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 supreme scalability, prompt flow migration and failure recovery, ... **Chuan:** add more detailed results. We also show that *NFActor* can enable several interesting applications, including live NF update, flow deduplication and correct MPTCP subflow processing, which cannot be efficiently achieved using previous designs.

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 [35], OpenBox [25], CoMb [40], xOMB [23], Stratos [26], OpenNetVM [29, 43], and ClickOS [32] **Chuan:** comment out from the list the none 'management' systems. They implement a broad range of management functionalities, including dynamic NF placement, elastic NF scaling, load balancing, etc. However, failure tolerance [38, 41] capabilities of the NF systems are always lacking, together with efficient support for flow migration [27, 39, 31].

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 [41].

Efficient flow migration is important for long-lived flows for dynamic system scaling. Existing NF systems 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 **Chuan:** add citations to these systems, 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 underloaded, 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][39], 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 [19]. Realizing this difficulty, Khalid *et al.* [31] 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] **Chuan:** add more references to these systems. 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, 34] 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* on a real-world testbed and open source the project [20]. **Chuan:** improve the result discussion The result shows that the performance of the runtime system is desirable. The runtimes have almost linear scalbility. The flow migration is blazingly fast. The flow replication is scalable, achieves desirable throughput and recover fast. The dynamic scaling of NFActor framework is good with flow migration. The result of the applications are good and positive.

Going beyond resilience, we also show that several interesting applications can be efficiently enabled on *NFActor*, including live NF update, flow deduplication 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 [21], a broad range of studies have been carried out, for bridging the gap between specialized hardware and network functions [29, 28, 32, 36], scaling and managing NFV systems [26, 35], flow migration [39, 31, 27], NF replication [38, 41], and traffic steering [37]. 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 [32] 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 [42] 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. **Chuan:** discuss OpenBox [25]'s idea of merging processing logic of VNFs

To implement flow migration, existing systems [27][39] 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 [41] [38]. Temporary pause of an NF process is typically needed [41], 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, 34, 33]. 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 [32]. This inspires us to customize an actor framework for *NFActor* with high performance.

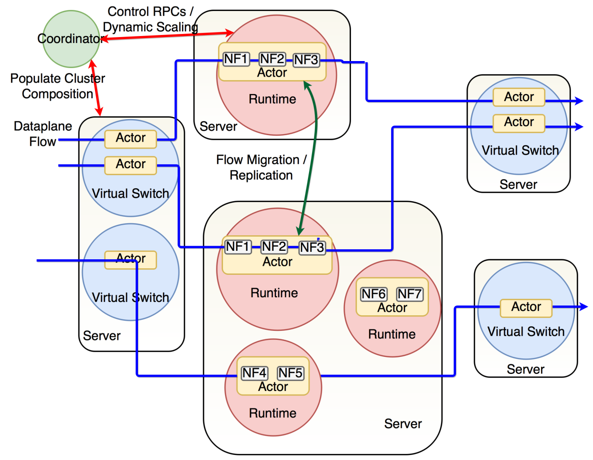


Figure 1: An overview of *NFActor*.

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, *i.e.*, they can be regarded as special 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

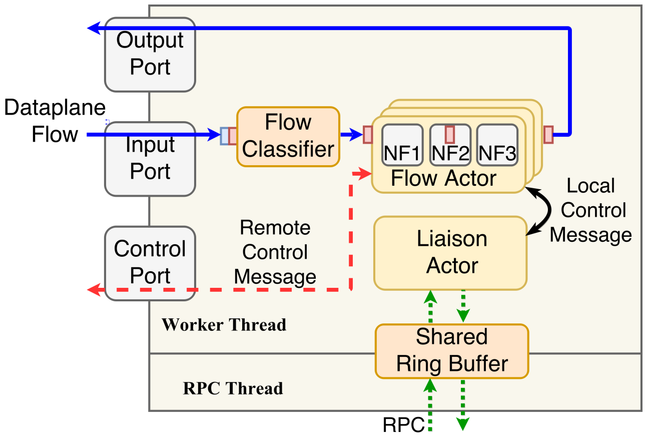


Figure 2: The internal structure of a runtime in *NFActor*.

flow to its final destination.[1]

The design of *NFActor* targets the following goals.

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 **Chuan:** cite papers mentioning this run-to-completion.

