NFVactor: A Resilient NFV System Using the Distributed Actor Model

Jingpu Duan, Xiaodong Yi, Shixiong Zhao, Chuan Wu, Heming Cui, and Franck Le

***Abstract*— Resilience functionality, including failure resilience and flow migration, is of pivotal importance in practical network function virtualization (NFV) systems. However, existing failure recovery procedures incur high packet processing delay due to heavyweight process checkpointing, while flow migration has poor performance due to centralized control. This paper proposes *NFVactor*, a novel NFV system that aims to provide lightweight failure resilience and high-performance flow migration. *NFVactor*enables these by using actor model to provide a per-flow execution environment, so that each flow can replicate and migrate itself with improved parallelism, while the efficiency of the actor model is guaranteed by a carefully designed runtime system. Moreover, *NFVactor*achieves transparent resilience: once a new network function (NF) is implemented for *NFVactor*, the NF automatically acquires resilience support. Our evaluation result shows that *NFVactor*achieves 10-Gbps packet processing, flow migration completion time that is 144 times faster than the existing system, and packet processing delay stabilized at around 20 µs during replication.**

***Index Terms*— NFV, SDN, actor model, failure resilience, flow migration, elastic scaling.**

I. Introduction

NETWORK function virtualization (NFV) advocates moving *network functions* (NFs) out of dedicated hardware middleboxes and running them as virtualized applications on commodity servers [1]. To enable effective large-scale deployment of virtual NFs, a number of NFV management systems have been proposed in recent years [2]–[7], 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 [8], [9] is crucial for stateful NFs. Many NFs maintain important per-flow states [10]: IDSs such as Bro [11] store and update protocol-related states for each flow for issuing alerts for potential attacks; firewalls [12] parse TCP SYN/ACK/FIN packets and maintain TCP connection related states for each flow; load balancers [13] 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 [8].

Efficient flow migration [14]–[16] is important for long-lived flows in case of dynamic system scaling. Existing NFV systems [2], [6] 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 [17]; video-streaming [18] and file-downloading [19] 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 [15].

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 [8], [9] is typically implemented through checkpointing: each NF process is regularly checkpointed by saving its process image and execution traces to a reliable storage, and any failed NF process can be recovered by the system using the saved image and traces. 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 [8], [9].

Flow migration in existing systems [14], [15] 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 [20]. However,

Manuscript received October 11, 2018; revised December 27, 2018; accepted January 11, 2019. Date of publication February 4, 2019; date of current version February 14, 2019. This work was supported in part by grants from Hong Kong RGC under Contracts HKU 17204715, 17225516, and C7036-15G (CRF), in part by NSFC under Grant 61628209, in part by the HKU URC Matching Funding, in part by Huawei HIRP under Grant HO2016050002BE, in part by the HK RGC Early Career Scheme under Grant 27200916, in part by the RGC General Research Fund under Grants 17207117 and 17202318, in part by the Croucher Innovation Award, and in part by the Project PCL Future Regional Network Facilities for Large-scale Experiments and Applications under Grant PCL2018KP001. *(Corresponding author: Jingpu Duan.)*

J. Duan is with the Institute of Future Networks, Southern University of Science and Technology, Shenzhen 518005, China, and also with the PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen 518005, China (e-mail: duanjp@sustc.edu.cn).

X. Yi, S. Zhao, C. Wu, and H. Cui are with the Department of Computer Science, The University of Hong Kong, Hong Kong (e-mail: xdyi@cs.hku.hk; sxzhao@cs.hku.hk; cwu@cs.hku.hk; heming@cs.hku.hk).

F. Le is with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: fle@us.ibm.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2019.2894287

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 [15] reports that thousands lines of patch code must be added to existing NF software [11], [21] 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 the design and implementation of *NFVactor*, a new software framework for building NFV systems with high-performance flow migration and lightweight failure resilience. Unlike previous systems [8], [9], [14], [15] 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 [22], 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* has three major novelties over existing systems.

▷ *Lightweight failure resilience.* With the actor abstraction and cleanly separated NF states, each flow actor in *NFVactor* can independently replicate itself by constantly saving its per-flow state to another actor that serves as a replica. This lightweight resilience operation eliminates the overhead of checkpointing the entire NF process. As a result, failure resilience in *NFVactor* achieves 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. The protocol designed for the migration achieves loss avoidance property, making packet loss a rare thing even when concurrently migrating many flows.

▷ *Transparent resilience. NFVactor* ensures that once the NFs are correctly implemented with the provided APIs, failure resilience of the NFs is immediately achieved. *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.

Our major technical challenge is to build an actor runtime system to satisfy the stringent performance requirement of NFV application. Even one of the fastest actor runtime systems [23] 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 [24] fast packet I/O framework [25] 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. When being compared with one of the fastest actor libraries [23], the packet processing throughput of our customized runtime increases by over 100% due to the improved actor scheduling strategy, while the speed for transmitting remote actor messages increases up to over 500% due to the removal of the kernel networking stack.

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 [15], flow migration completion time in *NFVactor* can be 144 times faster. Compare with FTMB [8] for replication performance, *NFVactor* achieves similar packet processing throughput and recovery time, but with packet processing latency stabilized at around 20 microseconds.

In summary, we make the following contributions in this paper:

- We introduce actor model to build resilient NFV system, and we demonstrate that the actor model can effectively improve the parallelism of resilience functionalities for better performance.
- We identify performance bottlenecks in existing actor runtime system when processing network function workload, and we design a new actor runtime to overcome these bottlenecks.
- We implement *NFVactor* and extensively evaluate *NFVactor* to verify its effectiveness. The source code and detailed documentation of *NFVactor* is available at [26].

