Building Scalable, Resilient and High-performance NFV Systems

by

Jingpu Duan

Department of Computer Science

The University of Hong Kong

(Temporary Binding for Examination Purposes)

Supervised by

Dr. Chuan Wu

A thesis submitted in partial fulfillment of the requirements for
the Degree of Doctor of Philosophy
at The University of Hong Kong

February, 2018

Declaration of Authorship

I declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualications.

Signed:

Jingpu Duan
February, 2018

Abstract of thesis entitled

"Building Scalable, Resilient and High-performance NFV Systems"

Submitted by

Jingpu Duan

for the degree of Doctor of Philosophy
at The University of Hong Kong
in February, 2018

Over the past years, Network Function Virtualization (NFV) becomes an important research topic. NFV technology advocates replacing dedicated hardware network functions (NFs) with virtualized NFs running on standard virtualization platform, which can significantly reduce the deployment cost and management complexity of NFs.

However, there has been several problems that bother the practitioners of NFV technology. First, the development of cloud technology calls for deploying network services across geo-distributed datacenters to server people living in a wide range. Yet NFV system suffers poor scalability when being deployed across geo-distributed datacenters, as scaling across multiple datacenters requires non-trivial interactions between various NFV system components. Second, resilience functionality, including failure resilience and flow migration, is of pivotal importance in NFV system. But, existing NFV systems implementing resilience functionalities usually have unsatisfactory performance due to heavyweight execution method and centralized control. Finally, NFs are more than simple packet processors, they need to contact external services to achieve advanced functionalities. To achieve best performance, NF software is often implemented using callback-based asynchronous programming style, which is complex and error-prone. Moreover, the increased implementation complexity brought by callbacks

further limits programmers from enriching NF software with more advanced functionalities.

Regarding the three problems, this thesis proposes three resulting NFV systems, including *ScalIMS*, *NFVactor* and *NetStar*, to tackle these problems using a systematic and empirical methodology.

ScalIMS designs a dynamic NFV service chain scaling system that is capable of scaling NFV service chains across geo-distributed datacenters, using IMS system as a case study. *ScalIMS* combines proactive and reactive approaches for timely, cost-effective scaling of the service chains, and is evaluated on a geo-distributed public cloud service. The evaluation result reveals that *ScalIMS* can effectively scale multiple NFV service chains across geo-distributed datacenters for improved traffic quality and reduced NF resource consumption.

NFVactor is a novel NFV system that aims to provide lightweight failure resilience and high-performance flow migration by exploiting actor model to improve the parallelism of resilience functionalities, while the efficiency of the actor model is guaranteed by a carefully designed runtime system. Moreover, *NFVactor* achieves transparent resilience: once a new NF is implemented for *NFVactor*, the NF automatically acquires resilience support. The evaluation result shows that *NFVactor* achieves 10Gbps packet processing, flow migration completion time that is 144 times faster than existing system, and packet processing delay stablized at around 20 microseconds during replication.

NetStar is a new NF programming framework that brings the future/promise abstraction to the NF dataplane. *NetStar* simplifies asynchronous NF programming via a carefully designed async-flow interface that exploits the future/promise paradigm by chaining multiple continuation functions for asynchronous operations handling. NF programs implemented using *NetStar* framework mimic simple synchronous programming, but are able to achieve full flow processing asynchrony, with simple, elegant code. The evaluation results demonstrates that *NetStar* can effectively simplify asynchronous NF programming by substantially reducing the lines of code, while still approaching line-rate packet processing speeds.

[**479 Words**]

Acknowledgements

First, I would like to express my sincere gratitude towards my supervisor, Prof. Chuan Wu, for her kind help throughout the course of my graduate study. She has been encouraging and motivating me to pursue my own research goal and I really appreciate all the precious discussions that we had during the past five years. Besides research, Prof. Chuan Wu also helped me get through so many obstacles that I had in life. Without Prof. Chuan Wu's kind help, I wouldn't have completed my graduate study.

Second, I would like to thank Dr. Heming Cui for his kind help and discussions on the project that we worked together. Dr. Heming Cui's passion for solving challenging research problems really motivates me to keep on improving the quality of my research work.

I would like to thank various group members who I have worked with: Dr. Yu Wu, Dr. Jian Zhao, Dr. Xuanjia Qiu, Zhizhong Zhang, Xiaoxi Zhang, Yanghua Peng, Yixin Bao, Qihang Sun, Xiaodong Yi, Junjie Wang and many others. This journey would not be so memorable without all the support and help from all of them. I would also like to thank my collaborators including Dr. Franck Le and Prof. Alex X. Liu, for their important discussions that make my research work better.

Finally, I would like to thank my parents for raising me and giving me power to complete my graduate study. I would like to thank my fiance, Bingjie Zhao, for all the love that she gave me.

The entire thesis is dedicated to my dearest mother.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

NFV	**N**etwork **F**unction **V**irtualization
VNF	**V**irtualized **N**etwork **F**unction
VM	**V**irtual **M**achine
MAB	**M**ulti-**A**rmed **B**andit
IDS	**I**ntrusion **D**etection **S**ystem
TSP	**T**ravelling **S**alesman **P**roblem

Dedicated to my mother.

Chapter 1

Introduction

1.1 Introduction of *ScalIMS*

Traditional hardware-based network functions are notoriously hard and costly to deploy and scale. The recent paradigm of network function virtualization (NFV) advocates deploying software network functions in virtualized environments (e.g., VMs) on off-the-shelf servers, to significantly simplify deployment and scaling at much lowered costs [27].

Despite the advantages, many problems remain when introducing NFV to the provisioning of practical network services. One problem is to design efficient VNF software, such that software VNFs can achieve packet processing speeds close to hardware middleboxes. Another is to design an efficient management system, which deploys and scales VNF service chains – an ordered collection of VNFs that altogether compose a network service, according to the traffic demand. There have been efforts targeting architectural improvement of VNF software [72]. A number of management systems have also been proposed [58, 79], which operate VNF service chains deployed in a single server cluster or datacenter. These management systems are adequate for service chains such as "firewall→ intrusion detection system (IDS)", which are typically used to provide access service to a

client-server Web system, deployed in the on-premise cluster/datacenter of the service provider.

There are many other service chains which render a geo-distributed nature, e.g., the service chains in IP Multimedia Subsystems (IMS) [1] and mobile core networks [11]. In these systems, the network functions are desirably deployed close to geo-dispersed users, and putting the service chain in a single datacenter would be unfavourable as compared to distributing its VNFs across several datacenters. The existing management systems cannot be directly applied to handle such geo-distributed service chains [49], due to the escalated challenges on efficient interconnection of VNFs over the WAN, dynamic decision making on how VNF instances are deployed in different datacenters, and optimally dispatching individual flows through the deployed instances.

This paper presents *ScalIMS*, a management system that enables dynamic deployment and scaling of VNF service chains across multiple datacenters, using representative control-plane and data-plane service chains of the IMS system [1]. *ScalIMS* is designed to provide good performance (minimal VNF instances deployment and guaranteed end-to-end flow delays), using both runtime statistics of VNFs and global traffic information. IMS is chosen as the target platform because of its important role in the telecom core networks as well as the accessibility of open-source software implementation of IMS [31]. *ScalIMS* has two important characteristics that distinguish itself from existing management systems:

▷ **Dynamic Scaling over Multiple Datacenters:** *ScalIMS* dynamically deploys multiple instances of the same network function onto different datacenters according to real-time traffic demand and user distribution. The network paths that a service chain traverses are optimized to provide QoS guarantee of user traffic (i.e., bounded end-to-end delays). This feature distinguishes *ScalIMS* from systems that can only scale service chains within a single datacenter [58, 79].

▷ **A Hybrid Scaling Strategy:** Most existing VNF management systems [91] [58] scale service chains using reactive approaches, adding/removing VNF instances by responding to changes of runtime status of existing VNFs. Novelly, *ScalIMS* combines reactive scaling with proactive scaling, using predicted traffic volumes

based on the history. This hybrid strategy exploits all opportunities for timely scaling of VNFs and significantly improves system performance.

We evaluate *ScalIMS* on IBM SoftLayer cloud. Experiment results show that *ScalIMS* significantly improves QoS of user traffic compared with scaling systems that use only reactive or proactive scaling approaches. Meanwhile, *ScalIMS* achieves this improvement using almost 50% less VNF instances. Even though *ScalIMS* is designed for IMS systems, similar design principles can be easily applied to other NFV systems, which benefit from service chain deployment across multiple datacenters.

1.2 Introduction of *NFVactor*

Network function virtualization (NFV) advocates moving *network functions* (NFs) out of dedicated hardware middleboxes and running them as virtualized applications on commodity servers [28]. To enable effective large-scale deployment of virtual NFs, a number of NFV management systems have been proposed in recent years [48, 52, 58, 79, 87, 95], implementing a broad range of management functionalities. Among these functionalities, resilience guarantee, supported by flow migration and failure recovery mechanisms, is of particular importance in practical NFV systems.

Resilience to failures [85, 88] is crucial for stateful NFs. Many NFs maintain important per-flow states [50]: IDSs such as Bro [4] store and update protocol-related states for each flow for issuing alerts for potential attacks; firewalls [21] parse TCP SYN/ACK/FIN packets and maintain TCP connection related states for each flow; load balancers [22] may retain mapping between a flow identifier and the server address, for modifying destination address of packets in the flow. It is critical to ensure correct recovery of flow states in case of NF failures, such that the connections handled by the failed NFs do not have to be reset – a simple approach strongly rejected by middlebox vendors [88].

Efficient flow migration [59, 83, 86] is important for long-lived flows in case of dynamic system scaling. Existing NFV systems [58, 79] mostly assume dispatching new flows to newly created NF instances when existing instances are overloaded, or waiting for remaining flows to complete before shutting down a mostly idle instance, which are efficient for short flows. Long flows are common on the Internet: a web browser uses one TCP connection to exchange many requests and responses with a web server [18]; video-streaming [14] and file-downloading [16] systems maintain long-lived TCP connections for fetching large volumes of data from CDN servers. When NF instances handling such flows are overloaded or under-loaded, migrating flows to other available NF instances enables timely hotspot resolution or system cost minimization [59].

Even though failure resilience and efficient flow migration are important for NFV systems, enabling light-weight failure resilience and high-performance flow migration within existing NF software architecture has been a challenging task.

Failure resilience in the existing systems [85, 88] is typically implemented through checkpointing: each NF process is regularly checkpointed, and if it fails, the system replays important log traces collected since the latest checkpoint to recover the failed NF. Compared to the normal packet processing delay of an NF that lies within tens of microseconds, the process of checkpointing is heavyweight and can cause extra delay up to thousands of microseconds [85, 88].

Flow migration in existing systems [59, 86] is typically governed by a centralized controller. It fully monitors the migration process of each flow by installing SDN rule to update the route of the flow and exchanging messages with the NFs over inefficient kernel networking stack [25]. However, a practical NFV system needs to manage tens of running NFs and handle tens of thousands of concurrent flows. To migrate these flows, the controller needs to sequentially execute the migration process of each flow, install a large number of SDN rules and exchange many migration protocol messages through inefficient communication channel, which may prolong the flow migration completion time and inhibit flow migration from serving as a practical NFV management task.

Besides, enabling flow migration with existing NF software is not trivial: OpenNF [59] reports that thousands lines of patch code must be added to existing NF software [4, 41] in order to extract and serialize flow states, communicate with the controller and control flow migration. This approach mixes the logic for controlling flow migration together with the core NF logic. To maintain and upgrade such an NF, the developer must well understand both the core NF logic and the complicated flow migration process, which adds additional burden on the developer.

In this paper, we present *NFVactor*, a new software framework for building NFV systems with high-performance flow migration and lightweight failure resilience. Unlike previous systems [59, 85, 86, 88] which augment existing NF software with resilience support, *NFVactor* explores new research opportunities brought by a holistic approach: *NFVactor* provides a general framework with built-in resilience support by exploiting the distributed actor model [2], and exposes several easy-to-use APIs for implementing NFs. Internally, *NFVactor* delegates the processing of each individual flow to an unique flow actor. The flow actors run in high-performance runtime systems, handle flow processing and ensure their own resilience in a largely decentralized fashion. *NFVactor* brings three major benefits.

▷ *Transparent resilience guarantee. NFVactor* ensures that once the NFs are implemented with the provided APIs, failure resilience of the NFs is immediately guaranteed. *NFVactor* decouples resilience logic from core NF logic by incorporating resilience operations within the framework and only exposing APIs for building NFs. Using the APIs, programmers are fully liberated from reasoning about details of resilience operations, but only focus on implementing the processing logic of NFs and handling simple interaction for synchronizing shared states of NFs during resilience operations. The exposed APIs also ensure a clean separation between the core processing logic and important NF states, facilitating resilience operations.

▷ *Lightweight failure resilience.* With the actor abstraction and cleanly separated NF states, *NFVactor* is able to replicate each flow independently without

checkpointing the entire NF process. Each flow actor can replicate itself by constantly saving its per-flow state to another actor that serves as a replica. This lightweight resilience operation guarantees good throughput, short recovery time and a small packet processing delay.

▷ *High-performance flow migration.* The use of the actor model enables *NFVactor* to adopt a largely decentralized flow migration process: each flow actor can migrate itself by exchanging messages with other flow actors, while a centralized controller only initiates flow migration by instructing a runtime system about the amount of the flow actors that should be migrated. As a result, *NFVactor* is able to concurrently migrate a large number of flows among multiple pairs of runtime systems. *NFVactor* also replaces SDN switch with a lightweight virtual switch for flow redirection, simplifying flow redirection from updating SDN rule into modifying an runtime identifier number. The increased parallelism and simplified flow redirection jointly enhance the performance of flow migration.

Our major technical challenge is to build an actor runtime system to satisfy the stringent performance requirement of NFV application. Even the fastest actor runtime systems [54] may fail to deliver satisfactory packet processing performance due to their actor scheduling strategies and the use of kernel networking stack. To address this challenge, we carefully craft a high-performance actor runtime system by combining the performance benefits of (i) a module graph scheduler to effectively schedule multiple flow actors, (ii) a DPDK-based [19] fast packet I/O framework [3] to accelerate network packet processing and (iii) an efficient user-space message passing channel which completely bypasses the kernel network stack and improves the performance of both failure resilience and flow migration.

We implement *NFVactor* and build several useful NFs using the exposed APIs. Our testbed experiments show that *NFVactor* achieves 10Gbps line-rate processing for 64-byte packets, concurrent migration of 600K flows using around 700 milliseconds, and recovery of a single runtime within 70 milliseconds in case

of failure. Compared with OpenNF [59], flow migration completion time in *NFVactor* can be 144 times faster. Compare with FTMB [88] for replication performance, *NFVactor* achieves similar packet processing throughput and recovery time, but with packet processing latency stabilized at around 20 microseconds. The source code of *NFVactor* is available at [43].

1.3 Introduction of *NetStar*

Network Functions (NFs) are more than simple packet processors that perform various transformations on each received packet. Modern NFs, *e.g.*, firewall [65], NAT [65], IDS [4], and proxies [17, 31], often need to contact external services while processing network flows, *e.g.*, for querying external databases [5, 42], or saving critical per-flow states on external reliable storage (for failure resilience purposes) [65]. To ensure high-speed packet processing while executing external queries, these NFs must fully exploit asynchronous programming: after generating a request to an external service, the NF should not block and wait for the response in a synchronous fashion; instead, it can save the current processing context and register an event handler function to handle the response upon its return, and switch to process other packets, potentially generating additional asynchronous requests.

For example, to detect whether a file transmitted over a TCP connection contains a piece of malware, a Bro IDS [4] issues a DNS query containing the SHA1 hash [37] of the file extracted from the reconstructed byte stream of the TCP flow to a Malware Hash Registry (MHR) [24]. Then, the MHR generates a DNS response indicating whether the hash matches that of some known malware. To ensure high performance, the Bro IDS does not block and wait for the MHR response to arrive. Instead, it registers a callback function to handle the MHR response upon its receipt, in an asynchronous fashion, and switches to process other flows/packets or handle other generated events (new packet arrival, new reconstructed payload, etc. [81]).

Compared with synchronous NF programs, asynchronous NF implementation using callbacks is significantly more efficient in packet processing, as it does not waste important CPU time. However, callback-based asynchronous programming has some inherent drawbacks that can prevent developers from building NFs with richer functionalities.

First, compared to a synchronous program, a callback-based asynchronous program is harder to implement and reason about. Such a program may define multiple callback functions, scattered within different source files, to achieve a series of asynchronous operations. For example, the Bro IDS can be configured to detect malware in flows using two nested callback functions, a first callback function to handle the reply from a local database query which may trigger another callback function to handle the reply to a MHR query (Sec. 4.2.1); and a NAT may replicate important per-flow states in an external database using 4 consecutive callbacks, to read from and write to a remote database while a TCP connection through the NAT is being established [65]. Dealing with multiple callbacks scattered in different source files can be confusing, and make it more difficult for a programmer to trace the execution order of the program.

Second, visiting saved context information inside a registered callback can be error-prone. Since an asynchronous program immediately switches to other tasks after saving the context and registering a callback, the program must ensure that the saved context is not accidentally freed until the callback is invoked. Failing to do so may lead to invalid memory access and program crash. However, when multiple callback functions are used, the programmer may accidentally free the context if he fails to correctly trace the execution order of the callbacks.

Third, redundant error handling code may be introduced in a callback-based asynchronous program. Since exceptions may happen when waiting for the external response, the program must properly handle the exceptions by either registering an error handling function or implementing exception handling logic in the callback registered to handle the response. When a series of asynchronous operations are executed, the programmer needs to add exception handling logic for each asynchronous operation. Since the asynchronous operations/callback

functions may well be scattered among multiple files, duplicate error handling code may need to be added in multiple places.

People have recognized the problems with callbacks when building event-driven systems such as web browsers [57, 66], programming language runtime systems [90], web servers [45] and database servers [33]. Their solution to counter the problems is to use a more advanced programming abstraction, such as the coroutine [9] and the future/promise paradigm [56, 69, 92]. Coroutine is a user-space cooperative thread that is able to execute asynchronous program in a fully synchronous fashion. However, the coroutine switching time may cause and suffer considerable overhead in NFs processing a large number of concurrent network flows. In contrast, the future/promise paradigm uses special runtime objects, futures, promises and continuation functions, to mimic synchronous programming while being fully asynchronous. Besides making asynchronous program easier to implement, the future/promise abstraction can also reduce redundant error handling code by effectively propagating exceptions to a consolidate error handling logic.

Though the future/promise paradigm is promising, the future/promise abstraction had only existed in high-level programming languages such as F# [90], Haskell [69] and OCaml [92]. It is known that high-level programming languages are not suitable for developing NF software due to their lower runtime efficiency as compared to C/C++-based implementation, and unpredictable processing delay caused by their garbage collectors [80]. A recent open-source C++ library, Seastar [36], is implementing the future/promise paradigm to build high-performance database servers [35]. The library is integrated with DPDK and provides a customized user-space TCP/IP stack for high-speed database queries. However, it does not expose any interface to manipulate raw network packets, and designing such an interface, which effectively processes network packets without diminishing the power of the future/promise abstraction, is non-trivial. A straightforward design may directly expose a regular packet handler function, that is invoked for each received packet. However, such a design falls back to

callback-based asynchronous programming and leaves no room for utilizing the future/promise abstraction.

This paper proposes *NetStar*, a future/promise-based programming framework for simple, elegant asynchronous programming in NFs. *NetStar* enables programming a series of asynchronous operations (when processing dataplane packets) in a manner similar to implementing a simple synchronous program, while not incurring any performance degradation due to blocking as in a synchronous program.

The power of *NetStar* is mainly attributed to a programming interface that we design, the async-flow interface, which effectively combines network packet handling process with the future/promise abstraction. The async-flow interface is powered by a simulated packet processing loop, and uses the returned future objects from a packet handler function for implementing core NF processing logic. With this interface, programmers can simplify the implementation of complex asynchronous operations in NFs by chaining a series of continuation functions, which mimics a synchronous program. Due to the future/promise paradigm, programmers can avoid redundant error handling logic but use a set of consolidate error handling code, allowing them to focus more on the core NF processing logic. The async-flow interface also simplifies context management: a programmer only needs to keep track of a pointer to a context object, and subsequent visits to the context object is guaranteed to be safe.

To evaluate the performance of *NetStar*, we build a number of NFs using the framework, including four NFs from the StatelessNF paper [65], an HTTP reverse proxy, an IDS and a malware detector. With extensive experiments, we show that NFs based on *NetStar* use substantially fewer lines of code to implement asynchronous packet processing, as compared to callback-based implementation, while delivering sufficiently good performance in terms of packet processing throughput and latency. We also compare *NetStar* with a coroutine based implementation, and show that the coroutine is a less desirable paradigm for implementing NFs processing a large number of concurrent network flows. The source code of *NetStar* is available at [26].

Chapter 2

ScalIMS: Scaling IMS Service Chains Across Geo-distributed Datacenters

2.1 Introduction

Traditional hardware-based network functions are notoriously hard and costly to deploy and scale. The recent paradigm of network function virtualization (NFV) advocates deploying software network functions in virtualized environments (e.g., VMs) on off-the-shelf servers, to significantly simplify deployment and scaling at much lowered costs [27].

Despite the advantages, many problems remain when introducing NFV to the provisioning of practical network services. One problem is to design efficient VNF software, such that software VNFs can achieve packet processing speeds close to hardware middleboxes. Another is to design an efficient management system, which deploys and scales VNF service chains – an ordered collection of VNFs that altogether compose a network service, according to the traffic demand. There have been efforts targeting architectural improvement of VNF software [72]. A number of management systems have also been proposed [58, 79], which operate

VNF service chains deployed in a single server cluster or datacenter. These management systems are adequate for service chains such as "firewall→ intrusion detection system (IDS)", which are typically used to provide access service to a client-server Web system, deployed in the on-premise cluster/datacenter of the service provider.

There are many other service chains which render a geo-distributed nature, e.g., the service chains in IP Multimedia Subsystems (IMS) [1] and mobile core networks [11] (examples presented in Fig. 2.1). In these systems, the network functions are desirably deployed close to geo-dispersed users, and putting the service chain in a single datacenter would be unfavourable as compared to distributing its VNFs across several datacenters. The existing management systems cannot be directly applied to handle such geo-distributed service chains [49], due to the escalated challenges on efficient interconnection of VNFs over the WAN, dynamic decision making on how VNF instances are deployed in different datacenters, and optimally dispatching individual flows through the deployed instances.

This chapter presents *ScalIMS*, a management system that enables dynamic deployment and scaling of VNF service chains across multiple datacenters, using representative control-plane and data-plane service chains of the IMS system [1]. *ScalIMS* is designed to provide good performance (minimal VNF instances deployment and guaranteed end-to-end flow delays), using both runtime statistics of VNFs and global traffic information. IMS is chosen as the target platform because of its important role in the telecom core networks as well as the accessibility of open-source software implementation of IMS [31]. *ScalIMS* has two important characteristics that distinguish itself from existing management systems:

▷ **Dynamic Scaling over Multiple Datacenters:** *ScalIMS* dynamically deploys multiple instances of the same network function onto different datacenters according to real-time traffic demand and user distribution. The network paths that a service chain traverses are optimized to provide QoS guarantee of user traffic (i.e., bounded end-to-end delays). This feature distinguishes *ScalIMS* from systems that can only scale service chains within a single datacenter [58, 79].

▷ **A Hybrid Scaling Strategy:** Most existing VNF management systems [91] [58] scale service chains using reactive approaches, adding/removing VNF instances by responding to changes of runtime status of existing VNFs. Novelly, *ScalIMS* combines reactive scaling with proactive scaling, using predicted traffic volumes based on the history. This hybrid strategy exploits all opportunities for timely scaling of VNFs and significantly improves system performance.

We evaluate *ScalIMS* on IBM SoftLayer cloud. Experiment results show that *ScalIMS* significantly improves QoS of user traffic compared with scaling systems that use only reactive or proactive scaling approaches. Meanwhile, *ScalIMS* achieves this improvement using almost 50% less VNF instances. Even though *ScalIMS* is designed for IMS systems, similar design principles can be easily applied to other NFV systems, which benefit from service chain deployment across multiple datacenters.

2.2 Background

2.2.1 IMS Overview

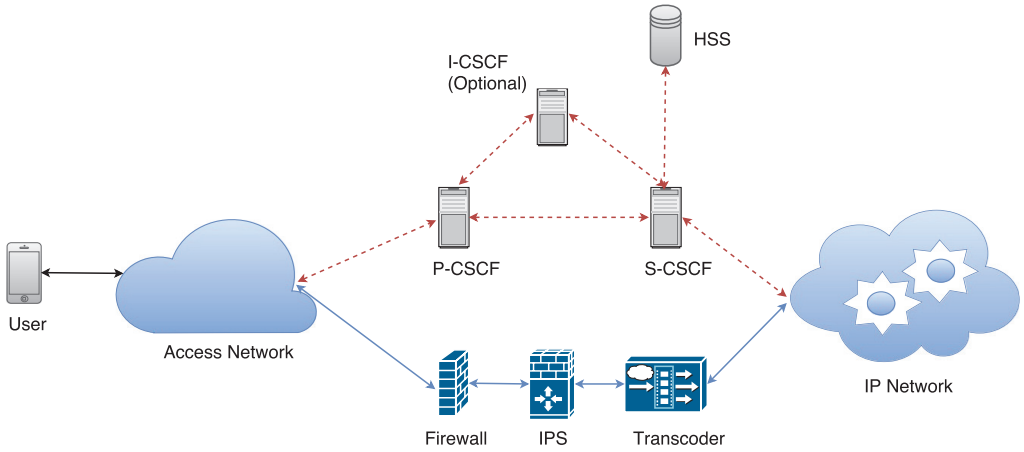


FIGURE 2.1: IMS: an architectural overview

An IP Multimedia Subsystem (IMS) [1] is a core part in 3G/4G telecom networks (e.g., 3GPP, 3GPP2) [46] [23], responsible for delivering multimedia services (e.g.,

voice, video, messaging) over IP networks. It is a complex system consisting of multiple service chains. We investigate two most important service chains as follows. An illustration is given in Fig. 2.1.

