NFActor: A Resillient NFV System using the Distributed Actor Model

Paper #44, 14 pages

ABSTRACT

Failure resilience is of pivotal importance in practical network function virtualization (NFV) systems, but has been mostly absent in the existing ones. The absence is mainly due to the challenge of patching source code of the existing NF software for extracting important NF states, a necessary step toward flow migration and replication to provide failure tolerance. This paper proposes *NFActor*, a novel NFV system that uses the actor programming model to provide transparent resilience, high scalability and low overhead in network flow processing. In *NFActor*, a set of efficient APIs are provided for constructing NFs, with inherent support for scalability and resilience. A per-flow management principle is advocated, different from the existing practice, which provides dedicated micro service chain services for individual flows, enabling decentralized flow migration and scalable flow replication. We implement *NFActor* and show that it achieves good scalability, prompt flow migration and failure recovery with large numbers of concurrent flows. We also show that *NFActor* can enable applications such as live NF update and correct MPTCP subflow processing, which cannot be efficiently achieved in previous systems.

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 [15]. With NFV, network operators can launch virtualized devices (virtual machines or containers) to run NFs on the fly and provision network services. A network service typically consists of a sequence of NFs for flow processing, described by a *service chain*, *e.g.*, "firewall → intrusion detection system (IDS) → loader balancer".

A number of NFV management systems have been designed in recent years, *e.g.*, E2 [36], OpenBox [25], CoMb [41], xOMB [23], Stratos [26] and OpenNetVM [45]. They implement a broad range of management functionalities, including dynamic NF placement, elastic NF scaling, load balancing, etc. However, failure tolerance [39, 42] capabilities of the NF systems are always lacking, together with efficient support for flow migration [27, 40, 32].

Failure tolerance is crucial for stateful NFs. Many NFs maintain important per-flow states [24]: IDSs such as Bro [3] store and update protocol-related states for each flow to alert potential attacks; firewalls [13] parse TCP SYN/ACK/FIN packets and maintain TCP connection related states for each flow; load balancers [14] 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 [42].

Efficient flow migration is important for long-lived flows for dynamic system scaling. Existing NF systems [36, 26] 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 only efficient in cases of short-lived flows. Long flows are common in the Internet: a web browser uses one TCP connection to exchange many requests and responses with a web server [11]; video-streaming [8] and file-downloading [9] systems maintain long-lived TCP connections for fetching large volumes of data from CDN servers. When NF instances handling long flows are overloaded or under-loaded, migrating flows to other available NF instances enables timely hotspot resolution and system scaling [27].

Why are failure tolerance and efficient flow migration missing in the existing NFV systems? The reason is simple: enabling them has been a challenging task on the existing NF software architectures. To provide resilience and enable flow migration, important NF states must be correctly extracted from NF software for transmitting to a backup NF instance. However, a separation between NF states and core processing logic is not enforced in the state-of-the-art implementation of NF software. Especially, important NF states may be scattered across the code base of the software, making extracting and serializing these states a daunting task.

Patch code needs to be manually added to the source code of different NFs for this purpose [27][40], which usually requires a huge amount of manual work to add up to thousands of lines of code for one NF. For example, Gember-Jacobson *et al.* [27] report that 3.3K lines of code is needed for patching Bro [3] and 7.8K for Squid caching proxy [20]. Realizing this difficulty, Khalid *et al.* [32] use static program analysis technique to automate this process. However, applying static program analysis itself is a challenging task and the inaccuracy of static program analysis may prevent some important NF states from being correctly retrieved.

Even if NF states can be correctly acquired and NF replicas created, flows need to be redirected to the new NF instances in cases of load balancing and failure recovery. The existing systems typically handle this using a centralized SDN controller [27, 40]. The controller initiates and coordinates the entire migration process of each flow, which involves multiple messages passes to ensure losslessness, leading to compromised scalability and additional delay.

This paper presents a software framework for building resilient NFV systems, *NFActor*, exploiting the distributed actor model [1, 18, 35] for efficient flow migration, replication and system scaling. Especially, the actor model enables lightweight extraction and decentralized migration of network flow states, based on which we enable highly efficient flow migration and replication. Transparent resilience, agile scalability and high efficiency in network flow processing are achieved in *NFActor* based on the following design highlights.

▷ *Clean separation between NF processing logic and resilience enabling operations.* We design a highly efficient flow actor architecture, which provides a clean separation between important NF states and core NF processing logic in a service chain using a set of easy-to-use APIs. Extracting and transmitting flow states hence become an easy task. Based on this, *NFActor* can carry out flow migration and replication operations, those needed to enable failure resilience, transparently to concrete NF processing, which we refer to as *transparent resilience.*

▷ *Per-flow micro management.* Fundamentally different from the existing systems, *NFActor* creates a micro execution context for each flow by providing a dedicated flow actor for processing the flow through its entire service chain, *i.e.*, a micro service chain service dedicated to the flow. Multiple flow actors run within one runtime system (*e.g.*, one container), with lightweight, efficient actor scheduling to achieve high packet processing throughput. *NFActor* consists of multiple uniform runtime systems, that we have carefully designed to provide high system scalability and easy flow replication.

▷ *Largely decentralized flow migration and replication.* Based on the actor framework, flow migration and replication processes in *NFActor* are automated through decentralized message passing. A central coordinator is involved only during initialization stage of flow migration and replication. In addition, runtimes in *NFActor*use the high-speed packet I/O library DPDK [12] for retrieving/transmitting flow packets and control messages for flow migration and replication, achieving high efficiency.

We implement *NFActor* and open source the project [21]. Our testbed experiments show the good scalability of *NFActor*to approach line-rate packet processing, zero packet loss during concurrent migration of more than 100K flows, and recovery of tens of thousands of flows within tens of milliseconds in case of runtime failures. Going beyond resilience, we also show that a couple of interesting applications can be efficiently enabled on *NFActor*, live NF update and correct MPTCP subflow processing. These applications require individual NFs to initiate flow migration, which is hard to achieve (without significant overhead) in the existing systems. Our decentralized and fast flow migration enable these applications with ease.

2. BACKGROUND AND RELATED WORK

2.1 Network Function Virtualization

Since the introduction of NFV [22], a broad range of studies have been carried out, for bridging the gap between specialized hardware and network functions [30, 28, 33, 37], scaling and managing NFV systems [26, 36], flow migration [40, 32, 27], NF replication [39, 42], and traffic steering [38]. NF instances are typically created as software modules running on standard VMs or containers. *NFActor* customizes a runtime system to run NFs, in the format of one-actor-one-flow in the runtime, enabling transparent resilience.

