDN controller [15]. We compare its performance with the state of the art in [146]. Our results for a chain of two to five middleboxes show that FTC improves the throughput of the state of art [146] by 2× to 3.5× with lower latency per middlebox.

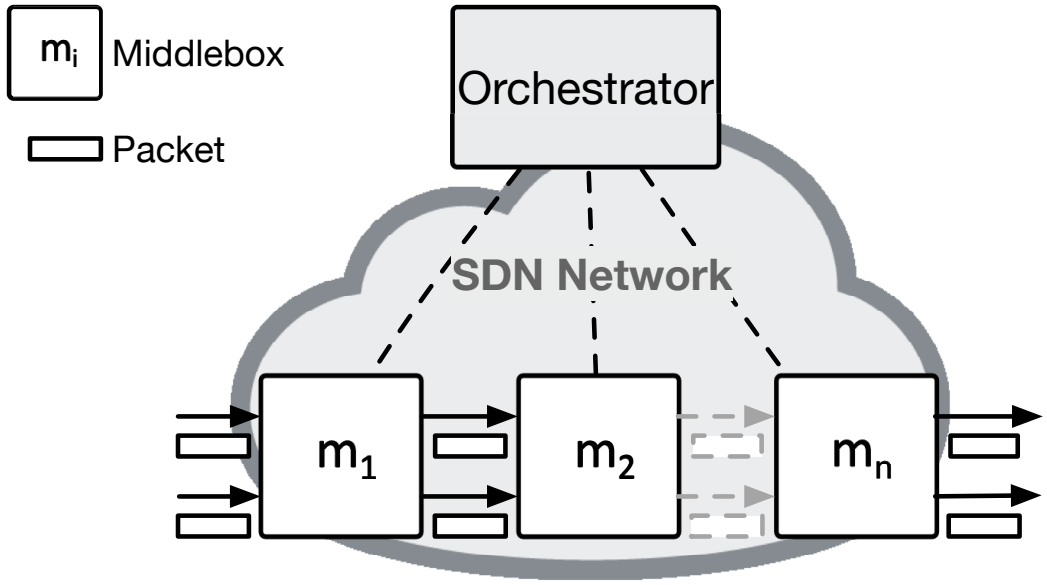


Figure 4.1: Service function chain model in NFV

4.1 Background

In an NFV environment, as shown in Figure 4.1, an orchestrator manages and steers traffic through a chain of middleboxes. Each middlebox runs multiple threads and is equipped with a multi-queue network interface card (NIC) [37, 124, 142]. A thread receives packets from a NIC's input queue and sends packets to a NIC's output queue. Figure 4.1 shows two threaded middleboxes processing two traffic flows.

Stateful middleboxes keep dynamic state for packets that they process [67, 151]. For instance, a stateful firewall filters packets based on statistics that it collects for network flows [10], and a network address translator (NAT) maps internal and external addresses using a flow table [70, 154].

Middlebox state can be *partitionable* or *shared* [10, 52, 58, 133]. Partitionable state variables describe the state of a single traffic flow (e.g., MTU size and timeouts in stateful firewalls [10, 52]) and are only accessed by a single middlebox thread. Shared state variables are for a collection of flows, and multiple middlebox threads query and update them (e.g., port-counts in an intrusion detection system).

A stateful middlebox is subject to both hardware and software failures that can cause the loss of its state [128, 146]. The root causes of these failures are for example CPU overloads, bit corruptions, cable problems, software bugs, and server failures due to maintenance operations and

power failures [55, 128]. We model these failures as *fail-stop* in which failures are detectable, and failed components are not restored.

4.1.1 Challenges

To recover from a middlebox failure, traffic must be rerouted to a redundant middlebox where the state of the failed middlebox is restored. State replication has two challenges that affect middlebox performance.

First, in a multithreaded middlebox, the order in which interleaving threads access shared state is non-deterministic. Parallel updates can lead to observable states that are hard-to-restore. The difficulty in achieving high performance multithreaded middleboxes is how we capture this state for recovery. One approach to accommodate non-determinism is to log any state read and write, which allows restoring any observable state from the logs [146]. However, this complicates the failure recovery procedure because of record/replay, and leads to high performance overheads during normal operation.

Second, to tolerate f failures, a packet is released *only* when at least $f+1$ replicas acknowledge that state updates due to processing of this packet are replicated. In addition to increasing latency, synchronous replication reduces throughput since expensive coordinations between packet processing and state replication are required for consistency (e.g., pausing packet processing until replication is acknowledged [81, 85, 90, 133]). The overhead of this synchrony for a middlebox depends on where its replicas are located, and how state updates are transferred to these locations. For a solution designed for individual middleboxes, the overheads can accumulate for each middlebox of a chain.

4.1.2 Limitations of Existing Approaches

Existing middlebox frameworks provide fault tolerance for individual middleboxes. Using these frameworks for a chain whose middleboxes are deployed over multiple servers significantly impacts the chain's performance. These frameworks use one of two approaches.

A class of frameworks take snapshots of middlebox state for state replication [90, 133, 146]. While taking snapshot, middlebox operations are stalled for consistency. These frameworks take snapshots at different rates. They take snapshots per packet or packet-batch introducing 400 μs to 8–9 ms of per packet latency overhead [90, 133]. Periodic snapshots (e.g., at every 20–200 ms intervals) can cause periodic latency spikes up to 6 ms [146]. We measure that per

middlebox snapshots cause 40% throughput drop going from a single middlebox to a chain of five middleboxes (see Section 4.6.4).

Other frameworks [81, 85] redesign middleboxes to separate and push state into a fault tolerant backend data store. This separation incurs high performance penalties. Accessing state takes at least a round trip delay. Moreover, a middlebox can release a packet only when it receives an acknowledgement from the data store that relevant state updates are replicated. Due to such overheads, the middlebox throughput can drop by ∼60% [81] and reduce to 0.5 Gbps (for packets with 1434 B median size) [85].

4.2 System Design Overview

The limitations of existing work lead us to design fault tolerant chaining (FTC); a new approach that replicates state along the chain to provide fault tolerance.

4.2.1 Requirements

We design FTC to provide fault tolerance for a wide variety of middleboxes. FTC adheres to four requirements:

Correct recovery: FTC ensures that the middlebox behavior after a failure recovery is consistent with the behavior prior to the failure [157]. To tolerate f failures, a packet can only be released outside of a chain once all necessary information needed to reconstruct the internal state of all middleboxes is replicated to $f + 1$ servers.

Low overhead and fast failure recovery: Fault tolerance for a chain must come with low overhead. A chain processes a high traffic volume and middlebox state can be modified very frequently. At each middlebox of a chain, latency should be within 10 to 100 μs [146], and the fault tolerance mechanism must support accessing variables 100 k to 1 M times per second [146]. Recovery time must be short enough to prevent application outages. For instance, highly available services timeout in just a few seconds [25].

Resource efficiency: Finally, the fault tolerance solution should be resource efficient. To isolate the effect of possible failures, replicas of a middlebox must be deployed on separate physical

servers. We are interested in a system that dedicates the fewest servers to achieve a fixed replication factor.

4.2.2 Design Choices

We model packet processing as a transaction. FTC carefully collects updated values of state variables modified during a packet transaction and appends them to the packet. As the packet passes through the chain, FTC replicates piggybacked state updates in servers hosting the middleboxes.

Transactional packet processing: To accommodate non-determinism due to concurrency, we model the processing of a packet as a transaction, where concurrent accesses to shared state are serialized to ensure that consistent state is captured and replicated. In other systems, the interleaved order of lock acquisitions and state variable updates between threads is non-deterministic, yet externally observable. Capturing and replaying this order is complex and incurs high performance overheads [146]. FTC uses transactional packet processing to avoid the complexity and overhead.

This model is easily adaptable to hybrid transactional memory, where we can take advantage of the hardware support for transactions [35]. This allows FTC to use modern hardware transactional memory for better performance, when the hardware is present.

We also observe that this model does not reduce concurrency in popular middleboxes. First, these middleboxes already serialize access to state variables for correctness. For instance, a load balancer and a NAT ensure *connection persistence* (i.e., a connection is always directed to a unique destination) while accessing a shared flow table [21, 154]. Concurrent threads in these middleboxes must coordinate to provide this property.

Moreover, most middleboxes share only a few state variables [81, 85]. Kablan et al. surveyed five middleboxes for their access patterns to state [81]. These middleboxes mostly perform only one or two read/write operations per packet. The behavior of these middleboxes allow packet transactions to run concurrently most of the time.

In-chain replication: Consensus-based state replication [94, 117] requires $2f + 1$ replicas for each middlebox to reliably detect and recover from f failures. A high-availability cluster approach requires $f + 1$ replicas as it relies on a fault tolerant coordinator for failure detection. For a chain of n middleboxes, these schemes need $n \times (2f + 1)$ and $n \times (f + 1)$ replicas. Replicas are placed on separate servers, and a *naïve* placement requires the same number of servers.

FTC observes that packets already flow through a chain; each server hosting a middlebox of the chain serves as a replica for the other middleboxes. Instead of allocating dedicated replicas,

FTC replicates the middleboxes state across the chain. In this way, FTC tolerates f failures without the cost of dedicated replica servers.

State piggybacking: To replicate state modified by a packet, existing schemes send separate messages to replicas. In FTC, a packet carries its own state updates. State piggybacking is possible, as a small number of state variables [86] are modified with each packet. Since state updated during processing a packet is replicated in servers hosting the chain, relevant state is already transferred and replicated when the packet leaves the chain.

No checkpointing and no replay: FTC replicates state values at the granularity of packet transactions, rather than taking snapshots of state or replaying packet processing operations. During normal operation, FTC removes state updates that have been applied in all replicas to bound memory usage of replication. Furthermore, replicating the values of state variables allows for fast state recovery during failover.

Centralized orchestration: In our system, a central orchestrator manages the network and chains. The orchestrator deploys fault tolerant chains, reliably monitors them, detects their failures, and initiates failure recovery. The orchestrator functionality is provided by a fault tolerant SDN controller [15, 84, 122]. After deploying a chain, the orchestrator is not involved in normal chain operations to avoid becoming a performance bottleneck.

In the following sections, we first describe our protocol for a single middlebox in Section 4.3, then we extend this protocol for a chain of middleboxes in Section 4.4.

4.3 FTC for a Single Middlebox

In this section, we present our protocol for a single middlebox. We first describe our protocol with a single threaded middlebox where state is replicated by single threaded replicas. We extend our protocol to support multithreaded middleboxes and multithreaded replication in Section 4.3.2 and Section 4.3.3.