▷ **Control Plane (CP) Service Chain** includes three main network functions, Proxy-CSCF (P-CSCF), Interrogating-CSCF (I-CSCF), and Serving-CSCF (S-CSCF), which collectively handle user registration, user authentication and call setup. These network functions rely on the Session Initiation Protocol (SIP) [38] to interoperate with users of the IMS system. Users can only contact with P-CSCF, which acts as a relay point between users and S-CSCF. Since I-CSCF acts as a middleman that forwards SIP messages between P-CSCF and S-CSCF, real-world implementation sometimes merges I-CSCF into S-CSCF as in [31] to simplify the structure of the IMS control plane service chain, making I-CSCF optional. S-CSCF dispatches SIP messages to their final destinations and constantly queries an external storage server called Home Subscriber Server (HSS), which is a database that contains identities of the users. We consider the control plane service chain P-CSCF $\rightarrow$ S-CSCF $\rightarrow$ P-CSCF in *ScalIMS*.

▷ **Data Plane (DP) Service Chain** contains a sequence of network functions that the actual multimedia traffic between users traverses, for security (e.g., firewall, deep packet inspection, intrusion detection), connectivity (e.g., NAT, IPv4-to-IPv6 conversion), quality of service (e.g., traffic shaping, rate limiting, ToS/DSCP bit setting), and media processing (e.g., transcoding). While 3GPP has standardized the IMS control plane for interoperability reasons, the exact set of deployed network functions for the data plane varies per operator.

The two service chains collectively handle two important procedures of the IMS system, which are user registration and call setup. To make a call, a user first registers his IP address to the IMS by initiating a SIP REGISTRATION transaction over the CP. When the registration is done, S-CSCF temporarily stores the binding between the identity of the user and the P-CSCF instance connected to the user for future calls. To setup a call between a caller and a callee, the caller initiates a SIP INVITE transaction to the IMS, specifying the identify of the callee. S-CSCF uses the binding saved during user registration to retrieve

the P-CSCF instance that the callee connects to and sends the message to the P-CSCF instance, which forwards to the callee. After the callee responds, a call is successfully set up. Subsequent media flows between the caller and the callee are routed through DP service chain. When the call is finished, a SIP BYE transaction between the caller and the callee is carried out to close the call over the CP.

2.2.2 Related Work

Running VNF software (e.g., DP packet processing software) on VMs incurs significant context switching cost [71], limiting the maximum throughput of a VNF. To solve this problem, ClickOS [72] maps packets directly from NIC receive queues to a shared memory region, and fetch packets directly from that shared memory region [19], which greatly improve packet processing throughput. However, this approach completely by-passes the existing kernel networking stack, unable to support VNFs (e.g., S-CSCF and P-CSCF used by IMS system) that use the traditional TCP/IP stack.

Scaling of service chains has been investigated in a single server, a computing cluster or a datacenter. CoMB [87] focus on scaling VNFs in a single server, by designing customized architecture to unify VNFs inside a single server. E2 [79] scales VNF service chains in a single datacenter, exploiting high-performance inter-VNF data paths through SDN-enabled switches. Stratos [58] jointly consider VNF placement and flow distribution within a datacenter, using on-demand VNF provisioning and VM migration to mitigate hotspots.

The management systems mentioned above cannot be directly extended to the multi-datacenter setting. One primary reason is that SDN controllers [73] are extensively used in these systems to facilitate routing, scaling and load-balancing within a datacenter. However, SDN controllers are rarely available in the WAN, except for among datacenters of a few large providers such as Google [89] and Microsoft [55]. *ScalIMS* is a NFV management system that efficiently coordinates service chain deployment and scaling, as well as flow routing, across multiple data

centers. *ScalIMS* uses similar methodologies as in [79] and [58] when scaling NFV service chains within a datacenter, but adopts a novel distributed flow routing approach and a proactive scaling strategy to scale NFV service chains across multiple datacenters.

Similar with *ScalIMS*, Klein [49] also scales NFV service chains across multiple datacenters. However, it focuses on scaling EPC system [11] and does not use a hybrid scaling strategy as *ScalIMS* does. Ren et al. [93] propose a VNF dynamic auto scaling algorithm for 5G networks, but it lacks a real-world implementation when compared to *ScalIMS*.

2.3 Challenges and Design Highlights

ScalIMS aims to address the following challenges, that arise when scaling service chains over multiple datacenters.

First, deciding service chain paths, as determined by the datacenters where instances of VNFs in a service chain should be deployed. The service chain path critically decides service quality of user traffic along the chain. For instance, a traffic flow sent by a user of the IMS system to another user may have two optional paths. The first path traverses a sequence of datacenters (a, c) while the second path traverses datacenters (a, b, c). The end-to-end delays on the two paths may vary over time. A multi-datacenter NFV scaling system should constantly update the service chain paths, so that a good service quality can be guaranteed for user traffic at all time.

Second, deciding scaling in/out of network functions, i.e., adding/removing instances of each VNF upon traffic rise/drop. This decision is coupled with service chain path selection in the multi-datacenter setting. If a service chain path is overloaded, instead of launching new VNF instances on the same datacenters, the system may search for available VNF instances on other datacenters, and set up new service chain paths using those instances.

Third, distributed flow routing. When a service chain path traverses multiple datacenters, it is difficult for a single controller to control the end-to-end route. When multiple SDN controllers are employed in different datacenters, they should work in coordination on constant updates of service chain paths, and correctly route user traffic towards destinations.

We make the following design decisions in *ScalIMS*. A functional overview of *ScalIMS* is given in Fig. 2.2.

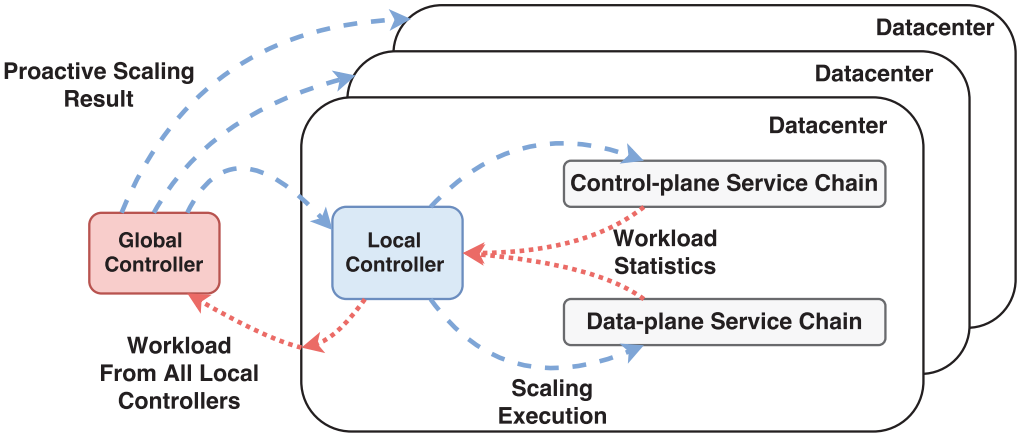


FIGURE 2.2: Functional overview of *ScalIMS*

▷ We adopt a hybrid scaling strategy, that combines proactive scaling and reactive scaling for both CP and DP service chains. We divide the system time into *scaling intervals*. At the end of each scaling interval, proactive scaling is invoked, which takes as input the predicted workload along each service chain, inter-datacenter latencies and the current VNF deployment (the numbers of instances of each VNF on each datacenter), and generates decisions on VNF scaling and service chain path deployment with bounded end-to-end delay simultaneously for the next scaling interval. Reactive scaling produces scaling decisions of each VNF based on runtime statistics of each instance within each data center. It compensates for the inaccuracy of workload prediction with proactive scaling, improving system performance under unpredicted traffic rate changes.

▷ *ScalIMS* enables a synergy of global and local controllers, to best execute the hybrid scaling strategy. The global controller runs on a standalone server. The local controllers are SDN controllers in each data center. For proactive scaling, the global controller coordinates with all local controllers: it collects statistics from each local controller, including CPU/memory usage and network traffic volume, runs the proactive scaling algorithm, generates scaling/deployment decisions, and broadcasts the decisions to local controllers. Each local controller executes the received decisions by launching new VNF instances and adjusting service chain paths. For reactive scaling, a local controller collects runtime statistics from each VNF instance running in its datacenter, and produces local, reactive scaling decision. For flow routing, a local controller uses flow tags and service chain paths received from the global controller to determine the VNF instances that a flow should traverse within its datacenter and be dispatched to in other datacenters.

▷ The architecture of *ScalIMS* follows ETSI NFV MANO framework [13], where the global controller closely resembles the NFV orchestrator and the local controller works as both VNF manager and virtual infrastructure manager. Though *ScalIMS* is designed for IMS systems, it can be easily adapted to handle other service chain systems, which provide user inter-connection services with users distributed over a large geographical span. For instance, *ScalIMS* can be adapted to manage the virtualized service chains in Evolved Packet Core (EPC) in 4G LTE network [11], by augmenting the CP service chain of EPC with an edge proxy that simulates the functionality of P-CSCF of IMS.

2.4 Scaling of Control Plane Service Chain

We first present the detailed design of *ScalIMS* in deployment and scaling of the control plane (CP) service chain.

2.4.1 Deployment and Entry Datacenter Binding

The CP service chain consists of P-CSCF and S-CSCF. We use a fixed placement strategy by deploying P-CSCF instances on every datacenter and S-CSCF instances on a fixed selected datacenter, such that the delay between the datacenter where we place S-CSCF instances and other datacenters falls within an acceptable range (the acceptable SIP transaction completion time is typically 250ms).

The rationale behind such a fixed placement strategy is the following. Even if we spread S-CSCF instances over different datacenters, each S-CSCF instance still needs to access a central HSS server and a memcached cluster to process most of the SIP transactions. This fact urges us to place S-CSCF instances together with the HSS server and memcached cluster in the same selected datacenter. Since P-CSCF instances act as relay points for user flows to access S-CSCF instances, it is desirable to place P-CSCF instances on every datacenter, to facilitate users' access to a P-CSCF instance on the closest datacenter. This fixed placement strategy also simplifies the routing of SIP messages along the CP service chain.

ScalIMS binds each user to the nearest datacenter according to his current geographical location, which is referred to as the user's *entry datacenter*. A user's CP and DP traffic can only enter and depart from the respective service chains from his entry datacenter. Such a datacenter binding is a natural design choice as each user should be pinned to a unique P-CSCF instance on a specific datacenter when he uses the IMS [1].

To implement entry datacenter binding, a DNS server is maintained in *ScalIMS*. When a user queries the IP address of an available P-CSCF instance by sending out a DNS request, the DNS server maps the user's IP address contained in the DNS request to a geographical location by querying an IP-location database (*e.g.*, IP location finder [20]), referred to as the *location service*, and then assigns a datacenter that is closest to the user's current location as his entry datacenter.

When a call is initiated between user a and user b, user b's entry datacenter is also referred to as user a's *exit datacenter*. In *ScalIMS*, each pair of datacenters may

form an entry-exit datacenter pair. CP workload and DP workload between each entry-exit datacenter pair are maintained in *ScalIMS*, which are the aggregate rates of traffic that callers associated with an entry datacenter send to callees bound to the exit datacenter, along the CP service chain and the DP service chain, respectively. The traffic rates among entry-exit datacenter pairs are used for traffic prediction and VNF instance provision.

2.4.2 Proactive Scaling

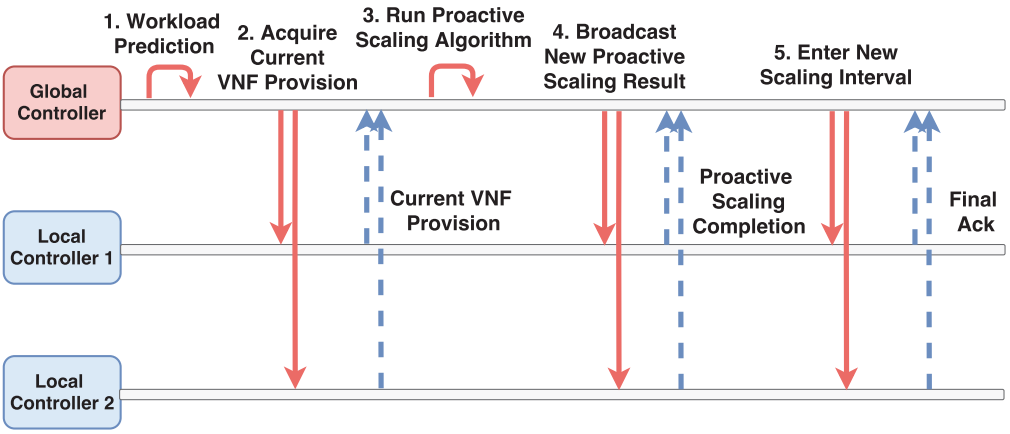


FIGURE 2.3: Proactive scaling protocol.

Proactive scaling is executed in each scaling interval according to the protocol illustrated in Fig. 2.3.

1. Workload Prediction. Workload along a CP service chain is described by the number of SIP transactions carried out between each entry-exit datacenter pair every second. When a SIP transaction finishes, the S-CSCF instance involved uses the location service to determine which entry-exit pair this transaction belongs to (according to the IP addresses of the caller and the callee). Each S-CSCF instance keeps a record of the number of SIP transactions on each entry-exit datacenter pair and reports this number to the local controller in its datacenter every second. The local controller accumulates CP workload for 5 seconds before relaying it to global controller.

At the end of a scaling interval t, the global controller predicts the workload $\hat{u}_{t+1}$ in the next scaling interval using historic data in past several intervals (10 as in our experiments), using auto regression [91]: $\hat{u}_{t+1} = \mu + \phi(u_t - \mu)$. Here μ is the mean of the historic workload values in the past several scaling intervals, u_t is the average CP workload collected in the current interval, and ϕ can be decided using the covariance of the historical workload divided by the variance of the historical workload. The workload between each entry-exit datacenter pair is predicted this way.

2. Acquire Current VNF Provision. The global controller then broadcasts a message to local controllers, asking them to send the numbers of instances of each VNF provisioned in the respective datacenter. Upon receiving this request, a local controller knows that proactive scaling computation is on going, stops its reactive scaling process (Sec. 2.4.3) so that it does not interfere with proactive scaling, and then sends its current VNF provision information to the global controller.

3. Run Proactive Scaling Alrogihm. After receiving current VNF deployment from all local controllers, the global controller computes the numbers of P-CSCF and S-CSCF instances to be deployed in each datacenter in the next scaling interval. Since all S-CSCF instances are placed in the same datacenter, the number is decided by dividing the total predicted workload between all pairs of entry-exit datacenters by the processing capacity of S-CSCF. The number of P-CSCF instances to be deployed in a datacenter is computed by dividing the overall predicted workload for the entry-exit datacenter pairs, which use this datacenter as either entry or exit datacenter, by the processing capacity of P-CSCF.

4. Broadcast Proactive Scaling Result. The global controller then broadcasts the computed numbers to local controllers. A local controller sends a completion message to the global controller, after creating new VNF instances (scale-out) or enqueueing unused VNF instances to the respective buffer queues (scale-in), according to the received numbers.

5. Enter New Scaling Interval. After the global controller receives completion messages from all local controllers, it broadcasts an "enter new scaling

interval" message to all local controllers. After receiving this message, a local controller increments its scaling interval index by 1, and shuts down some VNF instances from the head of the buffer queues. Then the local controller sends a final acknowledgement to the global controller. Upon receiving all final acknowledgements, the global controller increases its scaling interval index by 1.

Buffer Queue. In each datacenter, a double-ended buffer queue is maintained to temporarily hold unused VNF instances, with one queue for one type of VNF. When a VNF instance is to be removed, instead of directly shutting it down, the local controller tags it with the index of the current scaling interval, and enqueues it to the tail of the respective buffer queue. Once a VNF instance is enqueued, no more flows will be routed to it. Whenever more instances of a VNF are to be established, if there are available instances in the buffer queue of this VNF, buffered instances will be popped out from the tail of the queue, and transformed back to working instances, to fulfil the demand as much as possible. The purpose is to avoid creating new VNF instances frequently and improve flow loss rate, as the typical VNF creation time can last a few seconds and has a bad influence on flow loss rate. Unused buffered VNF instances are destroyed after τ scaling intervals (τ is set to 10 in our implementation), in Step 5 above.

2.4.3 Reactive Scaling

An agent running on each VNF instance reports to the local controller runtime statistics of the instance, e.g., CPU usage, memory usage and the number of input packets, in each second. The local controller maintains time series of these statistics for each VNF instance. It decides whether CPU, memory, or network is overloaded during the past several seconds by comparing the respective statistics with a threshold (Sec. 2.6). If overload is persistently identified for at least two types of statistics (*e.g.*, CPU and network usage) for some consecutive time (5 seconds as in our experiments), then that VNF instance is reported as overloaded. We make the decision using two types of statistics in order to eliminate false alarms brought by examining a single statistics. The local controller then avoids routing new traffic flows (*i.e.*, new calls) to overloaded instances if there

are other available instances. In a datacenter, if the states of a majority of instances of a VNF are "overloaded", scale-out is triggered by adding one new instance of that VNF. Note that no scale-in decisions (i.e., removing instances) are made reactively. They are solely handled by the proactive scaling protocol, as it improves system stability during workload fluctuation.

2.4.4 Flow Routing on Control Plane

When a user connects to the IMS system, he first issues a query to a DNS server, which determines the entry datacenter of this user and obtains the IP address of an available P-CSCF instance (non-overloaded) by querying the local controller of the entry datacenter. A P-CSCF instance learns the IP addresses of several available S-CSCF instances (non overloaded) by querying the local controller of the datacenter hosting S-CSCF instances and distributes its up-stream requests to these S-CSCF instances evenly. It regularly (every 30s in *ScalIMS*) updates the connections to up-stream S-CSCF instances by querying the local controller to obtain an updated view of S-CSCF instances.

A CP flow for establishment of a call (i.e., a SIP INVITE transaction) runs as follows. The caller sends out a SIP INVITE message, carrying caller's source IP address, receive port and send port, to the assigned P-CSCF instance. The P-CSCF instance forwards the message to one connected S-CSCF instance. The S-CSCF instance queries the HSS database to obtain the P-CSCF instance assigned to the callee that is saved when callee registers himself, and forwards the SIP INVITE message to that P-CSCF instance. The P-CSCF instance assigned to the callee modifies the source IP field in the message to an IP address located on the callee's entry datacenter, so that the callee can learn an IP address located on callee's entry datacenter. The P-CSCF instance then sends the modified SIP INVITE message to the callee. The callee responds with a SIP OK message, going through the same service chain in the reversed direction. When the SIP OK message passes through caller's entry datacenter, the P-CSCF instance modifies source IP field in the message to an IP address located on caller's entry datacenter as well. When such a SIP INVITE transaction ends, the caller and the callee use

TABLE 2.1: Mappings saved on local controller

Controller on Caller Entry DC	1. (caller IP, caller send port)→(callee IP)
	2. (callee IP, callee send port)→(an IP on caller's entry datacenter, caller IP, caller receive port)
Controller on Callee Entry DC	3. (callee IP, callee send port)→(caller IP)
	4. (caller IP, caller send port)→(an IP on callee's entry datacenter, callee IP, callee receive port)

the learned IP addresses as the destination IP addresses of the DP media flows and send their media flows to their entry datacenters, from where the media flows enter the DP service chain.

Besides SIP message modification, a P-CSCF instance sends two mappings (Table 2.1) to its local controller after it has received the SIP OK message. The local controller saves these mappings for use when processing the DP flows (Sec. 2.5).

2.5 Scaling of Data Plane Service Chain

The DP service chain adopts the same reactive scaling mechanism as discussed in Sec. 2.4.3. For proactive scaling, the same steps as shown in Fig. 2.3 is followed, with the following differences.

First, the proactive scaling algorithm for DP service chain, running in Step 3 in Fig. 2.3, not only decides how VNF instances are provisioned in each datacenter, but also updates the service chain path (Sec. 2.5.1) between each entry-exit datacenter pair. The new proactive scaling algorithm will be discussed in Sec. 2.5.2.

Second, workload along a DP service chain is described as the number of packets transmitted over each entry-exit datacenter pair every second. The local controller acquires input DP workload using workload measuring OpenFlow rules installed on the SDN switch at each datacenter, and reports DP workload measurements to the global controller every second. Local controllers also constantly measure inter-datacenter delays through a ping test among each other, and report the ping delays to the global controller every second. In Step 1 of Fig. 2.3,

not only workload but also delays between datacenters are predicted for the next interval, using the same approach as discussed in Sec. 2.4.2.

Next, each local controller also acts as the SDN controller to manage DP flow routing within the respective datacenter. When a local controller receives DP proactive scaling results in Step 4 of Fig. 2.3, it immediately adds/removes DP VNF instances accordingly, but saves the new DP service chain paths and uses them for routing only after receiving the "enter new scaling interval" message in Step 5 of Fig. 2.3 (Sec. 2.5.4).

2.5.1 DP Service Chain Path

ScalIMS employs one DP service chain path for all the DP media flows sharing the same entry-exit datacenter pair. A service chain path is a sequence of datacenters $l[0], \ldots, l[m+1]$, where $l[0]$ and $l[m+1]$ are the indexes of the entry datacenter and exit datacenter, respectively, and $l[i], 1 \leq i \leq m$, is the index of the datacenter hosting the ith VNF in a m-stage service chain. For example, the DP service chain in our implementation of *ScalIMS* is "firewall (stage 1)→IDS (stage 2) → transcoder (stage 3)", and a service chain path is a sequence of 5 datacenters. The DP proactive scaling algorithm constantly adjusts the service chain path for each entry-exit datacenter pair, to efficiently utilize deployed VNF instances.

The definition of service chain path augments an actual service chain with a virtual entry stage 0 and a virtual exit stage $m + 1$. These two stages are forced to be placed on the entry and exit datacenter respectively, so that entry and exit datacenters are guaranteed to be connected together. The two virtual stages have infinite capacities.

Each service chain path should satisfy two conditions. (i) Looplessness: if datacenter i hosts both stages x and y, $x < y$ on the service chain, then datacenter i must host stage z, where $x < z < y$ too; otherwise, a routing loop is created on the inter-datacenter network, which increases end-to-end delay and wastes important inter-datacenetr network bandwidth. There is no need for *ScalIMS* to tackle routing loops inside a datacenter, as routing loops inside datacentres

Algorithm 1: DP Proactive Scaling Algorithm

Input: Predicted delay between each datacenter pair, predicted workload of each entry-exit datacenter pair, current VNF deployment, current service chain paths

Output: New VNF instance provisioning, new service chain paths

1 Compute total available processing capacity of instances of each VNF in each datacenter;
2 **foreach** *entry-exit datacenter pair p, p.entry* $\neq$ *p.exit* **do**
3 **if** *there is enough VNF capacity on p's current service chain path and the end-to-end delay threshold is not violated along p's current path* **then**
4 use p's current service chain path as new path and reduce available processing capacities of VNFs on p's new path by predicted workload;
5 **foreach** *p, p.entry* $\neq$ *p.exit* && *p's new service chain path has not been determined* **do**
6 compute a new path for p using Alg. 2;
7 **if** *there is not enough capacity on p's new path* **then**
8 **foreach** *datacenter d on the new path* **do**
9 create $\lceil \max\{0, Q - Q'\}/C \rceil$ new instances of the VNF that datacenter d hosts for p, where Q is p's predicted workload, Q' is the total capacity of the VNF in d and C is per-instance processing capacity of that VNF;
10 reduce available processing capacities of instances of the VNF in d by predicted workload;
11 **foreach** *p, p.entry* = *p.exit* **do**
12 use p's current service chain path as new path, and carry out same actions as in line 7-10 ;
13 scale in by enqueuing un-used VNF instances to the respective buffer queues;

are not common and can be resolved by method described in [53]. (ii) Bounded end-to-end delay between the entry datacenter and the exit datacenter along the service chain path, by a pre-defined threshold.

2.5.2 DP Proactive Scaling Algorithm

The DP proactive scaling algorithm is given in Alg. 1, which computes a new service chain path for the next scaling interval, for each entry-exit datacenter pair. The algorithm first tries to reuse as many existing service chain paths as possible based on the current VNF provisioning, as long as the capacity is sufficient to handle predicted workload and the end-to-end delay threshold is still guaranteed (lines 2-4). In case that an existing service chain path can not be reused, a new service chain path is computed using Alg. 2 (lines 5-6), and scale-out is carried out if there is a shortage of VNF processing capacities (lines 7-10). For an entry-exit datacenter pair p where the entry and exit datacenters are the same, the entire service chain path of p is always deployed in this datacenter (lines 11-12). Finally, scale-in is carried out to enqueue un-used VNF instances into the buffer queue (line 13).

2.5.3 Service Chain Path Computation

Algorithm Overview. Alg. 2 presents the algorithm to compute a good service chain path between a given entry-exit datacenter pair, that aims to minimize the number of new VNF instances to be created while satisfing the end-to-end delay requirement. For a service chain of m VNFs (stages) and n datacenters, exhaustive search to identify such a service chain path incurs $O(m^n)$ running time. Instead, Alg. 2 seeks to optimistically find a good path in $O(mn)$ time.