Among the existing studies, ClickOS [33] also introduces modular design to simplify NF construction; however, advanced control functionalities, *e.g.*, flow migration, are still not easy to be integrated in the NFs following the design. Flurries [44] proposes fine-grained per-flow NF processing, by dynamically assigning a flow to a lightweight NF. Sharing some similarities, *NFActor* enables micro service chain processing of each flow in one actor, and focuses on providing transparent failure tolerance based on the actor model. OpenBox merges the processing logic of VNFs, therefore improving the modularity and processing performance of VNF. The idea of OpenBox could also be applied into *NFActor*, but *NFActor*focuses on mainstream service chain processing and leaves this to future work.

To implement flow migration, existing systems [27][40] require direct modification of the core processing logic of NF software, and mostly rely on a SDN controller to carry out migration protocol, involving non-negligible overhead. *NFActor* overcomes these issues using novelly designed NF APIs and a largely distributed framework, where flow states are easily extractable and migration is achieved directly in-between runtimes with only 3 steps of request-response.

Flow replication to provide failure tolerance usually involves check-pointing the entire process image (where the NF software is running), and replica creation using the image [42] [39]. Temporary pause of an NF process is typically needed [42], resulting in flow packet losses. *NFActor* is able to checkpoint all states of a flow in a lightweight, transparent fashion to minimize loss and delay, based on a clean separation between NF processing logic and flow state.

2.2 Actor

The actor programming model has been used for constructing massive, distributed systems [1, 18, 35, 34]. Each actor is an independent execution unit, which can be viewed as a logical thread. In the simplest form, an actor contains an internal actor state (*e.g.*, statistic counter, number of outgoing requests), a mailbox for accepting incoming messages, and several message handler functions. An actor processes incoming messages using message handlers, exchanges messages with other actors through a built-in message passing channel, and creates 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 [7, 18, 4], which schedules actors to execute and enables network transparency, *i.e.*, actors communicate with remote actors running on different runtime systems as if they were all running on the same runtime system.

The actor model is a natural fit for building distributed NFV systems. We can create one actor as one flow processing unit, and build flow packet processing and control messaging (*e.g.*, for flow migration and replication) as message handlers on the actor. Using capabilities of actors such as launching other actors, flow migration and replication can be achieved mostly by actors themselves in a distributed fashion. To the best of our knowledge, we are the first to build resilient NFV systems using the actor model, and to demonstrate its efficiency.

There are several popular actor frameworks, *e.g.*, Scala Akka [18], Erlang [7], Orleans [16] and C++ Actor Framework [4], which have been used to build a broad range of distributed applications [18]. None of these frameworks are optimized for building NFV systems. In our initial prototype implementation, we built *NFActor* on top of the C++ Actor Framework, but the message-passing efficiency turned out to be non-satisfactory. The cause mainly lies in transmitting actor messages uses kernel networking stack in the framework, with intolerable context switching overhead for NFV systems [33]. This inspires us to customize an actor framework for *NFActor* with high performance.

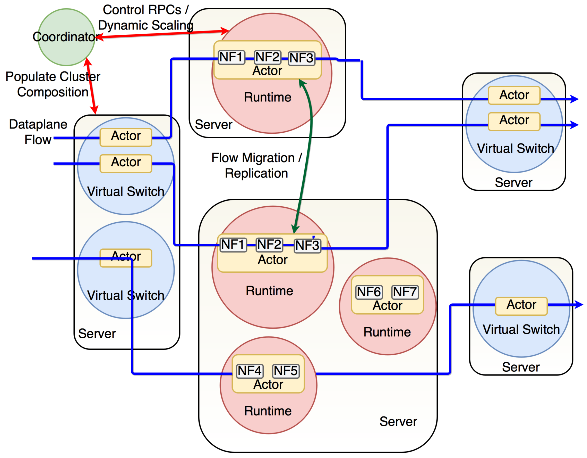


Figure 1: An overview of *NFActor*.

3. THE NFACTOR FRAMEWORK

We first present key design and modules in *NFActor*.

3.1 Overview

NFActor includes three modules: (i) runtime systems 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. An illustration of the *NFActor* system is given in Fig. 1.

A runtime system, referred to as *runtime* for short, is the execution environment of service chains. A runtime is running on a container, for quick launching and rebooting in cases of scaling out/in and failure recovery. There can be multiple runtimes (containers) running on the same physical server. In *NFActor*, the virtual switches are running in the same environments (containers) as those runtimes. Runtimes and virtual switches are connected through a L2 network.

Each virtual switch is configured with an entry IP address. The coordinator sets up flow rules to direct dataplane flows to virtual switches, which further dispatch them to runtimes hosting respective service chains. A runtime creates a dedicated flow actor for processing each flow. After processing, the outgoing flow packets are sent to another virtual switch, which forwards the flow to its final destination.[1]

The design of *NFActor* targets the following goals.

[1] When our flow replication mechanism is in place, the flow will be sent by the replica runtime to the virtual switch, to be further forwarded outside (Sec. 4.1).

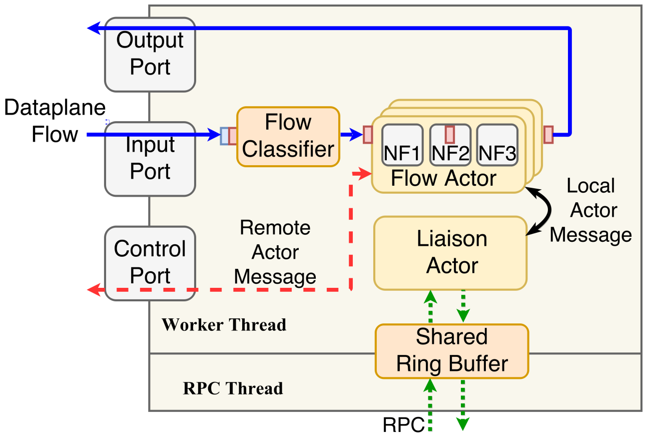


Figure 2: The internal structure of a runtime in *NFActor*.

Transparent Resilience. Flow migration and replication operations, those needed to enable failure resilience, are carried out transparently to flow processing through any service chain, in largely distributed fashion.

High Scalability. Good horizontal scalability is achieved for runtimes and virtual switches, so that *NFActor*adopts to varying workload timely with ease.