4.3.1 Middlebox State Replication

We adapt the *chain replication* protocol [163] for middlebox state replication. For reliable state transmission between servers, FTC uses *sequence numbers*, similar to TCP, to handle out-of-order

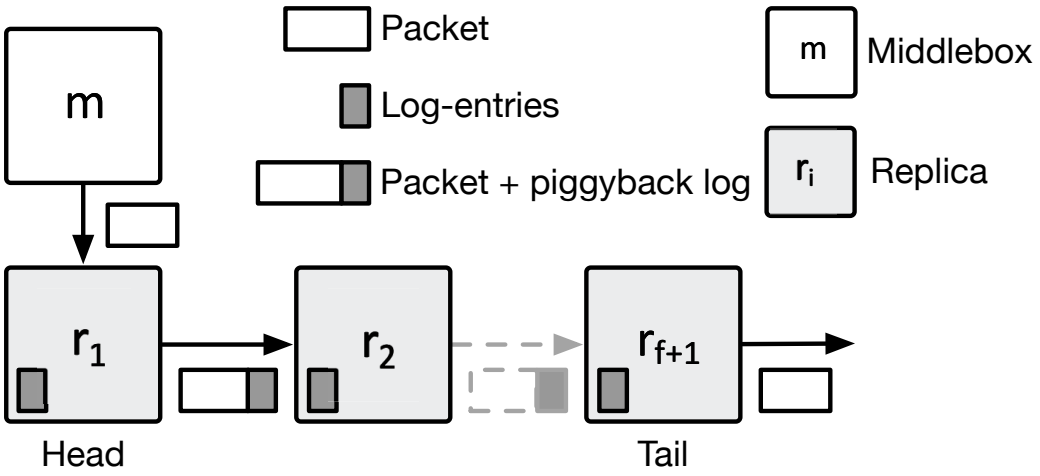


Figure 4.2: Normal operation for a single middlebox

deliveries and packet drops within the network.

Figure 4.2 shows our protocol for providing fault tolerance for a middlebox. FTC replicates the middlebox state in $f + 1$ replicas during normal middlebox operations. Replicas $r_1, \ldots, r_{f+1}$ form the *replication group* for middlebox m where r_1 and r_{f+1} are called the *head* and *tail* replicas. Each replica is placed on a separate server whose failure is isolated. With state replicated in $f + 1$ replicas, the state remains available even if f replicas fail.

The head is co-located with the middlebox in the same server. The middlebox state is separated from the middlebox logic and is stored in the head's *state store*. The head provides a state management API for the middlebox to read and write state during packet processing.

Normal operation of protocol: As shown in Figure 4.2, the middlebox processes a packet, and the head constructs and appends a *piggyback log* to the packet. The piggyback log contains a sequence number and a list of state updates during packet processing. As the packet traverses the chain, each subsequent replica replicates the piggyback log and applies the state updates to its state store. After replication, the tail strips the piggyback log and releases the packet.

The head tracks middlebox updates to state using a monotonically increasing sequence number. After a middlebox finishes processing a packet, the head increments its sequence number only if state was modified during packet processing. The head appends the state updates (i.e., state variables modified in processing the packet and their updated values) and sequence number to the packet as a piggyback log. If no state was updated, the head adds a no-op piggyback log. The

head then forwards the packet to the next replica.

Each replica continuously receives packets with piggyback logs. If a packet is lost, a replica requests its predecessor to re-transmit the piggyback log with the lost sequence number. A replica keeps the largest sequence number that it has received in order (i.e., the replica has already received all piggyback logs with preceding sequence numbers). Once all prior piggyback logs are received, the replica applies the piggyback log to its local state store and forwards the packet to the next replica.

The tail replicates state updates, strips the piggyback log from the packet, and releases the packet to its destination. Subsequently, the tail periodically disseminates its largest sequence number to the head. The sequence number is propagated to all replicas so they can prune their piggyback logs up to this sequence number.

Correctness: Each replica replicates the per-packet state updates in order. As a result, when a replica forwards a packet, it has replicated all preceding piggyback logs. Packets also pass through the replication group in order. When a packet reaches a replica, prior replicas have replicated the state updates carried by this packet. Thus, when the tail releases a packet, the packet has already traversed the entire replication group. The replication group has $f + 1$ replicas allowing FTC to tolerate f failures.

Failure recovery: FTC relies on a fault tolerant orchestrator to reliably detect failures. Upon failure detection, the replication group is repaired in three steps: adding a new replica, recovering the lost state from an alive replica, and steering traffic through the new replica.

In the event of a head failure, the orchestrator instantiates a new middlebox instance and replica, as they reside on the same server. The orchestrator also informs the new replica about other alive replicas. If the new replica fails, the orchestrator restarts the recovery procedure.

Selecting a replica as the source for state recovery depends on how state updates propagate through the chain. We can reason about this using the *log propagation invariant*: for each replica except the tail, its successor replica has the same or prior state, since piggyback logs propagate in order through the chain.

If the head fails, the new replica retrieves the state store, piggyback logs, and sequence number from the immediate successor to the head. If other replicas fail, the new replica fetches the state from the immediate predecessor.

To ensure that the log propagation invariant holds during recovery, the replica that is the source for state recovery discards any out-of-order packets that have not been applied to its state

store and will no longer admit packets in flight. If the contacted replica fails during recovery, the orchestrator detects this failure and re-initializes the new replica with the new set of alive replicas.

Finally, the orchestrator updates routing rules in the network to steer traffic through the new replica. If multiple replicas have failed, the orchestrator waits until all new replicas acknowledge that they have successfully recovered the state. Then, the orchestrator updates the necessary routing rules from the tail to the head.

4.3.2 Concurrent Packet Processing

To achieve higher performance, we augment our protocol to support multithreaded packet processing and state replication in the middlebox and the head. Other replicas are still single threaded. Later in Section 4.3.3, we will support multithreaded replications in other replicas.

In concurrent packet processing, multiple packets are processed in interleaving threads. The threads can access the same state variables in parallel. To accommodate this parallelism, FTC must consistently track parallel state updates. We introduce *transactional packet processing* that effectively serializes packet processing. This model supports concurrency if packet transactions access disjoint subsets of state.

Transactional Packet Processing: In concurrent packet processing, the effects on state variables must be serializable. Further, state updates must be applied to replicas in the same order so that the system can be restored to a consistent state during failover. To support this requirement, replay based replication systems, such as FTMB [146], track all state accesses, including state reads, which can be challenging to perform efficiently.

In transactional packet processing, state reads and writes by a packet transaction have no impact on another concurrently processed packet. This isolation allows us to only keep track of the relative order between transactions, without needing to track all state variable dependencies.

We realize this model by implementing a *software transactional memory* (STM) API for middleboxes. When a packet arrives, the runtime starts a new packet transaction in which multiple reads and writes can be performed. Our STM API uses *fine grained strict two phase locking* (similar to [36]) to provide serializability. Our API uses a *wound-wait scheme* that aborts transaction to prevent possible deadlocks if a lock ordering is not known in advance. An aborted transaction is immediately re-executed. The transaction completes when the middlebox releases the packet.

Using two phase locking, the head runtime acquires necessary locks during a packet transaction. We simplify lock management using *state space partitioning*, by using the hash of state variable

keys to map keys to partitions, each with its own lock. The state partitioning is consistent across all replicas, and to reduce contention, the number of partitions is selected to exceed the maximum number of CPU cores.

At the end of a transaction, the head atomically increments its sequence number only if state was updated during this packet transaction. Then, the head constructs a piggyback log containing the state updates and the sequence number. After the transaction completes, the head appends the piggyback log to the packet and forwards the packet to the next replica.

Correctness: Due to *mutual exclusion*, when a packet transaction includes an updated state variable in a piggyback log, *no* other concurrent transaction has modified this variable, thus the included value is consistent with the final value of the packet transaction. The head's sequence number maps this transaction to a *valid* serial order. Replicated values are consistent with the head, because replicas apply state updates of the transaction in the sequence number order.

4.3.3 Concurrent State Replication

Up to now FTC provides concurrent packet processing but does not support concurrent replication. The head uses a single sequence number to determine a total order of transactions that modify state partitions. This total ordering eliminates multithreaded replication at successor replicas.

To address the possible replication bottleneck, we introduce *data dependency vectors* to support concurrent state replication. Data dependency tracking is inspired by the *vector clocks* algorithm [48], but rather than tracking points in time when events happen for processes or threads, FTC tracks the points in time when packet transactions modify state partitions.

This approach provides more flexibility compared to tracking dependencies between threads and replaying their operations to replicate the state [146]. First, it easily supports vertical scaling as a running middlebox can be replaced with a new instance with different number of CPU cores. Second, a middlebox and its replicas can also run with different number of threads. The state-of-the-art [146] requires the same number of threads with a one-to-one mapping between a middlebox and its replicas.

Data dependency vectors: We use data dependency vectors to determine a partial order of transactions in the head. Each element of this vector is a sequence number associated to a state partition. A packet piggybacks this partial order to replicas enabling them to replicate transactions with more concurrency; a replica can apply and replicate a transaction in a different serial order that is still equivalent to the head.

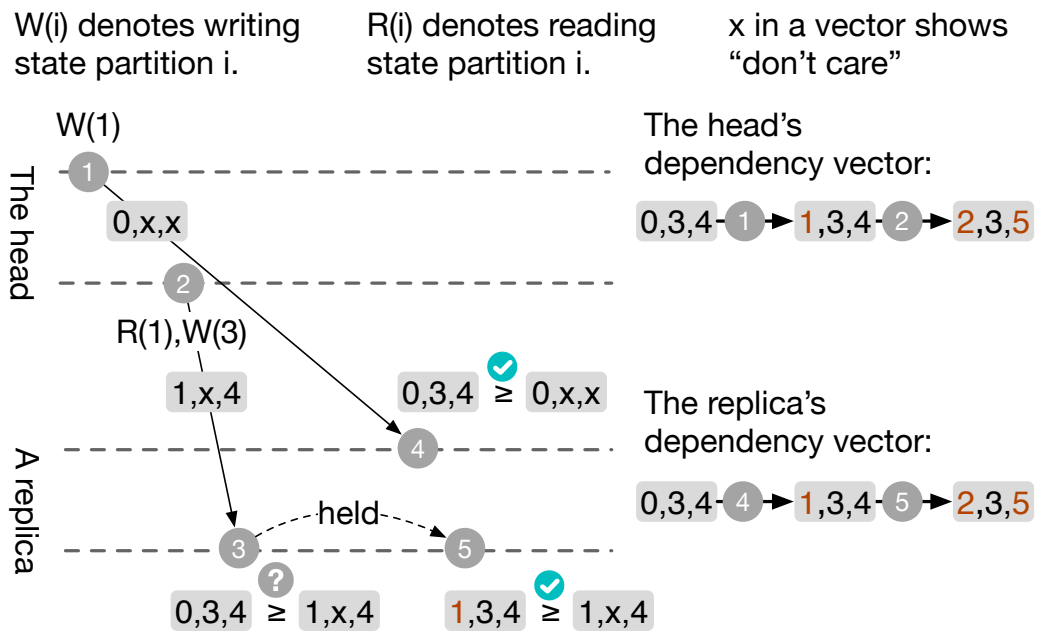


Figure 4.3: Data dependency vectors. The head and the replica run two threads and maintain a dependency vector for three state partitions.

The head keeps a data dependency vector and serializes parallel accesses to this vector using the same state partition locks from our transactional packet processing. The head maintains its dependency vector using the following rules. A read-only transaction does *not* change the vector. For other transactions, the head increments the sequence number of a state partition that received any read or write.