In Alg. 2, the *record* list retains the service chain path under investigation (line 3). The search starts by looping through all datacenters except the exit datacenter, to decide the one for hosting instances of stage-1 VNF (lines 2-3). For each subsequent VNF in the service chain, a datacenter (except the exit datacenter) with the largest available processing capacity of the respective VNF is chosen (lines 4-6). We might have created a loop in the service chain path. If so, the loop is eliminated (lines 7-8) using the method to be discussed next. Whenever the datacenter to host stage-x VNF is determined, a candidate path is produced

Algorithm 2: Service Chain Path Computation

Input: Predicted inter-datacenter delays, an entry-exit datacenter pair $p = (entry, exit)$, p's predicted workload, p's current service chain path, current available processing capacities of VNF instances, the service chain of m stages, n datacenters

Output: p's new service chain path

```
1  minProvPath = p's current path;
2  for v = 0, ..., exit − 1, exit + 1, ..., n − 1 do
3      record[0] = entry, record[1] = v, record[m + 1] = exit;
4      for x = 2, ..., m do
5          find out datacenter v_1 (v_1 ≠ exit) that has the largest available
           capacity for stage-x VNF;
6          record[x] = v_1;
7          if there is path loop on record then
8              eliminate loop by adjusting record;
9          path[0, ..., x] = record[0, ..., x];
10         path[x + 1, ..., m + 1] = exit;
11         if path leads to a smaller number of new VNF instances to be
           created for hanlding predicted workload than minProvPath and
           satisfies end-to-end delay requirement then
12             minProvPath = path
13     path[0] = entry, path[1] = v, path[2, ..., m + 1] = exit;
14     check whether path should be assigned to minProvPath as in lines
       11-12;
15 path1[0] = entry, path1[1, ..., m + 1] = exit;
16 path2[0, ..., m] = entry, path2[m + 1] = exit;
17 check whether path1 or path2 should be assigned to minProvPath as in
   lines 11-12;
18 if minProvPath violates end-to-end delay requirement then
19     find out the shortest-delay datacenter path between entry and exit;
20     run stage placement algorithm in Alg. 3 and assign the identified
       service chain path to minProvPath;
21 return minProvPath;
```

by assuming all the rest VNF stages ($x+1, \ldots, m$) will be hosted in the exit datacenter (lines 9-10). The candidate path is examined by calculating the number of new VNF instances to be created along this path and the end-to-end delay of this path. If the candidate path incurs addition of fewer new VNF instances

than the current best candidate path while satisfying end-to-end delay requirement, we retain it in $minProvPath$ (lines 11-12). The algorithm also checks some naive candidate paths that are not generated by the search loop (lines 13-17). Finally, it is possible that all candidate paths found so far fail to satisfy the delay requirement. If so, we compute the shortest end-to-end delay path using a shortest path algorithm, and run the stage placement algorithm in Alg. 3 to produce the service chain path (lines 18-20).

Loop Elimination. To eliminate a loop in the path introduced in lines 5-6 of Alg. 2, we adjust the *record* list following two ways: (i) place all VNF stages involved in the loop on the datacenter at the start of the loop. Suppose the path with loop is $(1, 2, 4, 2, 5)$. It is adjusted to $(1, 2, 2, 2, 5)$, i.e., the datacenter hosting stage 2 is changed from datacenter 4 to datacenter 2. (ii) Choose a new datacenter to replace the datacenter that leads to the loop, which has the second largest available capacity for hosting the respective VNF and will not create another loop. Suppose datacenter 3 in the above example has the second largest capacity of stage-3 VNF. Then the path is adjusted to $(1, 2, 4, 3, 5)$. The *record* list is adjusted in both ways and the two resulting lists are compared. The list requiring fewer new VNF instances is selected.

Stage Placement Algorithm. Alg. 3 gives the stage placement algorithm used in line 20 of Alg. 2, which calculates a service chain path, i.e., the sequence of datacenters to host VNF stages on the service chain, that minimizes the number of new VNF instances to be created on the path.

In Alg. 3, $num(i, j)$ is the number of new stage-j VNF instances to be created, if stage j is to be deployed on datacenter $d[i]$. $num(i, j)$ is computed in Eqn. 2.1 based on the following observation: if datacenter $d[i]$ is chosen to host stage j, then the remaining number of yet-to-be-placed stages, $(m+2)-(j+1)$, must be no smaller than the remaining number of datacenters which do not host any stage yet, $k-(i+1)$, as otherwise the resulting service chain is not able to traverse the shortest-delay datacenter path in sequence.

$N(i, j)$ is the minimum number of instances of stage 0 to stage j VNFs, if stage j is to be deployed on datacenter $d[i]$. The computation of $N(i, j)$ is a dynamic

Algorithm 3: Stage Placement Algorithm

Input: The shortest-delay datacenter path $d[0...k-1]$ as a list of distinct datacenter indices between entry-exit datacenter pair $(d[0], d[k-1])$; DP service chain with $m+2$ stages (augmented with virtual entry and exit stage discussed in Sec. 2.5.1) and per-instance capacity $C_j, 0 \leq j \leq m+1$, of stage-j VNF; overall processing capacities $Q'_{i,j}$ of all stage-j VNF instances in datacenter $d[i]$; predicted workload Q for entry-exit datacenter pair $(d[0], d[k-1])$

Output: service chain path $l[0], \ldots, l[m+1]$ for entry-exit datacenter pair $(d[0], d[k-1])$.

1 Calculate $N(i,j), \forall i = 0, \ldots, k-1, j = 0, \ldots, m+1$, as follows:

$$num(i,j) = \begin{cases} 0, & \text{if } Q'_{i,j} \geq Q \text{ and } m-j+1 \geq k-i-1 \\ \lceil (Q - Q'_{i,j})/C_j \rceil, & \text{if } Q'_{i,j} < Q \text{ and } m-j+1 \geq k-i-1 \\ +\infty, & \text{if } m-j+1 < k-i-1 \end{cases} \tag{2.1}$$

$$N(0,0) = 0, N(i,0) = +\infty, i = 1, \ldots, k-1 \tag{2.2}$$

$$N(0,j) = N(0,j-1) + num(0,j), j = 1, \ldots, m+1 \tag{2.3}$$

$$N(i,j) = min\{N(i-1,j-1) + num(i,j), N(i,j-1) + num(i,j)\}, \quad i = 1, \ldots, k-1, j = 1, \ldots, m+1 \tag{2.4}$$

2 Backtrack from $N(k-1, m+1)$ to derive the service chain path $l[0], \ldots, l[m+1]$;

3 **return** $l[0], \ldots, l[m+1]$;

programming problem (Eqn. (2.2) to (2.4) in Alg. 3). Eqn. (2.2) to Eqn. (2.3) are base cases, initialized according to the fact that stage 0 is on the entry datacenter $d[0]$. Eqn. (2.4) is derived based on the following observation: if datacenter $d[i]$ is chosen to host stage j, then stage $j-1$ can only be hosted on either datacenter $d[i-1]$ (the previous datacenter on the datacenter path) or datacenter $d[i]$.

When calculating $N(i,j)$ in Eqn. (2.4), we can construct a link connecting $N(i,j)$ to either $N(i-1,j-1)$ or $N(i,j-1)$, depending on whether $N(i-1,j-1) + num(i,j)$ or $N(i,j-1) + num(i,j)$ is smaller. This indicates whether datacenter $d[i-1]$ or $d[i]$ should host stage $j-1$, if stage j is to be deployed on datacenter $d[i]$. After $N(k-1, m+1)$ is computed (recall stage $m+1$ must be placed on

the exit datacenter $d[k-1]$), we can backtrack to obtain the best service chain path.

2.5.4 Flow Routing On Data Plane

Each DP media flow enters the system from the entry datacenter, is routed through the datacenters on its service chain path in a fully distributed fashion, and then departs from the exit datacenter. The DP media flow sent by the caller is used as an example to explain the distributed routing process in the following paragraphs.

Enter Service Chain Path. The caller learns an IP address located in his entry datacenter when the SIP OK message is received (Sec. 2.4.4). This IP address is used as the destination IP address to send the DP media flow to the entry datacenter.

Route through Service Chain Path. The local controller in a datacenter decides the service chain path used by a media flow, when the first packet of the flow arrives at the datacenter and triggers an OpenFlow Packet_IN message at the local controller. The local controller examines whether the packet header contains a special tag. In our implementation of *ScalIMS*, the tag is added to the destination port field by a special OpenFlow rule in the flow's entry datacenter. The tag contains the indices of the flow's entry and exit datacenters, and the current scaling interval number modulo 4.

If the special tag is not present, it indicates that this datacenter is the entry datacenter of the flow. Then the local controller uses the mapping 1 in Table 2.1, saved during the SIP INVITE transaction, to get the callee IP address and obtains the index of the flow's exit datacenter using the location service (Sec. 2.4). The local controller then selects the service chain path corresponding to the flow's entry-exit datacenter pair, recorded for the current scaling interval. If there is one or multiple VNF stages hosted in this entry datacenter, the local controller selects a sequence of VNF instances for those VNF stages, according to a smallest workload-first principle for load balancing. Next, it installs several OpenFlow

rules on the SDN switches in the datacenter, to route the flow along the selected sequence of VNF instances. The installed OpenFlow rule will also add the special tag to the header of all incoming packets of the flow. If the datacenter is not the exit datacenter, the flow is then routed to the next datacenter on the service chain path, through a VxLAN tunnel, that is set up between each pair of datacenters. Each VxLAN tunnel connecting a pair of datacenters is constructed over the inter-datacenter network connecting that pair of datacenters. Different service chain paths share the same VxLAN tunnel if they need to traverse the same pair of datacenters.

If the special tag is present, the datacenter hosts VNF stage(s) on the service chain path. The local controller selects the service chain path indicated in the tag. Then it selects VNF instance(s) for the VNF stage(s), installs OpenFlow rules to route the flow through the instance(s), and routes flow out to the next datacenter (if it is not the exit datacenter), in the same way as discussed in the previous case. The use of the special tag ensures that each flow is routed through a consistent service chain path.

Exit from Service Chain Path. At the exit datacenter, the local controller uses mapping 4 in Table 2.1 to retrieve the IP address on callee's entry datacenter, callee IP and callee's receive port. If the exit datacenter also hosts VNF stage(s) in the service chain, the flow is routed through the VNF instance(s). Then the local controller uses an OpenFlow rule to perform an address translation, which replaces the flow's source IP address by the IP address on callee's entry datacenter, flow's destination IP address by the callee IP, and flow's destination port by the callee's receive port. The flow is then delivered to the callee over the Internet.

2.5.5 Handling Scaling Interval Inconsistency

In Step 5 of Fig. 2.3, the global controller sends an 'enter new scaling interval' message to all local controllers to advance the scaling interval index on each local controller by 1. Due to network delay, the local controllers may not receive the

message at the same time, resulting in temporary inconsistency of scaling interval indices on different local controllers.

When a local controller is processing a flow, it may find that the encoded scaling interval in the header of the received flow packets does not match its current scaling interval index: one scaling interval ahead, or one scaling interval behind. In the first case, since the service chain paths for the next scaling interval are broadcast and saved at all local controllers before the 'enter new scaling interval' message can be sent and received by any local controller, the current local controller must have received the service chain paths for the next scaling interval, though not yet receiving the 'enter new scaling interval' message; it can use the service chain path for the next scaling interval to route the flow. In the second case, the local controller still uses the service chain path for the previous scaling interval to route the flow.

Since the difference between the scaling interval in the tag of received flow packets and the scaling interval on a local controller is among $-1, 0$, and 1, the scaling interval encoded in the tag is the actual value modulo 4, to reduce the number of bits required to only 2.

2.6 Implementation and Evaluation

2.6.1 Implementation

We implement a prototype of *ScalIMS* in Java and deploy *ScalIMS* code on both local controller and global controller. The local controller is implemented as a module in the FloodLight SDN controller [15]. The global controller is implemented as a multi-threaded server program, which communicates with local controllers over regular sockets. We also implement a traffic generator based on PJSIP [30]. We deploy one traffic generator in each datacenter, producing user arrivals bound to the datacenter at a configurable rate. A global traffic generator coordinator receives notifications of generated users and pairs up users in a first-come-first-match manner. A call process is launched between every

Table 2.2: VNF Capacity and Overload Threshold

VNF	Capacity	CPU threashold	Memory threshold	Input pkts/s threshold
P-CSCF	500 tran/s	70%	50%	1000 pkts/s
S-CSCF	200 tran/s	70%	50%	400 pkts/s
Firewall	35000 pkts/s	90%	50%	35000 pkts/s
IDS	20000 pkts/s	90%	50%	20000 pkts/s
Transcoder	15000 pkts/s	90%	50%	15000 pkts/s

pair of paired users, which includes SIP transactions to establish a call, followed by a one-minute voice call at the bit rate of 80kbit/s, as well as necessary SIP transactions to shutdown the call.

We use P-CSCF, S-CSCF and HSS components from the Project Clearwater [31] as our CP VNFs. I-CSCF is omitted by *ScalIMS* as I-CSCF is optional and merged into S-CSCF by Project Clearwater. The DP service chain contains a firewall (implemented using user space Click [72]), an intrusion detector (Snort IDS [39]) and a transcoder (implemented using user space Click). Each network function runs on a QEMU/KVM VM. A VM to run a CP VNF is configured with 1 core and 2GB RAM. A VM to run a DP VNF is configured with 2 cores and 2GB RAM. The capacity and overload threshold (to decide scale-out) of each instance of each VNF are given in Table 2.2, which are obtained by stress testing VNF instances to overloaded states.

2.6.2 Evaluation in IBM SoftLayer Cloud

We evaluate the performance of *ScalIMS* on IBM SoftLayer Cloud [40] by renting one bare-metal server in each of the 4 Softlayer datacenters, located in Tokyo, Hong Kong, London and Houston, respectively. Each server is equipped with two 6-core 2.4GHz Intel CPU, 64GB RAM, and 1TB SATA disk. In the SoftLayer cloud, servers in different datacenters are connected through a global private network [40], which provides a Gigabyte throughput. *ScalIMS* creates a VxLAN tunnel mesh in the private network to route DP traffic. CP VNFs are connected directly over the private network. The difference in the connection method is

because CP VNFs are addressable at L3 layer (IP layer) whereas DP VNFs are only addressable at L2 layer (Ethernet layer).

We evaluate the performance of *ScalIMS* based on two groups of metrics. (1) The total number of new VNF instances created over time: the smaller the number is, the more cost/resource effective *ScalIMS* is; (2) QoS of user traffic including the SIP transaction completion time for CP flows, the RTT and loss rate for DP flows: a large number of QoS data is collected and their cumulative distribution function (CDF) curves are shown (as in Fig. 2.5, 2.6, 2.7), so that the higher the CDF curve is on a figure, the better the QoS is.

We compare the performance achieved by using proactive scaling only (the local controller does not react to overload of instances), reactive scaling only (the global controller initializes a good set of service chain paths, but no subsequent proactive scaling decisions are sent to local controllers), and both proactive and reactive scaling enabled (*i.e.*, the combined scaling strategy of *ScalIMS*). The reactive-scaling-only is the base-line case as the reactive scaling used by *ScalIMS* inside a single datacenter is similar to that of [58, 79]. Each scaling interval is set to be 50 seconds long, and each buffer queue retains an unused VNF instance for at most 10 scaling intervals. The maximum allowed end-to-end flow delay is 250ms.

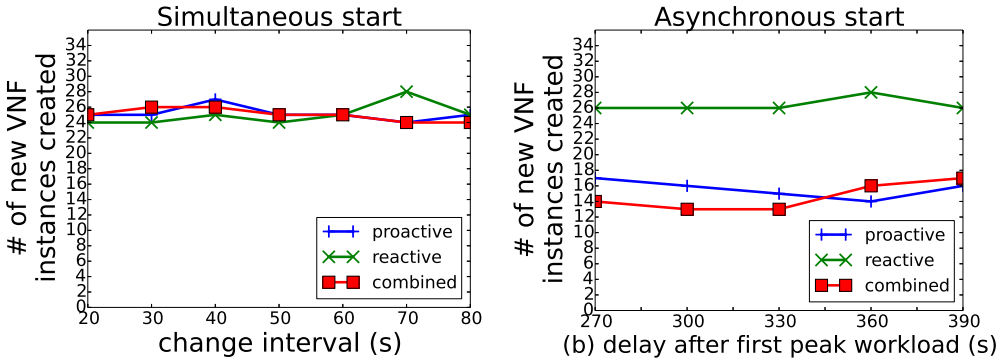


FIGURE 2.4: Overall number of new VNF instances created: (a) simultaneous start; (b) asynchronous start.

2.6.2.1 Simultaneous Start

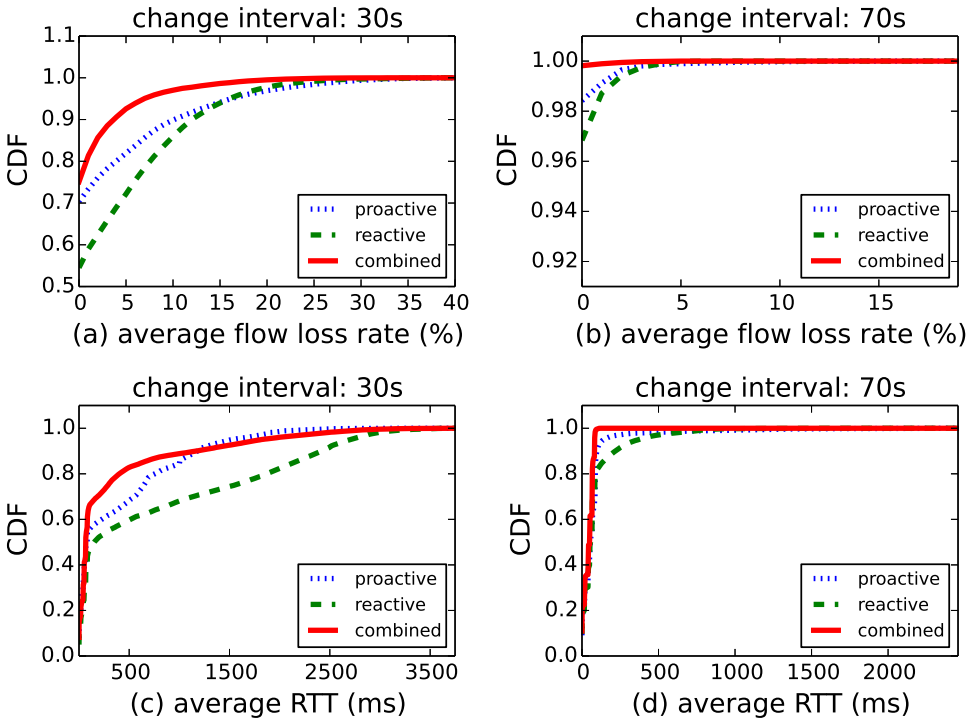


FIGURE 2.5: QoS of DP flows: simultaneous start.

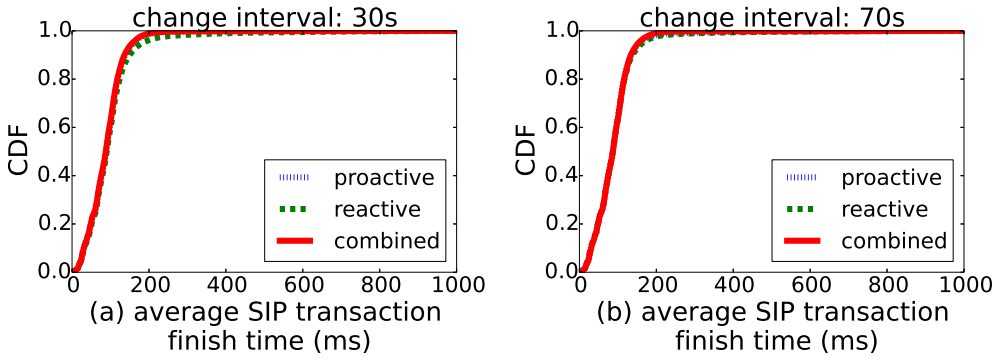


FIGURE 2.6: QoS of CP flows: simultaneous start.

In this set of experiments, each traffic generator starts generating users simultaneously at the rate of 1 $users/s$, which gradually increases to 15 $users/s$ and

then decreases to 6 *users/s*. The rate change happens once every *change interval*. The duration of change intervals is set to different values in different experiments. In this way, the peak workload arrives at each datacenter almost concurrently.

Fig. 2.4(a) shows that the total number of VNF instances provisioned (for both CP and DP service chains) does not differ much under the three schemes. Since the maximum workload on each datacenter arrives at around the same time, the proactive scaling algorithm has no opportunity to decrease the number of provisioned VNF instances by adjusting the service chain path, and therefore the numbers are similar.

Fig. 2.5 plots CDFs of the number of DP flows at different average packet loss rates and average RTTs, and shows that the combined scaling strategy of *ScalIMS* out-performs the other two strategies in terms of DP traffic quality. Pure reactive scaling adds new instances only when overload occurs. During the boot-up time of new instances, traffic continues to arrive at the overloaded VNF instances, resulting in a high packet loss rate and then high RTT. Pure proactive scaling adjusts VNF instances once every scaling interval. During each scaling interval, increased workload may have overloaded the system. The best performance is achieved combining both proactive and reactive scaling.

Fig. 2.6 shows that the average SIP transaction completion time is similar with the three schemes. This is because scaling of CP service chains is not triggered as often as DP service chains, since each instance of a CP VNF is able to handle a large number of SIP transactions.

2.6.2.2 Asynchronous Start

In this set of experiments, the traffic generator in the Tokyo datacenter is started first, followed by the traffic generators in Hong Kong, in Houston and then in London. In this way, the peak workload in different datacenters does not occur at the same time. Each traffic generator increases its user generation rate from

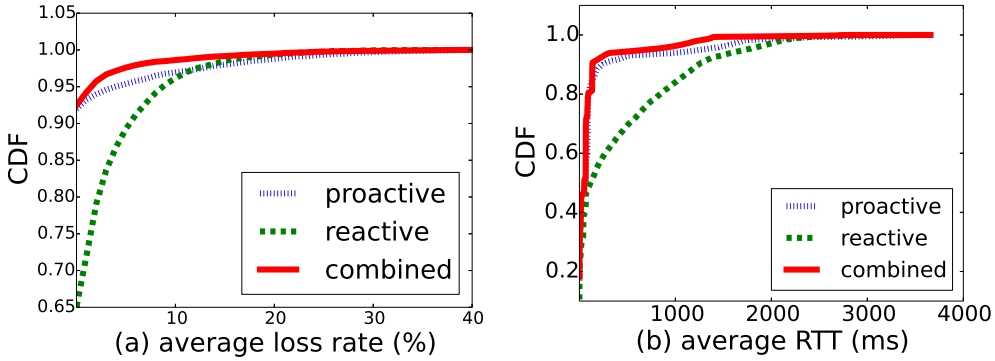


FIGURE 2.7: QoS of DP flows: asynchronous start.

5 *users*/*s* to 15 *users*/*s* and then reduces it to 5 *users*/*s*, with a 30s change interval.

In Fig. 2.4(b), the start delay between traffic generators in Tokyo and in Hong Kong is set according to the values on the x axis, while start delays between traffic generators in Hong Kong and in Houston, and between traffic generators in Houston and in London, are set to 120s. We observe that the number of VNF instances created with the combined scaling approach is similar to proactive scaling approach, but always smaller than reactive scaling approach. Fig. 2.7 shows that the traffic quality is the best with the combined approach as well. The performance of CP SIP transaction completion time under asynchronous start is very similar to the results presented in Fig. 2.6.

2.6.2.3 Explanation

Why can the combined scaling strategy perform well even when it creates a smaller number of VNF instances? The Tokyo datacenter sees the peak workload first, leading to the provisioning of many VNF instances in the datacenter, which later become redundant and will be buffered for 10 scaling intervals. When peak workload arrives at other datacenters, the global controller can re-use the buffered VNF instances by creating service chain paths traversing the Tokyo datacenter. These phenomena are exhibited in Fig. 2.8.

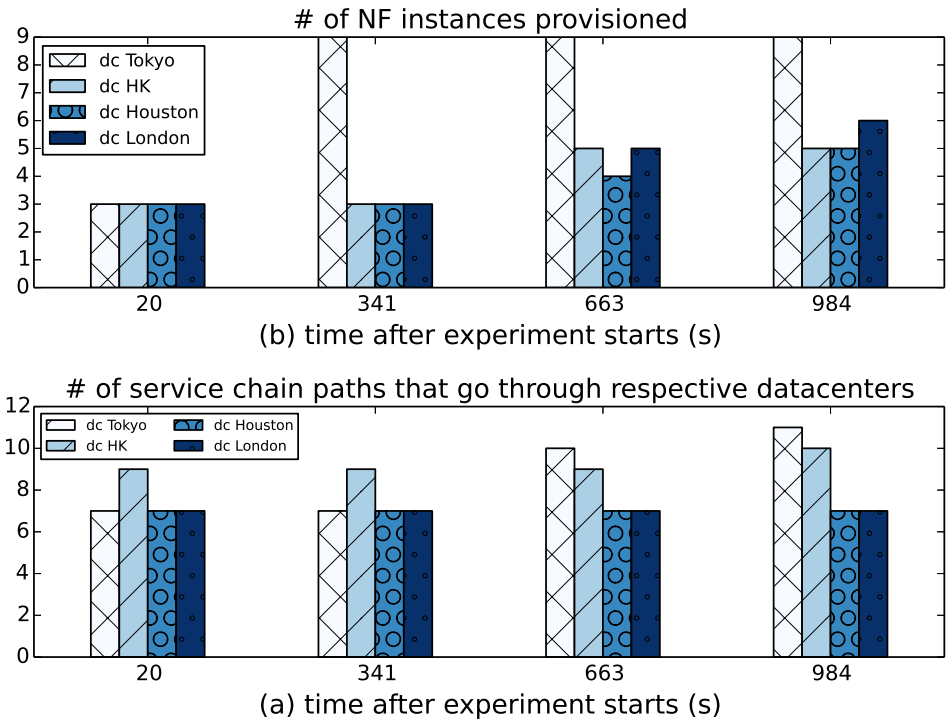


FIGURE 2.8: Number of VNF instances in/service chain paths through each datacenter: asynchronous start.