The rest of the paper is organized as follows. Section II discusses the related work of *NFVactor*. Section III motivates the use of actor model. Section IV presents the overall architecture of *NFVactor* framework. Section V describes the APIs exposed by *NFVactor* for building network functions. Section VI discusses how failure resilience and flow migration are done in *NFVactor*. Section VII introduces key techniques used to implement *NFVactor* runtime. Section VIII presents the evaluation results. Section IX discusses how to run *NFVactor* in multi-network environments. Section X gives a concluding remark and discusses future directions for improving *NFVactor*.

II. Related Work

Since the introduction of NFV [1], a broad range of studies have been carried out, to bridge the performance gap between specialized hardware boxes and virtualized network functions, dynamically manage NFV systems, migrate flows between different NF instances for better scalability, replicate important NF states to resist devastating failures, *e.t.c.* The rest of this section first reviews several existing works on various aspects of NFV technology. Then a comparison is given between *NFVactor* and multiple representative works that directly influence the design and implementation of *NFVactor*.

High-performance Packet Processing. A major challenge in NFV technology is how to improve the performance of packet processing in virtualized environment. Both NetVM [27] and ClickOS [28] use a mapped memory area as an efficient communication channel between hypervisor and virtual machine to accelerate packet processing. NetVM also relies on the DPDK library [24] to fetch packets from virtual NICs. Similar to NetVM, *NFVactor* relies on DPDK to provide fast packet accessing. A major difference between *NFVactor* and NetVM is that the runtimes of *NFVactor* run in lightweight containers instead of heavyweight virtual machines, to achieve better performance.

Modern NFV system usually runs inside a cluster, where multiple NF instances collaborate to work. Inspired by Click [29] modular router, the BESS [30] software switch is a high-performance tool for connecting different NF instances within a cluster. *NFVactor* reuses BESS when connecting different runtimes. The runtime scheduler of *NFVactor* is also inspired by the module graph scheduler of BESS, and becomes a key enabler for an efficient actor library that is suitable for various NF tasks.

NFV System Management. Complex NFV systems are usually designed as distributed systems running in high-performance clusters or geo-distributed datacenters. To better manage the NFV system, Stratos [6] predicts future workload and designs an algorithm to provision NF based on the prediction inside a single datacenter. E2 [2] uses a similar approach to provision NF instances according to workload prediction, but E2 uses BESS to construct a high-performance dataplane and ensure line-rate processing. Neither Stratos nor E2 support flow migration, so they can not promptly eliminate overload caused by long-lasting flows like *NFVactor*. Flurries [31] manages NFV system at the granularity of per-flow NFs. For each new flow, Flurries launch a new container running a per-flow NF. *NFVactor* manages NFV system at the granularity of flow actors. For each flow, a new actor is created to process the flow. Efficient flow migration and failure resilience are then enabled on top of the actor model of *NFVactor*, which are not available in Flurries.

The management of NFV system is further extended to geo-distributed datacenters to cover a wider area. Duan *et al.* [32] propose to dynamically scale distributed service chains over geo-distributed clouds. Qazi *et al.* [33] and Bagaa *et al.* [34] study how to scale EPC cores across geo-distributed datacenters. Qazi *et al.* [33] focus on implementing a global management system while Bagaa *et al.* [34] propose a placement algorithm for the core NFs of EPC. *NFVactor* is optimized for a high-performance cluster and can not directly run in a geo-distributed environment. The geo-distributed environment has to be divided into several clusters first, then *NFVactor* can be independently deployed inside each cluster.

Flow Migration for Dynamic Scaling. Flow migration is a key technique for dynamic scaling as it can promptly eliminate overload caused by long-lasting flows. To implement flow migration, existing systems [14], [15], [35] rely on a centralized SDN controller to carry out the migration protocol, involving non-negligible overhead. *NFVactor* overcomes this issues using a largely distributed framework to achieve efficient flow migration, where migration is achieved directly between runtimes with only 3 steps of request-response. StateAlyzr [36] uses static analysis to automate flow state extraction and simplify human effort for enabling flow migration. However, enabling high-performance flow migration still requires a holistic design like *NFVactor*. Qazi *et al.* [16] design a new ECP core and use flow migration to alleviate system overload. The migration protocol used by Qazi *et al.* [16] is not carefully designed and may cause packet loss during migration, while the migration protocol of *NFVactor* satisfies loss avoidance property and does not incur packet loss in practice. In [37], an approach for ensuring end-to-end QoS of user traffic is proposed using the queue support of OpenFlow. *NFVactor* preserves end-to-end QoS by dynamic scaling through flow migration, and achieves better performance with an efficient architecture.

Replicate NF State to Resist Failure. Failure resilience in existing systems [8], [9] usually involves check-pointing the entire NF process image, which may pause the NF process, lose packets and prolong the processing delay. *NFVactor* is able to replicate individual flow by saving the flow state to a replica, resulting in minimized loss and delay. StatelessNF [38] 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*: StatelessNF stores flow states in a reliable database [39] 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.