In a piggyback log, we replace the sequence number with a dependency vector that represents the effects of a transaction on state partitions. If the transaction does not access a state partition, the head uses a "*don't-care*" value for this partition in the piggyback log. The head obtains the sequence number of other partitions from the head's dependency vector before incrementing their sequence numbers.

Each successor replica keeps a dependency vector MAX that tracks the latest piggyback log that it has replicated in order, i.e., it has already received all piggyback logs prior to MAX. In case a packet is lost, a replica requests its predecessor to retransmit missing piggyback logs.

Upon receiving a packet, a replica compares the piggybacked dependency vector with its MAX. The replica ignores state partitions with "don't care" from this comparison. Once all prior piggyback logs have been received and applied, the replica applies and replicates the piggyback

log. For other state partitions, the replica increments their associated sequence numbers in MAX.

Example: Figure 4.3 shows an example of using data dependency vectors in the head and a successor replica with two threads. The head and the replica begin with the same dependency vector for three state partitions. First, the head performs a packet transaction that writes to state partition 1 and increments the associated sequence number. The piggyback log belonging to this transaction contains "don't care" value for state partitions 2 and 3 (denoted by $\times$), since the transaction did not read or write these partitions. Second, the head performs another transaction and forwards the packet with a piggyback log.

Third, as shown the second packet arrives to the replica before the first packet. Since the piggybacked dependency vector is out of order, the replica holds the packet. Fourth, the first packet arrives. Since the piggybacked vector is in order, the replica applies the piggyback log and updates its local dependency vector accordingly. Fifth, by applying the piggyback log of the first packet, the replica now can apply the piggyback log of the held packet.

4.4 FTC for a Chain

In this section, we describe our protocol for a chain to enable every middlebox to replicate the chain's state while processing packets. To accomplish this, we extend the original chain replication protocol [163] during both normal operation and failure recovery.

A chain consists of different middlebox applications. Thus, FTC must allow different middleboxes to run across the chain, while the original chain replication protocol supports running an identical process across the nodes. FTC's failure recovery instantiates a new middlebox at the failure position to maintain the chain order, while the traditional protocol appends a new node at the end of a chain.

Figure 4.4 shows our protocol for a chain of n middleboxes. Our protocol can be thought of as running n instances (per middlebox) of the protocol developed earlier in Section 4.3. FTC places a replica per each middlebox. Replicas form n replication groups, each of which provides fault tolerance for a single middlebox.

Viewing a chain as a *logical ring*, the replication group of a middlebox consists of a replica and its f succeeding replicas. Instead of being dedicated to a single middlebox, a replica is shared among $f+1$ middleboxes and maintains a state store for each of them. Among these middleboxes, a replica is the head of one replication group and the tail of another replication group. For instance in Figure 4.4, if $f = 1$ then the replica r_1 is in the replication groups of middleboxes m_1 and m_n,

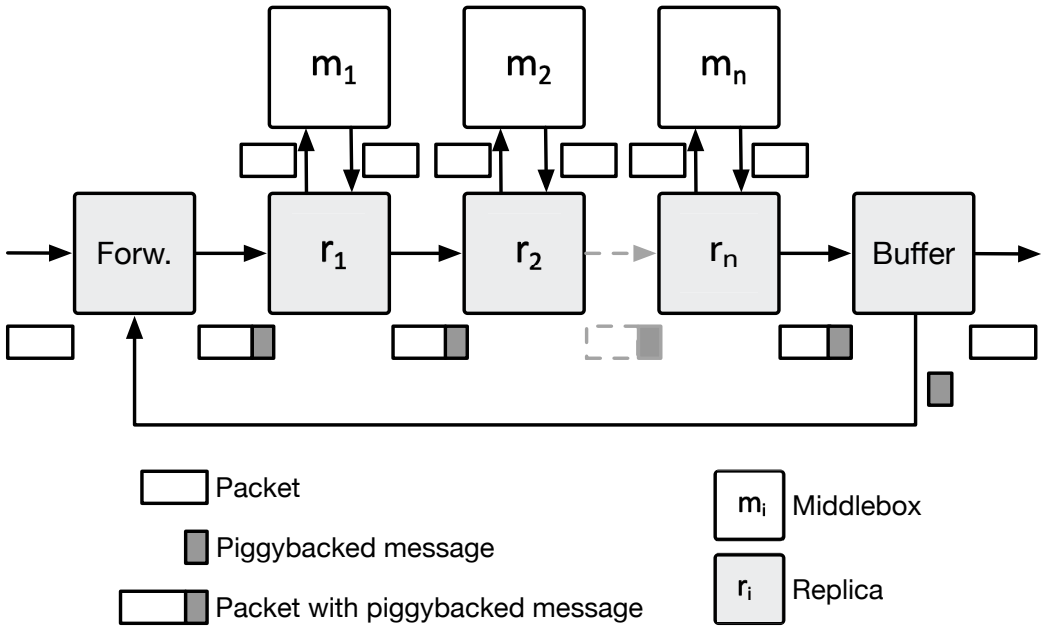


Figure 4.4: Normal operation: chain of n middleboxes.

and r_2 is in the replication groups of m_1 and m_2. Subsequently, the replicas r_n and r_1 are the head and the tail of middlebox m_n.

FTC adds two additional elements, the *forwarder* and *buffer* at the ingress and egress of a chain. The forwarder and buffer are also multithreaded, and are collocated with the first and last middleboxes. The buffer holds a packet until the state updates associated with all middleboxes of the chain have been replicated. The buffer also forwards state updates to the forwarder for middleboxes with replicas at the beginning of the chain. The forwarder adds state updates from the buffer to incoming packets before forwarding the packets to the first middlebox.

4.4.1 Normal Operation of Protocol

Figure 4.4 shows the normal operation of our protocol. The forwarder receives incoming packets from the outside world and *piggyback messages* from the buffer. A piggyback message contains middlebox state updates. As the packet passes through the chain, a replica detaches and replicates the relevant parts of the piggyback message and applies associated state updates to its state stores. A replica r_i tracks the state updates of a middlebox m_i and updates the piggyback message to include these state updates. Replicas at the beginning of the chain replicate for middleboxes

at the end of the chain. The buffer withholds the packet from release until the state updates of middleboxes at the end of the chain are replicated. The buffer transfers the piggyback message to the forwarder that adds it to incoming packets for state replication.

The forwarder receives incoming packets from outside world and piggyback messages from the buffer. A piggyback message consists of a list of piggyback logs and a list of *commit vectors*. The tail of each replication group appends a commit vector to announce the latest state updates that have been replicated $f + 1$ times for the corresponding middlebox.

Each replica constantly receives packets with piggyback messages. A replica detaches and processes a piggyback message before the packet transaction. As mentioned before, each replica is in the replication group of f preceding middleboxes. For each of them, the replica maintains a dependency vector MAX to track the latest piggyback log that it has replicated in order. The replica processes a relevant piggyback log from the piggyback message as described in Section 4.3.3. Once all prior piggyback logs are applied, the replica replicates the piggyback log, applies state updates to the associated state store, and updates the associated dependency vector MAX.

Once the middlebox finishes the packet transaction, the replica updates and reattaches the piggyback message to the packet, then forwards the packet. For the replication group where the replica is the head, it adds a piggyback log containing the state updates of processing the packet. If the replica is a tail in the replication group of a middlebox m, it removes the piggyback log belonging to middlebox m to reduce the size of the piggyback message. The reason is that a tail replicates the state updates of m for $f + 1$-th time. Moreover, it attaches its dependency vector MAX of middlebox m as a commit vector. Later by reading this commit vector, the buffer can safely release held packets. Successor replicas also use this commit vector to prune obsolete piggyback logs.

To correctly release a packet, the buffer requires that the state updates of this packet are replicated, specifically for each middlebox with a preceding tail in the chain. The buffer withholds a packet from release until an upcoming packet piggybacks commit vectors that confirm meeting this requirement. Upon receiving an upcoming packet, the buffer processes the piggybacked commit vectors to release packets held in the memory.

Specifically, let m be a middlebox with a preceding tail, and V_2 be the end of updated range from a piggyback log of a held packet belonging to m. Once the commit vector of each m from an upcoming packet shows that all state updates prior to and including V_2 have been replicated, the buffer releases the held packet and frees its memory.

Other considerations: There may be time periods that a chain receives no incoming packets. In such cases, the state is not propagated through the chain, and the buffer does not release packets.

To resolve this problem, the forwarder keeps a timer to receive incoming packets. Upon the timeout, the forwarder sends a *propagating packet* carrying a piggyback message it has received from the buffer. Replicas do not forward a propagating packet to middleboxes. They process and update the piggyback message as described before and forward the packet along the chain. The buffer processes the piggyback message to release held packets.

Some middlebox in a chain can filter packets (e.g., a firewall may block certain traffic), and consequently the piggybacked state is not passed on. For such a middlebox, its head generates a propagating packet to carry the piggyback message of a filtered packet.

Finally, if the chain length is less than $f + 1$, we extend the chain by adding more replicas prior to the buffer. These replicas only process and update piggyback messages.

4.4.2 Failure Recovery

Handling the failure of the forwarder or the buffer is straightforward. They contain only soft state, and spawning a new forwarder or a new buffer restores the chain.

The failure of a middlebox and its head replica is not isolated, since they reside on the same server. If a replica fails, FTC repairs $f + 1$ replication groups as each replica replicates for $f + 1$ middleboxes. The recovery involves three steps: spawning a new replica and a new middlebox, recovering the lost state from other alive replicas, and steering traffic through the new replica.

After spawning a new replica, the orchestrator informs it about the list of replication groups in which the failed replica was a member. For each of these replication group, the new replica runs an independent state recovery procedure as follows. If the failed replica was the head of a replication group, the new replica retrieves the state store and the dependency vector MAX from the immediate successor in this replication group. The new replica restores the dependency matrix of the failed head by setting each of its row to the retrieved MAX. For other replication groups, the new replica fetches the state from the immediate predecessors in these replication groups.

Once the state is recovered, the new replica notifies the orchestrator to update routing rules to steer traffic through the new replica. For simultaneous failures, the orchestrator waits until all new replicas confirm that they have finished their state recovery procedures before updating routing rules.

4.5 Implementation

FTC builds on ONOS SDN controller [15] and Click [87]. The forwarder and buffer are implemented as Click elements.

A replica consists of *control* and *data plane* modules. The control module is a daemon that communicates with the orchestrator and the control modules in other replicas. In failover, the control module spawn a thread to fetch state per each replication group. Using a reliable TCP connection, the thread sends a fetch request to the appropriate member in the replication group and waits to receive state.

The data plane module processes piggyback messages, sends and receives packets to and from a middlebox, constructs piggyback messages, and forwards packets to a next element in the chain (the data-plane module of the next replica or the buffer).