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 significantly reduces the overhead of installing many NFs and avoids service chain selection in one runtime, for higher packet processing efficiency (speed) and management simplicity. Our one-actor-one-flow design is useful for facilitating fast flow migration (Sec. 4.1), which migrates a flow by performing a flow actor migration.There are a few possible alternatives to our one-actor-one-flow design: (1) *One flow actor handles multiple flows.* It compromises the efficiency of flow migration, especially when multiple flows come from different virtual switch actors. In this case, the flow actor must synchronize the responses sent from different virtual switch actors after their destination runtimes are changed to ensure loss-avoidance migration (Sec. 4.1), adding overhead to

[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).

Table 1: APIs to be Implemented by NFs in *NFActor*.

API	Usage
nf.allocate_new_fs()	Create a new flow state object for a new flow actor, to be used for storing flow state
nf.deallocate_fs(fs)	Deallocate the flow state object when the flow actor expires
nf.process_pkt(input_pkt, fs)	Process the input packet using the current flow state

flow migration process. (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 working thread to carry out all tasks including poll dataplane packets from input port and remote actor messages from control port, run the scheduler to schedule flow actors to process these packets/messages, and send processed packets/messages out from output or control port; running the liaison actor to process RPC requests from the coordinator and dispatch messages to flow actors. Instead of using multiple worker threads in a runtime, the single-worker-thread design guarantees a sequential execution order of flow actors, thereby completely eliminating the need to protect message passing among flow actors by locks , and achieving higher efficiency.

3.3 NF APIs

To achieve transparent resilience together with the micro execution environments provided by runtimes in *NFActor*, an important step is to separate useful NF states from the core processing logic of each NF. With this separation, a flow actor can retrieve NF states for transmission whenever needed, without interfering with packet processing of the NF. In *NFActor*, we achieve this separation by designing a set of APIs that NF implementation should follow in *NFActor*.

The APIs are given in Table 1, provided as three public methods for each NF to implement. When a new flow actor is created to handle a new flow, it first calls $nf.allocate\_new\_fs()$ to create a flow state object. The actor then stores the flow state object in its internal storage. Whenever the actor receives a new packet, the actor passes the received packet and the flow state object to $nf.process\_pkt(input\_pkt, fs)$, for processing by the NFs, in sequence of the service chain. Any changes to the flow state when an NF has processed the packet is immediately visible to the flow actor. When the flow terminates, the flow actor expires and it calls $nf.deallocate\_fs(fs)$ to deallocate the flow state object. Using these three APIs, the flow actor always has direct access to the latest flow state, enabling it to transmit the flow state during flow migration and replication processes without disturbing packet processing of the NFs.

To implement an NF in *NFActor*, core processing logic of the NF needs to be implemented following the actor model and the APIs in Table 1. Nevertheless, porting the core processing logic of an existing NF software is relatively straightforward. We have implemented a broad range of NFs in *NFActor*and will present details in Sec. 6.

3.4 Virtual Switch

A virtual switch in *NFActor* is a special runtime where the actors do not run a service chain but only a load balancer function. Following the one-actor-one-flow principle, a virtual switch can create 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*.

Each virtual switch receives information of the runtimes that it can dispatch flows to from the coordinator, through RPC requests its liaison actor receives, including MAC addresses of the input ports of the runtimes. A virtual switch actor selects one of the runtimes to forward its flow in a round-robin fashion, upon creation of this actor. We choose a simple round-robin approach because the virtual switch must run very fast and a round-robin algorithm introduces the smallest amount of overhead while providing satisfactory load balancing performance. Whenever 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 MAC address of output port of the virtual switch, and then sends the packet out from the output port.

The architectural consistency of virtual switches and runtimes in *NFActor* facilitates flow migration and replication. The flow actor on a runtime can analyze the source MAC address of the incoming packets and determine which virtual switch this packet comes from. Then the flow actor can contact the virtual switch during flow migration and replication processes, to indicate change of the runtime that the respective virtual switch actor should dispatch packets to. This is done through sending a remote actor message to the virtual switch actor, which is further explained in Sec. 4.1.

The virtual switch is also used as a relay-point to the final destination of the dataplane traffic flows. By connecting the output ports of runtimes with the input port of a virtual switch and configuring proper SDN rules on the output port of the virtual switch, the virtual switch is able to forwards all the outgoing packets from the runtime to their final destination.