NFV in Mobile Networks NFV technology is extensively adopted in mobile networks to dynamically manage resources and improve user experience. There are many theoretical advancements for applying NFV technology in mobile networks. The works in [40]–[43] study how to optimally place

TABLE I

COMPARISON WITH REPRESENTATIVE WORKS

Project Name	Support Flow Migration	Support Failure Resilience	Rely On Special-purpose Hardware
OpenNF [15]	Yes, low-performance.	Yes, low-performance.	**No.**
Split/Merge [14]	Yes, low-performance.	Yes, low-performance.	**No.**
FTMB [8]	No.	**Yes, high-performance.**	**No.**
StatelessNF [38]	No.	**Yes, high-performance.**	Yes, rely on RDMA.
NFVactor	**Yes, high-performance**.	**Yes, high-performance.**	**No.**

core network functions of 5G mobile networks in both carrier and public clouds under different constraints. Addad *et al.* [44] propose an algorithm for computing the optimal network slice in a 5G mobile network to improve the efficiency of resource allocation. While the focus of *NFVactor* is to build an NFV system with high-performance resilience functionality, these theoretical results can be borrowed in the future version of *NFVactor* to improve the scheduling of runtimes in the cluster.

Comparison with Representative Works. We present a comparison between *NFVactor* and several representative works in the fields of flow migration and failure resilience. The result is shown in Table I. We can see that *NFVactor* is a complete solution as it provides high-performance flow migration and failure resilience, while not depending on special-purpose hardware.

III. MOTIVATIONS FOR USING THE ACTOR MODEL

Systems that enable failure resilience [8], [9] and flow migration [14], [15] 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 modeled by actors.

The actor programming model [22], [45]–[47] has a long history of being used to construct massive, distributed systems [22], [46], [48], [49]. 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 [47], 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.

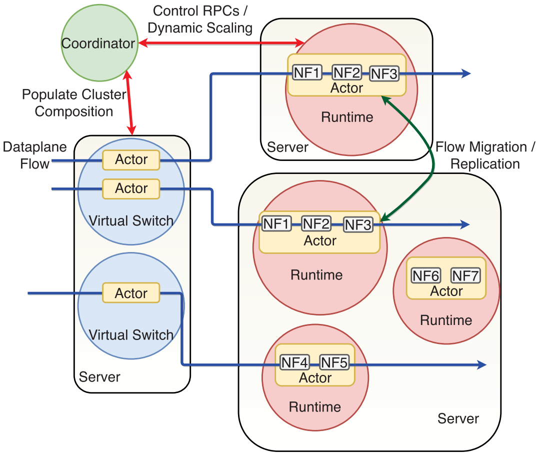


Fig. 1. An overview of the basic components of *NFVactor*. Three clusters are shown in this figure: a cluster for provisioning service chain ‘NF1→NF2→NF3’, a cluster for service chain ‘NF4→NF5’ and a cluster for service chain ‘NF6→NF7’.

There are several popular actor frameworks [45]–[47], [50], but none of these frameworks are optimized for building NFV systems. In our initial implementation, we built *NFVactor* on top of libcaf [47], one of the fastest actor system [23]. But the overall performance turned out to be less than satisfactory. This motivates us to customize a high-performance actor runtime system for *NFVactor*.

IV. THE NFVACTOR FRAMEWORK

A. Overview

At the highest level, *NFVactor* has a layered structure as shown in Fig. 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 (Section IV-B) is the execution environment of flow actors, running on a Docker container [51] for quick launching and rebooting, and is assigned a globally unique ID upon creation. A virtual switch is a special runtime (Section IV-C) and serves as the gateway to runtimes.

Runtimes and virtual switches are partitioned into several virtual clusters. In a cluster, runtimes are initialized with the same service chain (Section IV-B.3) and the virtual switches dispatch flows 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

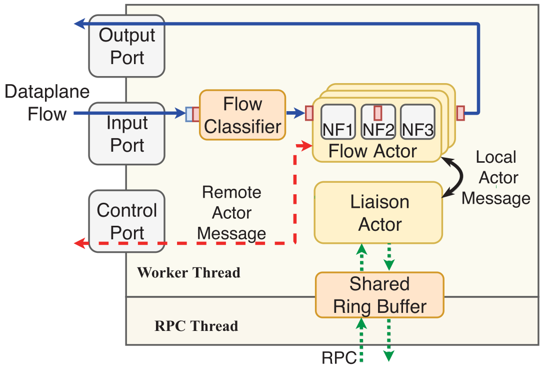


Fig. 2. The internal structure of a runtime.

runtimes and virtual switches through a series of control RPCs (Section IV-D).

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 Section VI.

B. 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.

1) Internal Structure: Fig. 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. 2 and works in a run-to-completion mode. It 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 (Section IV-D). The three ports are high-speed virtual NICs (ZeroCopyVPort in BESS [25]) 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.

2) Work Flow: The runtime has three basic work flows:

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 (Section VII) 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) 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 (Section V) of each NF according to the service chain structure to implement the service chain processing logic.

C. 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 [14], [15], the route update process is lightweight and improves flow migration performance for *NFVactor*.

D. Coordinator

The coordinator in *NFVactor* is responsible for basic cluster management routines. As compared to centralized controllers in the existing NFV systems [14], [15], 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 Table II. It uses $poll\_workload()$ to acquire the current workload on a runtime. It updates cluster composition (including MAC addresses of

TABLE II

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.

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.

The last three RPCs shown in Table II 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, containing the number of dropped packets on the input port, the current packet processing throughput and the number of active flows. In the current *NFVactor* prototype, the runtime is identified as overloaded if the number of dropped packets exceeds a fixed threshold (100 as in our experiments). This is an effective overload indicator for *NFVactor*: an overloaded runtime can not timely poll all the packets from its input port, therefore increasing the number of dropped packets significantly, while the CPU usage is rendered ineffective as the worker thread is a busy-polling thread and uses 100% of the CPU all the time.

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 (500 as in our experiments) from overloaded runtime to the new runtime, until half of the workload on the overloaded runtime is migrated away. If the new runtime becomes overloaded, more runtimes are added.We add new runtimes instead of moving flows across existing runtimes, since the load on existing runtimes is largely balanced, due to the load-balancing functionality of virtual switches.