FTC appends the piggyback logs to the end of a packet, and inserts an IP option to notify our runtime that a packet has a piggyback message. As a piggyback message is appended at the end of a packet, its process and construction can be performed in-place, and there is no need to actually strip and reattach it. Before sending a packet to the middlebox, the relevant header fields (e.g., the total length in IP header) is updated to not account for the piggyback message. Before forwarding the packet to next replica, the header is updated back to reconsider the piggyback message. For middleboxes that may extend the packet, the data plane module operates on the copy of a piggyback message.

4.6 Evaluation

We evaluate FTC in this section. We describe our setup and methodology in Section 4.6.1 and evaluate FTC's performance for middleboxes and chains in Section 4.6.3 and Section 4.6.4, respectively. Finally, we evaluate failure recovery in Section 4.6.5.

4.6.1 Experimental Setup and Methodology

We compare FTC with `NF`, a non fault-tolerant baseline system, and `FTMB`, our implementation of [146]. Our `FTMB` implementation is a performance upper bound of the original work that performs the logging operations described in [146] but does not take snapshots. Following the original prototype, `FTMB` dedicates a server in which a middlebox *master* (M) runs, and another server where the fault tolerant components *input logger* (IL) and *output logger* (OL) execute.

Packets go through IL, M, then OL. M tracks accesses to shared state using packet access logs (PALs) and transmits them to OL. In the original prototype, no data packet is released until all corresponding dropped PALs are retransmitted. Our prototype assumes that PALs are delivered on the first attempt, and packets are released immediately afterwards. Further, OL maintains only the last PAL.

We used two environments. The first is a local cluster of 12 servers. Each server has an 8-core Intel Xeon CPU D-1540 clocked at 2.0 Ghz, 64 GiB of memory, and two NICs, a 40 Gbps Mellanox ConnectX-3 MT27500 and a 10 Gbps Intel Ethernet Connection X552/X557. The servers run Ubuntu 14.04 with kernel 4.4 and are connected to 10 and 40 Gbps top-of-rack switches. We use MoonGen [42] and pktgen [162] to generate traffic and measure latency and throughput, respectively. Traffic from the generator server, passed in the 40 Gbps links, is sent through middleboxes and back to the generator. FTC uses a 10 Gbps link to disseminate state changes from `Buffer` to `Forwarder`.

The second environment is a distributed Cloud comprised of several core and edge data-centers deployed across a continent[1]. We use virtual machines (VMs) with 4 virtual processor cores and 8 GiB memory running Ubuntu 14.04 with Kernel 4.4. We use the published ONOS docker container [115] to control a virtual network of OVS switches [114] connecting these VMs. We follow the *multiple interleaved trials methodology* [1] to reduce the variability that come from performing experiments on a shared infrastructure.

We use the middleboxes and chains shown in Table 4.1. The middleboxes are implemented in Click [87] (read and write accesses to shared state variables are explicitly specified). `MazuNAT` is an implementation of the core parts of a commercial NAT [110], and `SimpleNAT` provides basic NAT functionalities. They represent read-heavy middleboxes with a moderate write load on the shared state (i.e., a flow table mapping private and public addresses). `Monitor` is a read/write heavy middlebox that counts the number of packets in a flow or across flows. It takes a *sharing level* parameter that specifies the number of threads sharing the same state variable. For example, no state is shared for the sharing level 1, and 8 `Monitor`'s threads share the same state variable for sharing level 8. `Gen` represents a write-heavy middlebox that takes a state size parameter, which allows us to test the impact of a middlebox's state size on performance. `Gen` is a write-heavy middlebox that takes a size parameter. `Gen` updates a state variable with this size per packet allowing us to test the impact of a middlebox's state size on performance. `Firewall` is a stateless firewall. Our experiments also test three chains (`Ch-`n, `Ch-Gen`, `Ch-Rec`) that are composed of combinations of these middleboxes.

For experiments in the first environment, we report latency and throughput. For a latency data-point, we report the average of hundreds of samples taken in a 10 second interval. For a

[1]The name of this Cloud is withheld to comply with the anonymity policy of the conference.

Table 4.1: Experimental middleboxes and chains

Middlebox	State reads	State writes
MazuNAT	Per packet	Per flow
SimpleNAT	Per packet	Per flow
Monitor	Per packet	Per packet
Gen	No	Per packet
Firewall	N/A	N/A

Chain	Middleboxes in chain
Ch-n	$\texttt{Monitor}_1 \rightarrow \cdots \rightarrow \texttt{Monitor}_n$
Ch-Gen	$\texttt{Gen}_1 \rightarrow \texttt{Gen}_2$
Ch-Rec	Firewall → Monitor → SimpleNAT

throughput data-point, we report the average of maximum throughput values measured every second in a 10-second interval. Unless shown, we do not report confidence intervals as they are negligible. Unless otherwise specified, the packet size in our experiments is 256 B, and $f = 1$.

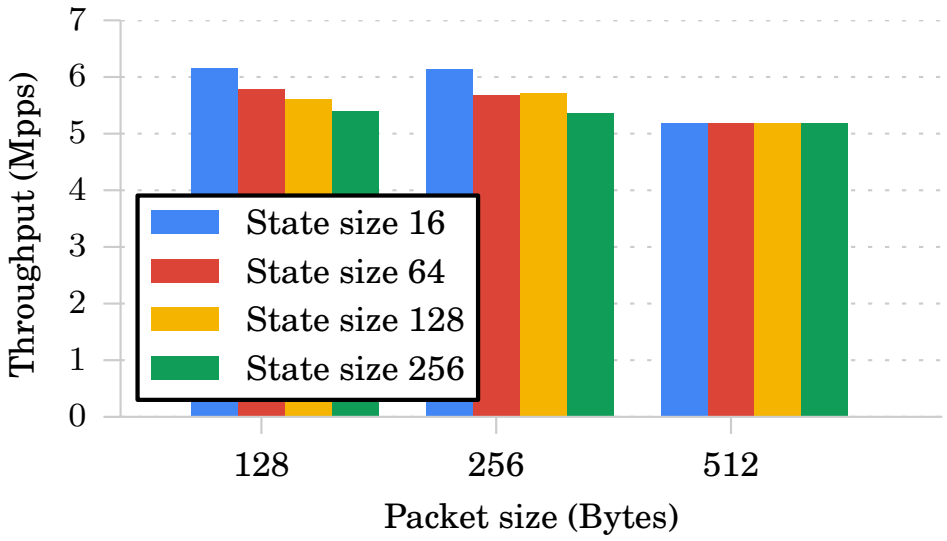


Figure 4.5: Throughput vs. state size

4.6.2 Micro-benchmark

We use a micro-benchmark to determine the impact of a state size on the performance of FTC. We measured the latency overhead for the middlebox Gen and the chain Ch-Gen. We observed

that under 2 Mpps for 512 B packets, varying the size of the generated state from 32-256 B has a negligible impact on latency for both `Gen` and `Ch-Gen` (the difference is less than 2 μs). Thus, we focus on the throughput overhead.

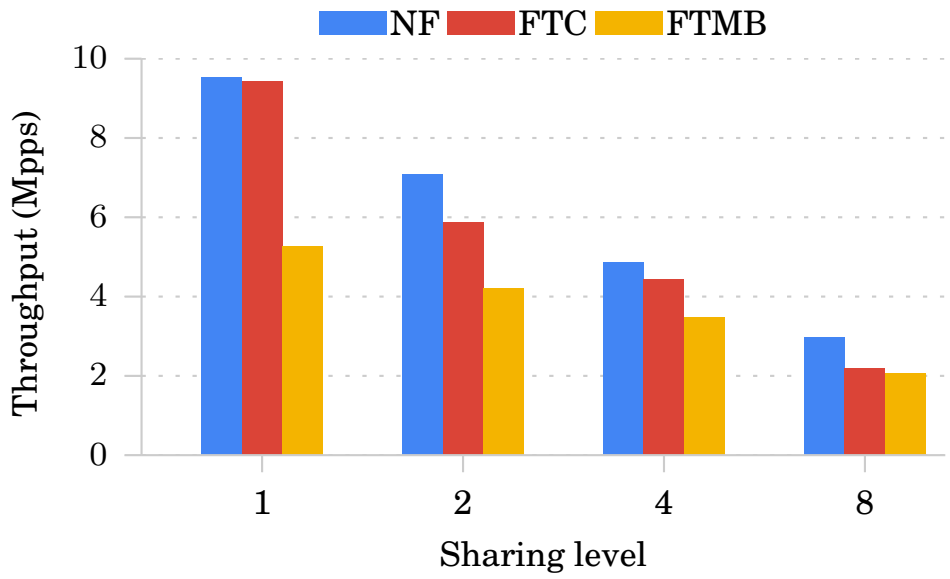


Figure 4.6: Throughput of `Monitor`

Throughput: Figure 4.5 shows the impact of state size generated by `Gen` on throughput. `Gen` runs a single thread. We vary the state size from 16 to 256 B and measure `Gen`'s throughput for packet sizes 128, 256, and 512 B. As expected, the size of piggyback messages impacts the throughput only if it is proportionally large compared to packet sizes. For 128 B packets, throughput drops by only 9% when `Gen` generates states that are 128 B in size or less. The throughput drops by less than 1% with 512 B packets and state up to 256 B in size. We expect popular middleboxes to generate state much smaller than some of our tested values. For instance, a load balancer and a NAT generate a record per traffic flow [21, 80, 154] that is roughly 32 B in size (2×12 B for the IPv4 headers in both directions and 8 B for the flow identifier).