High Packet Processing Throughput. Extra overhead for flow migration, replication, and system scaling is minimized, to ensure high-speed packet processing.

3.2 Runtime

NFActor employs a carefully designed, uniform runtime system as the basic unit for deployment and scaling. Within each runtime, we adopt the simple yet powerful design to create a micro execution context for each flow, and carry out packet processing within the flow over its entire service chain inside the micro execution context. Especially, a dedicated flow actor is created for handling each flow, which loads all the NFs of the service chain that the flow should traverse, and passes the received packets of the flow to these NFs in sequence. We refer to this as the *one-actor-one-flow* design principle. Packet processing by NFs and operations to support flow migration and replication are all implemented as message handlers of the flow actor.

Fig. 2 shows the complete structure of a runtime. The input and output ports are used for receiving and sending flow packets from and to virtual switches. The control port is used for transmitting and receiving *remote actor messages*, exchanged for distributed flow migration and replication among actors running on different runtimes. Input, output and control ports in each runtime are implemented with DPDK for high-speed raw packet transmission (Sec. 5). Input packets of dataplane flows are first sent to a flow classifier, which uses the 5-tuple of a flow (*i.e.*, source IP address, destination IP address, transport-layer protocol, source port and destination port) to identify packets belonging to the same flow. All packets of the same flow are sent to the same flow actor for processing. Other than flow processing, a flow actor participates in distributed flow migration and replication in response to certain actor messages.

Each runtime is configured with one specific service chain by the coordinator, and installs/initializes all the NFs of the service chain upon booting. Multiple flow actors may run concurrently in the same runtime, which all use the same service chain. Based on the actor model, passing packets to a NF for processing in a flow actor is essentially just a function call. Hence only one copy of each NF needs to be loaded in a runtime, used by all flow actors. A worker thread schedules flow actors in a runtime: when a flow packet or a remote actor message arrives, the flow actor which is responsible for the flow that the packet belongs to, or is the destination actor of the remote message, is invoked; the packet or message is processed completely, before the scheduler moves on to handle the next packet/message, *i.e.*, a run-to-completion scheduler [37].

A runtime also consists of a RPC thread for receiving and responding to RPC requests from the coordinator, for basic system management operations (Sec. 3.5). The RPC thread forwards received RPC requests to a *liaison actor* handled by the worker thread through a high-speed shared ring buffer. The liaison actor coordinates with flow actors through *local actor messages* to carry out RPC requests from the coordinator. Having a dedicated thread for receiving RPCs saves the worker thread from potential context switches due to using kernel networking stack, expediting flow processing.

Discussions on Runtime Design Choices. Supporting only one service chain in one runtime avoids the overhead of installing many NFs per runtime and simplifies runtime management. Our one-actor-one-flow design facilitates fast flow migration, as compared to a few alternatives: (1) *One flow actor handles multiple flows.* It compromises the efficiency of flow migration, especially when multiple flows come from different virtual switch actors (Sec. 3.4): the flow actor must synchronize responses sent from different virtual switch actors after their destination runtimes are changed, to ensure loss-avoidance during migration (Sec. 4.2). (2) *One flow actor runs one NF.* Additional overhead is needed for chaining multiple flow actors to constitute a service chain, lowering packet processing speed. In addition, we use one worker thread to carry out all tasks including polling dataplane packets from input port and remote actor messages from control port, scheduling flow actors, send processed packets/messages out from output or control port, and running the liaison actor to process RPC requests. Instead of using multiple worker threads, the single thread design guarantees a sequential execu-

Table 1: APIs for NFs in *NFActor*

API	Usage
nf.allocate_new_fs()	create a new flow state object for a new flow actor
nf.deallocate_fs(fs)	deallocate the flow state object upon expiration of the flow actor
nf.process_pkt(input_pkt, fs)	process the input packet using the current flow state

tion order of flow actors, thus completely eliminating the need to protect message passing among flow actors by locks, and achieving higher efficiency.

3.3 NF APIs

A clear separation of useful NF states from core processing logic of an NF is important to enable easy extraction of flow states during the runtime, that does not interfere with packet processing of the NF. We design a set of APIs for NFs to implement for this purpose, as listed in Table 1.

When a new flow actor is created to handle a new flow, it calls $allocate\_new\_fs()$ to create a flow state object for each NF in its service chain. Upon arrival of a new packet, the flow actor passes the packet and the flow state object to NFs for processing ($process\_pkt()$). Any changes to the flow state during NF processing is immediately visible to the flow actor. When the flow terminates, the flow actor expires and $deallocate\_fs()$ is called to deallocate the flow state objects. With these APIs, the flow actor always has direct access to the latest flow states, enabling it to sends them out upon flow migration or replication.

To implement an NF in *NFActor*, core processing logic of the NF should be ported to the actor model and the three APIs should be implemented. The implementation is relatively straightforward. We have implemented a number of NFs based on the model, such as IDS, firewall and load balander (Sec. 6).

3.4 Virtual Switch

A virtual switch is a special runtime where the actors do not run a service chains but only a flow forwarding function. Following the one-actor-one-flow principle, a virtual switch creates multiple actors each to dispatch packets belonging to one flow. We refer to a flow dispatching actor in a virtual switch as a *virtual switch actor*.

A virtual switch learns runtimes that it can dispatch flows to through RPC requests its liaison actor receives from the coordinator, which include MAC addresses of the input ports of those runtimes. A virtual switch actor selects one of the runtimes to forward its flow in a round-robin fashion. We adopt a simple round-robin approach because the virtual switch must run very fast and a round-robin algorithm introduces the smallest overhead while providing satisfactory load balancing performance even in case of mixed long and short flows.

When a virtual switch actor receives an incoming packet, it replaces the destination MAC address of the packet to MAC address of input port of the chosen runtime, modifies the source MAC address of the packet to that of output port of the virtual switch, and then sends the packet out from the output port. When a virtual switch actor receives a processed packet for forwarding out, it simply sends it to the output port, which uses pre-configured SDN rules to direct the packet to its final destination.

The architectural consistency of virtual switches and runtimes facilitates efficient flow migration and replication (Sec. 4). A virtual switch in *NFActor* is lightweight, only responsible for dispatching flows to runtimes or outside, but not routing flows from one NF to anther as SDN switches in existing systems do [26, 27].