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.

TABLE III

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.

V. NF APIs

To create an NF with full resilience support, the programmer should properly implement the APIs listed in Table III and ensure that the implemented APIs correctly satisfy the usage description in Table III. We follow two principles when designing these APIs.

First, StatelessNF [38] and Split/Merge [14] 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. He should further implement the last four APIs in Table III to release 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 (Section VI).

A. 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 Table III to synchronize the flow state with the shared state for each NF, followed by executing some clean-up procedures.

B. 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 [52], 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 Table III to synchronize the shared state.

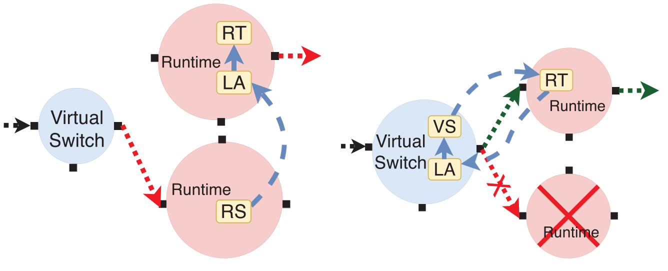
The load balancer forwards each input flow to a server with the smallest workload among a set of backend servers. To achieve this, after selecting a server (per-flow state), the load balancer increases the workload counter (shared state) of the selected server to reflect the load balancing decision. 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 Table III.

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 Table III: 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.

VI. System Management Operations

A. Fault Tolerance

1) Replicating Runtimes: 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 [8], [9]. 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.



(a) Flow replication. **(b)** Flow recover when the original runtime has failed.

Fig. 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.)

The detailed flow replication process is illustrated in Fig. 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 first stores latest flow states contained in the message, then sends the packet out, as shown in Fig. 3a.

This design guarantees the same *output-commit* property as in [8]: the packet is sent out from the system only when all the state changes caused by the packet have been replicated. This property is straightforward to verify. The output packet can only be sent out from the replica flow actor. To produce an output packet, the replica actor must have saved the updated flow state associated with the packet.

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 III) and the flow is successfully restored on runtime R (Fig. 3b). To recover a runtime processing n flows, the time complexity is $O(n)$ as each flow is recovered using one request-response.

2) Replicating Virtual Switches: Since a virtual switch is in fact a special runtime (Section IV-C), the virtual switch can be replicated in the same way as described in Section VI-A.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) 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 [53]. 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.

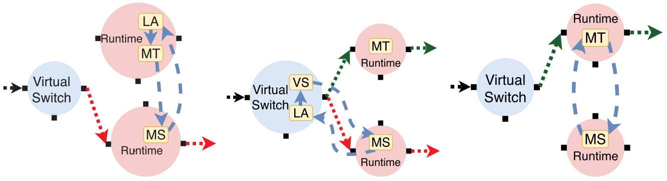
B. Flow Migration

1) Flow Migration Protocol: 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. 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. To migrate n flows out of a runtime, the time complexity is $O(n)$ as each flow is migrated within 3 steps of request-response.

First req-res 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.

Second req-res 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, which is encapsulated in a packet and traverses the same network path as the flow packets, 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 third step from the source actor. The migration source actor keeps processing received flow packets until it receives the response from the virtual switch.



(a) First req-res step. **(b)** Second req-res step. **(c)** Third req-res step.

Fig. 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.

Third req-res 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 III) 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 III) to synchronize the shared states and then destroys itself.

2) Loss Avoidance Property: If the following assumptions hold,

1) The network is lossless and does not reorder packets.
2) The buffer of the migration target actor does not overflow.
3) Runtimes involved in flow migration do not overload.

then the flow migration protocol of *NFVactor* satisfies the following property.

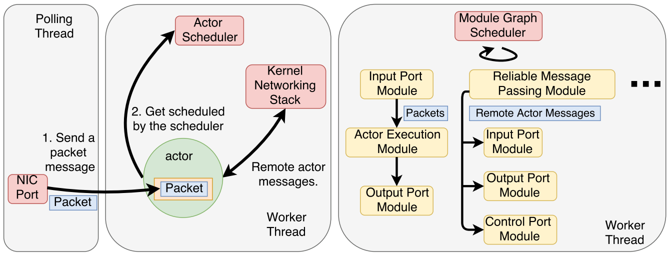
Loss Avoidance: All the packets are processed by migration source and target runtimes in the same order that they are received at the virtual switch. No packets are lost and no packets are processed out of order during flow migration.

To prove this property, let p_i be the i^{th} packet of a flow f received by the virtual switch, let S_i be the updated flow state of f after p_i is processed by the flow actor, let $req_i(1 <= i <= 3)$ be the request sent during the i^th req-res step and $res_i(1 <= i <= 3)$ be the response, let t_i be a time point during the flow migration process. Since runtime has a single thread, everything in the runtime happens in a linear order.

Consider the time point t_1 when the virtual switch actor receives req_2. Assume that the virtual switch actor has received $p_1 - p_k(1 < k)$ packets before t_1, then $p_1 - p_k$ are sent to the migration target actor as the destination runtime of virtual switch actor is not changed before t_1. After t_1, the virtual switch actor responds res_2 to migration source actor. When the migration source actor receives res_2, it should finish processing $p_1 - p_k$ and update the flow state to S_k. This is because res_2 is encapsulated in a single packet and traverses the same network path as $p_1 - p_k$. According to assumption 1, res_2 must have arrived at migration source after after p_k. Therefore, the flow state contained in req_3 is S_k.