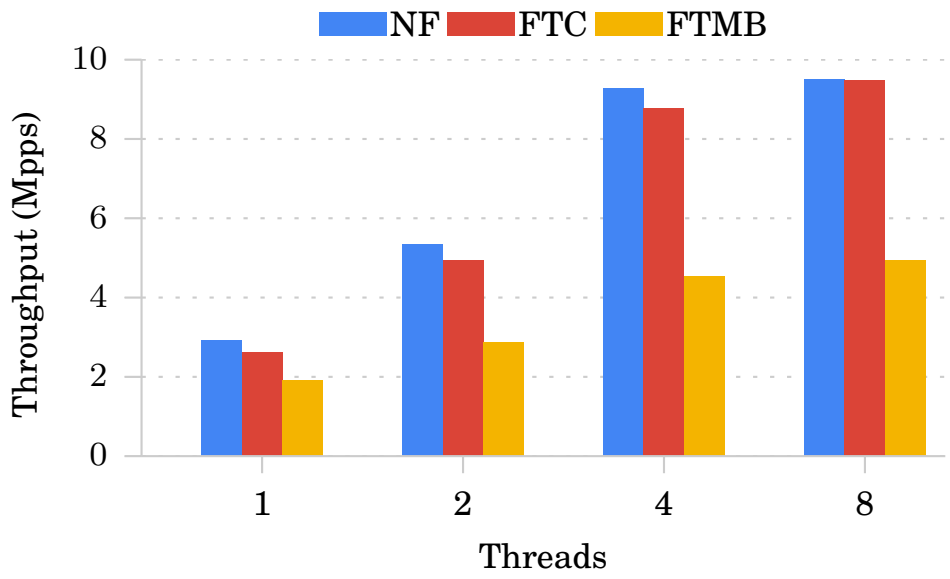


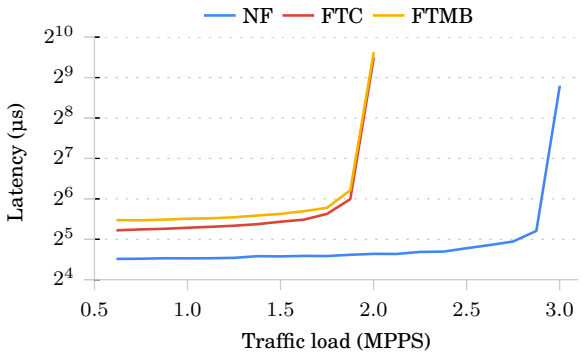
Figure 4.7: Throughput of MazuNAT

4.6.3 Fault Tolerant Middleboxes

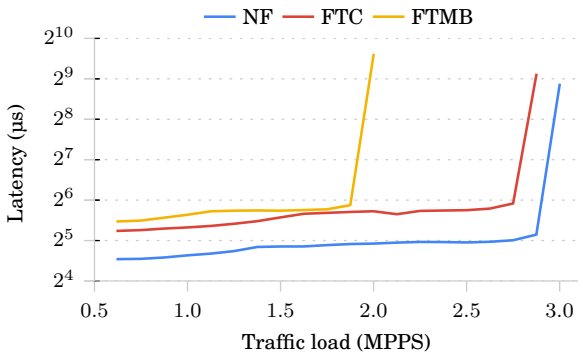
Throughput: Figures 4.6 and 4.7 show the maximum throughput of 2 middleboxes. In Figure 4.6, we configure Monitor to run with 8 threads and measure its throughput with different sharing levels. As the sharing level for Monitor increases, the throughput of all systems, including NF, drops due to the higher contention in reading and writing the shared state. For sharing levels of 8 and 2, FTC achieves a throughput that is 1.2-1.4 × that of FTMB's and incurs an overhead of 9-26% compared to NF. These overheads are expected since Monitor is a write-heavy middlebox, and the shared state is modified non-deterministically per packet. For sharing level 1, NF and FTC reach the NIC's packet processing capacity[2]. FTMB does not scale for sharing level 1, since for every data packet, a PAL is transmitted in a separate message, which limits FTMB's throughput to 5.26 Mpps.

For Figure 4.7, we evaluate the throughput of MazuNAT while varying the number of threads.

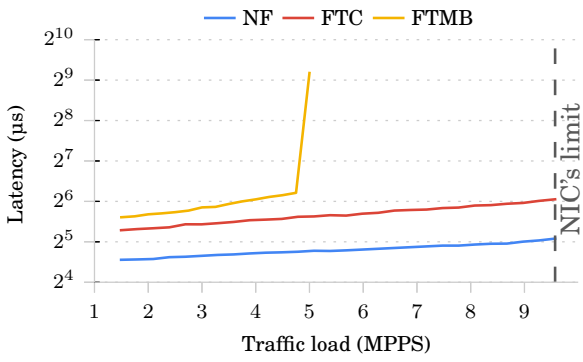
[2]Although the 40 GbE link is not saturated, our investigation showed that the bottleneck is the NIC's packet processing power. We measured that the Mellanox ConnectX-3 MT 27500, at the receiving side and working under the DPDK driver, at most can process 9.6-10.6 Mpps for varied packet sizes. Though we have not found any official document by Mellanox describing this limitation, similar behavior (at higher rates) has been reported for other vendors (see Sections 5.4 and 7.5 in [42] and Section 4.6 in [71]).



(a) `Monitor` – Sharing level 8



(b) `MazuNAT` – 1 thread



(c) `MazuNAT` – 8 threads

Figure 4.8: Latency of middleboxes

FTC's throughput is 1.37-1.94× that of FTMB's for 1-4 threads. Once a traffic flow is recorded in the NAT flow table, processing the next packets of this flow only requires reading the shared record (until the connection terminates or times out). The higher throughput compared for MazuNAT is because FTC does not replicate these reads, while FTMB logs them to provide fault tolerance [146]. We observe that FTC incurs 1-10% throughput overhead compared to NF. Part of this overhead is because FTC has to pay the cost of adding space to packets for possible state writes, even when state writes are not performed. The pattern of state reads and writes impacts FTC's throughput. Under moderate write workloads, FTC incurs 1-10% throughput overhead, while under write-heavy workloads, FTC's overhead remains less than 26%.

Latency: Figure 4.8 illustrates the latency of Monitor (8 threads with sharing level 8) and MazuNAT (two configurations, 1 thread and 8 threads) under different traffic loads. For both Monitor and MazuNAT, the latency remains under 0.7ms for all systems as the traffic load increases until the systems reach their respective saturation points. Past these points, packets start to be queued, and per-packet latency rapidly spikes.

As shown in Figure 4.8a, operating under sustainable loads, FTC and FTMB respectively add overhead within 14-25 μs and 22-31 μs to the per-packet latency, out of which 6-7 μs is due to the additional one-way network latency to forward the packet and state to the replica. For this write-heavy middlebox, FTC introduces a smaller latency overhead compared to FTMB.

Figure 4.8b shows that, when running MazuNAT with a single thread, FTC can sustain nearly the same traffic load as NF, and FTC and FTMB have similar latencies.

For 8 threads shown in Figure 4.8c, both FTC and NF reach the packet processing capacity of the NIC. The latency of FTC is largely independent of the number of threads, while FTMB experiences a latency increase of 24-43 μs when going from 1 to 8 threads.

4.6.4 Fault Tolerant Chains

In the following experiments, we evaluate the performance of FTC for a chain of middleboxes during normal operation. For a NF chain, each middlebox is deployed in a separate physical server. We use the same number of servers for FTC, while we dedicate 2× the number of servers to FTMB: A server for each middlebox (Master in FTMB) and a server for its replica (IL and OL in FTMB).

Chain length impact on throughput: Figure 4.9 shows the maximum traffic throughput passing in 4 chains (Ch-2 to Ch-5 listed in Table 4.1). Monitors in these chains run 8 threads with

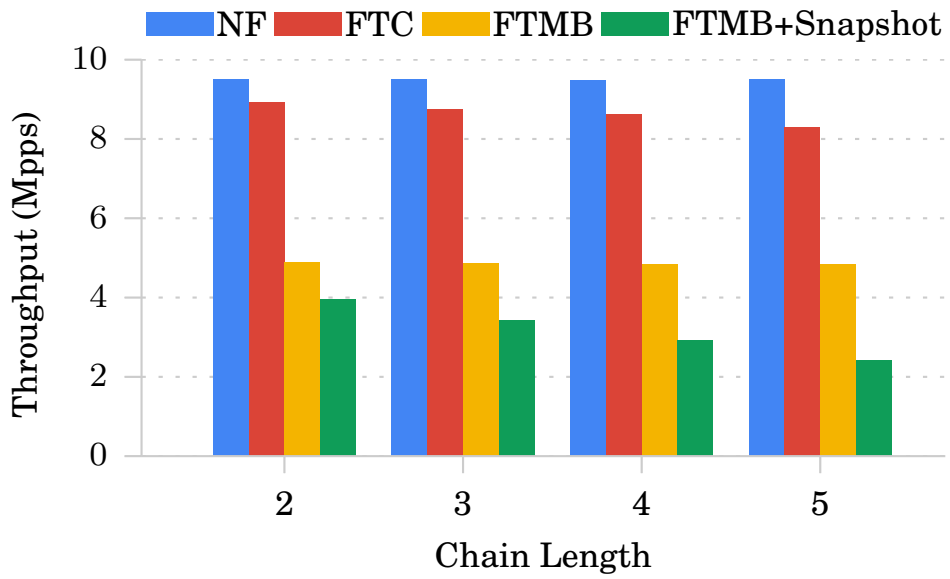


Figure 4.9: Throughput vs. chain length

sharing level 1. We also report for `FTMB+Snapshot` that is `FTMB` with snapshot simulation. To simulate the overhead of periodic snapshots, we add an artificial delay (6 ms) periodically (every 50 ms). We get these values from [146].

As shown in Figure 4.9, FTC's throughput is within 8.28-8.92 Mpps and 4.83-4.80 Mpps for `FTMB`. FTC imposes a 6-13% throughput overhead compared to `NF`. The throughput drop from increasing the chain length for FTC is within 2-7%, while that of `FTMB+Snapshot` is 13-39% (its throughput drops from 3.94 to 2.42 Mpps). This shows that throughput of FTC is largely independent of the chain length, while, for `FTMB+Snapshot`, periodic snapshots taken at all middleboxes significantly reduce the throughput. No packet is processed during a snapshot. Packet queues get full at early snapshots and remain full afterwards because the incoming traffic load is at the same rate. More snapshots are taken in a longer chain. Non-overlapping (in time) snapshots cause shorter service time at each period and consequently higher throughput drops. An optimum scheduling to synchronize snapshots across the chain can reduce this overhead; however, This is not trivial [22].

Chain length impact on latency: We use the same experimental settings as the previous experiment, except we run single threaded `Monitor`s due to a limitation of the traffic generator.

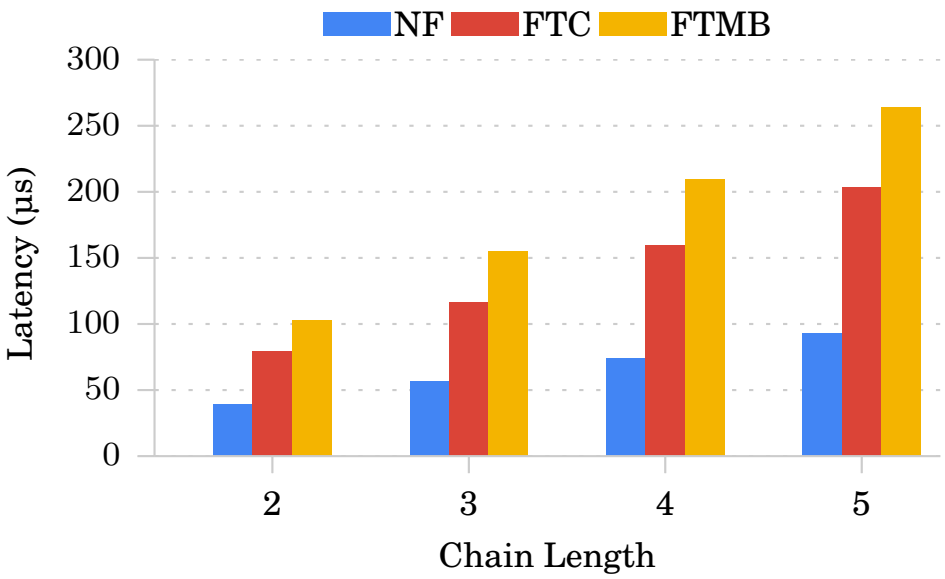


Figure 4.10: Latency vs. chain length

The latter is not able to measure the latency of the chain beyond size 2 composed of multi-threaded middleboxes. We resort to experimenting with single threaded `Monitors` under the load of 2 Mpps, a sustainable load by all systems.