3.5 Coordinator

The coordinator in *NFActor* is responsible for basic cluster management routines, *e.g.*, launching and shutting down virtual switches and runtimes, updating latest cluster composition to the virtual switches and runtimes, monitoring workload of runtimes, and participating in the initiation phase of flow migration and replication. As compared to SDN controllers in the existing NFV systems [27, 40], the coordinator is much lightweight. The design of the coordinator is hence simplified (as a single thread module).

To deploy a service chain in *NFActor*, the system operator first specifies composition of the service chain to the coordinator, as well as several rules to match input flows to service chains. The coordinator then launches a new virtual switch and a new runtime, configures the runtime with this service chain and installs several SDN rules that forward input flows using the service chain to the virtual switch. Each runtime or virtual switch is assigned a global unique ID to ease management. In *NFActor*, a virtual switch is responsible for dispatching flows using the same service chain, to runtimes installed with this service chain. The virtual switches and runtimes handling the same service chain constitute a *cluster*. Flow migration and replication occur within a cluster. These design choices are made since they simplify system management and improve efficiency of both virtual switches and runtimes, for avoiding extra service chain selection for each flow.

The controller constantly polls load information from all the runtimes, and perform scaling out and in case of runtime overload or under-load (Sec. 4.3). The coordinator does not take care of the scaling of virtual switches. The scaling of virtual switch is offloaded to NIC hardware using Receiver Side Scaling (RSS) [17, 31], which is a static scaling method that requires the

Table 2: 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.

coordinator to configure a fixed number of virtual switches to match the number of receiver queues of the NIC.

The coordinator communicates with runtimes via a number of control RPCs exposed by each runtime, as summarized in Table 2. It uses *poll_workload*() to acquire the current load on a runtime. The coordinator updates cluster composition (including mac addresses of input/output/control ports and IDs of all runtimes and virtual switches in the cluster, and address of virtual switch(s) to forward process packets to) to all the runtimes and virtual switches using $notify\_cluster\_cfg(cfg)$. The last three RPCs are used to initiate flow migration and replication. After issuing these three calls, migration and replication are executed among runtimes without further involving the coordinator.

4. MAJOR SYSTEM MANAGEMENT OPERATIONS

4.1 Fault Tolerance

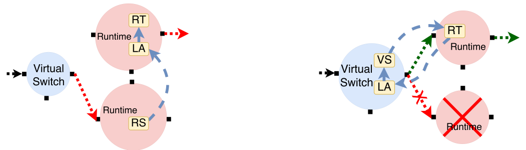
We next introduce the fault tolerance mechanisms in *NFActor*, for the coordinator, virtual switches and the runtimes, respectively.

4.1.1 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[29]. 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 reconstructs the system view by replaying logs.

4.1.2 Replicating Virtual Switches

The virtual switch can be replicated by check-pointing the memory image of the container running the virtual switch using CRIU [5], a popular tool for checkpointing/restoring Linux processes. One main technical challenge is that CRIU has to stop a process before checkpointing it. We tackle this challenge by letting the virtual switch call a $fork()$ periodically (by default, one minute), and then use CRIU to checkpoint the child process. In this way, the virtual switch can proceed without affecting system performance.



(a) Flow replication.

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

Figure 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.)

4.1.3 Replicating Runtimes

To perform lightweight runtime replication, we leverage the actor abstraction and state separation. In a runtime, important states associated with a flow is stored by the flow actor. The runtime can replicate each flow actor independently without check-pointing the entire container image [42, 39]. The biggest difference between *NFActor*'s replication strategy and the existing work (*e.g.*, [42]) is that *NFActor* replicates individual flows, not NFs, and the replication is transparent to the NFs. Each flow actor replicates itself on another runtime in the same cluster, without the need of dedicated back-up servers, achieving very good scalability. Meanwhile, this fine-grained replication provides the output-commit property [42], while achieving good throughput and fast flow recovery.

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 (desirably running on different physical servers from the server hosting this runtime) to its liaison actor via RPC $set\_replicas(runtime\_id\_list)$. The coordinator launches new runtimes if there are no available runtimes to host replicas in a cluster. When a flow actor is created on the runtime, it acquires its replication target runtime by sending a local actor message to liaison actor. The liason actor sends back the ID of the replica runtime, selected in the round-robin fashion among all available runtimes received from the coordinator.

When a flow actor has processed a flow packet, it sends a replication actor message, containing the current flow states and the packet, directly to the liaison actor on the replication target runtime. The liaison actor checks whether there exists a replica flow actor on this replication runtime using the 5-tuple of the packet. If not, it creates a new replica flow actor using the same 5-tuple and forwards all subsequent replication messages that share the same 5-tuple to that flow actor. A replica flow actor is created as a special flow actor, which processes received replication messages, stores latest flow states contained, and then directly sends the packets contained out from the output port of the replication target runtime. Though configured with the service chain of the flow, it does not invoke NFs; it will only do so after becoming the primary flow actor for processing the flow, when the runtime hosting the original flow actor has failed.

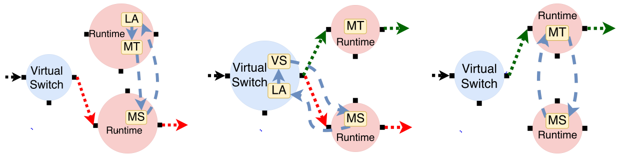
As shown in Fig. 3(a), when the runtime replication mechanism is in place, the path of dataplane flows in *NFActor* is as follows: virtual switch → runtime hosting flow actor to process the flow → runtime hosting replica flow actor → virtual switch → final flow destination. This design guarantees the same output-commit property as in [42]: the packet is sent out from the system only when all the state changes caused by the packet has been replicated.

When a runtime fails, the coordinator sends recovery RPC requests $recover(runtime\_id)$ to all the runtimes that it forwarded earlier to the failed runtime, as candidates to store flow replicas. When a runtime R receives this RPC, it checks if it indeed stores flow replicas of the failed runtime. If so, each replica flow actor on runtime R sends a request to the virtual switch responsible for forwarding the flow to the failed runtime, which further identifies the virtual switch actor in charge, and asks it to change the destination runtime to runtime R. After the acknowledge message from the virtual switch is received by the replica flow actor, packets of the flow start to arrive at the replica flow actor, and the flow is successfully restored on runtime R (Fig. 3(b)). Immediately after the replica flow actor becomes the primary one to handle the flow, it seeks another runtime to replicate itself, following the same procedure as described above.

