NFActor: A Resillient NFV System using the Distributed Actor Framework

Paper #44, xx pages

ABSTRACT

With the advent of Network Function Virtualization (NFV) paradigm, a few NFV management systems have been proposed, enabling NF service chaining, scaling, placement, load balancing, etc. Unfortunately, although failure resilience is of pivotal importance in practical NFV systems, it is mostly absent in existing systems. We identify 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. This paper proposes *NFActor*, a novel NFV system that uses the actor programming model to provide transparent resilience, easy scalability and high performance 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 systems - which provides dedicated service chain services for individual flows, enabling decentralized flow migration and scalable replication for each flow. Going beyond resilience, *NFActor* also enables several interesting applications, including live NF update, flow deduplication and reliable MPTCP subflow processing, which are not available in existing NFV systems due to the lack of decentralized flow migration. We implement *NFActor* on a real-world testbed and show that it achieves supreme scalability, prompt flow migration and failure recovery, ... **Chuan:** add more detailed results

1. INTRODUCTION

The recent paradigm of Network Function Virtualization (NFV) advocates moving Network Functions (NFs) out of dedicated hardware middleboxes and running them as virtualized applications on commodity servers [13]. With NFV, network operators no longer need to maintain complicated and costly hardware middleboxes. Instead, they may launch virtualized devices (virtual machines or containers) to run NFs on the fly, which drastically reduces the cost and complexity of deploying network services, usually consisting of a sequence of NFs such as "firewall→IDS→proxy", *i.e.*, a service chain.

A number of NFV management systems have been designed in recent years, *e.g.*, E2 [32], OpenBox [22], CoMb [37], xOMB [19], Stratos [23], OpenNetVM [27, 40], ClickOS [29]. They implement a broad range of NF management functionalities, including dynamic NF placement, elastic NF scaling, load balancing, etc., which facilitate network operators in operating NF service chains in virtualized environments. However, none of the existing systems enable failure tolerance [35, 38] and flow migration [24, 36, 28] capabilities simultaneously, both of which are of pivotal importance in practical NFV systems for resilience and scalability.

Failure resilience is crucial for stateful NFs. Many NFs maintain important per-flow states [20]. Intrusion detection systems such as Bro [3] parse different network/application protocols, and store and update protocol-related states for each flow to alert potential attacks. Firewalls [11] maintain TCP connection-related states by parsing TCP SYN/ACK/FIN packets for each flow. Some load-balancers [12] use a map between flow identifiers and the server address to modify the destination address in each flow packet. It is critical to ensure correct recovery of flow states in case of NF instance failures, such that the connections handled by the failed NF instances do not have to be reset. In practice, middlebox vendors strongly rejected the idea of simply resetting all active connections after failure as it disrupts users [38].

Flow migration is important for long-lived flows in various scaling cases. Existing NF management systems mostly assume dispatching new flows to newly created NF instances when existing instances are overloaded, or waiting for remaining flows to finish before shutting down a mostly idle instance, which is in fact only feasible in cases of short-lived flows. In real-world Internet systems, long-lived flows are common. Web applications usually multiplex application-level requests and responses in one TCP connection to improve performance. For example, a web browser uses one TCP connection to exchange many requests and responses with a web server [9]; video-streaming [6] and file-downloading [7] systems maintain long-lived TCP connection for fetching a large amount of data from CDN servers. When

NF instances handling long flows are overloaded, some flows need to be migrated to new NF instances, in order to mitigate overload of the existing ones in a timely manner [24]; when some NF instances are handling a few dangling long flows each, it is also more resource/-cost effective to migrate the flows to one NF instance while shutting the others down.