Existing NFV management systems either rely on SDN switches [26, 27] to route flows from one NF to the next, or build a customized data-plane for interconnecting different NF instances [35]. In comparison, our virtual switch is lightweight, only to dispatch flows to runtimes, but not route flows through individual NFs.

3.5 Coordinator

The coordinator in *NFActor* is responsible for basic cluster management (Sec. **??**), *e.g.*, updating latest cluster composition to the virtual switches and runtimes, monitoring workload of runtimes, control dynamic scaling. As compared to SDN controllers used in the existing NFV management systems [27, 39], the coordinator is much lightweight, without involving in the entire flow migration and replication process.Instead, flow migration and replication are handled in fully distributed fashion among flow actors, and only exploit the coordinator in the initialization phase.

The *NFActor*'s coordinator is responsible for launching new virtual switches and runtimes, monitoring the load on each runtime and executing dynamic scaling. Due to our distributed flow actor design, the coordinator only needs to participate in the initiation phase of flow migration and replication (Sec. 4.1). This differentiates *NFActor*'s coordinator with the controllers in existing NFV systems [27][39], which need to fully coordinate the entire flow migration process. The design of the coordinator is simplified (a single thread module) and the failure resilience of the system is improved, as the coordinator does not need to maintain complicated states associated with flow migration.

To deploy a service chain in *NFActor*, the system administrator first specifies composition of the service chain to the coordinator, as well as several rules to match the 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 the 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 are referred to as a *cluster* in *NFActor*. Flow migration and replication occur within a cluster. These design choices are made since it simplifies cluster management and improves the packet processing throughput of both virtual switches and runtimes, as they do not need to perform an extra service chain selection for each flow.

The controller constantly polls load information from all the runtimes. When the coordinator detects that a runtime is overloaded, it scales up the cluster by launching a new runtime and configures the service chain on that runtime (Sec. **??**). 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, 30], which is a static scaling method that requires the controller to configure a fixed number of virtual switches to match the number of the receiver queues of the NIC.

Table 2: Control RPCs Exposed at Each Runtime

Control RPC	Functionality
PollWorkload()	Poll the load information from a runtime.
NotifyClusterCfg(cfg)	Notify a runtime the current cluster view.
SetMigrationTarget(runtime_id, migration_number)	Initiate flow migration. It tells the runtime to migrate migration_num of flows to the runtime with runtime_id.
SetReplica(runtime_id)	Set the runtime with runtime_id as the replica.
Recover(runtime_id)	Recover all the flows replicated from runtime with runtime_id.

The coordinator communicates with runtimes via a series of control RPCs exposed by each runtime, as summarized in Table 2. It uses $PollWorkload()$ to acquire the current load on a runtime, to produce scaling decision. The coordinator maintains the composition of the system, which includes the mac addresses of input/output/control ports and the IDs of all runtimes and virtual switches of all clusters (handling different service chains). The coordinator updates the cluster composition to all the runtimes and virtual switches using $NotifyClusterCfg(cfg)$. The last three RPCs are used to initiate flow migration and replication. After issuing these three calls, migration and replication are automatically executed without further involving the coordinator. The use of RPC for coordinator to runtime communication decouples the execution of runtimes from the controller, therefore simplifying controller design.

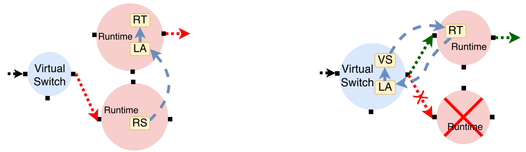
4. MAJOR SYSTEM MANAGEMENT OPERATIONS

4.1 Fault Tolerance

We next introduce the fault tolerance mechanisms in *NFActor*, for the coordinator, the virtual switches and the runtimes, respectively. Depending on the nature of these three components, we carefully design lightweight replication mechanisms, targeting robustness and little impact on performance of their normal operations.

4.1.1 Replicating Coordinator

Since the coordinator is a single-threaded module that mainly maintains composition of clusters in the system, we persistently log and replicate such information **Chuan:** where do you replicate the information to?. The liveness of the coordinator is monitored by a guard process and the coordinator is restarted immediately in case of failure. On a reboot, the coordinator reconstructs the system view by replaying logs. Each runtime or runtime monitors the connection status with the coordinator and reconnects to the coordinator in case of a connection failure **Chuan:** clarify whether the ad-



(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.)

dress of coordinator remains the same such that virtual switch and runtime knows how to reconnect.