Our primary-backup replication approach can tolerate the failure of one runtime (between runtimes that the actor and its replica are residing in). We consider this guarantee sufficient because the chance for both runtimes (on two servers) failing at the same time is very low. An alternative design is to restart the failed runtime, copy back replicated flows states, and rerun the flow actors on the recovered runtime. In comparison, our design avoids the addition delay for relaunching flow actors and minimizes interruption to consecutive flow processing. The downside is that routing output packets to the replica flow actor incurs additional bandwidth consumption - the cost for providing fast flow recovery.

4.2 Flow Migration

Based on the actor model, flow migration in *NFActor* 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.



(a) 1st req-rep. **(b)** 2nd req-rep. **(c)** 3rd req-rep.

Figure 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.

In *NFActor*, flow migration is primarily used to move flows from one runtime to another (the migration target runtime) for resolving hot spot (overloaded runtimes), or for shutting down largely idle runtimes. When a runtime is detected by the coordinator to be overloaded/underloaded, the coordinator starts migrating flows out from the runtime. It keeps calling the $set\_migration\_target$ RPC method on the runtime, asking it to migrate a number of flows to other available runtimes (Sec 4.3). After receiving ID of a migration target runtime, the flow actor starts migrating flows by itself.

Flow migration in *NFActor* only involves 3 passes of request-response messages in sequence, as illustrated in Fig. 4.

1st req-rep: 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: 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 responsible for forwarding the flow to itself. 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: The source flow actor then sends its flow state to the migration target actor. After receiving the flow states, the migration target actor saves them and immediately starts processing all the buffered packets while sending a response to the source actor. The migration source actor expires when it receives the response.

Besides being distributed, the flow migration mechanism achieves two properties.

1. Loss Avoidance: except for buffer overflow at the migration target actor or network packet reordering, no flow packets are dropped during the flow migration protocol. If it takes a long time for the request in the 3rd request-response step to arrive, the buffer at the migration target actor may overflow. In *NFActor*, a large collective buffer is used, and the distributed flow migration process is extremely fast. Buffer overflow rarely happens, even when concurrently migrating a very large number of flows (Sec. 6).

Packet reordering may also lead to packet loss: some flow packets may continue arriving at the source actor after it has sent the request to the target actor in the 3rd request-response step; those packets have been sent out by the virtual switch actor before its destination runtime is changed, but arrived later at the source actor than the response from the virtual switch. The source actor would drop packets received after it has sent the 3rd request, to avoid inconsistency. Nonetheless, packet reordering rarely happens in *NFActor* since it is deployed over a L3 network. Besides, *NFActor* minimizes the chance for packet reordering by having the virtual switch actor transmitting the response in a network packet, using the output port where it sends data packets to the source actor, rather than the control port. The response will then be received by input port on the migration source runtime, same as flow packets, ensuring that no more packets will be received by the source actor after the virtual switch's response.

2. Order Preserving: flow packets are always processed in the order that they are sent out from a virtual switch in *NFActor*.

Our loss avoidance property is slightly weaker than the loss-free property in OpenNF while the order-preserving guarantee is the same [27]. It has been a long-time understanding that providing good properties for flow migration would compromise the performance of flow migration [27]. *NFActor* breaks this curse using distributed flow migration based on actor model.

Error Handling. The three request-response steps may not always be successfully executed. In case of a request timeout, the migration source actor is responsible for recovering the destination runtime at the virtual switch actor (if its destination is changed) and resumes normal packet processing. The migration target actor created is automatically deleted after a timeout.

4.3 Dynamic Scaling

The coordinator performs dynamic scaling of the runtimes and virtual switches according to the workload variation. It fully exploits the fast and scalable distributed flow migration mechanism, to quickly resolve hot spot and immediately shut down mostly idle runtimes.

The coordinator periodically polls the load 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. The worker thread in each runtime keeps its CPU usage up to 100% all the time, due to using DPDK to busy poll the input port. Therefore, CPU usage is not a good indicator to tell whether a runtime has been overloaded. Instead, the coordinator uses the total number of dropped packets on the input port of a runtime to determine overload, which is a very effective indicator in *NFActor*: an overloaded runtime can not timely poll all the packets from its input port, therefore increasing the number of dropped packets significantly. The maximum packet processing throughput of each runtime is recorded by the coordinator, for identifying idleness in each cluster.

When the number of dropped packets on a runtime exceeds a threshold (100 as in our experiments), the runtime is identified as overloaded. If there is at least one overloaded runtime in a cluster, the coordinator launches a new runtime, configures it to run the same service chain, and keeps migrating a configurable number of flows from overloaded runtimes (500 as in our experiments) to the new runtime, until all the hotspots are resolved. If the new runtime becomes overloaded, more new 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 flow dispatching by the virtual switches.

If the current throughput of all runtimes in a cluster is smaller than half of the maximum throughput, 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 its flows have been successfully moved out.

5. IMPLEMENTATION

The core functionalities of *NFActor* are implemented by about 8500 lines of code in C++, except the code for NFs. We customize an actor library, and run our runtimes and virtual switches on Docker containers [6]. We use BESS [2] as the dataplane inter-connection tool for connecting different runtimes and virtual switches, building a virtual L2 network inside each server, and connecting each virtual L2 network to the physical L2 network connecting all the servers. The three ports in each runtime are BESS ZeroCopy VPorts - high-speed virtual ports implemented with DPDK for fetching and transmitting raw packets. With DPDK, the memory holding packet buffers is mapped directly into the address space of the runtime process, avoiding high context switching overhead if using the traditional kernel network stack and speeding up packet handling [33].

Resource Allocation to Runtimes. When launching runtimes on one server, the coordinator ensures that the worker threads of the runtimes are pinned to different CPU cores (excluding core 0 and the cores that are used by BESS), since they are busy polling threads which keep the CPU utilization to 100%. The RPC threads of all the runtimes in the same server are collective pinned to core 0, since they sleep for most of the time waiting for RPC requests.

Customized Actor Library. We implement our own actor library, including a fast actor memory allocator, a fast actor timer and a set of C++ base classes for implementing new actors. In this actor library, due to our single-worker-thread design, local actor message transmission is directly implemented as a function call, therefore eliminating the overhead of enqueing and dequeuing messages to and from an actor's mailbox [1]. For remote actor message passing, we assign a unique ID to each actor. A sender actor only needs to specify the receiver actor's ID and the hosting runtime's ID, and then the reliable transmission module (to be discussed soon) can correctly deliver the remote actor message to the receiver actor. We also implement a flow actor scheduler, which redirects both input flow packets and remote actor messages to corresponding flow actors, by looking up the flow actors using the 5-tuple. Even though functionalities provided by our customized actor library are much simpler than those in the existing actor frameworks [18] [4], its simple design leads to effective actor processing overhead reduction, enabling high-speed packet processing.