As shown in Figure 4.10, FTC's overhead compared to `NF` is within 39-104 μs for `Ch-2` to `Ch-5`, translating to roughly 20 μs latency per middlebox. The overhead of `FTMB` is within 64-171 μs, approximately 35 μs latency overhead per middlebox in the chain.

As shown in Figure 4.11, the tail latency of individual packets passing through `Ch-3` is only moderately higher than the minimum latency. FTC incurs 16.5-20.6 μs per middlebox latency which is respectively three and two orders of magnitudes less than Pico's and REINFORCE's, and is around $2/3$ of `FTMB`'s. In-chain replication eliminates the communication overhead with remote replicas. Doing so also does not cause latency spikes unlike snapshot-based systems. In FTC, packets experience constant latency, while the original FTMB reports up to 6 ms latency spikes at periodic checkpoints (e.g., at every 50 ms intervals) [146].

Replication factor impact on performance: For replication factors of 2-5 (i.e., tolerating 1-4 failures), Figure 4.12 shows FTC's performance for `Ch-5` in two settings where `Monitors` run

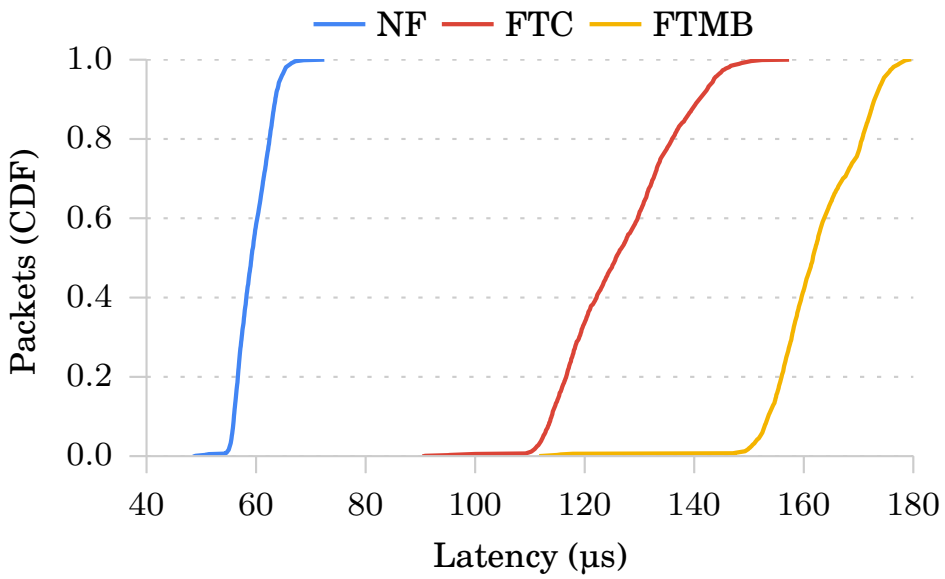


Figure 4.11: `Ch-3` per packet latency

with 1 or 8 threads. We report the throughput of 8 threaded `Monitor`, while only report the latency of 1 threaded `Monitor` due to a limitation of our test harness.

To tolerate $2.5\times$ failures, FTC incurs only 3% throughput overhead as its throughput decreases to 8.06 Mpps. The latency overhead is also insignificant as latency only increases by 8 μs. By exploiting the chain structure, FTC can tolerate a higher number of failures without sacrificing performance. However, the replication factor cannot be arbitrarily large as encompassing the resulting large piggyback messages inside packets becomes impractical.

4.6.5 Failure Recovery

Recall from Section 4.5, failure recovery is performed in three steps that incure initialization, state recovery, and rerouting delays. To evaluate FTC during recovery, we measure the recovery time of `Ch-Rec` (see Table 4.1). Each middlebox is placed in a different region of our Cloud testbed. As `Orch` detects a failure, a new `Replica` is placed in the same region as the failed middlebox. The `Head` of `Firewall` is deployed in the same region as `Orch`, while the `Heads` of `SimpleNAT` and `Monitor` are respectively deployed in a neighboring region and a remote region compared to `Orch`'s region. Since `Orch` is also a SDN controller, we observe negligible

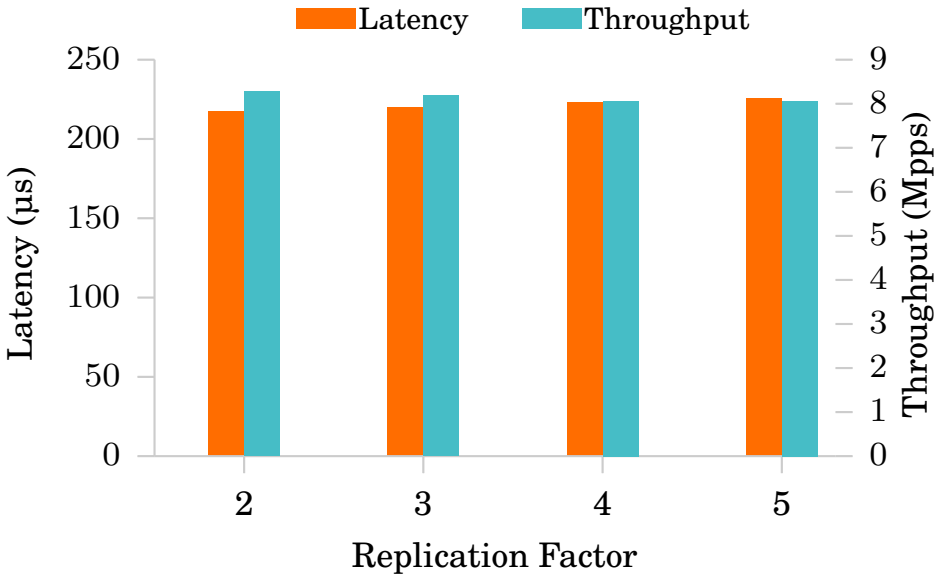


Figure 4.12: Replication factor

values for the rerouting delay, thus we focus on the state recovery delay and initialization delay.

Recovery time: As shown in Figure 4.13, the initialization delays are 1.2, 49.8, and 5.3 ms for `Firewall`, `Monitor`, and `SimpleNAT`, respectively. The longer the distance between `Orch` and the new `Replica`, the higher the initialization delay. The state recovery delays are in the range of 114.38 ± 9.38 ms to 270.79 ± 50.47 ms[3]. FTC replicates the values of state variables, and its state recovery delay is bounded by the state size of a middlebox. Moreover, upon any failure, a new `Replica` fetches state from a remote region. The WAN latency between two remote regions becomes the dominant delay during failover. Using ping, we measured the network delay between all pairs of remote regions, and the observed RTTs confirmed our results. Finally, since in FTC, a new instantiated `Replica` fetches state in parallel from other `Replica`s (see Section 4.5), the replication factor has a negligible impact on recovery time.

[3]The large confidence intervals reported are due to latency variability in the wide area network connecting different regions.

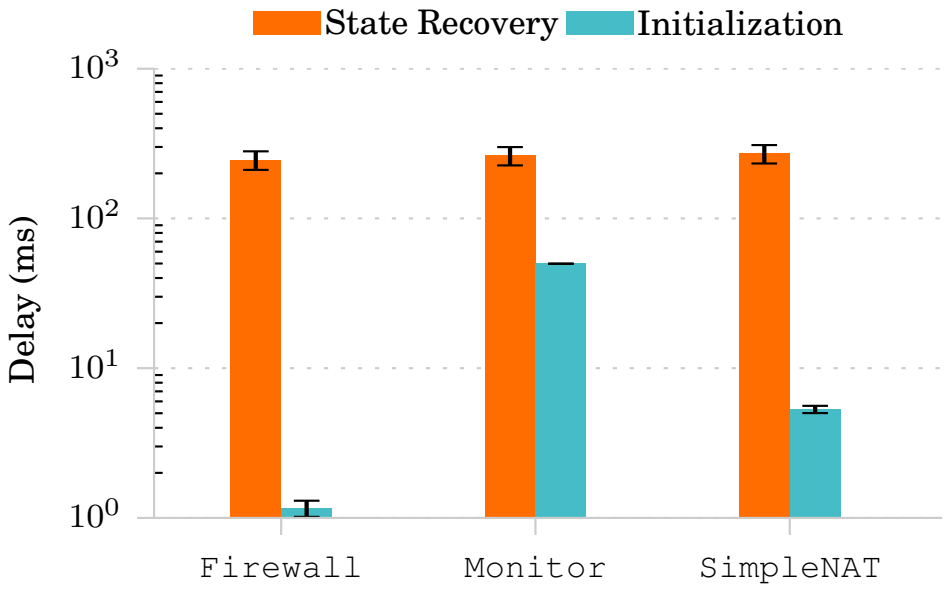


Figure 4.13: Recovery time

4.7 Related Work

We already discussed NFV related work in Section 4.1.2. Next, we position FTC related to three lines of work.

Fault tolerant storage: Prior to FTC, the distributed system literature used chain and ring structures to provide fault tolerance. However, their focus is on ordering read/write messages at the process level (compared to, middlebox threads racing to access shared state in our case), at lower non-determinism rates (compared to, per-packet frequency), and at lower output rates (compared to, several Mpps releases).

A class of systems adapt the chain replication protocol [163] for key-value storages. In HyperDex [43] and Hibari [53], servers shape multiple logical chains replicating different key ranges. NetChain [79] replicates in the network on a chain of programmable switches. FAWN [4], Flex-KV [125], and parameter server [96] leverage consistent hashing to form a replication ring of servers. Unlike these systems, FTC takes advantage of the natural structure of service function chains, uses transactional packet processing, and piggybacks state updates on packets.

Primary backup replication: In *active* replication [140], all replicas process requests. This scheme requires determinism in middlebox operations, while middleboxes are non-deterministic [37, 71]. In passive replication [20], only a *primary* server processes requests and sends state updates to other replicas. This scheme makes no assumption about determinism. Generic virtual machine high availability solutions [31, 38, 139] pause a virtual machine per each checkpoint. These solutions are not effective for chains, since the chain operations pauses during long checkpoints.

Consensus protocols: Classical consensus protocols, such as Paxos [94] and Raft [117] are known to be slow and cause unacceptable low performance if used for middleboxes.

4.8 Conclusion and Future Work

Existing fault tolerant middlebox frameworks can introduce high performance penalties when they are used for a service function chain. This chapter presented FTC, a system that takes advantage of the structure of a chain, transactional packet processing, and data dependency vectors to provide efficient fault tolerance. Our evaluation demonstrates that FTC can provide high degrees of fault tolerance with low overhead in terms of latency and throughput of a chain. As future work, we plan to rigorously prove the serializability of our data dependency vectors using *conflict equivalence* approach [65].

Chapter 5

Conclusions

5.1 Thesis Summary

In this thesis, we have presented three projects aiming to realize some of the main NFV promises: *cost efficiency*, *dynamic scaling*, and *fault tolerance*.

With **DSFC**, presented in Chapter 2, we optimized the cost of middlebox chain deployments. DSFC places middlebox instances distributedly and decouples the chain's performance from underlying hardware. By proving the NP-Hardness of a such deployment, we provided a better theoretical understanding of the complexity of a middlebox chain deployment. Kariz, our heuristic solution, enables operators to distribute a chain of middleboxes over a large network infrastructure with competitive performance.