On the other hand, as the destination runtime of virtual switch is changed to migration target actor after t_1, packets starting from p_{k+1} are sent to the migration target actor by the virtual switch actor after t_1.

Consider another time point t_2 when the migration target actor receives req_3. Assume that before t_2, migration target



(a) Libcaf runtime, which is abandoned due to its unsatisfactory performance.

(b) *NFVactor* runtime, which is designed to overcome the drawbacks of libcaf runtime.

Fig. 5. Comparison of runtime architecture.

actor has received and buffered $p_{k+1} - p_m (k+1 < m)$ packets without dropping a single packet (assumption 2). After t_2, the migration target actor starts processing all the buffered packets using flow state S_k, and updates the flow state to S_m. According to assumption 3, no packets are dropped due to overload when migration target actor processes the buffered packets. Therefore, when migration target finishes processing the buffer, it continues to process packets starting from p_{m+1}. This proves the loss avoidance property.

3) Summary: In practice, the assumptions of the loss avoidance property can be satisfied for most of the time. In fact, packet drop caused by the flow migration protocol rarely happens, even when concurrently migrating hundreds of thousands of flows (Section VIII-D). It has been a common understanding that providing good properties for flow migration would trade off the performance of flow migration [15]. *NFVactor* mitigates this trade-off using distributed, high-performance flow migration based on the actor model.

VII. Implementation

A. Libcaf Runtime

Throughout the development process of *NFVactor*, we initially choose libcaf [47] as the runtime, whose architecture is shown in Fig. 5a. We prioritize libcaf over other actor runtime systems [22], [45], [46] because libcaf has better performance [23]. On the other hand, it is easier to integrate a high-performance packet I/O framework into libcaf due to its C++-based implementation.

However, the performance of libcaf [47] 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. 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 using DPDK [24] and send received packets to actors running in another worker thread by enqueueing the packet into actor's message queue. As verified in Section VIII-A.1, there is an expensive synchronization overhead between the polling thread and the worker thread, which decreases packet processing throughput by over 100%. *Second*, libcaf runtime still relies on inefficient kernel networking thread to exchange remote actor messages with other runtimes.

B. NFVactor Runtime

Realizing the problems, we abandon libcaf runtime and design a new actor runtime for *NFVactor* as shown in Fig. 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.

1) Module Graph Scheduler: Inspired by the scheduler design of BESS [25] and Click [29], the module graph scheduler keeps scheduling several module graphs to run. 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. For our current prototype implementation, the scheduler uses round-robin algorithm to schedule all the module graphs, which is chosen to maximize the performance of the runtime.

Fig. 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.

2) 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. 5b

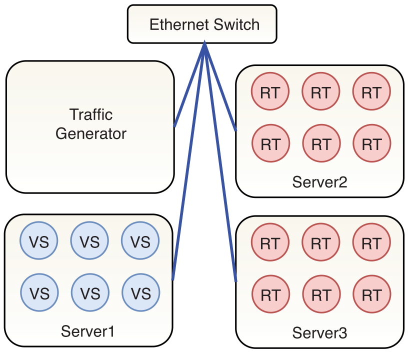


Fig. 6. The network topology of the testbed for evaluating *NFVactor*. **VS** - virtual switch runtime, **RT** - runtime.

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 (Section VIII-A.2).

Since our goal is to reliably transmit remote actor messages over an inter-connected L2 network, we do not use user-level TCP [54], 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 (Section VI-B.1), reliable actor message passing enables us to implement loss-avoidance migration (Section VI-B.2) with ease.

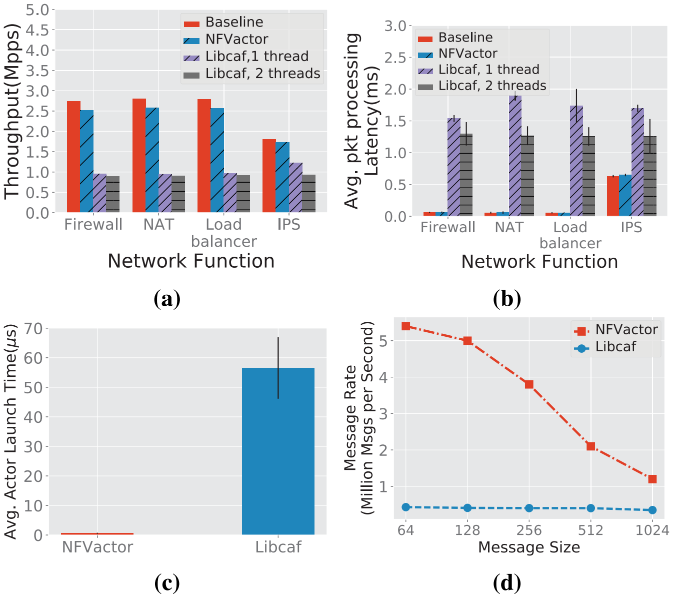
VIII. Evaluation

We evaluate *NFVactor* on a testbed with 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 topology of the testbed is shown in Fig. 6. The servers are connected through a 10GB Dell Ethernet switch. 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 two servers are used to run runtimes.

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 rely on the traffic generation module of BESS [25], which has been used for testing complex NFV system [2] and is capable of generating input traffic up to 10Gbps (at around 14Mpps) with 64-byte packets.

A. Performance of the Runtime

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 to verify the performance advantages of our customized actor 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 [55] without using the actor abstraction. By comparing with fast baseline NFs, we can observe the overhead imposed by the actor runtime.