2.7 Summary

This chapter proposes *ScalIMS*, a NFV management system designed to deploy and scale service chains spanning geo-distributed datacenters, using the case of an IMS. *ScalIMS* features joint proactive and reactive scaling of DP and CP service chains, for timely and cost-effective provisioning of practical network services. Evaluation of our prototype implementation on IBM SoftLayer cloud shows that: (i) *ScalIMS* improves QoS of user traffic by a large margin when compared with pure reactive scaling; (ii) when peak workload arrives asynchronously over the geographic span, *ScalIMS* effectively reduces the total number of VNF instances provisioned while guaranteeing excellent QoS.

Chapter 3

NFVactor: A Resilient NFV System using the Distributed Actor Model

3.1 Introduction

Network function virtualization (NFV) advocates moving *network functions* (NFs) out of dedicated hardware middleboxes and running them as virtualized applications on commodity servers [28]. To enable effective large-scale deployment of virtual NFs, a number of NFV management systems have been proposed in recent years [48, 52, 58, 79, 87, 95], implementing a broad range of management functionalities. Among these functionalities, resilience guarantee, supported by flow migration and failure recovery mechanisms, is of particular importance in practical NFV systems.

Resilience to failures [85, 88] is crucial for stateful NFs. Many NFs maintain important per-flow states [50]: IDSs such as Bro [4] store and update protocol-related states for each flow for issuing alerts for potential attacks; firewalls [21] parse TCP SYN/ACK/FIN packets and maintain TCP connection related states for each flow; load balancers [22] may retain mapping between a flow identifier

and the server address, for modifying destination address of packets in the flow. It is critical to ensure correct recovery of flow states in case of NF failures, such that the connections handled by the failed NFs do not have to be reset – a simple approach strongly rejected by middlebox vendors [88].

Efficient flow migration [59, 83, 86] is important for long-lived flows in case of dynamic system scaling. Existing NFV systems [58, 79] mostly assume dispatching new flows to newly created NF instances when existing instances are overloaded, or waiting for remaining flows to complete before shutting down a mostly idle instance, which are efficient for short flows. Long flows are common on the Internet: a web browser uses one TCP connection to exchange many requests and responses with a web server [18]; video-streaming [14] and file-downloading [16] systems maintain long-lived TCP connections for fetching large volumes of data from CDN servers. When NF instances handling such flows are overloaded or under-loaded, migrating flows to other available NF instances enables timely hotspot resolution or system cost minimization [59].

Even though failure resilience and efficient flow migration are important for NFV systems, enabling light-weight failure resilience and high-performance flow migration within existing NF software architecture has been a challenging task.

Failure resilience in the existing systems [85, 88] is typically implemented through checkpointing: each NF process is regularly checkpointed, and if it fails, the system replays important log traces collected since the latest checkpoint to recover the failed NF. Compared to the normal packet processing delay of an NF that lies within tens of microseconds, the process of checkpointing is heavyweight and can cause extra delay up to thousands of microseconds [85, 88].

Flow migration in existing systems [59, 86] is typically governed by a centralized controller. It fully monitors the migration process of each flow by installing SDN rule to update the route of the flow and exchanging messages with the NFs over inefficient kernel networking stack [25]. However, a practical NFV system needs to manage tens of running NFs and handle tens of thousands of concurrent flows. To migrate these flows, the controller needs to sequentially execute the migration process of each flow, install a large number of SDN rules and exchange many

migration protocol messages through inefficient communication channel, which may prolong the flow migration completion time and inhibit flow migration from serving as a practical NFV management task.

Besides, enabling flow migration with existing NF software is not trivial: OpenNF [59] reports that thousands lines of patch code must be added to existing NF software [4, 41] in order to extract and serialize flow states, communicate with the controller and control flow migration. This approach mixes the logic for controlling flow migration together with the core NF logic. To maintain and upgrade such an NF, the developer must well understand both the core NF logic and the complicated flow migration process, which adds additional burden on the developer.

In this paper, we present *NFVactor*, a new software framework for building NFV systems with high-performance flow migration and lightweight failure resilience. Unlike previous systems [59, 85, 86, 88] which augment existing NF software with resilience support, *NFVactor* explores new research opportunities brought by a holistic approach: *NFVactor* provides a general framework with built-in resilience support by exploiting the distributed actor model [2], and exposes several easy-to-use APIs for implementing NFs. Internally, *NFVactor* delegates the processing of each individual flow to an unique flow actor. The flow actors run in high-performance runtime systems, handle flow processing and ensure their own resilience in a largely decentralized fashion. *NFVactor* brings three major benefits.

▷ *Transparent resilience guarantee. NFVactor* ensures that once the NFs are implemented with the provided APIs, failure resilience of the NFs is immediately guaranteed. *NFVactor* decouples resilience logic from core NF logic by incorporating resilience operations within the framework and only exposing APIs for building NFs. Using the APIs, programmers are fully liberated from reasoning about details of resilience operations, but only focus on implementing the processing logic of NFs and handling simple interaction for synchronizing shared states of NFs during resilience operations. The exposed APIs also ensure a clean

separation between the core processing logic and important NF states, facilitating resilience operations.

▷ *Lightweight failure resilience.* With the actor abstraction and cleanly separated NF states, *NFVactor* is able to replicate each flow independently without checkpointing the entire NF process. Each flow actor can replicate itself by constantly saving its per-flow state to another actor that serves as a replica. This lightweight resilience operation guarantees good throughput, short recovery time and a small packet processing delay.

▷ *High-performance flow migration.* The use of the actor model enables *NFVactor* to adopt a largely decentralized flow migration process: each flow actor can migrate itself by exchanging messages with other flow actors, while a centralized controller only initiates flow migration by instructing a runtime system about the amount of the flow actors that should be migrated. As a result, *NFVactor* is able to concurrently migrate a large number of flows among multiple pairs of runtime systems. *NFVactor* also replaces SDN switch with a lightweight virtual switch for flow redirection, simplifying flow redirection from updating SDN rule into modifying an runtime identifier number. The increased parallelism and simplified flow redirection jointly enhance the performance of flow migration.

Our major technical challenge is to build an actor runtime system to satisfy the stringent performance requirement of NFV application. Even the fastest actor runtime systems [54] may fail to deliver satisfactory packet processing performance due to their actor scheduling strategies and the use of kernel networking stack. To address this challenge, we carefully craft a high-performance actor runtime system by combining the performance benefits of (i) a module graph scheduler to effectively schedule multiple flow actors, (ii) a DPDK-based [19] fast packet I/O framework [3] to accelerate network packet processing and (iii) an efficient user-space message passing channel which completely bypasses the kernel network stack and improves the performance of both failure resilience and flow migration.

We implement *NFVactor* and build several useful NFs using the exposed APIs. Our testbed experiments show that *NFVactor* achieves 10Gbps line-rate processing for 64-byte packets, concurrent migration of 600K flows using around 700 milliseconds, and recovery of a single runtime within 70 milliseconds in case of failure. Compared with OpenNF [59], flow migration completion time in *NFVactor* can be 144 times faster. Compare with FTMB [88] for replication performance, *NFVactor* achieves similar packet processing throughput and recovery time, but with packet processing latency stabilized at around 20 microseconds. The source code of *NFVactor* is available at [43].

3.2 Motivations for Using the Actor Model

Systems that enable failure resilience [85, 88] and flow migration [59, 86] typically achieve a low level of parallelism: a centralized controller governs the migration of all the flows among multiple NF instances, while an entire NF process has to be checkpointed for replication. If we can improve the parallelism by providing an efficient per-flow execution environment, then each flow can migrate and replicate itself without full-process checkpointing and centralized migration control, leading to improved resilience performance. Such a per-flow execution environment can be modelled by actors.

The actor programming model [2, 6, 12, 34] has a long history of being used to construct massive, distributed systems [2, 34, 74, 75]. Each actor is a lightweight and independent execution unit. In the simplest form, an actor contains a global unique address, a message queue (mailbox) for accepting incoming messages, several message handlers and an internal actor state (*e.g.*, statistic counter, number of outgoing requests). An actor can send messages to other actors by referring to their addresses, process incoming messages using message handlers, update its internal state, and create new actors. Multiple actors run asynchronously as if they were running in their own threads, simplifying programmability of distributed protocols and eliminating potential race conditions that may cause

system crash. Actors typically run on a powerful runtime system [6], which can schedule millions of lightweight actors simultaneously.

The actor model is a natural fit to provide a per-flow execution environment for resilient NFV system. We can create one actor as the basic execution environment for a flow and equip the actor with necessary message handlers for service chain processing, flow state replication and migration. Then each actor can process network packets and handle its own resilience by creating new actors and exchanging messages with other actors.

There are several popular actor frameworks [6, 12, 29, 34], but none of these frameworks are optimized for building NFV systems. In our initial implementation, we built *NFVactor* on top of libcaf [6], one of the fastest actor system [54]. But the overall performance turned out to be less than satisfactory, due to its actor scheduling strategy and the use of kernel networking stack. This inspires us to customize a high-performance actor runtime system (Sec. 3.6) for *NFVactor*.

3.3 The NFVactor Framework

3.3.1 Overview

At the highest level, *NFVactor* has a layered structure as shown in Fig. 3.1. There are three key elements in *NFVactor*: (i) runtime systems (referred to as *runtime* for short) that enable flow processing using actors; (ii) virtual switches for distributing flows to runtime systems and sending flows to final destinations; and (iii) a lightweight coordinator for basic system management.

A runtime (Sec. 3.3.2) is the execution environment of flow actors, running on a Docker container [10] for quick launching and rebooting, and is assigned a globally unique ID upon creation. A virtual switch is a special runtime (Sec. 3.3.3) and serves as the gateway to runtimes. Runtimes and virtual switches are partitioned into several virtual clusters. In a cluster, the runtimes are initialized with the same service chain (Sec. 3.3.2.3) and the virtual switches dispatch flows

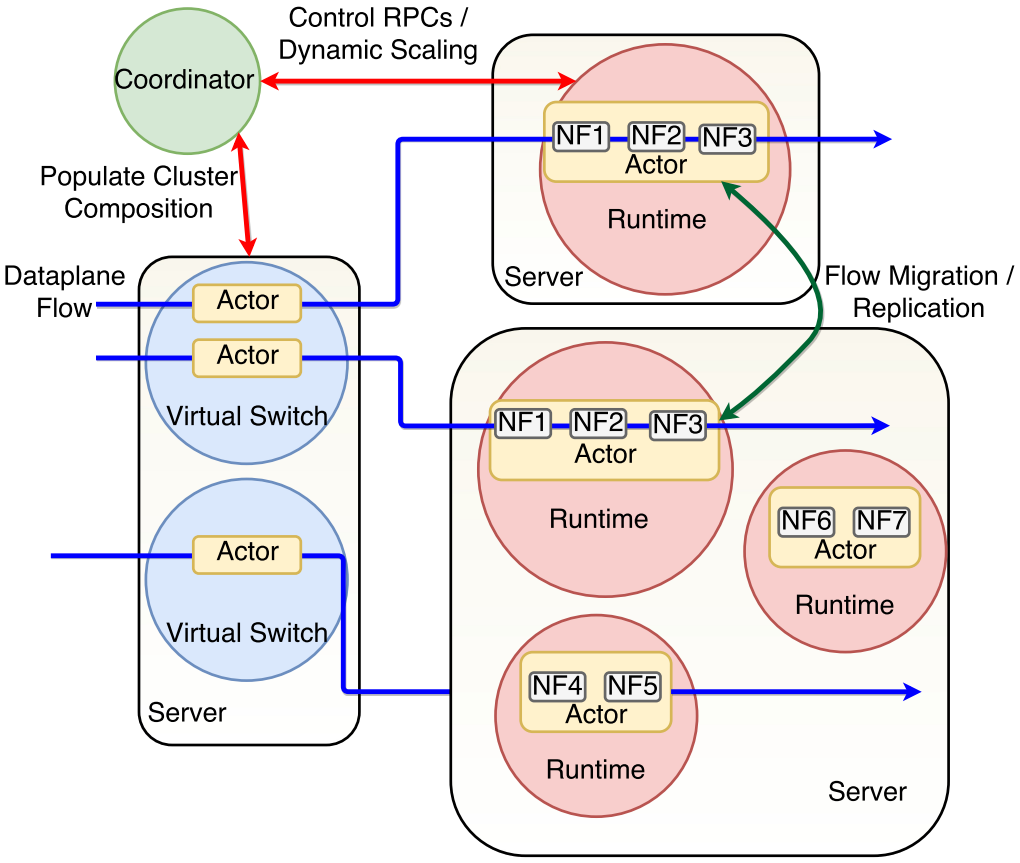


FIGURE 3.1: An overview of the basic components of *NFVactor*. Three clusters are shown in this figure: a cluster for provisioning service chain 'NF1→NF2→NF3', a cluster for service chain 'NF4→NF5' and a cluster for service chain 'NF6→NF7'

to the runtimes within the same cluster. The partitioning of virtual clusters enables *NFVactor* to simultaneously provision multiple service chains.

Each virtual switch is configured with an entry IP address. The coordinator sets up proper switching rules to direct dataplane flows to virtual switches, which further dispatch them to runtimes within the same cluster. A runtime creates a dedicated flow actor to process each flow and forward flow packet to its final destination. The coordinator also manages dynamic scaling and failure recovery of *NFVactor* by interacting with runtimes and virtual switches through a series of control RPCs (Sec. 3.3.4).

Dataplane flows can be migrated and replicated from one runtime to another runtime within the same cluster in a distributed fashion without persistent involvement of the coordinator. The details of flow migration and replication are further introduced in Sec. 3.5.

3.3.2 Runtime

NFVactor employs a carefully designed, uniform runtime system to run flow actors. Within a runtime, we adopt a *one-actor-one-flow* design principle: a dedicated flow actor is created to handle each flow received by the runtime. Packet processing by NFs and resilience operations are all implemented as reactive message handlers of the flow actor. The runtime timely schedules each flow actor to react to the various events, so that each flow actor can process flow packets and manage its own resilience in a largely decentralized fashion. Our one-actor-one-flow principle improves the parallelism of resilience operations and serves as the basis for high-performance resilience support.

3.3.2.1 Internal Structure

Fig. 3.2 shows the internal structure of a runtime. Each runtime is configured with one worker thread and one RPC thread. The worker thread actively polls the three ports shown in Fig. 3.2 and is pinned to a dedicated CPU core to minimize the performance impact caused by thread scheduling. The RPC thread responds to RPC requests sent from the coordinator, for basic system management operations (Sec. 3.3.4). The three ports are high-speed virtual NICs (ZeroCopyVPort in BESS [3]) and they are connected to a virtual L2 switch (L2Forward module of BESS) inside a physical server. The worker thread can bypass the kernel and directly fetch network packets from these ports.

3.3.2.2 Work Flow

The runtime has three basic work flows:

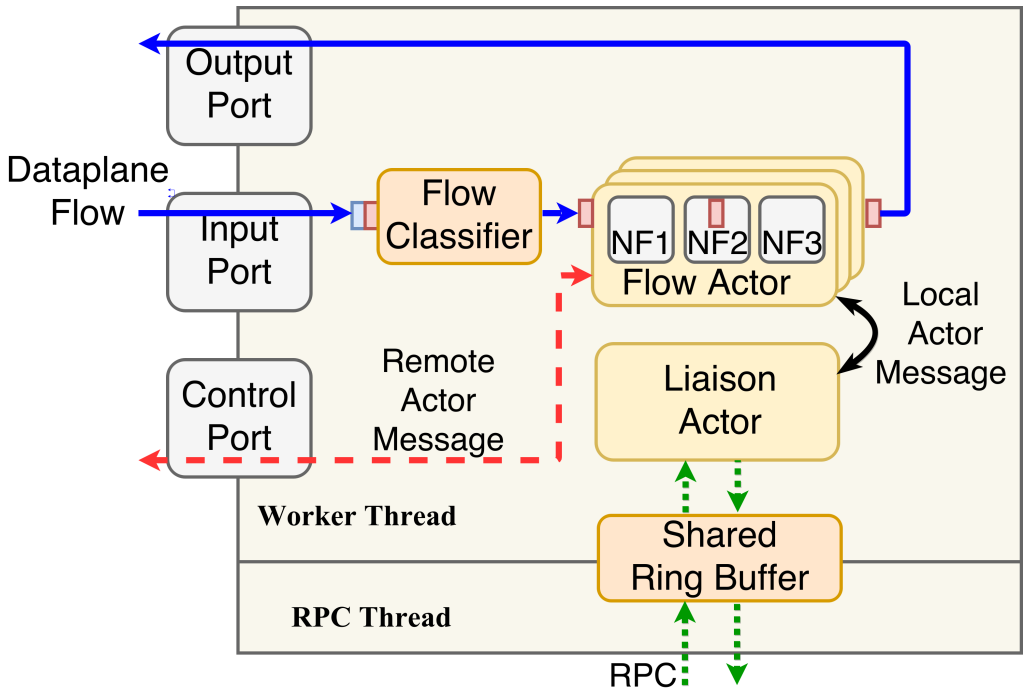


FIGURE 3.2: The internal structure of a runtime.

Process Dataplane Flows: The worker thread constantly polls dataplane flow packets from the input port. For each packet, the worker thread uses the 5-tuple of the packet (*i.e.*, source IP address, destination IP address, transport-layer protocol, source port and destination port) to retrieve the corresponding flow actor and sends the packet to the flow actor for processing. Running in its own logical thread, the flow actor processes the packet along the configured service chain and then sends the processed packet out from the output port.

Process Remote Actor Messages: During distributed flow migration and replication, *remote actor messages* are exchanged among actors running on different runtimes. The runtime is equipped with a reliable message passing channel (Sec. 3.6) to reliably send and receive remote actor messages over the control port. The received remote actor messages are handed over to the destination actors for processing. The sent remote actor messages are reliably delivered to their receivers.

Process Control RPCs: The RPC thread forwards received RPC requests to a *liaison actor* in the worker thread through the shared ring buffer. The liaison actor coordinates with flow actors via *local actor messages*, to handle RPC requests sent from the coordinator.

3.3.2.3 Service Chain

Each runtime is configured with a sequential service chain (*e.g.*, firewall→NAT→load-balancer) and initializes all the NFs along the service chain upon booting. When the flow actor processes packets, it calls the $process\_pkt(input\_pkt, fs, ss)$ API (Sec. 3.4) of each NF according to the service chain structure to implement the service chain processing logic.

3.3.3 Virtual Switch

A virtual switch is a special runtime where the actors do not run service chains but only a flow forwarding function. An actor in a virtual switch is referred to as a *virtual switch actor*. The virtual switch serves as a load-balancing gateway when forwarding flows and a lightweight flow redirector when executing resilience operations.

For flow forwarding, a virtual switch learns runtimes that it can dispatch flows to through RPC requests sent from the coordinator. When a new flow arrives, a virtual switch actor selects a runtime with the smallest workload as the destination and saves its ID. For each flow packet, the virtual switch actor replaces the destination MAC address with the MAC address of the input port of the destination runtime and forwards the packet.

During flow migration and replication, each virtual switch actor can independently update the flow route by simply modifying the ID of the destination runtime. Compared with installing flow rules on an SDN switch [59, 86], the route update process is lightweight and improves flow migration performance for *NFVactor*.

Table 3.1: Control RPCs Exposed at Each Runtime

Control RPC	**Functionality**
poll_workload()	Poll the load information from a runtime.
notify_cluster_cfg(cfg)	Notify a runtime/virtual switch the current cluster composition.
set_migration_target(runtime_id, migration_number)	Initiate flow migration. It tells the runtime to migrate migration_num of flows to the runtime with runtime_id.
set_replicas(runtime_id_list)	Set the runtimes with IDs in runtime_id_list as the replica.
recover(runtime_id)	Recover all the flows replicated from runtime with runtime_id.

3.3.4 Coordinator

The coordinator in *NFVactor* is responsible for basic cluster management routines. As compared to centralized controllers in the existing NFV systems [59, 86], the coordinator only uses light-weight RPC calls to initiate the flow migration and replication process.

The coordinator communicates with runtimes via a number of control RPCs summarized in Tbl. 3.1. It uses $poll\_workload()$ to acquire the current workload on a runtime. It updates cluster composition (including MAC addresses of input/output/control ports, workload status and IDs of all runtimes and virtual switches in the cluster) to all the runtimes and virtual switches in a cluster using $notify\_cluster\_cfg(cfg)$.

To deploy a cluster, the system operator first specifies the composition of a service chain to the coordinator. The coordinator then creates a new cluster with one runtime and one virtual switch, configures the runtime with the specified service chain and installs proper switching rules to forward matching input flows to the virtual switch. The cluster is then dynamically scaled and recovered under the control of the coordinator.

Table 3.2: APIs for implementing NFs in *NFVactor*

API	**Usage**
nf.allocate_shared_state()	Allocate a singleton object containing the shared states.
nf.allocate_new_fs()	Create and initialize a new flow state object.
nf.deallocate_fs(fs)	Deallocate the flow state object upon expiration of the flow actor.
⋆ nf.process_pkt(input_pkt, fs, ss)	Process the input packet using the current flow states of the flow and the shared states of the NF.
nf.flow_expires(fs, ss)	Update the shared states according to final flow states upon expiration of the flow actor.
nf. flow_migrate_out(fs, ss) nf. flow_migrate_in(fs, ss) nf. flow_recover(fs, ss)	Update the shared states using the flow states during flow migration and replication.

The last three RPCs shown in Tbl. 3.1 are used to initiate flow migration and replication. After issuing these three calls, migration and replication are automatically executed among runtimes without further involving the coordinator.

Dynamic Scaling. The coordinator performs dynamic scaling of the runtimes and virtual switches by exploiting the distributed flow migration mechanism. The coordinator periodically polls the workload statistics from all the runtimes. If there is an overloaded runtime in a cluster, the coordinator launches a new runtime in the same cluster and keeps migrating a configurable number of flows from overloaded runtime (500 as in our experiments) to the new runtime, until half of the workload on the overloaded runtime is migrated away. If runtimes in a cluster become largely idle, the coordinator carries out scale-in: it selects a runtime with the smallest throughput, migrates all its flows to the other runtimes, and shuts the runtime down when all the flows have been successfully moved out.

3.4 NF APIs

To create an NF with full resilience support, the programmer should properly implement the APIs listed in Tbl. 3.2. We follow two principles when designing

these APIs.

First, StatelessNF [65] and Split/Merge [86] demonstrate that it is possible to build practical NFs by processing each individual flow with its per-flow state and shared state. Inspired by this principle, *NFVactor* employs a core API $process\_pkt(input\_pkt, fs, ss)$ to accomplish the core NF processing logic. It is called by each flow actor when processing the input packet, taking per-flow state and shared state as additional arguments. Several supporting APIs are also provided to manage important NF states. This design ensures a clean separation between useful NF states and the core processing logic of an NF, so that the flow actor always has direct and efficient access to the latest flow states to ease flow migration and replication.

Second, to properly handle shared state, we treat shared state accessing by an NF as allocating resource from a shared resource pool. For instance, when a NAT processes a flow, accessing shared state usually means allocating an address from a shared address pool. Therefore, when the flow expires, the resource that the flow acquired should be properly released back to the shared resource pool. With the NAT example, this means that the allocated address should be put back into the address pool when the flow expires. However, when the flow is migrated or recovered on another NF instance, without proper synchronization, the resource obtained by the flow may not be correctly released back to the shared resource pool. To resolve this issue in *NFVactor*, the programmer should properly store the allocated resource in the per-flow state. They also need to implement the last four APIs shown in Tbl. 3.2 for properly releasing the acquired resource so that the shared state is correctly synchronized. Our runtime guarantees that the three APIs are timely invoked during flow migration and replication (Sec. 3.5).

3.4.1 How Runtime Uses the APIs

When a runtime is created, the shared state of each NF along the configured service chain is initialized by calling *allocate_shared_state*() and stored by a storage actor. After a flow actor is created to process a new flow, it first calls

$allocate\_new\_fs()$ to create a flow state for each NF and stores these flow states throughout its lifetime. The flow actor processes a packet along the service chain by sequentially calling $process\_pkt(input\_pkt, fs, ss)$ for each NF, passing in the per-flow state, and shared state obtained from the storage actor. The shared state is sent back to the storage actor when the flow actor finishes processing the packet. When the flow actor expires (this is triggered by a per-actor timer), the flow actor first calls $flow\_expires(fs, ss)$ for each NF to update the shared state and then executes some clean-up procedures, including calling $deallocate\_fs(fs)$ to free the flow state. When a flow is migrated or recovered, the flow actor calls the last three APIs shown in Tbl. 3.2 to synchronize the flow state with the shared state for each NF, followed by executing some clean-up procedures.

3.4.2 Example NFs

Using these APIs, we create four example NFs, i.e., a firewall, an intrusion prevention system (IPS), a load balancer and a NAT.

The firewall updates the connection information (per-flow state) and compare the 5-tuple of the flow with the access control list (shared state) to decide whether to drop the flow packet. The IPS scans the packet payload using an automaton (shared state) built with the Aho-Corasick algorithm [47], saves an index (per-flow state) to the current automaton state, and drops the flow packet if an attack signature is found. Since both shared states of the firewall and the IPS are read-only, i.e. the flow only reads the shared state without acquiring any resource from it, there is no need to implement the last three APIs in Tbl. 3.2 to synchronize the shared state.