Given the importance of failure resilience and flow migration in an NFV system, why are they absent in the existing NF management systems? The reason is simple: implementing flow migration and fault tolerance has been a challenging task on the existing NFV software architectures. To provide resilience, important NF states must be correctly extracted from the NF software for transmitting to a new NF instance, needed both for flow migration and replication (for resilience). 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 NF states a daunting task. Patch codes need to be manually added to the source code of different NFs to extract and serialize NF states [24][36]. This usually requires a huge amount of manual work to add up to thousands of lines of source code for one NF, *e.g.*, Gember-Jacobson *et al.* [24] report that it needs to add 3.3K LOC for Bro [3] and 7.8K LOC for Squid caching proxy [16]. Realizing this difficulty, Khalid *et al.* [28] 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 NF load balancing and failure recovery. In the existing systems, this is usually handled by a centralized SDN controller, which initiates and coordinates the entire migration process for each flow. Aside from compromised scalability due to the centralized control, for lossless flow migration, the controller has to perform complicated migration protocols that involve multiple passes of messages among the SDN controller, switches, migration source and migration target [24], which adds delay to flow processing and limits packet processing throughput of the system.

In this paper, we propose a software framework for building resilient NFV systems, *NFActor*, exploiting the actor framework for programming distributed services [1, 15, 31]. Our main observation is that actor provides the unique benefits for light-weight, decentralized migration of network flow states, based on which we enable highly efficient flow migration and replication. *NFActor* tracks each flow's state with our high-performance flow actor, whose design transparently separates flow state from NF processing logic. *NFActor* provides service chain processing of flows using flow actors on carefully designed uniform runtime environment, and enables fast flow migration and replication without relying much on centralized control. *NFActor* achieves transparent resilience, easy scalability and high performance in network flow processing based on the following design highlights:

▷ *Clean separation between NF processing logic and resilience support.* Unlike existing work [24, 38] that patch functionalities for failure resilience into NF software, *NFActor* provides a clean separation between important NF states and core NF processing logic in each NF using a unique API, which makes extracting, serializing and transmitting important flow states an easy task. Based on this, the *NFActor* framework can transparently carry out flow migration and replication operations, those needed to enable failure resilience, regardless of the concrete network function to be replicated, *i.e.*, which we refer to as *transparent resilience.* Using *NFActor*, programmers implementing the NFs only need to focus on the core NF logic, and the framework provides the resilience support.

▷ *Per-flow micro-management.* Fundamentally different from the existing systems, *NFActor* creates a micro execution context for each flow by providing a dedicated service chain on one actor for processing packets of this flow on the actor. This can be viewed as a micro (service chain) service dedicated to the flow. The micro execution context is constructed using actor framework, which has been proven to be a light-weight and scalable abstraction for building high-performance systems [31]. Scheduling actors to execute only incurs a small overhead, enabling *NFActor*to have a high packet processing throughput. The horizontal scalability of *NFActor*is also improved as actors can be scheduled to run on uniform runtime systems.

▷ *Largely decentralized implementation.* Based on decentralized message passing of the actor framework, flow migration and replication in *NFActor* are fully automated, achieved in a fully distributed fashion without continuous monitoring of a centralized controller, which distinguishes *NFActor* from the existing NFV systems [24]. The controller in *NFActor* is only used for controlling dynamic scaling and initiating flow migration and replication, thus light-weighted and failure resilient as the controller does not need to maintain complicated state generated by flow migration and can be easily replicated by storing its simple state on a reliable storage system like ZooKeeper [26]. In addition, *NFActor* is implemented on top of the high speed packet I/O library, DPDK [10], which further improves the performance of *NFActor*.

Going beyond resilience, our NFActor framework also enables several interesting applications that the existing NFV systems are difficult to support, including live NF update, flow deduplication and reliable MPTCP subflow processing. These applications require individual NFs to initiate flow migration, which is hard to achieve (without significant overhead) in existing systems where flow migration is initiated and fully monitored by a centralized controller. In *NFActor*, these applications can utilize our decentralized and fast flow migration to achieve live NF update with almost no interruption to high-speed packet processing of the NF, best flow deduplication to conserve bandwidth, and correct MPTCP subflow processing, with ease.

We implement *NFActor* on a real-world testbed and opensource the project code [17] **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.