Reliable Remote Actor Message Passing. Actor messages passed among flow actors on different runtimes should be reliably delivered. We build a customized reliable message passing module, which inserts those messages into a reliable byte 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. 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).

Since our goal is to reliably transmit remote actor messages over an inter-connected L2 network, we do not use user-level TCP [31], which may impose much more processing 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. 4.2).

Worker and RPC Threads. The worker thread on a runtime polls packets from the ports, schedules flow actors and transmits remote actor messages (Sec. 3.2). To schedule these tasks efficiently, we implement them as BESS modules and use the BESS module scheduler, which schedules these modules in a round-robin fashion. Each worker thread is pinned to one CPU core to avoid overhead due to OS scheduling. The RPC thread on each runtime is implemented with GRPC [10], and the RPC requests are sent over a reliable TCP connection between a runtime and the coordinator.

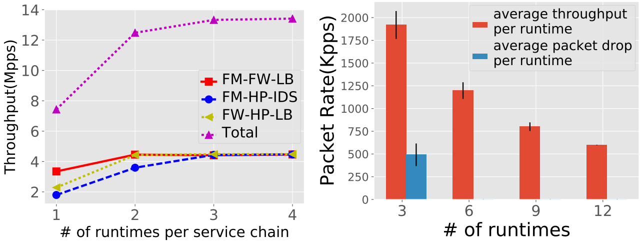
6. EXPERIMENTS

We evaluate *NFActor* using a cluster of around 10 Dell R430 servers, each containing 20 logical cores, 48GB memory and 2 Intel X710 10Gb NICs. The servers are connected through a 10GB switch. We implement a number of NFs using the APIs in Table 1: firewall (336 LOC), IDS (734 LOC), load balancer (47 LOC), flow monitor (169 LOC), HTTP parser (1471 LOC) and a NF for MPTCP subflow detection (223 LOC, which is used for experiments in Sec. 6.5). The firewall maintains several rules such as blocking a certain source IP address, and checks each received packet against the rules: if the packet violates any of the rules, a tag in the flow state is flipped and later packets are automatically dropped. The firewall also records the TCP connection status of the flow in the flow state. The IDS is a simplified version of Snort IDS [19]. The flow monitor updates an internal counter whenever it receives a packet. The HTTP parser parses received packets for HTTP requests and responses, and saves them together with the HTTP method in the flow state object.

We run at most 50000 actors per runtime according to memory allocation of the runtimes. The coordinator updates cluster composition to runtimes and virtual switches once every 5 seconds. Each collective buffer at a migration target runtime can store up to 4096 DPDK packets.A runtime is identified as overloaded if more than 100 packet drops are recorded on the input port of the runtime within one second.

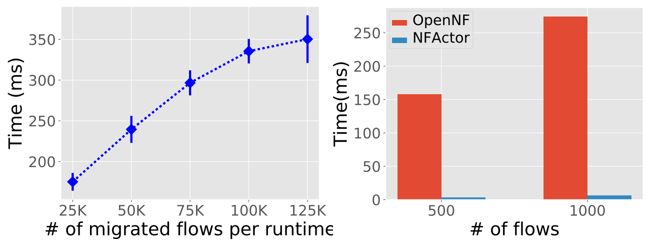
6.1 Packet Processing Throughput

We run six virtual switches on one server for dispatching flows to runtimes, run runtimes on other servers, and implement a traffic generator to produce flows to be sent



(a) (b)

Figure 5: Packet processing throughput.



(a) (b)

Figure 6: Time consumption for flow migration.

to virtual switches. The flows consist of 64 byte data packets. We deploy three service chains, 'flow monitor (FM) → firewall (FW) → load balancer (LB)', 'flow monitor (FM) → HTTP parser (HP) → IDS', and 'flow monitor (FM) → HTTP parser (HP) → load balancer (LB)'.[2] The same number of flows are processed by each of the service chains. We do not enable flow replication in this set of experiments.

We first evaluate the packet processing throughput of *NFActor* using uniform flows, each produced at 10pps (packets per second), lasting for 10 seconds. We inject an overall rate of all flows around 14$Mpps$ (60K flows) into virtual switches, which approaches line-rate of the server NICs. We evaluate the throughput when different numbers of runtimes are deployed for each service chain.

Fig. 5a shows that when there are more than 2 to 3 runtimes for each service chain, the total packet processing capacity of the runtimes is larger than the incoming flow rate (about 14$Mpps$), leading to a stablizing overall throughput about 14$Mpps$. In addition, we start to observe zero packet loss for all the three service chains when the number of runtimes is scaled up to 3.

We next evaluate packet processing throughput with flows whose duration spans 10 seconds to one minute, flow rate varies from 1kpps to 2kpps, and packet size varies among 64 bytes, 128 bytes and 256 bytes. The total number of injected flows is 550K. The runtimes are configured with a service chain 'flow monitor → HTTP parser → IDS'. Fig. 5b shows the average throughput and packet drop of each runtime, when different numbers of runtimes are deployed for each service chain. An error bar indicates the standard deviation. We can see that the flow processing load is quite balanced among different runtimes (using our simple round-robin scheduling algorithm) and the runtimes can handle all the incoming flows when the number of runtimes is scaled up to six (zero loss). The primary reason that the round-robin algorithm used by the virtual switch balances workload well enough even in case of non-uniform flows is because the total number of injected flows are large enough (550K), which is a common case for high-speed NFV system.

These results exhibit high efficiency of the runtime design and flow migration protocol in *NFActor*. The development of *NFActor* has gone through a few trials and errors. In our previous version of *NFActor* design and prototype implementation using Libcaf [4], we used 4 worker threads per runtime. Compared with the old design, our current design with one worker thread per runtime can achieve up to 4x throughput. We believe the reason is that the customized actor library and the single worker thread design completely eliminate the overhead of multi-threaded contention for shared resource, *i.e.*actor's mailbox.