4.1.2 *Replicating Virtual Switches*

Chuan: update the following paragraph on how virtual switch is to be replicated The most important state of the virtual switch process is its switching hash table in memory. In order to replicate the virtual switch for failure resilience, we constantly check-point the container memory image of 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, which may hurt the availability of the virtual switch. We tackle this challenge by letting the virtual switch call a fork() periodically (by default, one minute), and then we use CRIU to checkpoint the child process. Therefore, the virtual switch can proceed without affecting the system performance.

4.1.3 *Replicating Runtimes*

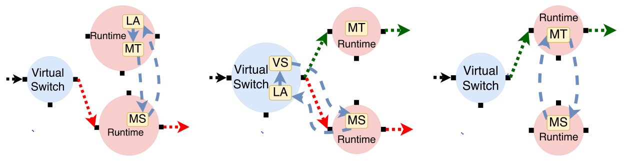
To perform lightweight runtime replication, we leverage the actor abstraction and state separation to create a lightweight flow replication strategy. In a runtime, important flow states associated with a flow is owned by a unique flow actor. The runtime can replicate each flow actor independently without check-pointing the entire container image [41, 38]. The biggest difference between *NFActor*'s replication strategy and the existing work (*e.g.*, [41]) 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 [41], achieving very good scalability. Meanwhile, this fine-grained replication strategy provides the same output-commit property as indicated in [41] with a desirable replication throughput and fast recovery time.

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 $SetReplica(runtime\_id)$ **Chuan:** revise this RPC to set multiple replica runtimes. The coordinator launches new runtimes if there are no available replication target runtimes 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 target 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 replica 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, and will only do so after it becomes 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 [41]: only one copy of the output packet is sent out from the system even when the replica is in place **Chuan:** check if this is what you meant by 'output-commit'.

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 respective 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.



(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.

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 minimize interruption to consecutive flow processing. When a replica runtime becomes overloaded, our scaling control takes over and moves out some of the flow actors (Sec. 4.3). Even if routing an output packet to the replica flow actor before sending it out incurs more inter-server communication, our experiments show that the impact is very small.

4.2 Flow Migration

We next present the lightweight, distributed flow migration mechanism in *NFActor*. Flow migration functionalities as built as message handlers of the actors, and transparent to NF processing logic. Based on the actor model, flow migration can be regarded as a transaction between a source flow actor and a target flow actor, where the source flow actor delivers its entire state and processing tasks to the target flow 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 fallback to regular packet processing and instruct to destroy the target flow actor.

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. **Chuan:** describe clearly how the decision is made based on load of runtimes, whether the coordinato or the runtime or the flow actor makes the decision, how $SetMigrationTarget(runtimeid, migrationnumber)$ is used, how migration number is decided When a runtime is to migrate a flow, it sends the ID of the migration target runtime to the flow actor handling the flow. The flow actor starts migrating the flow by itself.

Flow migration in *NFActor* only involves 3 passed 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. During the first request-response process, 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 it. 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 third request 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 from the target actor.

Besides being fully distributed, flow migration in *NFActor* guarantees 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. Before the migration target actor receives the request in the 3rd request-response step, it buffers incoming packets, which might lead to buffer overflow if it takes a long time for the request to arrive. In *NFActor*, a large collective buffer is used to buffer the packets for each migration target actor and the distributed flow migration process is extremely fast, so buffer overflow rarely happens, even when concurrently migrating a very large number of flows (Sec. 6).

Packet reordering in the network may lead to another cause for 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 in the 2nd request-response step. The source actor would drop the 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* further 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 the input port on the migration source runtime, same as the data 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 that the loss-free property in OpenNF while the order-preserving guarantee is the same as in OpenNF [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 the novel distributed flow migration design.

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

In *NFActor*, 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. In *NFActor*, each runtime has a worker thread that keeps its CPU usage up to 100% all the time (due to using DPDK to busy poll the input port Sec. **??**). 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 from each runtime during the previous overload **Chuan:** what is 'previous overload'? is recorded by the coordinator, for identifying idleness in each cluster. **Chuan:** how is the number of active flows on each runtime used in the scaling alg?