The load balancer forwards each input flow to a server with the smallest workload among a set of backend servers. To achieve this, after selecting a server (per-flow state), the load balancer increases the workload counter (shared state) of the selected server to reflect the load balancing decision. Therefore, when the flow expires, or when the flow is migrated or recovered, the workload counter on

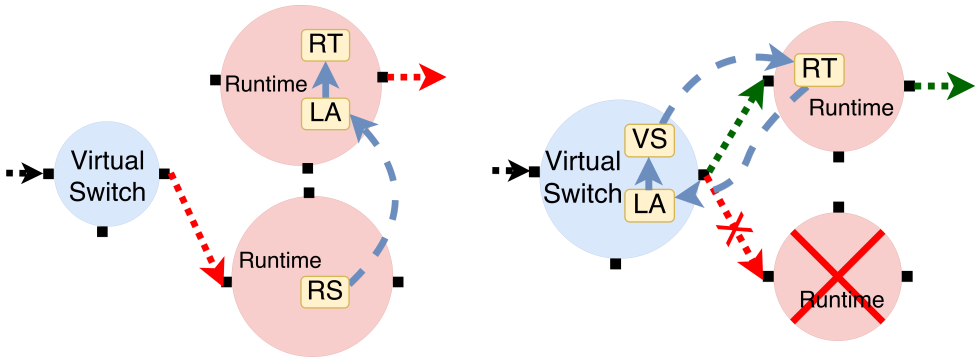
that server should be properly decreased by implementing the last three APIs in Tbl. 3.2.

The NAT operates by substituting the source IP address and source port of the flow packet with an allocated address (per-flow state) from a shared address pool (shared state). Within a cluster, the address pool of each NAT contains non-overlapping addresses. There is no need to implement the last 3 APIs in Tbl. 3.2: we treat the address allocation from the address pool as persistent allocation that lasts throughout the lifetime of the flow, i.e., the flow only releases the address back to the address pool when it expires.

3.5 System Management Operations

3.5.1 Fault Tolerance

3.5.1.1 Replicating Runtimes



(A) Flow replication.

(B) Flow recover when the original runtime has failed.

FIGURE 3.3: Flow replication and recovery: **RT** - replication target actor, **RS** - Replication source actor, **LA** - liaison actor, **VS** - virtual switch actor; **dotted line** - flow packets, **dashed line** - actor messages.)

To perform lightweight runtime replication, we leverage the actor abstraction and state separation. In a runtime, important states associated with a flow are stored by the flow actor. The runtime can replicate each flow actor independently

without check-pointing the entire container image [85, 88]. While achieving good throughput and fast flow recovery, this replication strategy also improves the packet processing delay and has good scalability, as each flow actor can replicate itself on another runtime without the need of dedicated back-up servers.

The detailed flow replication process is illustrated in Fig. 3.3. When a runtime is launched, the coordinator sends a list of runtimes in the same cluster to its liaison actor via RPC $set\_replicas(runtime\_id\_list)$. When a flow actor is created on the runtime, it acquires its replication target runtime from the liaison actor, selected in a round-robin fashion among all available runtimes received from the coordinator.

When a flow actor has finished processing a flow packet, it sends a replication actor message, containing the current flow states of all the NFs and the packet, directly to the liaison actor on the replication target runtime. The liaison actor further forwards the replication message to a replica flow actor sharing the same 5-tuple as the flow actor. The replica flow actor stores latest flow states contained in the message, and then directly sends the packet out, as shown in Fig. 3.3a. This design guarantees the same *output-commit* property as in [88]: the packet is sent out from the system only when all the state changes caused by the packet have been replicated.

The coordinator monitors the liveness of a runtime by sending heartbeat messages to the liaison actor of the runtime. When a runtime fails, the coordinator sends recovery RPC requests $recover(runtime\_id)$ to all the runtimes containing replica flows of the failed runtime. When a runtime R receives this RPC, it instructs each replica flow actor on runtime R to send a request to the virtual switch actor, asking it to change the destination runtime to runtime R. After the acknowledge message from the virtual switch actor is received, the replica flow actor synchronizes the shared states by calling $flow\_recover$ (Table 3.2) and the flow is successfully restored on runtime R (Fig. 3.3b).

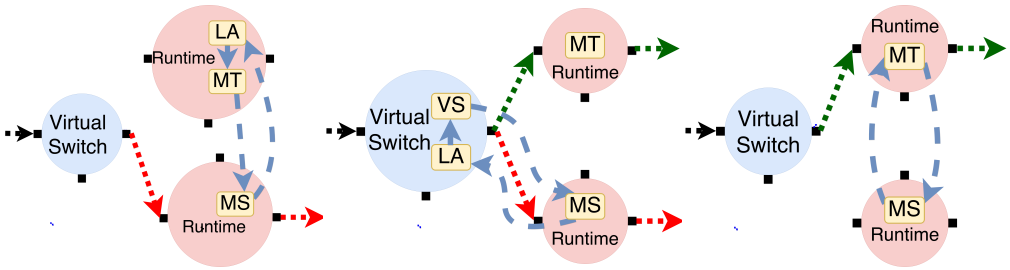
3.5.1.2 Replicating Virtual Switches

Since a virtual switch is in fact a special runtime (Sec. 3.3.3), the virtual switch can be replicated in the same way as described in Sec. 3.5.1.1. The only difference is that when the source virtual switch fails, the replica flow actors in the replication target virtual switch immediately become the primary flow actors without sending out a request to change the forwarding route. Instead, the coordinator takes control and updates the SDN rules to forward the input flows to the replication target virtual switch.

3.5.1.3 Replicating Coordinator

Since the coordinator is a single-threaded module, we can log and replicate information it maintains into a reliable storage system such as ZooKeeper[61]. The liveness of the coordinator is monitored by a guard process and it is restarted immediately in case of failure. On a reboot, the coordinator can reconstruct the system view by replaying logs.

3.5.2 Flow Migration



(A) 1st req-rep step. (B) 2nd req-rep step. (C) 3rd req-rep step.

FIGURE 3.4: The 3 flow migration steps: **MT** - migration target flow actor, **MS** - migration source flow actor, **LA** - liaison actor, **VS** - virtual switch actor; **dotted line** - flow packets, **dashed line** - actor messages.

Based on the actor model, flow migration in *NFVactor* can be regarded as a transaction between a source flow actor and a target flow actor, where the source

actor delivers its entire state and processing tasks to the target actor. Flow migration is successful once the target actor has completely taken over packet processing of the flow. In case of unsuccessful flow migration, the source flow actor can fall back to regular packet processing and instruct to destroy the target actor.

In *NFVactor*, the coordinator starts flow migration by calling *set_migration_target* RPC method on a runtime, asking it to migrate a number of flows to another runtime. After receiving the ID of a migration target runtime, the flow actor starts migration by itself. The flow migration protocol used by flow actors is shown in Fig. 3.4, consisting of three request-response steps. In case of request timeout, the migration source actor performs clean-up procedures and reverts to normal packet processing.

1st req-rep step: The source flow actor sends 5-tuple of its flow to the liaison actor on the migration target runtime. The liaison actor creates a migration target actor using the 5-tuple, and sends a response back to the migration source actor. Meanwhile, migration source actor continues to process packets as usual.

2nd req-rep step: The source flow actor sends the 5-tuple of its flow and the ID of the migration target runtime to the liaison actor on the virtual switch. The liaison actor uses the 5-tuple to identify the virtual switch actor in charge and notifies it to change the destination runtime to the migration target runtime. After this change, the virtual switch actor sends a response back to the source actor, and the migration target actor starts to receive packets. Instead of processing the packets, the target actor buffers all the received packets until it receives the request in the 3rd step from the source actor. The migration source actor keeps processing received flow packets until it receives the response from the virtual switch.

3rd req-rep step: The source flow actor sends its flow states to the migration target actor. After receiving the flow states, the migration target actor saves them, calls *flow_migrate_in* (Table 3.2) to synchronize the shared states, and immediately starts processing all the buffered packets while sending a response to

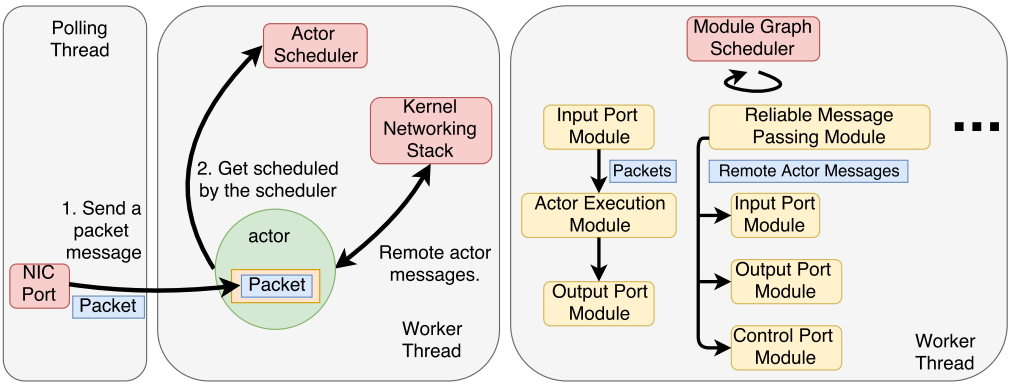
the source actor. The migration source actor calls $flow\_migrate\_out$ (Table 3.2) to synchronize the shared states and then destroys itself.

Loss Avoidance. It is possible for our flow migration protocol to drop flow packets. However, packet drop caused by the flow migration protocol rarely happens in practice, even when concurrently migrating hundreds of thousands of flows (Sec. 3.7.4). We refer to this as loss-avoidance property, which is slightly weaker than the loss-free property [59] in OpenNF.

In the 3rd step, before the request arrives at the migration target actor, the migration target actor has already received and buffered several flow packets. The buffer may overflow, causing migration target actor to drop several packets. However, our distributed flow migration finishes fast within several microseconds and such drop rarely happens in practice.

Still in the 3rd step, after the request is sent out by the source actor, it is possible for some flow packets to continue arriving at the source actor. These packets are sent out by the virtual switch actor before its destination runtime is changed in the 2nd step, but arrive later at the source actor than the response of the 2nd step. The source actor has to discard these packets because it has already sent out its flow state. *NFVactor* minimizes the occurrence of this problem by having the virtual switch actor transmitting the response message of the 2nd step in the same network path as the flow packets, so that we rarely observe this problem in practice.

It has been a common understanding that providing good properties for flow migration would trade off the performance of flow migration [59]. *NFVactor* mitigates this trade-off using distributed, high-performance flow migration based on the actor model.



(A) Libcaf runtime. (B) *NFVactor* runtime.

FIGURE 3.5: Comparison of runtime architecture.

3.6 Implementation

Throughout the development process of *NFVactor*, we initially chose libcaf [6] as the runtime, whose architecture is shown in Fig. 3.5a. However, the performance of libcaf [6] runtime is less than satisfactory, for both packet processing and resilience operations. We believe that the performance problems of libcaf runtime come from two aspects. *First*, the actor scheduler of libcaf is not suitable for handling IO of raw network packets. According to Fig. 3.5a, the scheduler schedules an actor run according to whether the actor has received a message. To inject dataplane packets into libcaf runtime, we have to set up a dedicated polling thread to poll the NIC port and send received packets to actors running in another worker thread by enqueueing the packet into actor's message queue. As verified in Sec. 3.7.1.1, there is an expensive synchronization overhead between polling thread and worker thread, which may decrease packet processing throughput by over 100%. *Second*, libcaf runtime still relies on inefficient kernel networking thread to exchange remote actor messages with other runtimes.

Realizing these problems, we design a customized actor runtime for *NFVactor* as shown in Fig. 3.5b, by leveraging two optimizations. *First*, we implement a module graph scheduler to schedule actors according to several pre-defined module graphs. The module graph scheduler combines various IO operations (polling

NIC port, exchanging remote actor messages) with actor scheduling inside a single worker thread, effectively eliminating the thread synchronization overhead as in libcaf. *Second*, we bypass inefficient kernel networking stack and implement a high-performance, reliable message passing module running in user-space.

The tradeoff point when designing the customized runtime is programmability, as the programming interface exposed by the customized runtime is not as easy to use as the libcaf runtime. However, we believe that such a tradeoff is worthwhile due to improved performance.

Module Graph Scheduler. Inspired by the scheduler design of BESS [3] and Click [68], the module graph scheduler keeps scheduling several module graphs to run in a round-robin fashion. A module graph consists of several processing modules, connected together into an acyclic graph. Inside a module graph, the actor messages are generated by a source module, flow through the connected module for processing before reaching the sink module, which consumes each message by either freeing it or sending it to the outside. Inside a module, the message handler of the corresponding actor is called for each actor message. The actor then pushes the processed message to the next connected module. When all the generated actor messages are consumed, the scheduler moves on to run the next module graph.

Fig. 3.5b illustrates two module graphs. The first module graph polls dataplane packets from the input port and generates packet messages, which are pushed along the module graph, processed by the flow actors and sent out from the output port. The second module graph fetches remote actor messages from the reliable message passing module and sends the remote actor message out from one of the three ports. There are two other module graphs that are used for receiving reliable actor messages and interacting with the RPC requests.

Reliable Message Passing. Based on the module graph scheduler, we build a reliable message passing module, which inserts remote actor messages into a reliable packet stream for transmission. The module creates one ring buffer for each remote runtime and virtual switch. When a flow actor on this runtime sends a

remote actor message, the module creates a packet, copies the content of the message into the packet and then enqueues the packet into the respective ring buffer. These packets are configured with a sequential number each, and appended with a special header to differentiate them from dataplane packets. When the second module graph in Fig. 3.5b is scheduled to run, the worker thread dequeues these packets from the ring buffers, and sends them to respective remote runtimes. A remote runtime acknowledges receipt of such packets. Retransmission is fired in case that the acknowledgement for a packet is not received after a configurable timeout (*e.g.*, 10 times the RTT in our current implementation). Running entirely in user-space, the performance of the reliable message passing module is good enough to saturate a 10Gbps link (Sec. 3.7.1.2).

Since our goal is to reliably transmit remote actor messages over an interconnected L2 network, we do not use user-level TCP [64], which may impose additional overhead for reconstructing byte streams into messages. In addition, packet-based reliable message passing provides additional benefits during flow migration and replication. Because the response in 2nd request-response step of flow migration is sent as a packet using the same path as the dataplane packets, reliable actor message passing enables us to implement loss-avoidance migration with ease (Sec. 3.5.2).

3.7 Evaluation

We evaluate *NFVactor* using 4 Dell R430 servers, each equipped with an Intel Xeon E5-2650 CPU running at 2.30GHz with 20 logical CPU cores, 48GB memory and 2 Intel X710 10Gb NICs. The servers are connected through a 10GB Dell switch. In each server, the worker thread of each runtime is pinned to a dedicated logical core, while the RPC threads of all the runtimes are collectively pinned to logical core 0 to minimize the performance impact on the worker thread. To generate test traffic, we use a customized traffic generation module (the FlowGen module) of BESS [3], which is capable of generating input traffic up to 10Gbps (at around 14Mpps) with 64-byte packets.

We use a single server to generate the traffic. We set up 6 virtual switches on another server, which are capable of handling input traffic at 10Gbps line rate and do not render bottlenecks. The rest of the servers are used to run runtimes.

3.7.1 Performance of the Runtime

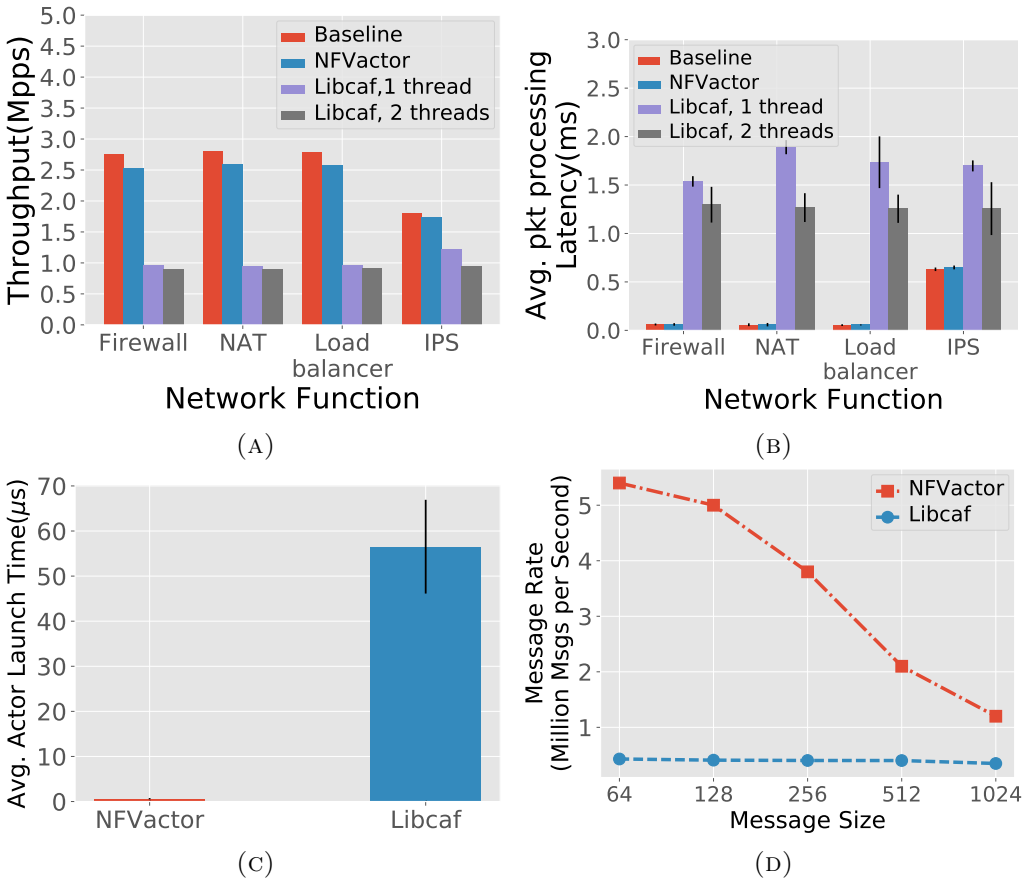


FIGURE 3.6: Performance of the Runtime.

3.7.1.1 Packet processing throughput and latency

We first evaluate the packet processing throughput (number of packets processed per second) and latency (difference between the time that a packet enters the

runtime to the time this packet is released from the runtime) of the *NFVactor* runtime by running the four implemented NFs. The traffic generator produces flows with 64-byte TCP packets, a 10pps (packets per second) flow rate and a 10-second active time. The overall packet rate of the input traffic is 5Mpps. For comparison, we also evaluate performance of libcaf runtime. We vary the number of worker threads used by the libcaf runtime to reflect the performance overhead of thread synchronization. Finally, we compare with four baseline NFs. The baseline NFs are implemented using a normal packet processing loop, sharing similar processing logic as the *process_pkt* API. The per-flow state in each baseline NF is stored in a fast hash table [78] without using the actor abstraction.

Fig. 3.6a and Fig. 3.6b show that *NFVactor* runtime achieves significantly larger throughput and smaller processing latency than libcaf runtime, and the performance of the NFs in *NFVactor* is close to that of the baseline NFs, as the actor abstraction does introduce a small overhead. According to Fig. 3.6a, when the number of the worker threads used by libcaf is increased, the total throughput drops by a small margin, due to increased synchronization overhead between the polling thread and multiple worker threads.

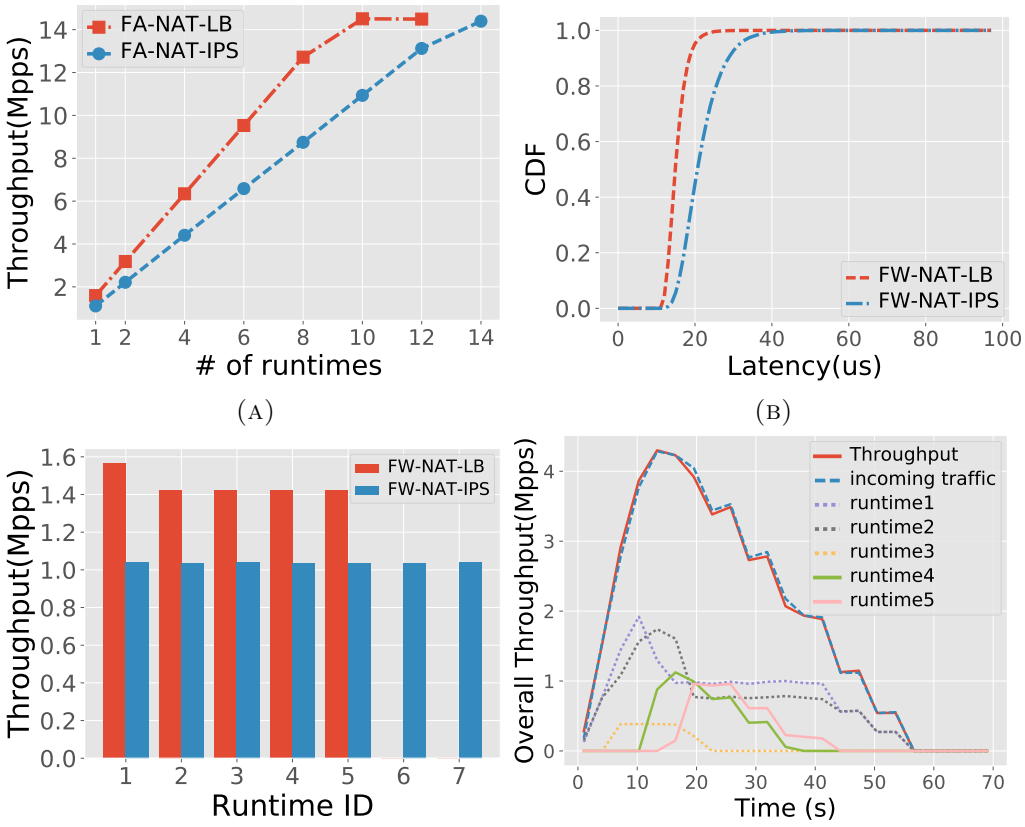
3.7.1.2 Actor launch time and sending rate of remote actor messages

The actor launch time is the time when the first packet of a flow is received to the time when this packet is processed by the launched actor. It is a strong performance indicator of the actor runtime system. In Fig. 3.6c, the input traffic is the same as in Sec. 3.7.1.1. We see that the average actor launch time in *NFVactor* is much smaller than that of the libcaf runtime, as *NFVactor* preallocates flow actors into a ring buffer to speed-up actor launch time.

Fig. 3.6d shows the average sending rate of remote actor messages between two runtimes running on different servers. For various message sizes, the message rate achieved by *NFVactor* is significantly larger than that of libcaf. Especially, when the message size is larger than 256 bytes, we measure the consumed bandwidth of *NFVactor* to be around 9.1Gbps, which is close to the 10Gbps line rate.

This result reflects that our user-space message passing module can significantly improve the performance of remote actor communication.

3.7.2 System Scalability



(A) (B) (C) (D) Dynamic scaling.

FIGURE 3.7: System scalability.

We now evaluate the maximum packet processing throughput of *NFVactor* as the number of runtimes increases. We use two servers and configure the runtimes with service chain 'firewall (FW) → NAT → load balancer (LB)' (in one set of experiments) or service chain 'firewall (FW) → NAT → IDS' (in another set of experiments). To fully stress the system, we configure traffic generators to produce a mixture of short flows and long flows up to 10Gbps line rate. A short flow consists of 64-byte TCP packets with a 10pps packet rate and lasts

for 1 second. A long flow consists of 64-byte TCP packets with a 10pps packet rate and lasts for 10 seconds. Each type of flow consumes half of generated bandwidth. We gradually increase the number of active runtimes and collect the total throughput achieved by the runtimes.

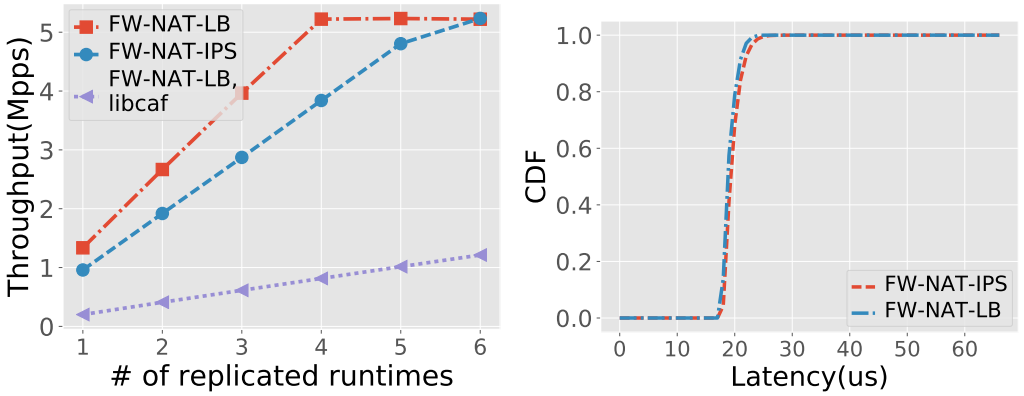
Fig. 3.7a shows the overall packet processing throughput increases linearly with the increase of the runtimes. Overall throughput of 14.49Mpps (9.70Gbps) and 14.39Mpps (9.67Gbps) are achieved when the runtimes run service chain 'FW → NAT → LB'and 'FW → NAT → IDS', respectively. This verifies that *NFVactor* can approach 10Gbps line-rate packet processing for 64-byte small packets, even when the input traffic consists of many short-lived flows.

Fig. 3.7b shows the CDF of packet processing latencies, collected during a 20s period when 10 and 14 runtimes are used to run 'FW → NAT → LB' and 'FW → NAT → IDS', respectively. The average latency for both service chain is around 20μs.

We next run the two service chains concurrently in the system. We run 5 runtimes in one server configured with 'FW → NAT → LB' and 7 runtimes in another server configured with 'FW → NAT → IDS'. The input traffic has a total packet rate of 14.50Mpps and shares the same mixed pattern to produce Fig. 3.7a. The input traffic is evenly split among the two service chains. Fig. 3.7c shows the throughput of each runtime. We can see that a total throughput of 7.25Mpps can be reached by each service chain. The workload is also evenly balanced among runtimes in the same server, demonstrating the effectiveness of our virtual switches for load balancing under mixed short and long flows.