The rest of the paper is organized as follows. **Chuan:** to complete

2. BACKGROUND AND RELATED WORK

2.1 Network Function Virtualization

NFV was introduced by a 2012 white paper [18] by telecommunication operators that propose running virtualized network functions on commodity hardware. Since then, a broad range of NFV studies has been seen in the literature, including bridging the gap between specialized hardware and network functions [27, 25, 29, 33], scaling and managing NFV systems [23, 32], flow migration among different NF instances [36, 28, 24], NF replication [35, 38], and traffic steering [34]. In these systems, the NF instances are created as software modules running on standard VMs or containers. *NFActor* customizes a uniform runtime platform to run network functions, which enables transparent resilience support for all network functions/service chains in the runtimes. In addition, a dedicated service chain instance is provisioned for each flow, enabled by the actor framework, achieving failure tolerance and high packet processing throughput with ease. Even though modular design introduced by ClickOS [29] simplifies the way how NFs are constructed, advanced control functionalities, *e.g.*, that to enable flow migration, are still not easy to be integrated in NFs following the design.

A number of NFV systems [32, 22, 37, 19, 27, 40, 29, 39] have been proposed to manage NF service chains or graphs in an effective and high-performance way. Among these work, Flurries [39] proposes to do fine-grained per-flow NF processing, and is able to dynamically assign a flow to a light-weight NF. While sharing some similarities, *NFActor* focuses on applying actor model to perform micro service chain processing for each flow, and uses actor model to provide transparent resilience. It is possible to expand the service chain processing in *NFActor* to service graph processing as in E2 [32] and OpenBox [22] because *NFActor* uses a run-to-completion scheduling strategy to process flow packets. But *NFActor* sticks to the service chain processing as it still represents the mainstream processing method [27, 29].

To achieve flow migration, existing work such as OpenNF [24] require direct modification of the core processing logic of NF software, which is tedious and difficult to achieve. In addition, existing NFV management [36] [24] systems mostly rely on a centralized SDN controllers to carry out the flow migration protocol, involving non-negligible message passing overhead that lowers packet processing speed of the system. *NFActor* overcomes these issues using a clean separation between NF processing logic and resilience support functionalities, as well as a system design based on the distributed actor framework. The actors can be migrated by communicating among themselves without the coordination from a centralized controller. A fast virtual switch is designed to achieve the functionality of a dedicated SDN switch. Only 3 rounds of request-response are needed for achieving flow migration, based on the actor framework and the customized virtual switch, enabling fast flow migration and high packet processing throughput.

Flow replication usually involves check-pointing the entire process image runing the NF software and creating a replica for the created process image [38] [35]. The checkpointing method, such as the one used by [38], may require temporary pause of an NF process, leading to flow packet losses. *NFActor* is able to checkpoint all states of a flow in a lightweight fashion without introducing large delay, due to that the clean separation between NF processing logic and NF flow state enables the actor to directly store all the flow states of the service chain and transmit the flow states at any time without interfering the normal execution of the NF, enabling transparent replication of NFs and service chains. Existing work [38] rely on automated tools to extract important state variables for replicating, which relies on static program analysis technique and may not accurately extract all the important state variables if the NF program is complicated and uses a new architecture.

2.2 Actor

The actor programming model has been used for constructing massive, distributed systems [1, 15, 31, 30]. 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 the out-going request), a mailbox for accepting incoming messages and several message handler functions. An actor can process incoming messages using its message handlers, send messages to other actors through the built-in message passing channel, and create new actors. Actors are well suited to implement state machine, that modify its internal state based on the received message, therefore facilitates distributed protocol implementation. In an actor system, actors are asynchronous entities that can receive and send messages as if they are running in their own threads, simplifying programmability of actors and eliminating potential race conditions which may cause the program to crash. The actors usually run on a powerful runtime system [5, 15, 4], which is a uniform platform to schedule actors to execute , enabling them to achieve network transparency, as actors could transparently communicate with remote actors running on different runtimes as they are all running on the same runtime. An actor could launch a remote actor and communicates with the remote actor to migrate/replicate all of its internal state. The remote actor could directly substitute the identity of the original actor whenever necessary. Therefore actor model provides a natural and unified way for migrating/replicating actors.