With **Constellation**, discussed in Chapter 3, we designed a framework for the dynamic deployment of middleboxes across geo-distributed network infrastructures. Constellation uses asynchronous state replication of convergent state objects to achieve high performance and scalability for middleboxes. Constellation improves the packet processing throughput of competing designs by two orders of magnitude.

Lastly, we presented **FTC** in Chapter 4, where we made a chain of middleboxes fault tolerant. FTC considers the entire chain as a fault tolerant unit, instead of making individual middleboxes fault tolerant. FTC provides a simpler design for a fault tolerant chain and reaches 2–3.5$\times$ throughput of the state of art, with a performance overhead on the order of only tens of microseconds.

In the following, we discuss future research directions.

5.2 Future Research Directions

Fault tolerance for a chain of multi-instance middleboxes: To scale to large traffic load, a chain can compose *multi-instance* middleboxes. Achieving fault-tolerance with high-performance for such a chain deployment is a challenging problem. The difficulty lies in the fact that stateful middlebox instances share state. A distributed set of middleboxes – that can be deployed across separate physical servers – must be able to read and write shared state with high performance – e.g., per packet or per flow – while the requirements for fault tolerance of that state – which can be with high performance overhead – must be met.

In accessing shared state information, fault tolerance demands expensive co-ordinations among middlebox instances, while high performing packet processing avoids such costly synchronizations. Co-ordinations and synchronizations provide *consistency* guarantees but deteriorate performance drastically. Specifically, a fault tolerant system design for such chain deployment must find a trade-off between desired levels of consistency and concurrency.

Middlebox framework with multi-state object convergence: Many middlebox applications contain a collection of state objects. In some applications, concurrent updates across middlebox instances must hold collective invariants for a subset of state objects. Although such invariants can be specific to a particular middlebox application, the middlebox framework can provide mechanisms that facilitate resolving the conflicting state updates that may lead to violating such invariants.

The Constellation framework dedicates a state channel per state object. Alternatively, we can allocate a state channel per subset of state objects. State updates that are relevant to a subset of state objects are exchanged over the associated state channel. The transmission and delivery of relevant state updates can be performed transparently. Moreover, the framework can expose certain APIs to facilitate the task of developers in resolving conflicting updates.

References

[1] Ali Abedi and Tim Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 287–292, New York, NY, USA, 2017. ACM.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. *SIGCOMM Comput. Commun. Rev.*, 44(4):503–514, August 2014.

[3] Amazon. Elastic load balancing. https://aws.amazon.com/elasticloadbalancing/. [Online].

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[5] Fabien André, Stéphane Gouache, Nicolas Le Scouarnec, and Antoine Monsifrot. Don't share, don't lock: Large-scale software connection tracking with krononat. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 453–465, Berkeley, CA, USA, 2018. USENIX Association.

[6] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. Programming slick network functions. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 14:1–14:13, New York, NY, USA, 2015. ACM.

[7] Azeem Aqil, Karim Khalil, Ahmed O.F. Atya, Evangelos E. Papalexakis, Srikanth V. Krishnamurthy, Trent Jaeger, K. K. Ramakrishnan, Paul Yu, and Ananthram Swami. Jaal:

Towards network intrusion detection at isp scale. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 134–146, New York, NY, USA, 2017. ACM.

[8] Katerina Argyraki, Salman Baset, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Eddie Kohler, Maziar Manesh, Sergiu Nedevschi, and Sylvia Ratnasamy. Can software routers scale? In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 21–26, New York, NY, USA, 2008. ACM.

[9] S. Ayoubi, S. R. Chowdhury, and R. Boutaba. Breaking service function chains with khaleesi. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 64–72, May 2018.

[10] P Ayuso. Netfilter's connection tracking system. *;login*, 31(3), 2006.

[11] Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, and Raouf Boutaba. On orchestrating virtual network functions. In *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, CNSM '15, pages 50–56, Washington, DC, USA, 2015. IEEE Computer Society.

[12] Barracuda. Barracuda cloudgen firewall. https://www.barracuda.com/products/cloudgenfirewall. [Online].

[13] Barracuda. Barracuda WAF. https://goo.gl/QmLv8E. [Online].

[14] Ruben Becker, Maximilian Fickert, and Andreas Karrenbauer. A novel dual ascent algorithm for solving the min-cost flow problem. In *2016 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 151–159, 01 2016.

[15] 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 '14, pages 1–6, New York, NY, USA, 2014. ACM.

[16] Boost.org. boost.coroutine. https://github.com/boostorg/coroutine. [Online].

[17] Bevin Brett. Memory performance in a nutshell. https://software.intel.com/en-us/articles/memory-performance-in-a-nutshell, jun 2016. [Online].

[18] Bro. The bro network security monitor. https://www.bro.org. [Online].

[19] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[20] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Distributed systems (2nd ed.). In Sape Mullender, editor, *Distributed Systems (2Nd Ed.)*, chapter The Primary-backup Approach, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[21] B. Carpenter and S. Brim. Middleboxes: Taxonomy and issues. RFC 3234, RFC Editor, 2002.

[22] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[23] Chen Chen, Changbin Liu, Pingkai Liu, Boon Thau Loo, and Ling Ding. A scalable multi-datacenter layer-2 network architecture. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 8:1–8:12, New York, NY, USA, 2015. ACM.

[24] Chen Chen, Changbin Liu, Pingkai Liu, Boon Thau Loo, and Ling Ding. A scalable multi-datacenter layer-2 network architecture. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 8:1–8:12, New York, NY, USA, 2015. ACM.

[25] Elden Christensen. Tuning failover cluster network thresholds. https://bit.ly/2NC7dGk, 2020. [Online].

[26] P.C. Chu and J.E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, Jun 1998.

[27] Adrian Cockcroft. A closer look at the christmas eve outage. http://techblog.netflix.com/2012/12/a-closer-look-at-christmas-eve-outage.html. [Online].

[28] P. Coelho and F. Pedone. Geographic state machine replication. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 221–230, Oct 2018.

[29] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[30] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[31] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA, 2008. USENIX Association.

[32] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[33] Julien Desgats. How we built rate limiting capable of scaling to millions of domains. https://blog.cloudflare.com/counting-things-a-lot-of-different-things, 2017. [Online].

[34] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 44–51, Aug 2003.

[35] Dave Dice, Yossi Lev, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 325–334, New York, NY, USA, 2010. Association for Computing Machinery.

[36] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, page 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[37] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[38] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *Proceedings*

of the 4th Annual Symposium on Cloud Computing, SOCC '13, pages 3:1–3:16, New York, NY, USA, 2013. ACM.

[39] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Predicting the resource consumption of network intrusion detection systems. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, pages 135–154, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[40] Norbert Egi, Mihai Dobrescu, Jianqing Du, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, Laurent Mathy, et al. Understanding the packet processing capability of multi-core servers. Technical report, Technical Report, 2009.

[41] 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 *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, 2016.

[42] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 275–287, New York, NY, USA, 2015. ACM.

[43] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[44] ETSI. NFV Whitepaper. Technical report, European Telecommunications Standards Institute, 2017.

[45] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 184–193, Oct 1975.

[46] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[47] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using

flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 543–546, Seattle, WA, 2014. USENIX Association.

[48] Colin J Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987.

[49] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys Tutorials*, 15(4):1888–1906, Fourth 2013.

[50] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 333–346, San Jose, CA, 2013. USENIX.

[51] Open Networking Foundation. Openflow switch specification v.1.3.1. https://goo.gl/llwvyE. [Online].

[52] N. Freed. Behavior of and requirements for internet firewalls. RFC 2979, RFC Editor, 2000.

[53] Scott Lystig Fritchie. Chain replication in theory and in practice. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, Erlang '10, pages 33–44, New York, NY, USA, 2010. ACM.

[54] Fujitsu. Onos e-cord proof of concept demonstrates open disaggregated roadm. https://www.fujitsu.com/us/Images/ONOS-E-CORD-use-case.pdf. [Online].

[55] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 21:1–21:16, New York, NY, USA, 2016. ACM.

[56] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, abs/1305.0209, 2013.

[57] 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.

[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 *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[59] Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali Jose Mashtizadeh. Fault tolerant service function chaining. In *Proceedings of the 2020 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '20, New York, NY, USA, August 2020. Association for Computing Machinery (ACM).

[60] Milad Ghaznavi, Ali Jose Mashtizadeh, Bernard Wong, and Raouf Boutaba. Constellation: A high performance geo-distributed middlebox framework. Technical Report arXiv:2003.05111 [cs.NI], ArXiV, March 2020.

[61] Milad Ghaznavi, Nashid Shahriar, Shahin Kamali, Reaz Ahmed, and Raouf Boutaba. Distributed service function chaining. *IEEE Journal on Selected Areas in Communications*, 35(11):2479–2489, November 2017.

[62] Chaima Ghribi, Marouen Mechtri, and Djamal Zeghlache. A dynamic programming algorithm for joint vnf placement and chaining. In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, CAN '16, pages 19–24, New York, NY, USA, 2016. ACM.

[63] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36(4):873–886, October 1989.

[64] M. G. Gouda and A. X. Liu. A model of stateful firewalls and its properties. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 128–137, June 2005.

[65] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, page 428–451, New York, NY, USA, 1975. Association for Computing Machinery.

[66] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery.

[67] Y. Gu, M. Shore, and S. Sivakumar. A framework and problem statement for flow-associated middlebox state migration. https://tools.ietf.org/html/draft-gu-statemigration-framework-03, 2013.

[68] Guanyao Huang, A. Lall, C. Chuah, and Jun Xu. Uncovering global icebergs in distributed monitors. In *2009 17th International Workshop on Quality of Service*, pages 1–9, July 2009.

[69] S. Guha, A. Meyerson, and K. Munagala. Hierarchical placement and network design problems. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 603–612, 2000.

[70] T. Hain. Architectural implications of nat. RFC 2993, RFC Editor, 11 2000.

[71] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206, August 2010.

[72] HAProxy. Haproxy load balancer's development branch. https://github.com/haproxy/haproxy. [Online].

[73] Hewlett-Packard. HP Virtual Router Series. http://goo.gl/hPWrTt. [Online].

[74] A. Hirwe and K. Kataoka. Lightchain: A lightweight optimisation of vnf placement for service chaining in nfv. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 33–37, June 2016.

[75] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 181–194, New York, NY, USA, 2011. ACM.

[76] Marios Iliofotou, Michalis Faloutsos, and Michael Mitzenmacher. Exploiting dynamicity in graph-based traffic analysis: Techniques and applications. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 241–252, New York, NY, USA, 2009. ACM.

[77] Marios Iliofotou, Prashanth Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 315–320, New York, NY, USA, 2007. ACM.

[78] DPDK Intel. Data plane development kit, 2015.

[79] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, 2018. USENIX Association.

[80] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network*, 22(5):20–25, September 2008.

[81] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, 2017. USENIX Association.