Finally, the performance of the dynamic scaling controlled by the coordinator is shown in Fig. 3.7d. The traffic generator generates traffic with increasing packet rate for the first 15 seconds, and then gradually decreases the packet rate of the traffic for the last 45 seconds. The cluster is initialized with two runtimes (runtime 1 and 2) running 'FW → NAT → IDS' service chain. Starting from the 10th second, runtimes 1 and 2 become overloaded and their workload is gradually migrated away to runtime 4 and 5. With dynamic scaling, the achieved throughput can correctly match the input traffic during the 60 seconds.

3.7.3 Performance of Flow Replication



(A) (B)

FIGURE 3.8: Performance of flow replication.

TABLE 3.3: Recovery time and # of flows recovered

	FW→NAT→LB	FW→NAT→IDS	FW→NAT→LB, libcaf
Average recovery time for each runtime	65.3ms	66.6ms	934.2ms
Number of flows recovered on each runtime	at least 87k flows	at least 87k flows	at least 20k flows

In this set of experiments, input flows are produced following the same mixed pattern to produce Fig. 3.7a. The coordinator chooses two servers and launches the same number of runtimes on them. The coordinator instructs each runtime on a server to replicate its flows to a distinct runtime in another server. We gradually increase packet rate of the input traffic and the number of runtimes running on each server, to investigate throughput and scalability when flow replication is enabled.

Fig. 3.8a shows that both service chains can scale up to handle the 5.22Mpps input packet rate when six runtimes running on a server concurrently replicate their traffic to six replica runtimes on another server. In particular, when the input rate reaches 5.22Mpps, the measured bandwidth on the link for transmitting replication messages reaches almost 10Gbps. When the libcaf version of implementation is used, the replication throughput is significantly lower.

Fig. 3.8b shows the CDF of packet processing latencies of *NFVactor* when flow replication is enabled. The latency measured in this experiment is difference between the time that the packet enters replication source runtime to the time this packet is released from the replication target runtime. For both service chains, the number of runtimes on each of the two servers is 6 while the input packet rate is 5.22Mpps. The average latency is around 20μs for both service chains.

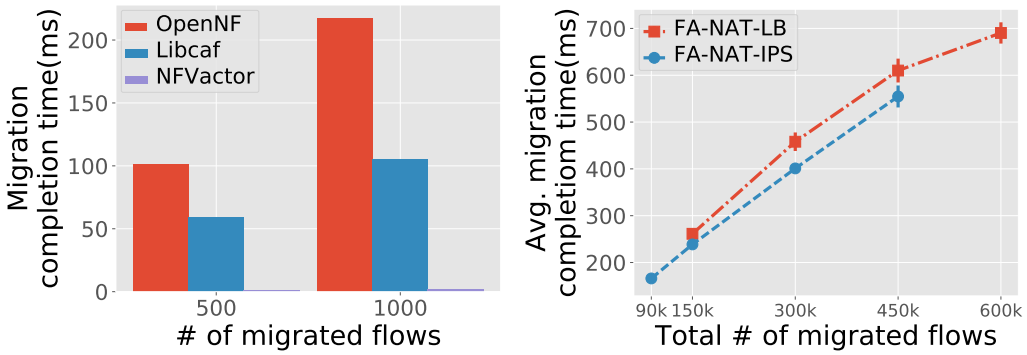
Table 3.3 shows the average recovery time of 6 replication target runtimes when their corresponding replication source runtimes fail simultaneously. We can see that *NFVactor* has a much shorter recovery time than the libcaf version even when processing a larger number of flows.

3.7.3.1 Comparison with FTMB

We compare the performance of flow replication in *NFVactor* with the reported performance of FTMB [88]. Both systems achieve throughput up to millions of packets per second and recovery time of tens of milliseconds with flow replication enabled. *NFVactor* has a more stable packet processing latency (according to Fig. 3.8b, smaller than 70 microseconds, with an average of 20 microseconds) because it does not need to checkpoint the runtime, whereas FTMB introduces a relatively high packet processing latency (up to 3000 microseconds) when checkpointing kicks in.

3.7.4 Performance of Flow Migration

We first compare flow migration performance among *NFVactor*, libcaf runtime, and OpenNF. Both *NFVactor* and libcaf runtime run the example firewall. We also port the example firewall to work with OpenNF. We send the same number of flows to the three firewalls and each flow has a 10pps packet rate. In Fig. 3.9a, the time to migrate the respective number of flows is much smaller with *NFVactor* (0.7ms for 500 flows and 1.5ms for 1000 flows), when compared



(A) (B)

FIGURE 3.9: Flow migration completion time.

to OpenNF (101ms for 500 flows and 217ms for 1000 flows) and the libcaf runtime (59ms for 500 flows and 105ms for 1000 flows). Due to efficient runtime design, *NFVactor* can out-perform OpenNF by 144 times when migrating 1000 flows.

We next show the time taken for concurrently migrating a large number of flows among multiple pairs of runtimes. The traffic generator produces a number of flows with a 10pps flow rate, each lasting for 1 minute. The flows are first sent to three runtimes running in one server. Then the coordinator instructs the three runtimes to concurrently migrate all the flows to three runtimes running in another server.

Fig. 3.9b shows the average migration completion time of the three migration source runtimes. The standard deviation of the completion time is shown as error bar in Fig. 3.9b as well. *NFVactor* can migrate 600K flows (with 6Mpps total throughput) from three migration source runtimes running in one server to three migration target runtimes running in another server using around 700ms. Besides the performance boost enabled by efficient runtime design, the decentralized flow migration also contributes to the performance. Since the flow migration are concurrently carried out among three pairs of runtimes, each pair of runtime only needs migrates around 200K flows. This significantly reduce the eventual flow migration completion time for all the 600K flows. If the 600K flows are

sequentially migrated, the resulted flow migration completion time may be prolonged to over 2 seconds. Finally, throughout the evaluation in Fig. 3.9b, we observe zero packet loss caused by our flow migration protocol.

3.8 Related Work

Since the introduction of NFV [28], a broad range of studies have been carried out, for bridging the gap between specialized hardware and network functions [60, 62, 72, 80], scaling and managing NFV systems [58, 79], flow migration [59, 67, 86], NF replication [85, 88], and traffic steering [82]. In these work, NF instances are typically created as software modules running on separate VMs or containers. *NFVactor* customizes a runtime system to run flow actors and embeds service chain processing within the flow actor abstraction, to achieve transparent and highly efficient failure resilience guarantee.

Among the existing studies, StatelessNF [65] shares similar design goals with *NFVactor*, *i.e.*, to enable transparent system scalability and failure resilience. However, the methodology used by StatelessNF is orthogonal to that of *NFVactor*: it stores flow states in a reliable database [76] to achieve failure resilience, while *NFVactor* exploits the actor model. Compared with StatelessNF, *NFVactor* can approach line-rate packet processing and does not rely on RDMA equipment.

Besides, the Click modular router [68] is the first work to introduce modular design for simplifying NF construction. The module graph scheduler used by *NFVactor* is partially inherited from the scheduler of Click. However, *NFVactor* uses such a scheduler to speed up actor processing. Flurries [94] proposes fine-grained per-flow NF processing, by dynamically assigning a flow to a lightweight NF. Sharing some similarities, *NFVactor* enables micro service chain processing of each flow in one actor, but focuses on providing transparent failure tolerance based on the actor model. StateAlyzr [67] uses static analysis to automate flow state extraction and simply human effort for enabling flow migration. However, enabling high-performance flow migration still requires a holistic design like *NFVactor*.

3.9 Conclusions and Discussions

We have presented *NFVactor*, an NFV system using actor model to achieve transparent and highly efficient failure resilience. *NFVactor* advocates a novel one-flow-one-actor principle to improve the parallelism and performance of resilience operations, while the efficiency of the actor model is guaranteed by a high-performance runtime. Our experiments show that *NFVactor* achieves good scalability and high packet processing speed, as well as fast flow migration and failure recovery.

Powered by a completely new architecture, *NFVactor* does require NFs to be rewritten or ported using the provided APIs. We believe that there will be a growing need for implementing new NFs with further adoption of the NFV paradigm and the flexible API design of *NFVactor* makes it possible to port a wide range of existing NFs. Besides, our holistic design approach will be among the method that NF developers may choose from, valuable for decoupling complexity of critical system services from the core logic of NF.

Chapter 4

NetStar: A Future/Promise Framework for Asynchronous Network Functions

4.1 Introduction

Network Functions (NFs) are more than simple packet processors that perform various transformations on each received packet. Modern NFs, *e.g.*, firewall [65], NAT [65], IDS [4], and proxies [17, 31], often need to contact external services while processing network flows, *e.g.*, for querying external databases [5, 42], or saving critical per-flow states on external reliable storage (for failure resilience purposes) [65]. To ensure high-speed packet processing while executing external queries, these NFs must fully exploit asynchronous programming: after generating a request to an external service, the NF should not block and wait for the response in a synchronous fashion; instead, it can save the current processing context and register an event handler function to handle the response upon its return, and switch to process other packets, potentially generating additional asynchronous requests.

For example, to detect whether a file transmitted over a TCP connection contains a piece of malware, a Bro IDS [4] issues a DNS query containing the SHA1 hash [37] of the file extracted from the reconstructed byte stream of the TCP flow to a Malware Hash Registry (MHR) [24]. Then, the MHR generates a DNS response indicating whether the hash matches that of some known malware. To ensure high performance, the Bro IDS does not block and wait for the MHR response to arrive. Instead, it registers a callback function to handle the MHR response upon its receipt, in an asynchronous fashion, and switches to process other flows/packets or handle other generated events (new packet arrival, new reconstructed payload, etc. [81]).

Compared with synchronous NF programs, asynchronous NF implementation using callbacks is significantly more efficient in packet processing, as it does not waste important CPU time. However, callback-based asynchronous programming has some inherent drawbacks that can prevent developers from building NFs with richer functionalities.

First, compared to a synchronous program, a callback-based asynchronous program is harder to implement and reason about. Such a program may define multiple callback functions, scattered within different source files, to achieve a series of asynchronous operations. For example, the Bro IDS can be configured to detect malware in flows using two nested callback functions, a first callback function to handle the reply from a local database query which may trigger another callback function to handle the reply to a MHR query (Sec. 4.2.1); and a NAT may replicate important per-flow states in an external database using 4 consecutive callbacks, to read from and write to a remote database while a TCP connection through the NAT is being established [65]. Dealing with multiple callbacks scattered in different source files can be confusing, and make it more difficult for a programmer to trace the execution order of the program.

Second, visiting saved context information inside a registered callback can be error-prone. Since an asynchronous program immediately switches to other tasks after saving the context and registering a callback, the program must ensure that the saved context is not accidentally freed until the callback is invoked. Failing

to do so may lead to invalid memory access and program crash. However, when multiple callback functions are used, the programmer may accidentally free the context if he fails to correctly trace the execution order of the callbacks.

Third, redundant error handling code may be introduced in a callback-based asynchronous program. Since exceptions may happen when waiting for the external response, the program must properly handle the exceptions by either registering an error handling function or implementing exception handling logic in the callback registered to handle the response. When a series of asynchronous operations are executed, the programmer needs to add exception handling logic for each asynchronous operation. Since the asynchronous operations/callback functions may well be scattered among multiple files, duplicate error handling code may need to be added in multiple places.

People have recognized the problems with callbacks when building event-driven systems such as web browsers [57, 66], programming language runtime systems [90], web servers [45] and database servers [33]. Their solution to counter the problems is to use a more advanced programming abstraction, such as the coroutine [9] and the future/promise paradigm [56, 69, 92]. Coroutine is a user-space cooperative thread that is able to execute asynchronous program in a fully synchronous fashion. However, the coroutine switching time may cause and suffer considerable overhead in NFs processing a large number of concurrent network flows. In contrast, the future/promise paradigm uses special runtime objects, futures, promises and continuation functions, to mimic synchronous programming while being fully asynchronous. Besides making asynchronous program easier to implement, the future/promise abstraction can also reduce redundant error handling code by effectively propagating exceptions to a consolidate error handling logic.

Though the future/promise paradigm is promising, the future/promise abstraction had only existed in high-level programming languages such as F# [90], Haskell [69] and OCaml [92]. It is known that high-level programming languages are not suitable for developing NF software due to their lower runtime

efficiency as compared to C/C++-based implementation, and unpredictable processing delay caused by their garbage collectors [80]. A recent open-source C++ library, Seastar [36], is implementing the future/promise paradigm to build high-performance database servers [35]. The library is integrated with DPDK and provides a customized user-space TCP/IP stack for high-speed database queries. However, it does not expose any interface to manipulate raw network packets, and designing such an interface, which effectively processes network packets without diminishing the power of the future/promise abstraction, is non-trivial. A straightforward design may directly expose a regular packet handler function, that is invoked for each received packet. However, such a design falls back to callback-based asynchronous programming and leaves no room for utilizing the future/promise abstraction.

This paper proposes *NetStar*, a future/promise-based programming framework for simple, elegant asynchronous programming in NFs. *NetStar* enables programming a series of asynchronous operations (when processing dataplane packets) in a manner similar to implementing a simple synchronous program, while not incurring any performance degradation due to blocking as in a synchronous program.

The power of *NetStar* is mainly attributed to a programming interface that we design, the async-flow interface, which effectively combines network packet handling process with the future/promise abstraction. The async-flow interface is powered by a simulated packet processing loop, and uses the returned future objects from a packet handler function for implementing core NF processing logic. With this interface, programmers can simplify the implementation of complex asynchronous operations in NFs by chaining a series of continuation functions, which mimics a synchronous program. Due to the future/promise paradigm, programmers can avoid redundant error handling logic but use a set of consolidate error handling code, allowing them to focus more on the core NF processing logic. The async-flow interface also simplifies context management: a programmer only needs to keep track of a pointer to a context object, and subsequent visits to the context object is guaranteed to be safe.

To evaluate the performance of *NetStar*, we build a number of NFs using the framework, including four NFs from the StatelessNF paper [65], an HTTP reverse proxy, an IDS and a malware detector. With extensive experiments, we show that NFs based on *NetStar* use substantially fewer lines of code to implement asynchronous packet processing, as compared to callback-based implementation, while delivering sufficiently good performance in terms of packet processing throughput and latency. We also compare *NetStar* with a coroutine based implementation, and show that the coroutine is a less desirable paradigm for implementing NFs processing a large number of concurrent network flows. The source code of *NetStar* is available at [26].

4.2 Motivation and Background

4.2.1 Asynchronous Programming in Representative NFs

Bro. We use a concrete example, malware detection in Bro IDS [4], to demonstrate how asynchronous callbacks are used to program NFs. Bro can be configured to detect malware contained in a transmitted file as follows. A local database server stores hash values of commonly-seen malware. For each TCP connection that goes through, Bro reconstructs its byte stream. If a transmitted file is detected within the byte stream, Bro computes a SHA1 hash value over the file content, and queries the local database to obtain a quick response whether the hash value matches any malware's. In case of no hit in the local database, Bro can turn to the more complete and up-to-date Malware Hash Registry (MHR) [24], which provides a special DNS service: Bro can send a DNS request carrying the file hash to MHR, and the MHR responds whether there is a match with any of the malware hash values it has. If either of the two queries detects some malware, Bro raises an alarm in the log file.

The C++ code covering the execution path of the malware detection steps is given in Fig. 4.1, as extracted from Bro version 2.5-359. In the packet processing

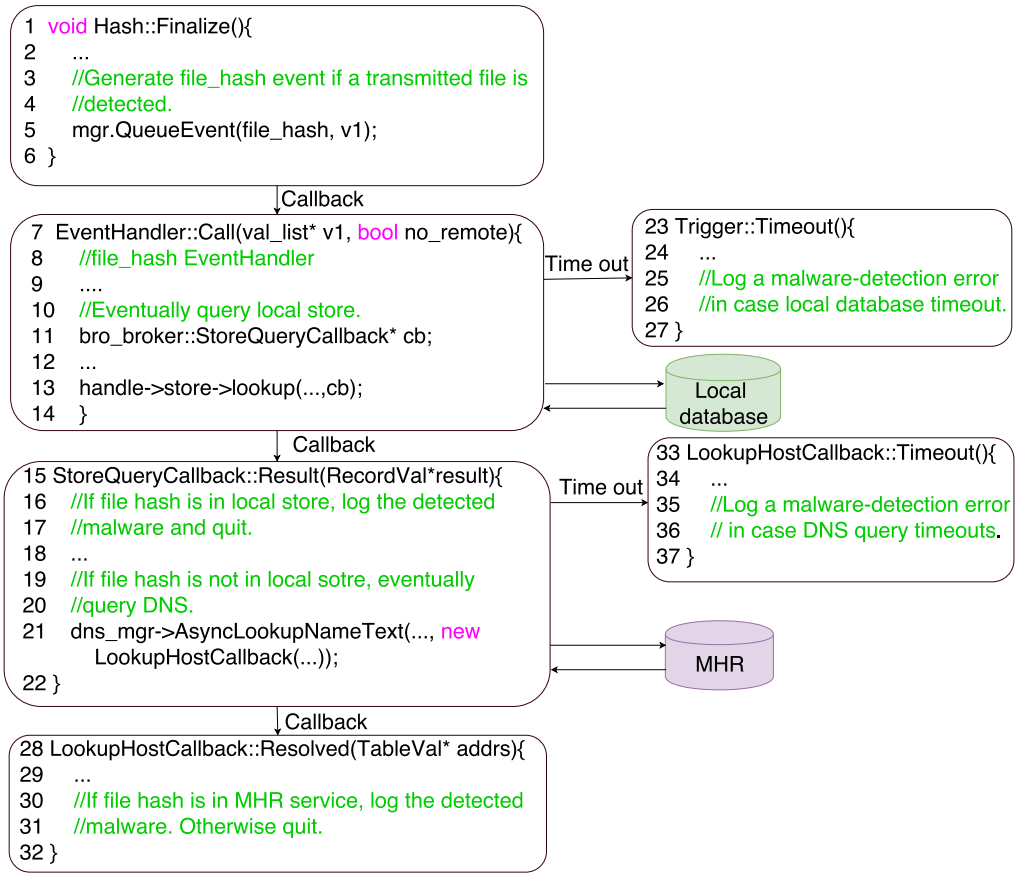


FIGURE 4.1: Malware detection in Bro.

loop of Bro, after a transmitted file is detected, `Hash::Finalize()` is called to calculate the file's hash value and post a `file_hash` event to Bro's event engine. Bro handles the event in the callback `EventHandler::Call`, by performing an asynchronous query to the local database (lines 11,13). The response of the database query is handled by the callback `StoreQueryCallback::Result`: if the file hash is not found in the local database, Bro performs another asynchronous query to the MHR (line 21). The DNS response is handled by a second callback, `LookupHostCallback ::Resolved`, which logs the detection of a piece of malware if the DNS response indicates so. Since any of the database and MHR queries may fail, two additional callbacks, `Trigger::Timeout()` and `LookupHostCallback::Timeout()`, are registered to handle the database and MHR query timeout error, respectively.

The code excerpt highlights the following disadvantages in callback-based asynchronous programming: (1) The asynchronous code is more complex to implement as compared to a synchronous program. For instance, if `StoreQueryCallback::Result` is to be implemented in a synchronous fashion, the programmer can directly proceed to handle the DNS response or the timeout event after calling a blocking DNS query function, without the need for saving the context information (omitted from line 21) and defining two extra callback functions elsewhere (lines 28 and 33). (2) Omitted in line 21, saving and retrieving context information can be quite troublesome, as Bro relies on reference counting to keep allocated objects alive, and needs to correctly increase the reference counts of all the objects that it intends to keep within the context when registering a callback. (3) Two error handling functions are defined to handle potential errors in the two asynchronous operations. The two functions in fact implement the same logic, *i.e.*, to log a malware detection error to the database, which are quite redundant.

HAProxy. We have also inspected the asynchronous program in HAProxy [17], a high-speed proxy due to its completely asynchronous design. HAProxy accepts incoming user TCP connections, relays received packet contents, such as HTTP requests, to several servers (*e.g.*, for load-balancing or connecting an internal network to the public network), and then relays the responses received from the servers back to the client. In order to poll a socket file descriptor to retrieve the incoming data over the TCP connection, HAProxy invokes three nested callbacks, `iocb` (for accessing the socket), `recv` (for receiving data from the socket) and `rcv_buf` (for putting received data into buffer). These three callbacks are registered multiple times in various source files, making it hard to trace the execution flow of HAProxy without a deep inspection of the files.

4.2.2 Advanced Programming Abstractions

There is a tradeoff between simplicity and performance when implementing a NF program: On one hand, synchronous programming is easier but incurs poor performance. On the other hand, asynchronous programming using callbacks achieves much better performance at the cost of implementation complexity.

Similar tradeoffs have been identified when people build server programs, such as web [45] and database servers[33]. The solution in those domains has been to use advanced programming abstractions, including coroutine [9] and future/promise [56, 69, 90, 92], to manage the complexity of asynchronous programming while achieving good performance.

Coroutine is a user-space cooperative thread: a coroutine explicitly yields its execution to other coroutines when it waits for asynchronous operations to complete. The coroutine paradigm has found its success in building asynchronous web servers [45] and databases [33].

With coroutine, asynchronous NFs can be very easy to implement. The NF program can be directly written as a synchronous program, which runs as multiple coroutines (threads) to handle different flows. Whenever a coroutine needs to perform an asynchronous operation, it yields its execution to other coroutines after saving its thread context, and then waits to be resumed when the asynchronous response arrives in the future.

Typical coroutine switching time can be up to hundreds of nano seconds. A typical high-performance NF needs to process hundreds of thousands of flows concurrently and millions of packets per second. The coroutine switching time may significantly slow down NF processing, which we have verified with experiments (to be presented in Sec. 4.6.7).

In search for a suitable advanced programming abstraction for significantly reducing implementation complexity of asynchronous NFs, we have identified that the future/promise paradigm can be more suitable.

4.2.3 Future/Promise

The future/promise abstraction is a light-weight, elegant model for asynchronous programming. It simplifies asynchronous programming by mimicking synchronous programming, and reduces redundant error handling logic by propagating exceptions.

4.2.3.1 Future

A future object is of type `future<T>` and contains a value of type `T`. There are two states for each future object, available and unavailable. When in the available state, a future object either contains a concrete value of type `T` that can be directly used, or an exception. In the unavailable state, a future object is associated with a promise object, and can be turned into available state with the help of the associated promise object.

4.2.3.2 Promise

A promise object is of type `promise<T>`, and associated with an unavailable future object. When needed (*e.g.*, when the response of a remote query arrives, or the response times out and an exception is generated), the programmer can set either a concrete value, or an exception, to the promise object with the `promise::set_value` method. The value/exception is then automatically propagated to the associated future object and turns it into the available state.

4.2.3.3 Continuation Function

The future/promise abstraction works together with continuation functions to achieve asynchronous programming. A continuation function can be appended to a future object using the `future::then` method. The following code shows a continuation function appended to the `fur` future object.

```
future<T> fur=pr.get_future();
fur.then([captured_variables...](T t){
...
});
```

Here, the continuation is presented as a C++ lambda function [7]. `captured_variables` is a list of variables passed to the continuation function, which it is free to use. For instance, `captured_variables` may contain a pointer to another object and the continuation function can visit this object by following the pointer. `(T t)` represents the input arguments to the continuation function.

A continuation function can be considered as a special callback function. If the future object that it is appended to is available, the continuation function is immediately invoked; otherwise, it is invoked the moment when the future object becomes available. When a continuation function is called, the future object passes its concrete value as the argument `T` into the continuation function.

4.2.3.4 Transforming Future Object

If a continuation function is appended to a future object `a` of type `future<A>`, creates a future object `b` of type `future<B>` and returns it as a future object `transformed` of type `future<B>`, then we say the future object `a` is transformed into future object `transformed` of type `future<B>`, and `transformed` only becomes available when both `a` and `b` are available. When `transformed` becomes available, it contains the exact value of `b`.

Consider the following example for asynchronous database and DNS queries:

```
// Query database.
future<db_res> db_fur=_db.lookup(_hash);
future<dns_res> fur=db_fur.then([this](db_res res){
  // Do something with the database query response.
  ...
  // Query DNS.
  future<dns_res> dns_fur=_dns.lookup(domain_name);
  return dns_fur;
});
```

We first issue an asynchronous database query `db.lookup()` to acquire a future object `db_fur`, which contains the query response when it returns. Then, we append a continuation function to `db_fur`. Inside the continuation function, an asynchronous DNS query `dns.lookup()` is issued and a future object `dns_fur` is obtained, which contains the DNS response once it is received. When the continuation function is done, a future object `fur` is obtained, which becomes available when both `db_fur` and `dns_fur` are available (*i.e.*, when both the database response and DNS response are received), and contains the received DNS response once available (passed from `dns_fur`).

The returned future object can be associated with another continuation function, which further returns anther future object. In this way, we are able to chain multiple continuation functions to carry out a series of asynchronous operations. It allows us to mimic synchronous programming while achieving full execution asynchrony. In the above example, the database and DNS queries appear to be sequentially executed but are in fact asynchronously performed. Also, there is no need to define callbacks elsewhere: the continuation function for handling the database response is defined right in place (in the same file where the database query is issued), but is invoked only when the database response arrives and `db_fur` becomes available.

4.2.3.5 Consolidated error handling.

In previous example, to catch database and DNS query exceptions, we can simply append another continuation function to the future object `fur` as follows:

```
fur.then_wrapped([](future<dns_res> f){
  try{
    f.get();
  }
  catch(...){
    // catch whatever exceptions thrown
    // during database or DNS query
  }
});
```

Here, `fur` may contain an exception that is either generated by database query or DNS query. This is due to the capability of the future/promise abstraction to propagate exceptions and bypass uncalled continuation functions in the chain, and push the exception to the final returned future object. For example, if an exception occurs during database query, `db_fur` will contain the exception, and then pass it to `fur` without running the continuation functions it is associtaed with; if the database query is alright but DNS query raises an exception, `dns_fur` contains the exception and passes it to `fur`. Finally, the exception is passed to `f` when the continuation function `fur.then_wrapped([](future<dns_res> f)` is invoked, thrown by `f.get`, and caught inside a try-catch pair [8].