The actor model is a natural fit when building distributed NFV systems. We can create one actor as one flow processing unit (a NF or a service chain, while the later is our design choice in *NFActor*), and map flow packet processing to actor message processing. Meanwhile, flow migration and replication functions can be implemented as message handlers on the actors. Even though there exists this natural connection between the actor model and NF flow processing functions, we are not aware of any existing work that leverages the actor model to build an NFV system. To the best of our knowledge, we are the first to exploit the actor model in enabling resilient NFV systems and relevant applications, as well as to demonstrate the benefits of this actor-based approach.

There are several popular actor frameworks, *e.g.*, Scala Akka [15], Erlang [5], Orleans [14] and C++ Actor Framework [4]. These frameworks have been used to build a broad range of distributed applications. For example, Blizzard (a famous PC game producer) and Groupon/Amazon/eBay (famous e-commerce websites) all use Akka in their production environment [15]. However, 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 [4], but the message-passing performance of that prototype turned out to be non-satisfactory , due mainly to that C++ Actor Framework uses kernel networking stack to transmit actor messages and the context switching overhead is intolerable in NFV system [29].

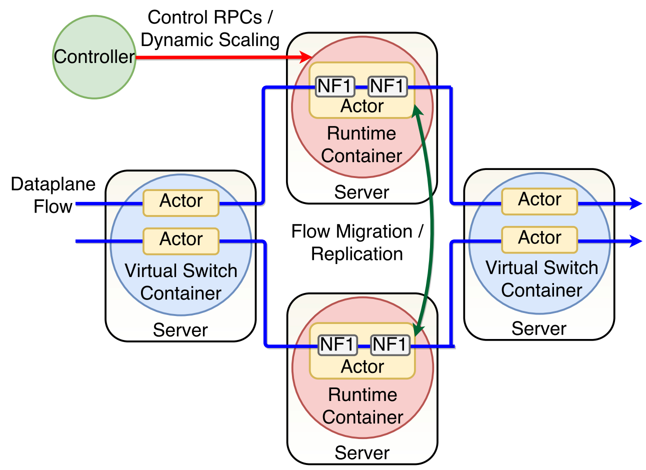


Figure 1: An overview of *NFActor*.

This inspires us to create a customized actor framework for *NFActor* with significantly improved performance.

3. THE NFACTOR SYSTEM

We present the design and key modules of *NFActor* in this section.

3.1 Overview

NFActor includes three key modules: (i) runtime systems that enable flow processing using actors; (ii) virtual switches for distributing and balancing flows to runtime systems; and (iii) a light-weight controller for basic system management. An illustration of the architecture of *NFActor* is given in Fig. 1.

A runtime system, referred to as *runtime* for short, is the execution environment of NFs and service chains. A runtime is running on a container, for quick launching and rebooting in cases of scaling up 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 and run actors for flow distribution, *i.e.*, a virtual switch can be regarded as a special runtime that runs a load balancer function. Runtimes and virtual switches are inter-connected through a L2 network.

The virtual switch is configured with an entry IP address and the controller sets up corresponding flow rules to direct the dataplane flow to the virtual switch,which dispatches it to a runtime hosting the NF service chain that the flow is to traverse (Sec. 3.4). When a runtime receives a new flow, it creates a new flow actor to process the flow. Our runtime design follows the one-actor-one flow principle (Sec. 3.2): the flow actor loads all the required NFs of the service chain, and passes the received packets of the flow to these NFs in sequence of the service chain. Once a packet has been processed by all NFs in the service chain, the runtime sends the packet