When the number of dropped packets on a runtime exceeds a threshold (Sec. 5), 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 as others in the cluster, and keeps migrating a configurable number of flows from overloaded runtimes (500 as in our implementation in Sec. 5) to the new runtime, until all the hotspots are resolved. If the new runtime becomes overloaded during flow migration, 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 implemented at the virtual switches.

If the current throughput of all runtimes in a cluster is smaller than half of the maximum throughput, the coordinator scales-in the cluster: 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 migrated out. **Chuan:** what does the following mean in the alg. that I commented out due to lack of space: while active flows on selected runtime is larger than 0, migrate 500 flows to other runtimes in a round-robin way.

Chuan: describe how virtual switch scaling is done Scaling of virtual switches is done in a similar fashion...

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 xxx**Chuan:** complete, avoiding high context switching overhead if using the traditional kernel network stack and speeding up packet handling [32].

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 thread 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. To minimize the overhead of actor programming, we choose to implement our own actor library, including xxx **Chuan:** summarize what this library includes. 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 enqueuing and dequeuing messages 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 **??** 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 of the flow. Even though the functionalities implemented by our customized actor library is very simple compared with the existing actor frameworks [18] [4], its simple design can effectively reduce actor processing overhead, enabling high-speed packet processing.

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 these tasks 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 and improve packet processing performance. 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.

Reliable Remote Actor Message Passing. Actor messages passed among flow actors on different runtimes during flow migration/replication should be reliably delivered. We build a customized reliable message passing module for *NFActor*, 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. A message may be split into several packets. These packets are configured with a sequential number each, and appended with a special header to differentiate them from dataplane packets. When the fourth task of the worker thread is scheduled to run (Sec. 3.2), these packets are dequeued from the ring buffers, and sent to their respective remote runtimes. A remote runtime acknowledges receipt of these packets. Retransmission is fired in case that the acknowledgement for a packet is not received after a configurable timeout (10 times of the RTT).

We do not use user-level TCP [**?**] to implement reliable actor message passing, since our goal is to reliably transmit remote actor messages over an inter-connected L2 network, and using user-level TCP connections may imposes much more processing overhead for reconstructing byte streams into messages. What's more, the packet-based reliable message passing provides additional benefits during flow migration and replication. Because the second response in 2nd request-response step of flow migration is sent as a packet using the same path as the dataplane flow packets, reliable actor message passing enables us to implement loss-avoidance migration with ease (Sec. 4.1). During flow replication, we can directly send the output packet as a message to the replica actor, without the need of additional packet copy **Chuan:** I do not get what additional packet copy might be needed.

6. EVALUATION

We evaluate *NFActor*framework using three Dell R430 Linux servers, containing 20 logical cores, 48GB memory and 2 Intel X710 10Gb NIC. The Dell servers are connected through a 10GB switch. We use one server to run traffic generators and six virtual switches, which are capable of generating 64 byte data packets at almost 14Mpps, achieve line-rate throughput. We use the reset of the two servers to run runtimes.

To evaluate the performance of *NFActor*, we implement 5 customized NF modules using the API provided by *NFActor* framework, which are firewall, flow monitor, load balancer, HTTP parser and an IDS. Among these 5 NFs, the first three carry out light-weight tasks where as the later two carry out heavy-weight tasks.

The rest of the section tries to answer the following questions. *First,* what is packet processing throughput of *NFActor*, does it scales well? *Second,* how good is the flow migration performance of *NFActor*when compared with existing works like OpenNF? *Third,* how is the performance of flow replication? *Fourth,* how good is *NFActor*'s dynamic scaling algorithm performs. The section concludes with the performance of the unique applications that are enabled through *NFActor*'s distributed flow migration.

6.1 Packet Processing Throughput

To evaluate the packet processing throughput of *NFActor*, we configure *NFActor*cluster with three service chains and use the same number of runtimes to process packets for each service chain. The traffic generators are configured to generate flows with 10pps flow rate and 10 seconds duration. The flow packet is uniformly configured to be 64 byte. For each service chain, we run two traffic generators to generate traffic into the service chain, resulting in a maximum throughput at around 4.4Mpps. We scale up runtimes in each service chain concurrently to test the scalability of the *NFActor*when running different service chains.