Leveraging this consolidated approach to handle various errors that may occur during a series of asynchronous operations, we can effectively reduce redundant error handling code when developing asynchronous programs.

4.2.4 Bring Future/Promise to Dataplane

We use the open-source Seastar [36] library as the base for implementing our future/promise NF programming model: (i) Seastar is one of the most mature C++-based future/promise libraries, and such C++ based implementation introduces minimum runtime overhead, which is critical for high-performance NFs; (ii) Seastar is integrated with DPDK and has a built-in user-space TCP/IP stack, rendering a feasible ground for building various NFs.

However, Seastar is originally designed for creating high-performance database servers [35]. It only exposes a socket-like interface for interacting with application-layer payload and has no interface for directly retrieving and sending raw network packets. Yet, an interface for manipulating raw network packets is crucial for implementing L4 NFs, such as firewall, NAT and IDS.

This leads to the design of *NetStar*, a future/promise paradigm for building various NFs. *NetStar* is built on top of Seastar, and novelly designs and integrates two stacks to handle raw network packets and application-layer traffic in a coherent framework.

4.3 The *NetStar* Framework

We design *NetStar*, a future/promise based programming framework for building various NFs that may carry out L4 or L7 packet processing tasks, and extensive asynchronous interactions with external services. For example, an IDS handles L4 packet processing by receiving TCP flows, reconstructing byte streams, and detecting whether they contain any malware. A proxy handles L7 application payload by forwarding application requests and responses between clients and servers. An overview of the *NetStar* framework is given in Fig. 4.2.

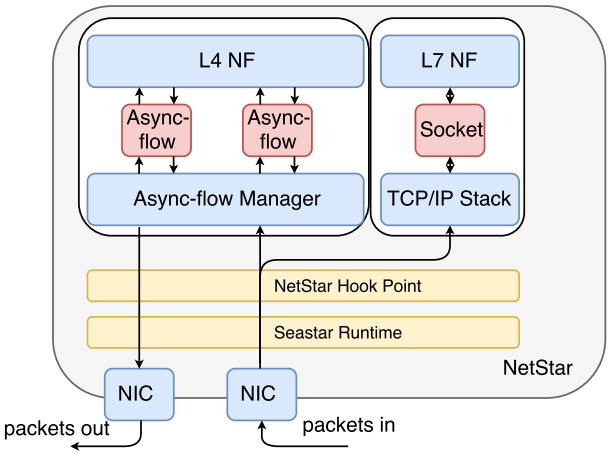


FIGURE 4.2: An Overview of *NetStar*.

NetStar is integrated with Seastar's runtime module, which uses DPDK to fetch packets from NIC queues directly into the user space. *NetStar* adopts a 'dual-stack' design, including modules to handle packets in L4 and L7 application traffic. When *NetStar* is used to implement an NF which handles L4 network packets (*i.e.*, transport-layer packets), the hook point is configured to forward the packets received through the runtime to the async-flow manager. When *NetStar* is used to implement an NF handling L7 application-level payload (*e.g.*, a proxy), the hook point forwards the flows to the user-space TCP/IP stack. In an NF which involves both L4 and L7 traffic, the hook point classifies the packets according to their IP addresses, forwards packets whose IP addresses matches configured destination IPs to the TCP/IP stack, and others to the async-flow manager. For an IDS that carries out malware detection as in Sec. 4.2.1, it handles L4 traffic (the TCP flows being inspected) and L7 traffic (database and MHR queries); packets carrying IP addresses of the database and MHR servers will be forwarded to the TCP/IP stack and the others to the async-flow manager.

Build an L4 NF. To process L4 packets, a special interface, referred to as the *async-flow interface*, is designed, which consists of an async-flow manager and several async-flow objects. The manager classifies received packets from the hook point into different network flows and pushes each flow to one async-flow object. The async-flow objects implement NF processing logic inside a simulated

packet processing loop, using the future/promise abstraction for asynchronous operations. We will discuss our detailed design of the async-flow interface in Sec. 4.4.

Build an L7 NF. To handle L7 application-level payload, we reuse Seastar's TCP/IP stack. The socket interface exposed by the TCP/IP stack is a modern re-implementation of the traditional POSIX socket based on the future/promise abstraction. An established TCP connection exposes a socket, through which an input stream for reading and an output stream for writing application data are available.

An NF built with *NetStar* uses a share-nothing thread model. Each NF runs multiple threads. Each thread is created and pined to one CPU core by the runtime. Each thread creates a complete set of modules, including the async-flow interface (one async-flow manager and multiple async-flow objects), the TCP/IP stack and the NF logic, and does not share these modules with any other thread. Incoming packets are distributed to different threads in a load balanced fashion, by configuring RSS [32] on the incoming NIC port. In this way, performance overhead associated with thread scheduling and shared resource contention is avoided.

4.4 Async-flow Interface

Our major challenge when designing the async-flow interface is how to exploit the power of the future/promise abstraction while exposing easy-to-use interfaces for building NFs. We carefully design a simulated packet processing loop and use returned future objects to concatenate asynchronous operations, to address the challenge.

4.4.1 Async-flow Manager

After a packet is delivered to the async-flow manager from the hook point, the manager first retrieves flow information from the packet header to build a flow

key. By default, the flow key is based on the flow 5 tuple (source/destination IP addresses, source/destination port and protocol type) of each TCP/UDP packet. The manager uses the key to identify an async-flow object from a hash map: if a corresponding async-flow object is found, the received packet is sent to the async-flow object; otherwise, the packet belongs to a new flow, and the manager creates a new async-flow object for the key, and updates the hash map. The manager also configures the new async-flow object with interested events (file hash event) and core processing logic of the NF, using programming interfaces exposed by the async-flow object.

4.4.2 Async-flow Object

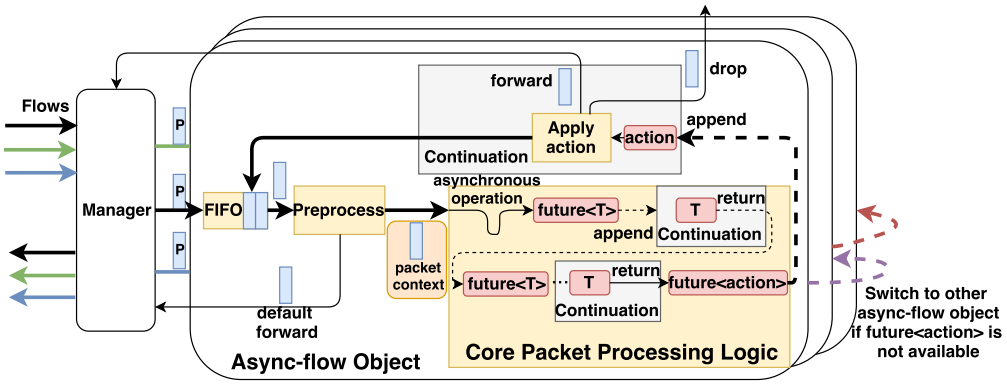


FIGURE 4.3: Workflow of async-flow objects (P represents a packet).

The async-flow objects are key components to implement asynchronous packet processing in an NF. We target the following design objectives in its design. (1) **Ease of Use.** The exposed programming interfaces should be easy to use, especially for programmers to implement NFs. (2) **Full Processing Asynchrony.** After an async-flow object launches an asynchronous operation when processing a packet, it should yield its execution immediately to other async-flow objects, without blocking.

In *NetStar*, the programming interfaces exposed by the async-flow object to a programmer include an interface for registering a special packet handler function that implements the core NF processing logic, and an interface for configuring

what events to be reported to the registered packet handler function. The interfaces are easy to use: registration of a packet handler function is very much the same as implementing a packet handler in existing NF architectures [4, 39].

To achieve the second objective, we strategically simulate a packet processing loop within each async-flow object via a series of future-continuation chains, pause a packet processing loop if any intermediate future object is unavailable, and only resume it when the future object becomes available. An illustration of the workflow within and among async-flow objects is presented in Fig. 4.3.

FIFO Queue. When a packet is sent from the manager to the async-flow object, the packet is first pushed into an FIFO queue and waits to be fetched by the packet processing loop.

Preprocessing. When a packet goes into the packet processing loop, it is first preprocessed to generate a set of events. For instance, if an in-sequence TCP packet arrives, the preprocessing may retrieve new payload from the reordering buffer to reconstruct the TCP byte stream and report it as an new event, along with arrival event of the TCP packet. In this way, some functionalities within the core processing logic can be offloaded to the preprocessing step for simplification. *NetStar* is equipped with four default preprocessors: a simple UDP preprocessor and a simple TCP preprocessor report packet arrival events; a TCP tracking preprocessor reports events on TCP packet arrival, TCP connection status change and the reconstruction of the TCP bytestream; a file extraction preprocessor extracts transmitted files from the byte stream and reports hash values of the files as events. All four preprocessors are equipped with a timer to report flow time-out events, when the async-flow object can gracefully shut down its packet processing loop.

After preprocessing, a packet *context* object is constructed, containing the current packet and all the generated events. If none of the events matches any of the events that the NF is interested in (*e.g.*, an IDS may only be interested in reconstructed TCP byte stream, not the arrival of an out-of-order TCP packet), a default action is performed, *i.e.*, forwarding the packet out (to the next-hop

NF or the flow destination), and another packet is fetched for preprocessing from the FIFO queue; otherwise, the packet context is sent to the core processing logic for further processing.

Core Packet Processing Logic. In the core processing logic, events from the packet context are retrieved and packet processing logic is executed together with a number of asynchronous operations, carried out through a chain of future/promise objects and continuation functions. To handle one asynchronous operation, a future object is constructed, associated with a promise object, and a continuation function is appended to the future object. When the promise object turns the future object into available state (when the asynchronous operation is completed), the appended continuation function is invoked, which carries out corresponding processing based on the received response and returns another future object for handling the next asynchronous operation. Multiple asynchronous operations can be handled using such a 'future/promise-continuation' chain.

Using malware detection in IDS as an example, the core processing logic includes two asynchronous operations: querying the local database and querying the MHR. A future object is first constructed to receive query response from the local database, and the database response is obtained as the concrete type of value in the future object. A promise is associated to turn the future into available state when the response returns. The appended continuation function is invoked then, and returns another future object, which receives query response form the MHR. The second future object obtains the MHR response as its concrete type of value and passes it to an appended continuation function, which may raise an alert in case that the response indicates the detection of a malware.

Final Future Object and Its Continuation Function. The last continuation function in the asynchronous 'future/promise-continuation' chain is forced to return a future object containing an action decision (*e.g.*, drop or forward in the malware detection example). This future object becomes available when all asynchronous operations in the core processing logic have been completed. A continuation function is appended to this future object, which receives the action decision from the future object and carries out the action accordingly (*e.g.*, drops

the packet or forwards it to the next hop. The current packet context is then destroyed, and the packet processing loop moves on to fetch another packet from the FIFO queue (if it is not empty), and recursively restarts itself.

Switching to Another Async-flow Object. After an async-flow object has issued an asynchronous operation, the thread that the async-flow interface is running in moves on to handle another async-flow object, *i.e.*, the packet processing loop in another async-flow object starts to fetch packets from its own FIFO queue and processes them. Different async-flow objects implement exactly the same logic in an NF. The programming style in implementing packet processing logic is very much like synchronous programming, while full asynchronous packet processing can be achieved.

4.4.3 Asynchronous Programming With *NetStar*: the IDS Example

Now we present the implementation of a malware detector using the async-flow interface designed above, which achieves similar functionalities as the Bro example in Sec. 4.2.1.

Our code sketch is given in Fig. 4.4. `run_async_flow_manager` runs the async-flow manager, which constantly checks (lines 2-3) incoming new flows. If a new TCP flow has arrived, a new async-flow object is created to handle the flow (line 4), and a malware detector object is associated with the new async-flow object (line 5). The malware detector object expresses its interest in the file hash event by registering this event (line 7). The core processing logic of the malware detector is then registered with the async-flow object by calling `register_loop_fn` (line 15), which is carried out upon occurrence of the file hash event.

First, the preprocessor reconstructs the TCP byte stream as it receives each new flow packet (code omitted from Fig. 4.4). After processing a packet, if the preprocessor detects a newly transmitted file, it calculates the hash value of this file and raises a the file hash event. In case that the preprocessor fails to detect

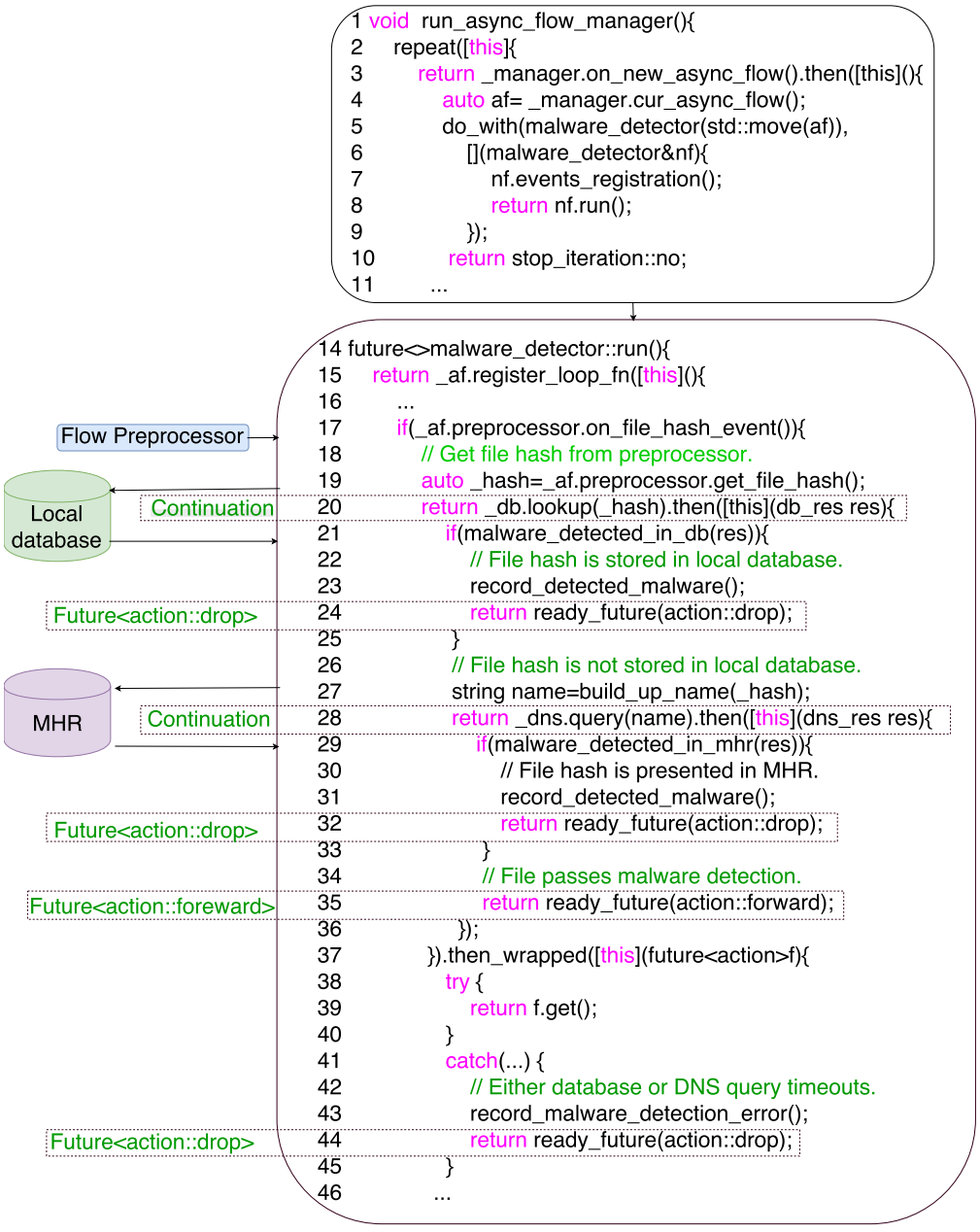


FIGURE 4.4: Malware detector using *NetStar*.

a transmitted file, it only raises a packet arrival event, which is ignored by the core processing logic, and the packet is forwarded out directly.

After a file hash event is generated (line 17), the core processing logic (lines 19-37)

implements similar malware detection steps as discussed in Sec. 4.2.1. A database query is initiated by `_db.loopkup` and a future object is obtained which contains the database response (line 20); a continuation function is appended for checking the query result. If some malware is detected, the packet is immediately dropped by returning a future object containing a drop action (line 24). Otherwise, it moves on to query the MHR with a DNS query using `_dns.query` (line 28). The continuation function appended to the future containing the DNS query response checks the query result. If some malware is detected, the continuation function returns a future object containing a drop action as well (line 32); otherwise, the continuation function returns a future object containing a forward action (line 35). The code from line 38 to line 47 handles potential exceptions generated during database or DNS queries, as a continuation function appended to the returned future object.

Comparison. The malware detector implemented with *NetStar* has the following advantages over that in Fig. 4.1. (1) *Simplified Implementation.* The malware detection process can be concisely implemented in *NetStar* using only 18 lines of code (lines 19-37). Our code mimics the sequential execution of database and DNS queries within a single function, instead of spreading the execution flow across multiple functions. (2) *Simplified Context Management.* In our code, the context information is put directly into the malware detection object. Different continuation functions (lines 15, 20, 28, 37) can easily visit the saved context by capturing a pointer (`this`) to the malware detection object. Since the malware detector object is only destroyed when the packet processing loop stops running (line 5), the above code guarantees that the malware detector object is always alive when the continuation functions are being called. (3) *Consolidated Error Handling.* Instead of defining two error handling functions, our implementation consolidates the error handling logic within a single continuation function (lines 38-47). The programmer can be more focused on the core NF processing logic, while simply chaining another continuation function at the end for handling all errors that might be generated from the core code.

4.5 Implemented NFs

4.5.1 Enhancing Seastar

NetStar is implemented on top of the Seastar framework, to exploit its C++-based future/promise implementation and DPDK support. Besides enabling raw L2-L4 packet processing that Seastar does not support, *NetStar* makes several improvements.

Eliminating Packet Copy. In its TCP/IP stack, Seastar does not use DPDK's RTE packet buffer [19]. Instead, it defines its own packet object and introduces two additional packet copies for copying the payload content of each RTE packet buffer to and from a packet object While the additional packet copy does not affect the performance of the TCP/IP stack for building L7 applications, it does pose additional overhead for L2-L4 NFs. To eliminate the packet copy and speed up packet processing, we directly use RTE packet buffer for L2-L4 NFs.

Multiple TCP/IP Stacks. Seastar treats its TCP/IP stack as a singleton. But an NF program may need more than one TCP/IP stack if the NF, *e.g.*, proxy [17], handles more than one NIC port. We improve Seastar to allow the creation of multiple TCP/IP stacks.

4.5.2 Implementing Representative NFs

We have built multiple representative NFs using *NetStar*.

NFs from the StatelessNF paper [65]. We reimplement four NFs from the StatelessNF paper, *i.e.*, firewall, NAT, IDS and load balancer. Our implementation follows the pseudo-code logic in the paper, and leverages our async-flow interface. The major differences are: (1) we do not need to set up a dedicated polling thread for each worker thread to poll each NIC queue; each thread in *NetStar* performs all the tasks including port polling and packet processing. (2) We use a fast in-memory key-value store, mica [70], which has a larger throughput than the RAMCloud database [77] used in StatelessNF. (3) In StatelessNF, a

unique thread is dedicated to contact RAMCloud for storing the per-flow states; with *NetStar*, the async-flow objects running in each thread can use the thread-local mica client library to contact mica server concurrently.

An HTTP reverse proxy. We use the TCP/IP stack in *NetStar* to implement an HTTP reverse proxy, whose functionality is similar to HAProxy (see Sec. 4.2.1) and TinyProxy []: it intercepts incoming TCP connections from clients, and relays HTTP requests to servers; it then receives HTTP responses from the servers and pushes them back to the clients.

An IDS that inspects HTTP payload. This IDS detects potential intrusion from the reconstructed HTTP request payload using our async-flow interface, which is more complicated than the IDS implemented following the StatelessNF.

A Malware Detector as introduced in Sec. 4.4.3. It utilizes the async-flow interface and the TCP/IP stack in *NetStar* concurrently.

4.6 Evaluation

In this section, we evaluate the performance of the various NFs built with *NetStar*. Even though future/promise paradigm can simplify asynchronous programming, using this paradigm adds additional overhead to dynamically create special runtime objects (Sec. 4.2.3). Therefore a natural question to ask is whether the performance of *NetStar* is good enough to approach line-rate processing. To answer this question, we design a series of experiments to evaluate the maximum throughput achieved by the *NetStar* NFs. We also compare *NetStar* with several NFs that are implemented with fast, low-overhead callback-based method to reveal the overhead associated with using future/promise abstraction. The second question involves the effectiveness of future/promise abstraction for simplifying the implementation. To answer this question, we adopt a similar quantitative methodology [90] used for evaluating F# future/promise abstraction and compare the required lines of code for implementing the core processing logic between *NetStar* NFs and NFs built with callback method.

4.6.1 Methodology

Testbed. Our testbed consists of 5 servers: three Dell R430 servers, each equipped with one Intel Xeon E5-2650 CPU 2.30GHz with 10 physical cores and 48GB memory, and two Supermicro servers, each with one Intel Xeon E5-1620 CPU 3.50GHz with 4 physical cores and 32GB memory. All servers are equipped with two Intel X710 10Gbps NICs and they are connected through a Dell 10Gbps Ethernet switch. We divide the servers to run our NFs and traffic generators (flow sources and destinations).

Traffic generation. We use two types of traffic generators. The first is a custom packet generator that we built on top of *NetStar*, which can generate 64-bytes UDP/TCP packets at 14Mpps (packets per second), *i.e.*, 10Gbps. The number of flows and the packet size for generation are adjustable. This generator is used to produce flows to NFs such as the packet forwarder (Sec. 4.6.2 and Sec. 4.6.7) and the NFs from StatelessNF paper (Sec. 4.6.3). To measure packet processing latency, this generator tags each produced packet with a generation time and computes the packet processing latency by an NF after receiving the packet back from the NF (in this case, source and destination of the flows are the same). The second traffic generator is the default HTTP benchmarking tool provided by Seastar. The tool runs in a client-server setting by sending a large number of HTTP requests from a client and receiving corresponding HTTP responses at the server. We modified this tool, including adding support for HTTP POST method for a client to upload files to the server and recording the HTTP transaction completion time, which is the time interval between sending HTTP request and receiving HTTP response. We use flows produced by this traffic generator to evaluate NFs such as the HTTP reverse proxy (Sec. 4.6.4), the IDS (Sec. 4.6.5) and the malware detector (Sec. 4.6.6).

Metric. We focus on three types of key performance metrics. (1) **Packet processing throughput** achieved by an NF, measured in the number of processed packets per second (Sec. 4.6.2, 4.6.3, 4.6.7), the number of processed HTTP requests per second (Sec. 4.6.4) and total bandwidth consumed by all the HTTP connections (Sec. 4.6.5, 4.6.6). (2) **Latency**, computed as average

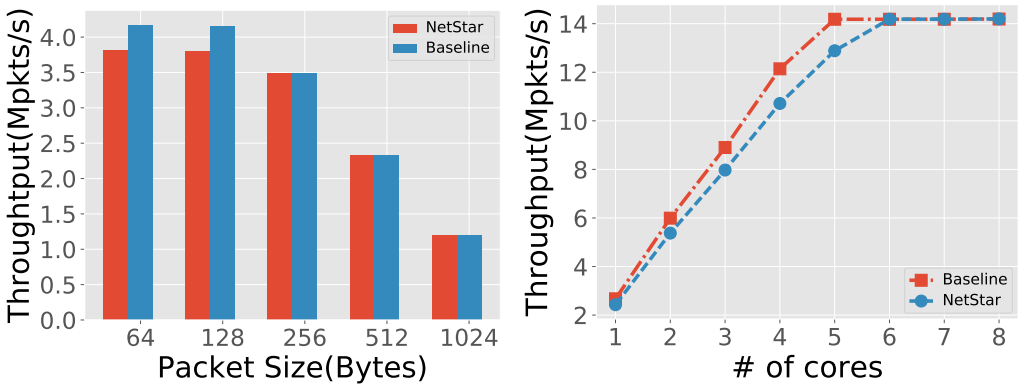
packet processing delay (Sec. 4.6.3) or the average HTTP transaction completion time (Sec. 4.6.4, 4.6.5, 4.6.6). (3) **Lines of code (LOC)** for implementing the core packet processing logic, meant for comparing the implementation difficulty using our future/promise based framework and the callback-based asynchronous programming.

Baselines. For comparison with NFs implemented using *NetStar*, we implement various NFs using callback based asynchronous programming (except for HAProxy [17] and TinyProxy [44]), following the practice in existing NF implementation.

4.6.2 Micro Benchmarking

We first carry out a set of micro benchmarking experiments to evaluate the performance of *NetStar* for basic packet processing and asynchronous operations.

4.6.2.1 Packet Processing Throughput



(A) (B)

FIGURE 4.5: Micro benchmarking on packet processing speed.