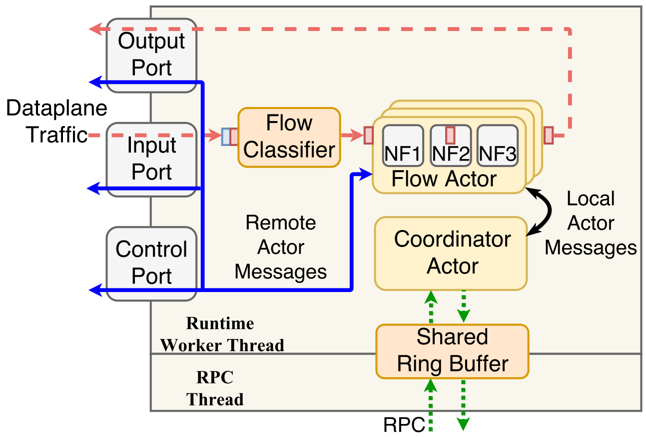


Figure 2: The internal structure of a runtime in *NFActor*.

to a virtual switch (which can be the same or a different virtual switch where the flow comes into the system), where the packet is forwarded to its destination.

The coordinator in *NFActor* is responsible for basic cluster management (Sec. **??**), *e.g.*, updating latest cluster composition to the virtual switches, monitoring workload of runtimes, control dynamic scaling. As compared to SDN controllers used in the existing NFV management systems [24, 36], the coordinator is much lightweight, without involving in the entire flow migration process.Flow migration and replication are handled in fully distributed fashion among the runtimes directly, and only exploit the coordinator in the initialization phase.

The design of *NFActor* targets the following goals.

1. Transparent Resilience. Flow actor should be able to transparently perform resilience operations, including flow migration and replication, regardless of the service chain that is configured for it.

2. High Scalability. The runtime should have good horizontal scalability so that *NFActor*could easily scale-up by launching additional runtimes.

3. Low Overhead. High speed packet processing must be achieved by *NFActor*framework to suit the requirement of modern NFV system.

3.2 Runtime

The concept of a uniform runtime system, as a basic flow processing and scaling unit in *NFActor*, does not appear in most existing work [21, 23, 32]. In existing NFV systems, the basic flow processing and scaling unit is an NF instance, which is a virtual machine or container hosting an instance of a NF. The primary reason that we design such a runtime is to enable NFs/service chains to achieve failure resilience automatically without coordinator intervention, as the runtime provides a network transparent abstraction **Chuan:** change this 'network transparent' to a more accurate wording to communicate and exchange messages among each other, which are crucial for flow migration and replication. Especially, in a runtime, we adopt the simple yet powerful design to create a micro execution context for each flow, and encapsulate processing functions of the flow over its entire service chain inside the micro execution context. Then we can enable failure resilience on the basis of each micro execution context (Sec. 4). To be able to process multiple flows, the runtime is capable of handling multiple micro execution contexts concurrently.

In *NFActor*, we exploit the actor programming model to implement the micro execution context. Each micro execution context is a flow actor. Flow processing by NFs in the service chain, flow migration and replication functionalities are all implemented as message handlers of the flow actor. The runtime provides the basic runtime environment for all the flow actors that it has created.

Fig. 2 shows the internal 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 control messages exchanged among different runtimes for flow migration and replication, which are directly encapsulated inside L3 packets and sent/received using DPDK. Input packets of dataplane flows are first sent to a flow classifier, which uses the classical 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. One flow actor is created for each new flow. The flow actor loads NFs of the service chain configured in the runtime. All packets of the same flow are sent to the same flow actor, which processes them in sequence by passing them through NFs in the service chain.

Each runtime is configured with a specific service chain by the coordinator during its booting phase. The runtime installs and initializes all the NFs as specified in the service chain upon booting. When a flow actor is created, it loads these NFs and uses a number of carefully defined NF APIs, as given in Table 1 in Sec. 3.3, to allocate flow states **Chuan:** not clear what 'allocate flow states' means. extract? and xxx **Chuan:** describe what it does to facilitate flow migration and replication by rewriting the idea of the following sentence: 'Besides service chain processing, the flow actor also provides an execution context for distributed flow migration and replication, in response to certain messages (Sec. 4)'.