6.2 Time Consumption for Flow Migration

To inspect performance of flow migration in *NFActor*, we show time taken for migrating large numbers of flows. We run three runtimes on each server, configured to run service chain "firewall → http parser → load balancer". The traffic generator produces flows of rate 20pps each, and virtual switches dispatch flows evenly to runtimes on one of the servers. After the traffic stabilizes, the coordinator asks all runtimes on that server to migrate all their flows to three runtimes on another server, concurrently.

We vary the number of flows injected to each runtime and show in Fig. 6a completion time for migrating all flows from a runtime, averaged over all three runtimes. We can see that it takes 350ms for migrating about 125000 flows (with 2.5Mpps packet processing throughput). The time only increases sublinearly with the increase of flows. Considering the migration is concurrently carried out among three pairs of runtimes, the results show good efficiency of our distributed flow migration and the following design: (i) flow states are directly copied into remote actor messages without the need for serialization and deserialization, and (ii) remote actor messages are directly encapsulated in L2 network packets and transmitted using DPDK.

We also observe zero packet loss throughout the experiment. This shows that the collective buffer of a capacity of 4096 packets (for all migration target actors

[2] *NFActor* can handle service chains with branches similarly, due to encapsulating each service chain entirely in one flow actor and the run-to-completion scheduling strategy in runtimes. We evaluate simle service chains without branches which represent the mainstream [30, 33].

to buffer received flow packets in a runtime before the request in the 3rd request-response step is received), is sufficient even when migrating a large number of flows.

We also compare *NFActor* with OpenNF [27] for flow migration. We send the same number of flows to *NFActor* runtimes and NFs controlled by OpenNF, and present the total time to migrate these flows in Fig. 6b.[3] Flow migration in *NFActor* takes much less time. Though this may not be a very fair comparison as OpenNF uses legacy NFs while *NFActor* relies on NFs implemented following our design, we believe the results are still illustrative.

6.3 Throughput during Dynamic Scaling

We further evaluate performance of dynamic scaling mechanism in *NFActor*. The traffic generator produces a varying number of flows overtime. Each flow has a rate of 20pps, lasting 60 seconds. The runtimes are configured with service chain "firewall → HTTP parser → IDS". Fig. 5a shows that the cluster is scaled up to 6 runtimes to handle the peak rate. We observe that whenever a runtime is overloaded (runtime 1 and runtime 4), our mechanism promptly resolves the hotspot (considering each flow lasts 60s).

We further observe in our experiment that the CPU usage of the coordinator remains under 5%, due to its lightweight design.

6.4 Performance of Flow Replication

We next enable flow replication in *NFActor*. The traffic generator produces 300K flows with a flow rate of 30pps. Four different service chains are used in this experiment. Three runtimes are running on each server.

Fig. 8a shows that with the increase of runtimes deployed for each service chain, the packet processing throughput from runtimes running each service chain increases, but is always lower than the throughput if no flow replication is in place. At the peak througput, the bandwidth on the L2 network has been saturated and becomes the bottleneck. This is because for each input packet, an additional packet is transmitted from the original runtime to the replication target runtime, containing the flow states and the output packet processed by the flow actor. These packets consume additional bandwidth in the system. We believe such additional bandwidth consumption is unavoidable if to ensure the output-commit property. To mitigate this issue, a flow actor can replicate its flow states after it has processed multiple packets, at the cost of weaker flow state consistency.

The biggest advantage brought by *NFActor*'s flow replication is the fast recovery time in case of failures. We emulate a server crash by shutting down the server, and recover flows processed by failed runtimes on their replicas on other servers. Fig. 9 shows that 50000 flows can be recovered within tens of milliseconds. This is because flow recovery in *NFActor* is extremely lightweight, involving only one request-response pass between the replica runtime and the virtual switch.

We do not show results evaluating failure resilience of the coordinator and the virtual switches, since their handling in cases of failures is mostly standard as in the literature.

6.5 Other Applications

In addition, we build two applications based on *NFActor*, which exploit its lightweight, distributed flow migration capability to achieve useful functionalities.

Live NF update. *NFActor* can achieve dynamic NF update (*e.g.*, software version, important NF configuration files) without interfering active flows, by dynamically migrating the flows out to a replica runtime, performing update and migrating the flows back. Fig. 9a illustrates the throughput of a runtime running a firewall NF, during dynamic update of its firewall rule. No active flows are dropped during the update with *NFActor*, while significant throughput drop occurs if shutting down the firewall for its update.

MPTCP sub-flow processing. When an MPTCP [43] flow traverses an NFV system, its sub-flows may be sent to different NF instances for processing. Some network functions require all subflows to be processed by the same instance, *e.g.*, IDS. In *NFActor*, we can add an MPTCP sub-flow detection function to each flow actor, such that when the flow actor processes the first packet of a flow, it can check whether it belongs to an MPTCP flow. If so, the flow actor performs a consistent hashing using the MPTCP header to decide a migration target runtime in the cluster, and migrates the flow to the target. In this way, different sub-flows belonging to the same MPTCP flow can be processed by the same flow actor. This is hard to be achieved in the existing NFV systems without significant central coordination.

In the experiment of Fig. 9b, we inject a total number of about 320K MPTCP flows to 3 runtimes. With subflow detection enabled, the total number of correctly processed MPTCP flows is always the same as the number of input flows, *i.e.*, those whose subflows are all processed by the same runtime. Without this detection, most of the subflows are processed by different runtimes.

7. DISCUSSIONS

We point out a few limitations of *NFActor*, and intriguing future directions.

To achieve clean separation between flow state and NF core logic, *NFActor* requires NFs rewritten using

[3] We were not able to test under large numbers of flows due to unexpected failures when running OpenNF, possibly due to our lack of experience in operating its authors' program.