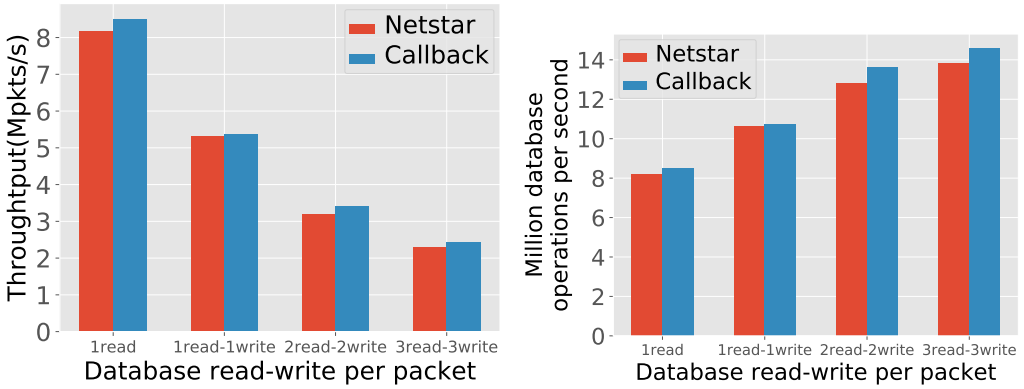
To scale to line-rate processing when handling complex asynchronous operations, *NetStar* should at least have adequate performance when being used as a simple packet forwarder. To evaluate this, we build a simple packet forwarder using *NetStar*, where the processing logic in each async-flow object is to directly forward

all the received packets. For comparison, we also build a packet forwarder using DPDK [19], which classifies packets into different flows by checking their flow-5-tuple against a cuckoo hash table adopted from the BESS virtual switch [3], and passes packets belonging to a flow to its flow object for forwarding. Compared with *NetStar*, the overhead associated with creating future object and chaining continuation functions are removed for the baseline forwarder, making it run faster.

Fig. 4.5a shows packet processing throughput when the NFs are running on a single thread. 10000 UDP flows with different packet sizes are dispatched to an NF at the rate that is slightly larger than the maximum throughput achieved by the NF. The throughput achieved by *NetStar* is very close to the baseline. When each NF runs on multiple threads (CPU cores) and 100000 UDP flows with 64-byte packets at 10Gbps line rate are injected, Fig. 4.5b shows that the throughput of *NetStar* is always close to the baseline. With 6 cores, *NetStar* achieves 10Gbps line rate. We evaluate *NetStar* using small packets to better understand its packet forwarding performance, since larger packets are easier to saturate the NIC bandwidth, making NIC a bottleneck rather than revealing any performance bottleneck in our framework. We believe that the throughput of *NetStar* can easily scale up to 40Gbps when it runs on a multi-core server and handles larger input packets.

4.6.2.2 Asynchronous Database Query

We next develop a simple NF in which each async-flow object reads from and writes to a mica database as its core processing logic. Each write stores a key-value pair, where the key is the packet's flow-5-tuple and the value is a 24-byte random array. Each read retrieves a key-value pair using the packet's flow-5-tuple. We also implement the same NF functionality on a callback-based framework built on top of DPDK: a callback function is registered with each database query; when the database response arrives, the callback function is invoked to handle the response. Compared with the *NetStar*-based implementation, this framework imposes minimum runtime overhead when executing asynchronous



(A) (B)

FIGURE 4.6: Micro benchmarking on asynchronous DB queries.

operations: the registered callback is simply a function pointer without any heap allocation, whereas dynamic allocation and deallocation of future/promise object on the heap are involved in a future/promise-based framework.

We use 100000 UDP flows with 64-byte small packets at 10Gbps line rate. We vary the number of database read/write carried out by the NF when processing each packet. After receiving each packet, those read/write operations are consecutively carried out, before sending the packet out. Both *NetStar* and the callback implementation run using 10 threads. Fig 4.6 shows that the packet processing throughput and the number of database queries that *NetStar* can achieve is very close to that of the callback-based implementation. The average performance gap under various database access patterns is 4.23% for packet throughput and 3.98% for database operations per second.

4.6.3 NFs from the StatelessNF paper[65]

We compare our *NetStar* based implementation of the four NFs with callback-based implementation (implemented following the pseudo-code logic), as well as with the reported performance of NFs implemented by StatelessNF authors. We inject into each NF 100000 TCP flows at 10Gbps and vary the size of the flow

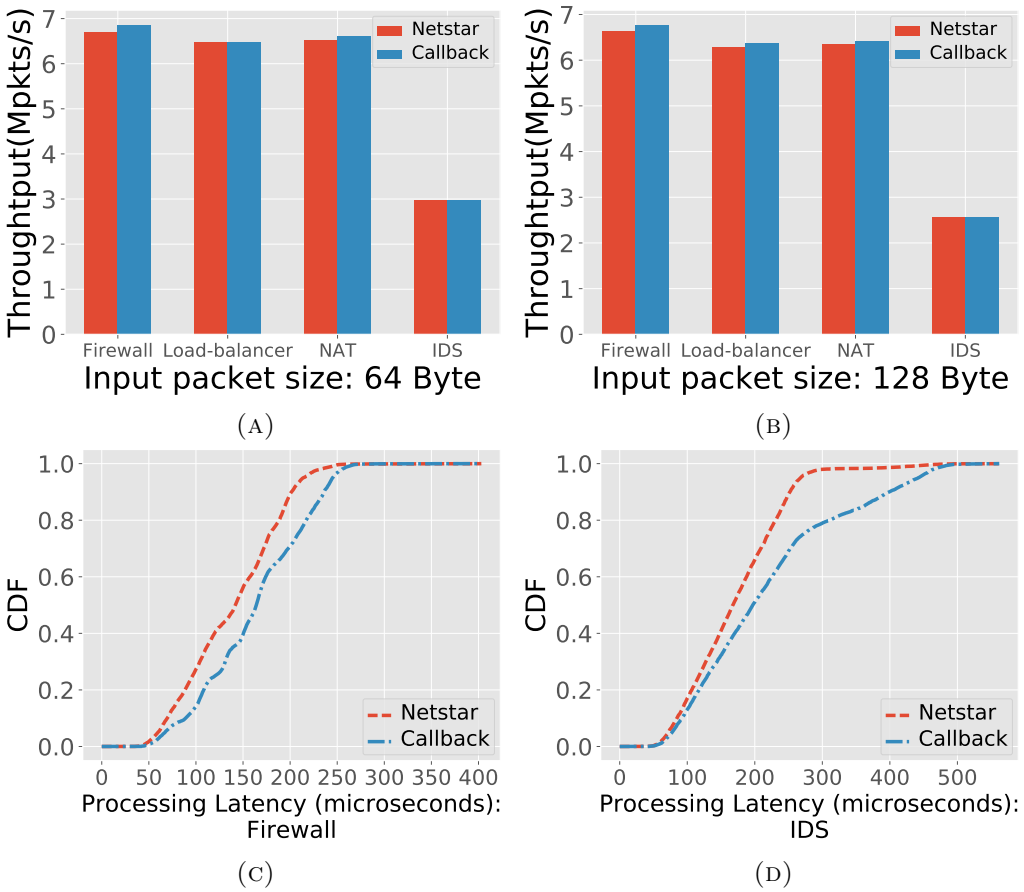


FIGURE 4.7: Performance comparison: NFs from the StatelessNF paper.

packets. Each NF runs using 10 threads on a server, and accesses the mica database on a different server.

In Fig. 4.7a and Fig. 4.7b, we observe that the packet processing throughput of our *NetStar* NFs is very comparable with the callback-based NFs. As compared to our observations in Sec. 4.6.2.2, when more complicated packet processing logic is involved here to process each packet besides executing asynchronous operations, the performance gap between *NetStar* and the callback-based implementation decreases to only 2%. We have also measured packet processing throughput when the packet size is 256, 512 or 1024 bytes. With a larger packet size, the NFs (except the IDS) can approach a 10Gbps processing rate. For IDS, the rate can reach 6.78 Gbps when processing 1024 bytes packets.

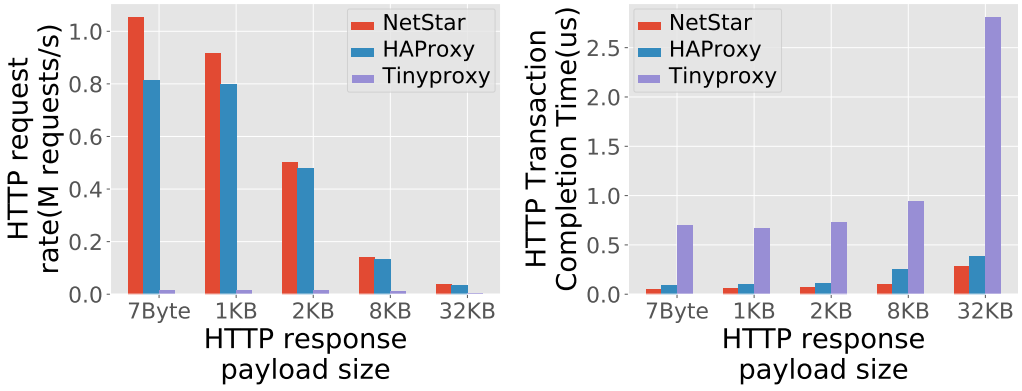
Fig. 4.7c and Fig. 4.7d show the CDF of packet processing latency at the firewall and the IDS when the packet size is 64 bytes. The packet processing latency between *NetStar* and the baseline is close to each other.

TABLE 4.1: LOC Comparison: NFs from the StatelessNF Paper

NF	Callback	*NetStar*	Reduction %
Firewall	52(44+8)	41(34+7)	21.1%
Load balancer	82(66+16)	64(57+7)	21.9%
NAT	99(79+20)	80(73+7)	19.1%
IDS	50(42+8)	43(36+7)	14%

To compare the implementation difficulty, we list the LOC for implementing the core processing logic of the four NFs using *NetStar*and the callback framework in Table 4.1. We only count the core processing code because a full-fledged implementation of each NF involves large volumes of auxiliary code on memory management, packet preprocessing and mica client library implementation. We can see that with the future/promise abstraction, the LOC can be reduced by as much as 21.9%. In the additions in Table 4.1, the number on the left is the total LOC for implementing packet processing functionalities, and the number on the right is the LOC for error handling. Due to consolidated error handling in *NetStar*, the same 7 lines of code are used for handling all errors in all NFs.

4.6.4 HTTP Reverse Proxy



(A) (B)

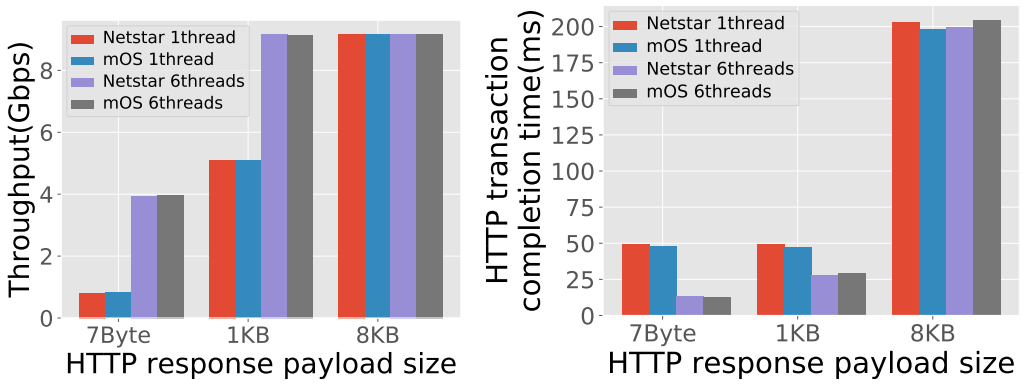
FIGURE 4.8: Performance comparison: different proxies.

We compare our proxy implemented using *NetStar* with both HAProxy version 1.8 [17] and TinyProxy version 1.8.4 [44]. TinyProxy does not use a callback-based asynchronous design; instead, it creates a new thread to handle each TCP flow in a synchronous manner, and relies on the kernel scheduler to schedule different threads. Each proxy runs on 10 threads. We use the HTTP benchmarking tool to generate 200 connections that go through the proxy, and then keeps producing HTTP requests at the generator's maximal capability.

In Fig. 4.8a, we see that *NetStar* out-performs HAProxy by up to 20% and is better than TinyProxy much more. As the payload size of HTTP response increases, the consumed bandwidth on the NIC gradually reaches 10Gbps, making the NIC a bottleneck and hence decreasing the performance gap. Fig. 4.8b shows that the smallest HTTP transaction completion time is achieved by *NetStar*.

Note that we implement our proxy by translating the processing logic in TinyProxy, as our future/promise based framework can easily mimic a synchronous execution flow. Such a translated *NetStar* proxy is easy to implement, runs in full asynchrony and has superior performance even compared to HAProxy, as shown by the above results.

4.6.5 IDS



(A) (B)

FIGURE 4.9: Performance comparison: IDS.

We compare the IDS built with *NetStar* and one built with mOS [63], one of the fastest frameworks for building middleboxes that handle application-level payload reconstruction. mOS extensively uses the callback-based event-driven model. Our IDS implementation on mOS relies on registering various callback functions to the mOS framework to obtain reconstructed TCP byte stream and parsing HTTP request. We generate 24K concurrent HTTP connections to pass through each IDS. The number of threads used by each IDSes varies, from 1 thread to 6 threads, to test the scalability.

In Fig. 4.9, we can see that the performance of *NetStar* is very close to that achieved by the mOS-based implementation. On the other hand, 493 LOC are written to implement the core processing logic in our *NetStar* IDS, whereas 689 LOC are used in the mOS-based implementation, achieving a 28% reduction.

4.6.6 Malware Detector

TABLE 4.2: Performance of Malware Detectors

	Throughput (file size: 8k)	Throughput (file size: 32k)	HTTP transaction completion time (file size: 8k)	HTTP transaction completion time (file size: 32k)
Asynchronous Malware Detector	5.02 Gbps	6.19 Gbps	12.74 ms	41.33 ms
Local Malware Detector	5.86 Gbps	6.79 Gbps	10.77 ms	37.84 ms

We compare our asynchronous malware detector that queries an external database and a DNS server (run in the same cluster as the malware detector) with a local malware detector that only tries to identify if the file hash exists in a local in-memory malware hash table. Both malware detectors are implemented using *NetStar*. We generate 1000 HTTP connections and send files in these HTTP connections from the client to the server. We randomly populate the content of the file with malware. The size of the files vary from 8K bytes to 32K bytes. Both malware detectors run using one thread.

In Table 4.2, we can see that compared with a local version, the asynchronous malware detector experiences a 14% throughput drop when the size of the transmitted file is 8K and 8.8% throughput drop when the size of the transmitted

file is 32K. The remote malware detection process adds a 1.97ms/3.48ms latency respectively.

We purposely compare a malware detector accessing external services with a local malware detector, instead of a callback-based malware detector which also accesses external services, in order to show the following: our NF's performance is very comparable with a fast local detector, not to mention one accessing external services; with *NetStar*, complicated asynchronous operations can be enabled on NFs without large performance drop, and the future/promise abstraction provided by *NetStar* enables easy implementation of the asynchronous operations.

4.6.7 Comparison with Coroutine

Finally, we compare *NetStar* with a coroutine based NF framework. In the coroutine framework, for each new flow, a new coroutine (thread) is created and a similar packet processing loop as in our async-flow object runs within the coroutine. Whenever a coroutine needs to waits for an asynchronous event, it yields its control to other coroutines in the NF.

We compare the *NetStar* NF carrying out database queries in Sec. 4.6.2.2 with a coroutine based implementation. 10000 UDP flows are sent to the two NFs, both running in one thread. Each NF reads the database once when processing one packet. A 750K pps throughput is achieved by the NF implemented with *NetStar*, while the throughput of the coroutine-based NF is only 147K pps. Further profiling reveals that the average coroutine switching time is around 352 nanoseconds. Even though the time is small, constant coroutine switching adds considerable overhead when an NF processes a large number of flows. Therefore, coroutine is a less desirable paradigm for implementing asynchronous NFs when compared with the future/promise abstraction.

4.7 Discussions

In our current *NetStar* implementation, after issuing an asynchronous operation, an async-flow object waits for the asynchronous operation to complete, while *NetStar* runs other async-flow objects. During this time, new flow packets coming to this async-flow object are buffered in its FIFO queue. This design choice is consistent with the semantics in most NFs, which requires processing packets in a flow strictly in sequence, and moving on to process the next packet only when all the processing on the current packet is completed. On the other hand, some NFs may carry out other asynchronous operations besides packet processing, *e.g.*, writing logs into external storage. Such an extra operation can be done concurrently with packet processing. To handle this case, we can launch a stand-alone asynchronous operation from the core NF processing logic and obtain a returned future object. As long as we do not add this future object into the "future-continuation" chain that leads to the creation of `future<action>` in Fig 4.3, the async-flow object will not wait for the completion of this stand-alone asynchronous operation and it proceeds with packet handling concurrently.

We have shown that using the future/promise paradigm can simplify asynchronous programming. Learning to use the future/promise abstraction may take some efforts, which makes use of various advanced C++14 features including move semantics, lambda functions and template meta programming. Our own experience is that, once a programmer has spent some time getting familiar with the future/promise abstraction, he can greatly improve his productivity when programming asynchronous code.

In addition, porting existing NF code to our *NetStar* framework is feasible, but may require some extra efforts on converting the callback-based programming interfaces to the future/promise abstraction, which usually involves exposing a new interface that returns a future object containing an asynchronous response. Once the concept of future/promise is mastered, this process can be made relatively easy.

4.8 Related Work

NetBricks proposes a fast and secure framework based on the Rust [80] language for building NFs. *NetStar* leverages future/promise abstraction to process dataplane traffic, and simplifies asynchronous programming with the async-flow interface. P4 [51] provides a high-level programming language for describing dataplane packet processing pipelines, but it has no intrinsic support for manipulating dataplane packets asynchronously as in *NetStar*. mOS [63] proposes a unified interface for managing connection oriented middleboxes. Using its interface, a middlebox can extract application-level payload and apply different callback functions to process the payload. *NetStar* shares a similar event layer as in mOS, where the packet is preprocessed to expose interested events to the core NF processing logic; however, *NetStar* uses the future/promise abstraction for event handling, and can effectively simplify implementation in case the NF requires to contact external services.

Except for being fast, NFs should be designed to resist various failures and handle large workload. StatelessNF [65] proposes a new architecture that separates the storage of per-flow states from the processing of packets. This advanced architecture requires efficient and simplified asynchronous programming support, which is nicely provided by *NetStar*. OpenNF [59], Split/Merge [86] and PEPC [84] all use flow migration for dynamical scaling of NF instances (by migrating flows out of hotspots). Implementation of a flow migration protocol typically involves complex asynchronous interactions, and *NetStar* can be potentially useful to simplify flow migration implementation.

4.9 Conclusion

This paper proposes *NetStar*, the first attempt to bring the future/promise abstraction to the NF dataplane for flow processing. *NetStar* adopts a dual-stack approach, enabling it to implement NFs processing packets at different layers. To fully utilize the power of future/promise, we carefully design an async-flow

interface that chains a number of future/promise and continuation functions for efficiently handling a series of asynchronous operations. Using *NetStar*, asynchronous programming in NFs is made easy, elegant, while good packet processing performance is still guaranteed. Our extensive evaluation shows that *NetStar* can effectively simplify asynchronous programming asynchronous in NFs, while easily achieve line-rate packet processing for NFs.

Bibliography

[1] 3GPP specification: 23.228. http://www.3gpp.org/ftp/Specs/html-info/23228.htm.

[2] Actor modle. https://en.wikipedia.org/wiki/Actor_model.

[3] Bess: Berkeley extensible software switch. https://github.com/NetSys/bess.

[4] Bro. https://www.bro.org/.

[5] Bro Scripting Tutorial. https://www.bro.org/sphinx/scripting/.

[6] C++ actor framework. http://actor-framework.org/.

[7] C++ lambda expressions. http://en.cppreference.com/w/cpp/language/lambda.

[8] C++ try-block. http://en.cppreference.com/w/cpp/language/try_catch.

[9] Coroutine. https://en.wikipedia.org/wiki/Coroutine.

[10] Docker container. https://www.docker.com/.

[11] EPC. http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core.

[12] Erlang. https://www.erlang.org/.

[13] ETSI NFV MANO framework. https://wiki.opnfv.org/display/mano/ETSI-MANO.

[14] Ffmpeg. https://ffmpeg.org/.

[15] Floodlight. http://www.projectfloodlight.org/floodlight/.

[16] Ftp. https://en.wikipedia.org/wiki/File_Transfer_Protocol.

[17] HAProxy. https://www.snort.org/.

[18] Http keep alive. https://en.wikipedia.org/wiki/HTTP_persistent_connection.

[19] Intel data plane development kit. http://dpdk.org/.

[20] IP Location Finder. https://www.iplocation.net/.

[21] iptables. https://en.wikipedia.org/wiki/Iptables.

[22] Linux virtual server. www.linuxvirtualserver.org/.

[23] LTE. http://www.3gpp.org/technologies/keywords-acronyms/98-lte.

[24] Malware Hash Registry. http://www.team-cymru.org/MHR.html.

[25] Netmap. info.iet.unipi.it/~luigi/netmap/.

[26] Netstar source code. https://github.com/sigcomm18p126/netstar.

[27] NFV. http://www.3gpp.org/DynaReport/23228.htm.

[28] Nfv white paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[29] Orleans. research.microsoft.com/en-us/projects/orleans/.

[30] PJSIP. http://www.pjsip.org/.

[31] Project Clearwater. http://www.projectclearwater.org/.

[32] Receiver side scaling. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[33] Rethinkdb blog post. https://rethinkdb.com/blog/improving-a-large-c-project-with-coroutines/.

[34] Scala akka. akka.io/.

[35] Scylla database. http://www.scylladb.com/.

[36] Seastar Project. http://www.seastar-project.org/.

[37] Secure Hash Algorithm 1. https://en.wikipedia.org/wiki/SHA-1.

[38] SIP: Session Initiation Protocol. https://www.ietf.org/rfc/rfc3261.txt.

[39] Snort ids. https://www.snort.org/.

[40] SoftLayer. www.softlayer.com.

[41] Squid caching proxy. www.squid-cache.org/.

[42] Telephone number mapping. https://en.wikipedia.org/wiki/Telephone_number_mapping.

[43] The NFVActor Project. https://github.com/duanjp8617/nfvactor.

[44] Tinyproxy. https://tinyproxy.github.io/.

[45] Tornado web framework. http://www.tornadoweb.org/.

[46] UMTS. http://www.3gpp.org/technologies/keywords-acronyms/103-umts.

[47] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[48] James W Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proc. of the eighth ACM/IEEE symposium on Architectures for networking and communications systems (ANCS'12)*, 2012.

[49] Q. Z. Ayyub, P. P. Krishna, and S. Vyas. Klein: A minimally disruptive design for an elastic cellular core. In *SOSR*, 2016.

[50] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. Enabling End-host Network Functions. In *Proc. of ACM SIGCOMM*, 2015.

[51] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[52] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proc. of ACM SIGCOMM*, 2016.

[53] S. Brent, C. Alan, and F. Wes. Past: Scalable ethernet for data centers. In *ACM Proc. of CoNext*, 2012.

[54] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, 45:105–131, April 2016.

[55] H. Chi-Yao, K. Srikanth, and M. Ratul. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.

[56] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[57] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in javascript. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[58] Aaron Gember, Robert Grandl, Ashok Anand, Theophilus Benson, and Aditya Akella. Stratos: Virtual Middleboxes as First-class Entities. Technical report, UW-Madison 2012.

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

[60] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical report, EECS Department, University of California, Berkeley, 2015.

[61] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. of the USENIX Annual Technical Conference (ATC '10)*, 2010.

[62] Jinho Hwang, KK Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, 2015.

[63] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 113–129, Boston, MA, 2017. USENIX Association.

[64] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *Proc. of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*, 2014.

[65] 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.

[66] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, page 3. ACM, 2013.

[67] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *Proc. of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, 2016.

[68] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[69] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *Acm sigplan notices*, volume 42, pages 189–199. ACM, 2007.

[70] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[71] R. Luigi, L. Giuseppe, and M. Vincenzo. Speeding up packet i/o in virtual machines. In *ANCS*, 2013.

[72] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*, 2014.

[73] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[74] Sanjeev Mohindra, Daniel Hook, Andrew Prout, Ai-Hoa Sanh, An Tran, and Charles Yee. Big Data Analysis using Distributed Actors Framework. In *Proc. of the 2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[75] Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. Optimizing Distributed Actor Systems for Dynamic Interactive Services. In *Proc. of the Eleventh European Conference on Computer Systems (EuroSys'16)*, 2016.

[76] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[77] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[78] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. In *Proc. of the 9th Annual European Symposium on Algorithms*, 2001.

[79] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a Framework for NFV Applications. In *Proc. of the 25th Symposium on Operating Systems Principles (SOSP'15)*, 2015.

[80] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[81] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[82] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. of ACM SIGCOMM*, 2013.

[83] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. A high performance packet core for next generation cellular networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 348–361. ACM, 2017.

[84] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. A high performance packet core for next generation cellular networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 348–361, New York, NY, USA, 2017. ACM.

[85] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. of the 4th Annual Symposium on Cloud Computing (SOCC'13)*, 2013.

[86] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.

[87] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012.

[88] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, et al. Rollback-Recovery for Middleboxes. In *Proc. of SIGCOMM*, 2015.

[89] J. Sushant, K. Alok, and M. Subhasree. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[90] Don Syme, Tomas Petricek, and Dmitry Lomov. The f# asynchronous programming model. In *International Symposium on Practical Aspects of Declarative Languages*, pages 175–189. Springer, 2011.

[91] W. Timothy, S. Prashant, and V. Arun. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[92] J. Vouillon. Lwt: a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ACM, 2008.

[93] R. Yi, P. Tuan, C. Cheng, and Y. Wei. Dynamic auto scaling algorithm (dasa) for 5g mobile networks. In *GLOBECOM*, 2016.

[94] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, KK Ramakrishnan, and Timothy Wood. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *Proc. of the 12th International on Conference on emerging Networking EXperiments and Technologies (CoNEXT '16)*, 2016.

[95] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phil Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proc. of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox'16)*, 2016.