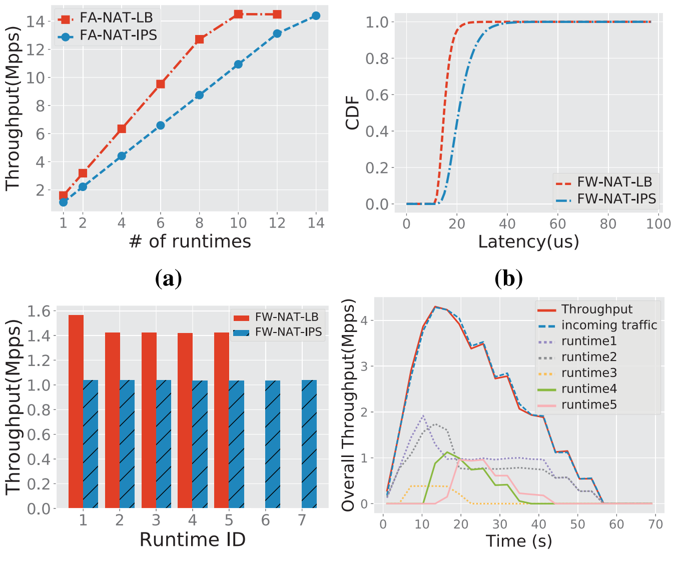
(a) (b)

(c) (d)

Fig. 7. Performance of the Runtime.

Fig. 7a and Fig. 7b 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. 7a, 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.

2) Actor Launch Time and Sending Rate of Remote Actor Messages: An important performance indicator of actor system is the actor launch time [23]. In *NFVactor*, the actor launch time is measured as the interval between the time when the first packet of a flow is received by the runtime and the time when the flow actor is created to handle the first packet. A small launch time indicates that the runtime is capable of creating a large number of actors instantly to handle the increasing workload. Fig. 7c illustrates the



(d) Dynamic scaling.

Fig. 8. System scalability.

average actor launch time achieved by *NFVactor* runtime and libcaf runtime respectively, using the same input traffic as in Section VIII-A.1. We see that the average actor launch time in *NFVactor* is much smaller than that of the libcaf runtime, as *NFVactor* pre-allocates flow actors into a ring buffer to speed-up actor launch time.

Fig. 7d 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.

B. 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. 8a 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. 8b 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.

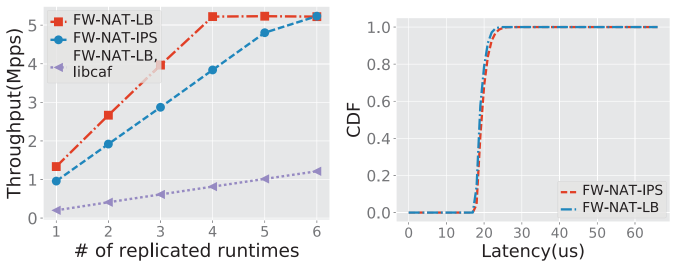
We next run the two service chains concurrently in the system. We run 5 runtimes in one server configured with 'FW → NAT → LB' and 7 runtimes in another server configured with 'FW → NAT → IDS'. The input traffic has a total packet rate of 14.50Mpps and shares the same mixed pattern to produce Fig. 8a. The input traffic is evenly split among the two service chains. Fig. 8c 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. 8d. We initialize the cluster with two runtimes (runtime 1 and 2) running 'FW → NAT → IDS' service chain. 40K flows are injected into the cluster, and each flow lasts for 60 seconds. For the first 15 seconds, the total packet rate of the 40K flows is increased from 0Mpps to 4.2Mpps. For the last 45 seconds, the total packet rate decreases 0.7Mpps for every 7 seconds, until it reaches 0Mpps. Starting from the 10th second in Fig. 8d, runtime 1 and 2 are detected as overloaded and their workload is gradually migrated away to runtime 4 and 5, respectively. With flow migration, *NFVactor* can effectively scale-out and eliminate system overload created by long-lasting flows, as the achieved throughout always matches the input traffic during the experiment.

C. Performance of Flow Replication

In this set of experiments, input flows are produced following the same mixed pattern to produce Fig. 8a. 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. 9a shows that both service chains can scale up to handle the handle the maximum replication packet rate of 5.22Mpps when six runtimes running on a server concurrently replicate their traffic to six replica runtimes on another server. We can see that the maximum replication packet rate can not reach the line rate, which is around 14.4Mpps for 64-byte packets. This is because when the replication throughput reaches 5.22Mpps, the bandwidth for transmitting replication messages reaches around 10Gbps, fully saturating the link for transmitting the replication messages. If the bandwidth of this link is increased



(a) **(b)**

Fig. 9. Performance of flow replication.

TABLE IV

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

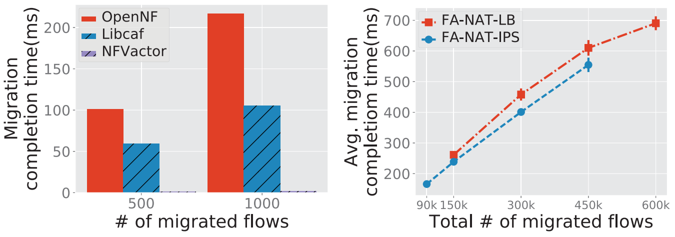
to 40Gbps or more, the maximum replication throughput achieved by *NFVactor* can be further improved. Finally, when the libcaf version of implementation is used, the replication throughput is significantly lower.

Fig. 9b 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 IV shows the average recovery time of 6 replication target runtimes. We simulate a server crash which is a common failure in datacenters by shutting down all the 6 replication source runtimes simultaneously. We can see that *NFVactor* has a much shorter recovery time than the libcaf version even when processing a larger number of flows.