Figure 8 illustrates the performance under this set-

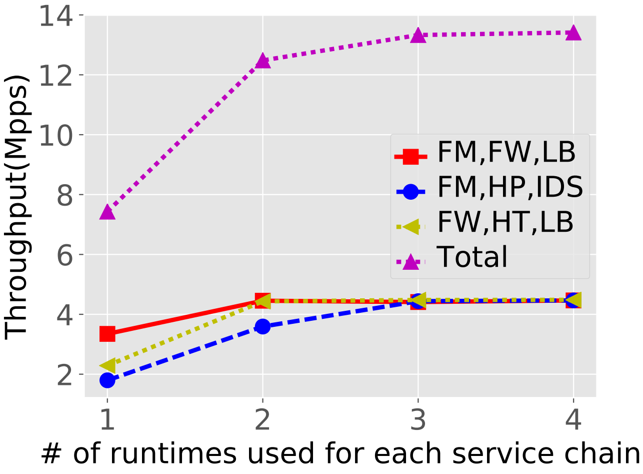
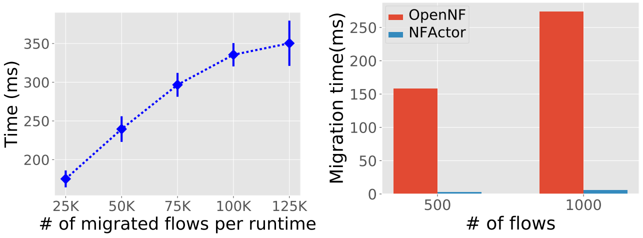


Figure 5: The packet processing throughput using different number of runtimes and different service chains. (**FM**: Flow Monitor. **HP**: HTTP Parser. **FW**: Firewall. **LB**: Load Balancer. **IDS**: Intrusion Detection System.)



(a)

(b)

Figure 6: (a) The total time to migrate different numbers of flows concurrently on three runtimes. (b) The flow migration performance of NFActor. Each flow in NFActor runtime goes through the same service chain as in Figure 6a. OpenNF controlls PRADS asset monitors. The flow packet rate is 20pps.

tings. We can see that for each service chain, the runtime scales good and handles all the input traffic. The total throughput of the *NFActor*when handling three service chains easily scale up to around 14Mpps, which basically saturate a 10Gb NIC card, therefore achieving line-rate throughput. Therefore *NFActor*can scale well and satisfies the line-rate processing requirement of modern NFV system.

The good packet processing performance achieved by *NFActor*is primarily due to our efficient actor design. The performance of a single runtime using our own actor framework out-performs our initial prototype, implemented using Libcaf [4] as the underlying library, by up to 4x even when the original runtime is configured with 4 worker threads. While the runtime using our own actor framework only uses a single core.

6.2 Flow Migration Performance

To evaluate *NFActor*'s distributed flow migration performance, we configure the three runtimes on each server. Each runtime is configured with the “firewall, http parser, load balancer” service chain. The traffic generators generate flows at 20pps and the virtual switch are only configured to generate output flow packets to runtimes on the first server, therefore each runtime on the first server processes approximately the same number of flows. After the traffic stabilizes, the coordinator concurrently initiates three migrations, asking each runtime on the first runtime to migrate all of its flows to the pair runtime on the second server.

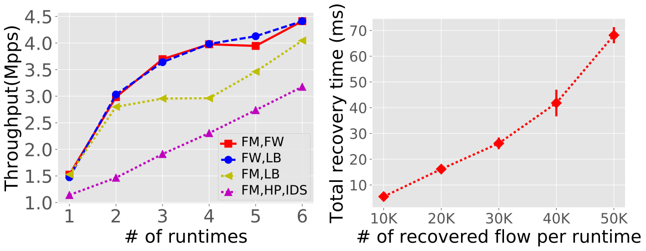
Figure 6a demonstrates average flow migration completion time calculated for all three pairs of runtimes. We can see that *NFActor* achieves excellent flow migration performance, even when migrating more than 300000 flows, with 6Mpps processing throughput. Also, the migration are concurrently executed among three pairs of runtimes, further proving the effectiveness of *NFActor*'s distributed flow migration. The key reason that *NFActor*is able to achieve such a good flow migration performance is because (i) flow states are directly copied to remote actor messages without the need for serialization and deserialization and (ii) remote actor messages are directly encapsulated in L2 network packet and transmitted with high-performance packet I/O.

We also keep the number of lost packet due to migration target buffer overflow or packet reordering. Throughout the entire test, this number is zero. The primary reason comes from the use of a large collective buffer, with the size to hold 4096 packet, for migration target actors to buffer the received packets before the third request is received, which basically eliminate migration target buffer overflow even when migrating a large number of flows. This verifies the loess-avoidance property of *NFActor*'s flow migration.