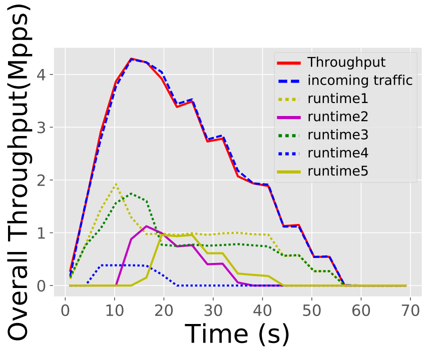
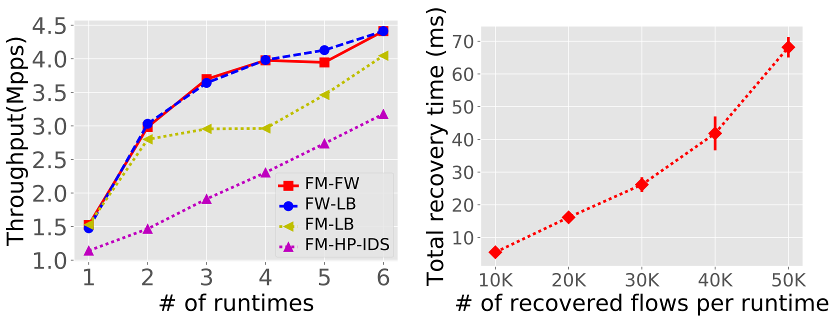


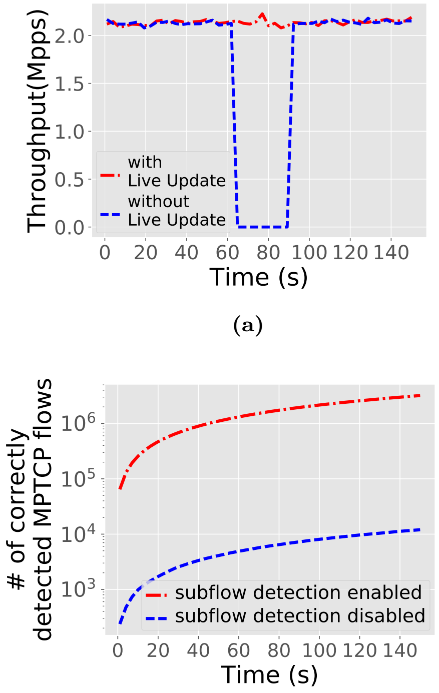
Figure 7: Throughput during dynamic scaling.



(a)

(b)

Figure 8: Performance of flow migration



(a)

(b)

Figure 9: New applications enabled with distributed flow migration capability of *NFActor*.

a set of APIs (Sec. 3.3). With the advance of NFV, more and more new NFs will created. If using the actor model for constructing NFV systems is accepted by the community, we believe it feasible to create new NFs following our design.

Due to the use of light-weight actors, *NFActor*could handle non-stateful NFs with ease. Non-stateful NFs could also benefit from the fast, distributed flow migration of *NFActor*because it eliminates potential packet re-ordering caused by directly changing the path of the flow.

NFActor focuses on handling per-flow state, consisting of states of stateful NFs along the flow path. The current *NFActor* framework does not correctly handle shared states, *i.e.*, the states shared by a bunch of flows such as [3].The reason is that our current NF API design does not achieve correct separation of shared states. Migrating and replicating flows that share states with other flows may lead to un-predictable errors. A potential solution to this issue is to implement a handler on the respective flow actor that explicitly deals with state inconsistency during flow migration and replication. We leave this to our future work.

In addition, *NFActor* may incorrectly handle flows with packet encapsulation. *NFActor* uses the 5-tuple to differentiate flows. Different flows may share the same 5-tuple if their original flow packets are encapsulated using similar headers. This is a common for flows sent over the same VxLAN tunnel. In that case, those flows are handled by the same flow actor using the same service chain, potentially incorrect. If we know what kind of encapsulation the input flows use, we may add a decapsulation function in the virtual switch to correctly extract different flows. This will also be further investigated in our future work.

8. CONCLUSION

NFActor is an NFV system built using the distributed actor model, to achieve transparent resilience, high scalability and high packet processing efficiency. It advocates a novel per-flow micro execution principle, to provide a dedicated service chain service for each individual flow, while guaranteeing low overhead and high performance simultaneously. *NFActor* includes a set of APIs for NFs to enforce clear separation of state and processing logic, as well as lightweight runtimes and virtual switches that carry out prompt flow migration, replication and scaling, largely among themselves. We have implemented the prototype *NFActor* system and open-sourced the project at [21]. Our experiments under intensive workload exhibit good scalability, fast flow migration speed, satisfactory flow replication throughput and fast runtime failure recovery speed.

9. REFERENCES

[1] Actor Modle. https://en.wikipedia.org/wiki/Actor_model.

[2] BESS: Berkeley Extensible Software Switch. https://github.com/NetSys/bess.

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

[4] C++ Actor Framework. http://actor-framework.org/.

[5] CRIU. https://criu.org/Main_Page.

[6] Docker Container. https://www.docker.com/.

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

[8] FFMPEG. https://ffmpeg.org/.

[9] FTP. https://en.wikipedia.org/wiki/File_Transfer_Protocol.

[10] GRPC. http://www.grpc.io/.

[11] HTTP Keep Alive. https://en.wikipedia.org/wiki/HTTP_persistent_connection.

[12] Intel Data Plane Development Kit. http://dpdk.org/.

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

[14] Linux Virtual Server. www.linuxvirtualserver.org/.

[15] NFV White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf.

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

[17] Receiver Side Scaling. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[18] Scala Akka. akka.io/.

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

[20] Squid Caching Proxy. www.squid-cache.org/.

[21] The NFActor Project. http://github.com/sigcomm17p44/nfactor 2017.

[22] Network Functions Virtualization - White Paper. https://portal.etsi.org/NFV/NFV_White_Paper2.pdf.

[23] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. 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.

[24] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End-host Network Functions. In *Proc. of ACM SIGCOMM*, 2015.

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

[26] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-class Entities. Technical report, UW-Madison 2012.

[27] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proc. of ACM SIGCOMM*, 2014.

[28] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical report, EECS Department, University of California, Berkeley, 2015.

[29] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. of the USENIX Annual Technical Conference (ATC '10)*, 2010.

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

[31] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. 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.

[32] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. 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.

[33] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*, 2014.

[34] S. Mohindra, D. Hook, A. Prout, A.-H. Sanh, A. Tran, and C. Yee. Big Data Analysis using Distributed Actors Framework. In *Proc. of the 2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[35] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein. Optimizing Distributed Actor Systems for Dynamic Interactive Services. In *Proc. of the Eleventh European Conference on Computer Systems (EuroSys'16)*, 2016.

[36] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a Framework for NFV Applications. In *Proc. of the 25th Symposium on Operating Systems Principles (SOSP'15)*, 2015.

[37] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking

the V out of NFV. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[38] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. of ACM SIGCOMM*, 2013.

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

[40] S. Rajagopalan, D. Williams, H. Jamjoom, and A. 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.

[41] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. 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.

[42] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-Recovery for Middleboxes. In *Proc. of SIGCOMM*, 2015.

[43] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*, 2011.

[44] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. 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.

[45] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. 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.