1) Comparison With FTMB: Due to the unavailability of FTMB's source code, we only compare the performance of flow replication in *NFVactor* with the reported performance of FTMB paper [8]. While OpenNF [15] can also be used for failure resilience, its performance is not good enough for a direct comparison according to Section VIII-D.

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. 9b, 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. Finally, the recovery time complexity of FTMB is $O(m)$ where m is the number of packet logs replayed during recovery, while the recovery complexity of *NFVactor* is $O(n)$ where n is the number of recovered flows.



(a) **(b)**

Fig. 10. Flow migration completion time.

D. 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. 10a, 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 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. The flow migration time complexities of both OpenNF and *NFVactor* are $O(n)$ where n is the number of flows for migration.

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. 10b 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. 10b 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. 10b, we observe zero packet loss caused by our flow migration protocol.

IX. DISCUSSIONS

NFVactor holds a basic assumption about running in a cluster where all the servers are connected through a high-speed LAN network. Under this assumption, runtimes of *NFVactor*enjoy high-speed, low-latency network connections which can deliver input packets at line rate and exchanges messages within microseconds. This is the basic assumption that is shared by other high-performance NFV frameworks like E2 [2], StatelessNF [38] and FTMB [8].

However, this assumption is violated when *NFVactor* runs in a multi-network environment. Under this setting, the environment consists of multiple networks, which may be connected over a slow WAN network. When packets are transmitted from one network to another, they may experience prolonged transmission delay and limited bandwidth due to complex routing schemes of the WAN network. In this case, *NFVactor*can not achieve a similar performance as in a local cluster.

To avoid this, it is highly recommended that the network operators divide a multi-network environment into distinct clusters, guarantee high-performance LAN connection within each cluster, and deploy different instances of *NFVactor* for each cluster.

X. Conclusions and Future Work

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.

We identify two future directions to improve *NFVactor*. First, we will continue to develop the runtime system of *NFVactor*, making it faster and more robust. Second, to facilitate porting existing NFs to *NFVactor*, we plan to develop a compiler framework that can automatically transform existing NF code to *NFVactor*-compatible code. This may require parsing the structure of existing NF code, replacing important API calls with *NFVactor* APIs, and automatically implementing the APIs for releasing acquired resource.

Acknowledgment

The authors would like to thank all the reviewers for their helpful comments on improving the paper. The major part of this work was done when J. Duan was with The University of Hong Kong, Hong Kong.

References

[1] (2018). *NFV White Paper*. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[2] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. 25th Symp. Oper. Syst. Princ. (SOSP)*, 2015, pp. 121–136.

[3] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, 2016, pp. 511–524.

[4] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," presented as the 9th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2012, pp. 323–336.

[5] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible open middleboxes with commodity servers," in *Proc. ACM/IEEE Symp. Architectures Netw. Commun. Syst. (ANCS)*, Oct. 2012, pp. 49–60.

[6] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella, "Stratos: Virtual middleboxes as first-class entities," UW-Madison, Madison, WI, USA, Tech. Rep. TR1771, 2012.

[7] W. Zhang *et al.*, "OpenNetVM: A platform for high performance network service chains," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization (HotMIddlebox)*, 2016, pp. 26–31.

[8] J. Sherry *et al.*, "Rollback-recovery for middleboxes," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 227–240.

[9] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. 4th Annu. Symp. Cloud Comput. (SOCC)*, 2013, pp. 1:1–1:15.

[10] H. Ballani *et al.*, "Enabling end-host network functions," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 493–507.

[11] (2018). *Bro*. [Online]. Available: https://www.bro.org/

[12] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *Proc. 7th ACM Conf. Comput. Commun. Secur.*, 2000, pp. 190–199.

[13] (2018). *Linux Virtual Server*. [Online]. Available: www.linuxvirtualserver.org/

[14] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," presented as the 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2013, pp. 227–240.

[15] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. ACM Conf. SIGCOMM (SIGCOMM)*, 2014, pp. 163–174.

[16] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker, "A high performance packet core for next generation cellular networks," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 348–361.

[17] (2018). *Hypertext Transfer Protocol (HTTP) Keep-Alive Header*. [Online]. Available: https://tools.ietf.org/id/draft-thomson-hybi-http-timeout-01.html

[18] (2018). *FFmpeg*. [Online]. Available: https://ffmpeg.org/

[19] (2018). *File Transfer Protocol (FTP)*. [Online]. Available: https://tools.ietf.org/html/rfc959.html

[20] (2018). *Netmap*. [Online]. Available: info.iet.unipi.it/~luigi/netmap/

[21] (2018). *Squid Caching Proxy*. [Online]. Available: www.squid-cache.org/

[22] G. A. Agha, "ACTORS: A model of concurrent computation in distributed systems," MIT Artif. Intell. LAB, Cambridge, MA, USA, Tech. Rep. AITR-844, 1985.

[23] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting actor programming in C++," *Comput. Lang. Syst. Struct.*, vol. 45, pp. 105–131, Apr. 2016.

[24] (2018). *Intel Data Plane Development Kit*. [Online]. Available: http://dpdk.org/

[25] (2018). *Bess: Berkeley Extensible Software Switch*. [Online]. Available: https://github.com/NetSys/bess

[26] (2018). *The NFVActor Project*. [Online]. Available: https://github.com/duanjp8617/nfvactor

[27] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manage.*, vol. 12, no. 1, pp. 34–47, Mar. 2015.