Finally, we compare the flow migration performance of *NFActor* against OpenNF [27]. We generate the same number of flows to both *NFActor* runtimes and NFs controlled by OpenNF and calculate the total time to migrate these flows. The evaluation result is shown in figure 6b. Under both settings, the performance of *NFActor* is much better than OpenNF. Even though this is not a fair comparison, as OpenNF uses legacy NFs while *NFActor*relies on newly implemented NFs, one can still get to know the good performance achieved by *NFActor*.

6.3 Replication Performance

We evaluate *NFActor*'s flow replication both in terms of overall packet processing throughput achieved during flow replication and the recovery time when a runtime fails. Therefore we set up several runtimes on a server, configure each of these runtimes with a dedicated replica runtime on anther server. Then we use traffic generators to generate traffic to the runtimes being replicated, and calculate the accumulated packet throughput from the replica. We run this test under different service



(a) The packet throughput of a *NFActor* cluster when replication is enabled.

(b) The recovery time of three failed runtimes under different settings. The tuple on the x axis represents the number of the runtime used in the evaluation and the total input packet rate.

Figure 7: The flow migration performance of *NFActor*

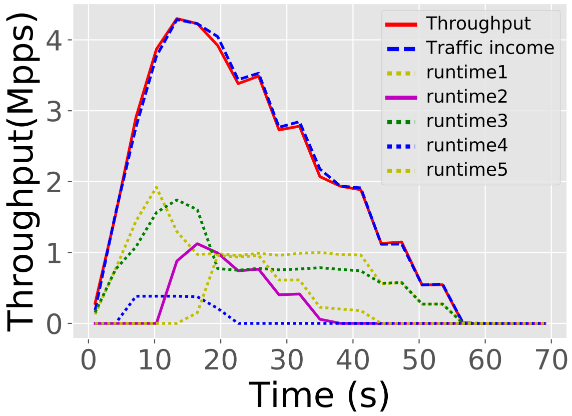


Figure 8: The performance of dynamic scaling in *NFActor*.

chain configurations. The result is shown in Figure 7a. We can see that for the service chains that consists of light-weight NFs (flow monitor, firewall and load-balancer), the overall throughput is gradually increased to around 4Mpps. At this time, the bandwidth on the L2 network has been saturated and becomes the bottleneck. This is because ensure output-commit property, for each input packet, the runtime generates at least two output packets, containing the flow state and the output packet processed by the flow actors. Being transmitted through reliably to the remote side, these packets are also appended with a header (the average size is 25bytes for each packet), further increase the bandwidth usage. We argue that our flow replication scheme remains its applicability because it achieves the output-commit property. If the requirement on consistency is weaker, *NFActor*could be easily adapted to use a light-weight replication scheme, that replicates the the flow state after it has processed multiple packets, to increase the throughput.

The biggest advantage brought by *NFActor*'s flow replication is its fast recovery time. Following the settings as in Figure 7a, we simulate a server crash by shutting down three runtimes running on one, and recovering the flows on their replicas running on the other server. Figure 9 shows that *NFActor*is able to quickly recover within tens of milliseconds even when recovering 50000 flows. This is because the recovery process of *NFActor*is extremely light-weight that only involves passing one request-response between the replica runtime and the virtual switch for each replicated flow actor.