[82] Rishi Kapoor, Alex C. Snoeren, Geoffrey M. Voelker, and George Porter. Bullet trains: A study of nic burst behavior at microsecond timescales. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 133–138, New York, NY, USA, 2013. ACM.

[83] Katran. Katran, a high performance layer 4 load balancer. https://github.com/facebookincubator/katran. [Online].

[84] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 4:1–4:12, New York, NY, USA, 2015. ACM.

[85] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 501–516, Boston, MA, 2019. USENIX Association.

[86] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the way for nfv: Simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 239–253, Santa Clara, CA, 2016. USENIX Association.

[87] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[88] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[89] Jeff Kronlage. Asymmetric routing and firewalls. http://brbccie.blogspot.com/2013/03/. [Online].

[90] Sameer G Kulkarni, Guyue Liu, KK Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, and Xiaoming Fu. Reinforce: Achieving efficient failure resiliency for network function virtualization based services. In *15th USENIX International Conference on emerging Networking EXperiments and Technologies (CoNEXT) 18)*. USENIX Association, 2018.

[91] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 177–188, New York, NY, USA, 2004. ACM.

[92] S Kumar, M. Tufail, S. Majee, C. Captari, and S. Homma. Service function chaining use cases in data centers. Internet draft, RFC Editor, 2 2017.

[93] T. W. Kuo, B. H. Liou, K. C. J. Lin, and M. J. Tsai. Deploying chains of virtual network functions: On the relation between link and server usage. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[94] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[95] Hongda Li, Hongxin Hu, Guofei Gu, Gail-Joon Ahn, and Fuqiang Zhang. vnids: Towards elastic security with safe and efficient virtualization of network intrusion detection systems. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 17–34, New York, NY, USA, 2018. ACM.

[96] Mu Li, David G. Anderson, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Operating Systems Design and Implementation (OSDI)*, pages 583–598, 2014.

[97] Linux. fork - create a child process. http://man7.org/linux/man-pages/man2/fork.2.html. [Online].

[98] Linux. madvise - give advice about use of memory. http://man7.org/linux/man-pages/man2/madvise.2.html. [Online].

[99] Linux. tc - show / manipulate traffic control settings. https://linux.die.net/man/8/tc. [Online].

[100] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 334–350. ACM, 2019.

[101] H. Long, Y. Shen, M. Guo, and F. Tang. Laberio: Dynamic load-balanced routing in openflow-enabled networks. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 290–297, March 2013.

[102] Y. Lu, B. Prabhakar, and F. Bonomi. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*, pages 2774–2778, July 2006.

[103] Marcelo Caggiani Luizelli, Weverton Luis da Costa Cordeiro, Luciana S. Buriol, and Luciano Paschoal Gaspary. A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining. *Computer Communications*, 102:67 – 77, 2017.

[104] Tamás Lukovszki, Matthias Rost, and Stefan Schmid. It's a match!: Near-optimal and incremental middlebox deployment. *SIGCOMM Comput. Commun. Rev.*, 46(1):30–36, January 2016.

[105] J. Ma, F. Le, A. Russo, and J. Lobo. Detecting distributed signature-based intrusion: The case of multi-path routing attacks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 558–566, April 2015.

[106] Mohammad Mahdian and Martin Pál. Universal facility location. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003*, pages 409–421, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[107] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

[108] MAWI. Mawi working group traffic archive - 2019. https://mawi.wide.ad.jp/mawi/ditl/ditl2019-G/201904090900.html, 2019. visited on April 2020.

[109] MAWI. Mawi working group traffic archive - 2020. https://mawi.wide.ad.jp/mawi/samplepoint-F/2020/202002161400.html, 2020. visited on April 2020.

[110] MazuNAT. mazu-nat.click, 2019. https://github.com/kohler/click/blob/master/conf/mazu-nat.click.

[111] Microsoft. Asymmetric routing with multiple network paths. https://docs.microsoft.com/en-us/azure/expressroute/expressroute-asymmetric-routing. [Online].

[112] H. Moens and F. D. Turck. Vnf-p: A model for efficient placement of virtualized network functions. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 418–423, Nov 2014.

[113] Juniper Networks. Contrail sd-wan. https://www.juniper.net/us/en/products-services/sdn/contrail/contrail-sd-wan. [Online].

[114] NiciraNetworks. Openvswitch: An open virtual switch. http://openvswitch.org.

[115] NiciraNetworks. The published onos docker images. https://hub.docker.com/r/onosproject/onos/.

[116] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, Renton, WA, 2018. USENIX Association.

[117] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[118] OpenBSD. pf — packet filter. https://man.openbsd.org/pf.4. [Online].

[119] Christoph Paasch and Olivier Bonaventure. Multipath tcp. *Commun. ACM*, 57(4):51–57, April 2014.

[120] M. Pal, T. Tardos, and T. Wexler. Facility location with nonuniform hard capacities. In *Proceedings 2001 IEEE International Conference on Cluster Computing*, pages 329–338, Oct 2001.

[121] 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 Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[122] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. SCL: Simplifying distributed SDN control planes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 329–345, Boston, MA, 2017. USENIX Association.

[123] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, 31(23-24):2435–2463, December 1999.

[124] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, New York, NY, USA, 2012. ACM.

[125] Amar Phanishayee, David G. Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. Flex-kv: Enabling high-performance and flexible kv systems. In *Proceedings of the 2012 Workshop on Management of Big Data Systems*, MBDS '12, pages 19–24, New York, NY, USA, 2012. ACM.

[126] Open Network Automation Platform. Onap architecture overview. https://www.onap.org/wp-content/uploads/sites/20/2018/11/ONAP_CaseSolution_Architecture_112918FNL.pdf. [Online].

[127] Diana Popescu, Noa Zilberman, and Andrew Moore. Characterizing the impact of network latency on cloud-based applications' performance, 2017.

[128] 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 '13, pages 9–22, New York, NY, USA, 2013. ACM.

[129] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, SECURE NETWORKS INC CALGARY ALBERTA, 1998.

[130] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM.

[131] Paul Quinn and Thomas Nadeau. Problem Statement for Service Function Chaining. Internet-draft, IETF, 2015.

[132] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. *SIGCOMM Comput. Commun. Rev.*, 37(4):337–348, August 2007.

[133] 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 '13, pages 1:1–1:15, New York, NY, USA, 2013. ACM.

[134] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, Lombard, IL, 2013. USENIX.

[135] Karthikeyan Ranganathan. Netflix shares cloud load balancing and failover tool: Eureka! https://bit.ly/2Xa9PBW. [Online].

[136] Mindaugas Rasiukevicius. Npf: packet filter with stateful inspection, nat, ip sets, etc. http://rmind.github.io/npf/. [Online].

[137] riverbed. Steelhead Product Family Spec Sheet. http://goo.gl/g2XfNs. [Online].

[138] M. Rost and S. Schmid. On the hardness and inapproximability of virtual network embeddings. *IEEE/ACM Transactions on Networking*, 28(2):791–803, April 2020.

[139] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. The design and evaluation of a practical system for fault-tolerant virtual machines. Technical report, Technical Report VMWare-RT-2010-001, VMWare, 2010.

[140] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[141] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 207–212, New York, NY, USA, 2004. ACM.

[142] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *NSDI 12*, pages 323–336, 2012.

[143] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[144] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[145] Justine Sherry. *Middleboxes as a Cloud Service*. PhD thesis, EECS Department, University of California, Berkeley, November 2016.

[146] 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. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 227–240, New York, NY, USA, 2015. ACM.

[147] Justine Sherry, Sylvia Ratnasamy, and Justine Sherry At. A survey of enterprise middlebox deployments, 2012.

[148] S. Sinha et al. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets*, 2004.

[149] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA, 2017. ACM.

[150] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a nids. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 89–98, Dec 2006.

[151] Robin Sommer, Matthias Vallentin, Lorenzo De Carli, and Vern Paxson. Hilti: An abstract execution environment for deep, stateful network traffic analysis. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 461–474, New York, NY, USA, 2014. ACM.

[152] Squid. Squid frequently asked questions - memory. http://www.comfsm.fm/computing/squid/FAQ-8.html. [Online].

[153] Squid. Squid: Optimising web delivery. http://www.squid-cache.org. [Online].

[154] P. Srisuresh and K. Egevang. Traditional ip network address translator (traditional nat). RFC 3022, RFC Editor, 2001.

[155] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids - a graph-based intrusion detection system for large networks. In *In Proceedings of the 19th National Information Systems Security Conference*, pages 361–370, 1996.

[156] R. Stewart. Stream control transmission protocol. RFC 4960, RFC Editor, September 2007.

[157] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, August 1985.

[158] The AWS Team. Summary of the october 22, 2012 aws service event in the us-east region. https://aws.amazon.com/message/680342/. [Online].

[159] The Google Apps Team. Data center outages generate big losses. http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/us/appsstatus/ir/plibxfjh8whr44h.pdf. [Online].

[160] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. Dynamics of hot-potato routing in ip networks. *SIGMETRICS Perform. Eval. Rev.*, 32(1):307–319, June 2004.

[161] J. A. Tomlin. Minimum-cost multicommodity network flows. *Oper. Res.*, 14(1):45–51, February 1966.

[162] Daniel Turull, Peter Sjödin, and Robert Olsson. Pktgen: Measuring performance on high speed networks. *Computer Communications*, 82:39 – 48, 2016.

[163] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[164] Guido VAN ROOIJ. Real stateful tcp packet filtering in ip filter. *SANE 2000*, 2000.

[165] Shobha Venkataraman, Dawn Song, Phillip B Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2004.

[166] Javier Verdú, Mario Nemirovsky, and Mateo Valero. Multilayer processing - an execution model for parallel stateful packet processing. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 79–88, New York, NY, USA, 2008. ACM.

[167] O. Huang M. Boucadair N. Leymann Z. Cao J. Hu W. Liu, H. Li. Service function chaining use-cases. https://tools.ietf.org/html/draft-liu-sfc-use-cases-01. [Online].

[168] L. Wang, Z. Lu, X. Wen, R. Knopp, and R. Gupta. Joint optimization of service function chaining and resource allocation in network function virtualization. *IEEE Access*, 4:8084–8094, 2016.

[169] Limin Wang, Kyoung Soo Park, Ruoming Pang, Vivek Pai, and Larry Peterson. Reliability and security in the codeen content distribution network. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, page 14, USA, 2004. USENIX Association.

[170] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An untold story of middleboxes in cellular networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):374–385, August 2011.

[171] T. Wen, H. Yu, G. Sun, and L. Liu. Network function consolidation in service function chaining orchestration. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2016.

[172] Shinae Woo. S6: Elastic scaling of stateful network functions. https://github.com/NetSys/S6. [Online].

[173] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, Renton, WA, 2018. USENIX Association.

[174] Jiarong Xing, Wenqing Wu, and Ang Chen. Architecting programmable data plane defenses into the network with fastflex. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, pages 161–169, New York, NY, USA, 2019. ACM.

[175] Ying Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, and M. Tatipamula. Steering: A software-defined networking for inline service chaining. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, Oct 2013.