[28] J. Martins *et al.*, "Clickos and the art of network function virtualization," in *Proc. 11th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2014, pp. 459–473.

[29] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[30] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep., 2015.

[31] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained NFs for flexible per-flow customization," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2016, pp. 3–17.

[32] J. Duan, C. Wu, F. Le, A. X. Liu, and Y. Peng, "Dynamic scaling of virtualized, distributed service chains: A case study of IMS," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2501–2511, Nov. 2017.

[33] Z. A. Qazi, P. K. Penumarthi, V. Sekar, V. Gopalakrishnan, K. Joshi, and S. R. Das, "KLEIN: A minimally disruptive design for an elastic cellular core," in *Proc. Symp. SDN Res. (SOSR)*, 2016, pp. 2:1–2:12.

[34] M. Bagaa, T. Taleb, A. Laghrissi, and A. Ksentini, "Efficient virtual evolved packet core deployment across multiple cloud domains," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2018, pp. 1–6.

[35] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han, "U-haul: Efficient state migration in nfv," in *Proc. 7th ACM SIGOPS Asia–Pacific Workshop Syst. (APSys)*, 2016, p. 2:1–2:8.

[36] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using statealyzr," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 239–253.

[37] D. L. C. Dutra, M. Bagaa, T. Taleb, and K. Samdanis, "Ensuring end-to-end QoS based on multi-paths routing using SDN technology," in *Proc. IEEE Global Commun. Conf.*, Dec. 2017, pp. 1–6.

[38] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 97–112.

[39] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proc. 23rd ACM Symp. Oper. Syst. Princ.*, 2011, pp. 29–41.

[40] T. Taleb, M. Bagaa, and A. Ksentini, "User mobility-aware virtual network function placement for virtual 5G network infrastructure," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2015, pp. 3879–3884.

[41] A. Laghrissi, T. Taleb, M. Bagaa, and H. Flinck, "Towards edge slicing: VNF placement algorithms for a dynamic & Realistic edge cloud environment," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2017, pp. 1–6.

[42] M. Bagaa, T. Taleb, and A. Ksentini, "Service-aware network function placement for efficient traffic handling in carrier cloud," in *Proc. IEEE Wireless Commun. Netw. Conf.*, Apr. 2014, pp. 2402–2407.

[43] A. Ksentini, M. Bagaa, and T. Taleb, "On using SDN in 5G: The controller placement problem," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2016, pp. 1–6.

[44] R. A. Addad, T. Taleb, M. Bagaa, D. Dutra, and H. Flinck, "Towards modeling cross-domain network slices for 5G," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2018, pp. 1–6.

[45] (2018). *Erlang*. [Online]. Available: https://www.erlang.org/

[46] (2018). *Scala Akka*. [Online]. Available: akka.io/

[47] (2018). *C++ Actor Framework*. [Online]. Available: http://actor-framework.org/

[48] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein, "Optimizing distributed actor systems for dynamic interactive services," in *Proc. 11th Eur. Conf. Comput. Syst. (EuroSys)*, 2016, pp. 38:1–38:15.

[49] S. Mohindra, D. Hook, A. Prout, A.-H. Sanh, A. Tran, and C. Yee, "Big data analysis using distributed actors framework," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2013, pp. 1–5.

[50] (2018). *Orleans*. [Online]. Available: research.microsoft.com/en-us/projects/orleans/

[51] (2018). *Docker Container*. [Online]. Available: https://www.docker.com/

[52] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.

[53] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. Conf. USENIX Annu. Tech. Conf. (ATC)*, 2010, p. 11.

[54] E. Jeong, S. Woo, M. Jamshed, and H. Jeong, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. 11th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2014, pp. 489–502.

[55] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Proc. 9th Annu. Eur. Symp. Algorithms*, 2001.

Jingpu Duan received the B.E. degree from the Huazhong University of Science and Technology in 2013 and the Ph.D. degree from The University of Hong Kong in 2018. He is currently an Assistant Researcher with the Institute of Future Networks, Southern University of Science and Technology. His research interests include designing and implementing high-performance networking systems.

Xiaodong Yi received the B.E. degree from the Department of Computer Science, Huazhong University of Science and Technology, in 2017. He is currently pursuing the Ph.D. degree with the Department of Computer Science, The University of Hong Kong. His research interests include network function virtualization, GPU, and deep learning.

Shixiong Zhao received the B.E. degree from The University of Hong Kong and the M.Sc. degree from The Hong Kong University of Science and Technology. He is currently pursuing the Ph.D. degree with the Department of Computer Science, The University of Hong Kong. His research interests include distributed systems for high-performance computing, de-centralized distributed systems, and system security.

Chuan Wu received the B.E. and M.E. degrees from Tsinghua University, China, in 2000 and 2002, respectively, and the Ph.D. degree from the University of Toronto, Canada, in 2008. She is currently an Associate Professor with the Department of Computer Science, The University of Hong Kong. Her research interests include cloud computing, network function virtualization, and distributed machine learning.

Heming Cui received the Ph.D. degree from Columbia University, New York City, NY, USA, in 2014. He is currently an Assistant Professor with the Department of Computer Science, The University of Hong Kong. His research interests include operating systems, programming languages, distributed systems, and cloud computing, with a particular focus on building software infrastructures and tools to improve reliability and security of real-world software.

Franck Le received the Ph.D. degree from Carnegie Mellon University, Pittsburgh, PA, USA, and the Diplome d'Ingenieur degree from the Ecole Nationale Superieure des Telecommunications de Bretagne, France. He is currently a Researcher with the IBM T. J. Watson Center, Yorktown Heights, NY, USA.