6.4 Dynamic Scaling

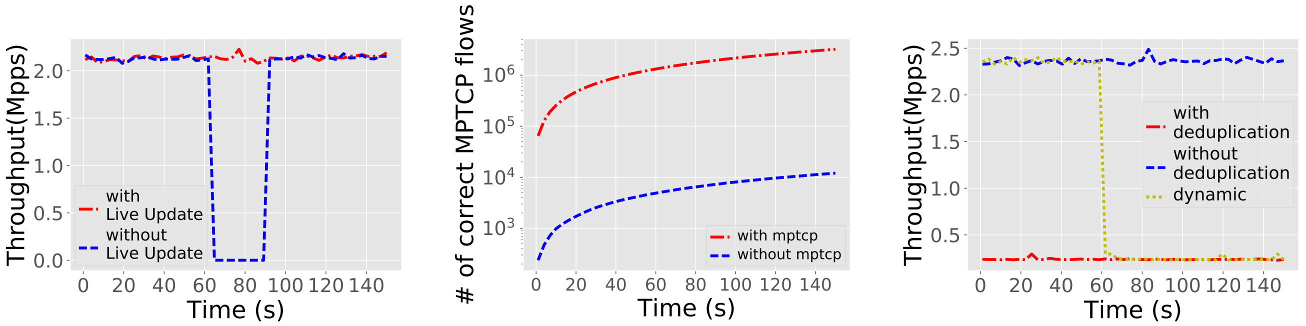
To evaluate the the performance of dynamic scaling in *NFActor*. We set up the traffic generator to inject a varying workload into the *NFActor*cluster. The runtime is configured with "firewall, HTTP parser, IDS" service chain. To evaluate the effectiveness of our flow migration method, we set up the traffic generator to generate long-lived flows, lasting 60s with 20pps. The coordinator uses the dynamic scaling algorithm to scale the cluster. As we can see from Figure 8, the cluster is scaled up to 5 runtimes. The flow migration is very effective in decreasing the the workload from an overload runtime (runtime 1 and runtime 3) to a normal status. Considering the flows lasts for 60s, without the efficient and distributed flow migration, *NFActor*would not be able to achieve this, but continue to be overloaded for a long period of time, resulting in severe performance drop. Even though not shown in the figure, due to the light-weight design, the CPU usage of the coordinator remains under 5%.

6.5 New Applications

We also build several applications based on *NFActor*, which exploits 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 running on a runtime by dynamically migrating the flows out to a replica runtime, performing update and migrating the flows back. Figure 9a illustrates the packet processing throughput of a runtime running firewall NF during the dynamic update to its firewall rule. Dynamic update ensures that no active flows are dropped during the update. On contrary, traditional way to carry out update has to shutdown the NF, causing a significant throughput drop.

MPTCP sub-flow processing. When a MPTCP [] 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.*, the IDS requires that all subflows of the same MPTCP flow to be examined by the same IDS



(a) The packet processing throughput when dynamically update the firewall rule of a runtime configured with firewall NF.

(b) Total number of correctly processed MPTCP flows by three runtimes when MPTCP detection is enabled on runtimes or disabled on runtimes.

(c) The output throughput of three runtimes when deduplication is disabled, enabled or enabled half way.

Figure 9: The new applications enabled with the distributed flow migration capability of *NFActor*.

instance to detect potential netowrk attacks.In *NFActor*, we can add a 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 a 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 existing NFV systems without significant central coordination.

Figure 9b illustrates the total number of correctly processed MPTCP flows when MPTCP sub-flow detection function is enabled or disabled. A MPTCP flow is correctly processed only if all of its subflows are processed consistently by the same runtime. We can see that *NFActor*'s subflow detection ensures that all the MPTCP flows are correctly processed. When this function is disabled, most of the subflows are processed by different runtimes, therefore only a small amount of MPTCP flows are correctly processed.

Efficient Flow Deduplication. Flow deduplication, as an important functionality of WAN optimizers [22], is useful for reducing bandwidth consumption due to transfer redundant data across different flows. *NFActor* can efficiently support flow deduplication: when a runtime receives a flow, it checks for duplicated content contained in the flow packet by performing a consistent hash over the entire packet content. In case it finds out the duplication, the flow actor initiates a migration to move the flow to a specific runtime, as indicated by the hash value of the packet content. In this way, duplicated flows can be processed on the same runtime, eliminating the output bandwidth on each runtime. When the duplicated flows arrive on the same runtime, before the packets are sent out from the runtime, we use the liason actor to intercept duplicated packet, and only generate an output data packet for every 10 duplicated packet it receives. The output data packet is packed with useful information (*i.e.*packet header of dupliated packet plus an identifier for identifying the duplicated content), so that the receiver on the other end could reconstruct all the original duplicated packets. Figure 9c illustrates deduplication performance. We can see that deduplication decreases the number of output packets by around 90% through effective deduplication.

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 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.

Chuan: briefly discuss what if nfactor is used to handle non-stateful NFs

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 xxx**Chuan:** give an example. 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 [20]. Our experiments under intensive workload exhibit xxx **Chuan:** add experiment result

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

[20] The NFActor Project. http:// 2017.

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

[22] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large cams for high performance data-intensive networked systems. In *Proc. of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, 2010.

[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] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.