Each runtime can host one or multiple flow actors for flows passing through the same service chain that the runtime is configured with, depending on its resource availability and performance isolation requirements. In case of a multi-tenant NFV system, we can run actors processing flows of the same tenant on the same runtime, but those of different tenants on different runtimes, for better security and isolation. When multiple flow actors are concurrently running on one runtime, they are scheduled by a worker thread: **Chuan:** clearly describe how the flow actors are scheduled whenever a message is received at the actor's mailbox, ... In addition, our design of the NF modules in the next section will show that passing packets to a NF for processing in a flow actor is essentially just a function call; only one copy of each NF software needs to be loaded in a runtime, while the flow actors can all make use of it.

The runtime also consists of a RPC thread for receiving RPC requests from the coordinator (for flow migration, replication, etc.) and responding to them. The RPC thread and the worker thread share a ring buffer, used for relaying RPC requests received by the RPC thread to a liaison actor in the worker thread. We use a high-speed shared ring buffer to achieve fast inter-thread communication **Chuan:** add citation. The liaison actor is responsible for coordinating with flow actors to execute the RPC requests from the coordinator.

Discussions on Runtime Design Choices. The design of 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), which migrates a flow by migrating the actor that processes it **Chuan:** revise to more accurate description. 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 **Chuan:** clarify what are 'responses sent from different virtual switch actors' and why the flow actor needs to sync them, adding overhead to 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. 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 by locks **Chuan:** message passing among flow actors or what?, 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 and serialize 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*.

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

The APIs are given in Table 1, provided as four 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. 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. 7.

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 remote control message exchanges with the virtual switch in the same way as message exchanges with other runtimes.

Chuan: describe how the virtual switches are used for receiving outgoing packets from runtime and send them to final destinations

Existing NFV management systems either rely on SDN switches [23, 24] to route flows from one NF to the next, or build a customized data-plane for interconnecting different NF instances [32]. In comparison, our virtual switch is lightweight, only to dispatch flows to runtimes, but not route flows through individual NFs.

3.5 Coordinator

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). This differentiates *NFActor*'s coordinator with the controllers in existing NFV systems [24][36], which need to fully coordinate the entire flow migration process. The design of the coordinator is simplified and improve 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. **Chuan:** describe how you use SDN or openflow like rules to enable dispatching flows to correct virtual switches. The coordinator then launches a new virtual switch and a new runtime, configures the runtime with this service chain, and directs incoming 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 xxx **Chuan:** give the rationale.

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.

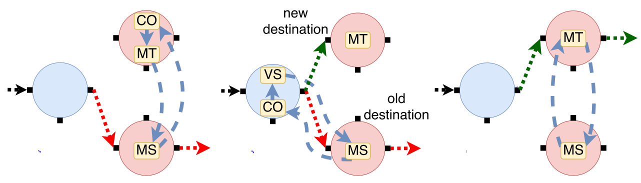
Each runtime in *NFActor* regularly sends heartbeat messages to the coordinator, containing the current load information (*i.e.*, CPU usage, memory usage) of the runtime. When the coordinator detects that a runtime is overloaded, it scales up the cluster by launching a new runtime and configures the service chain on that runtime (Sec. **??Chuan:** point to the scaling section). **Chuan:** does the coordinator takes care of virtual switch scaling as well? describe it

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 a runtime composition of the runtime it belongs to using $NotifyClusterCfg(cfg)$ **Chuan:** does coordinator sends it to virtual switches as well?. The cluster composition learned by different runtimes in a cluster do not have to be consistent because our flow migration and replication protocols perform safety checking by exchanging requests and responses to eliminate the inconsistency **Chuan:** check if this is true and revise accordingly. 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. **Chuan:** briefly describe why you use RPC for coordinator to runtime communication.

4. FLOW MANAGEMENT FOR RESILIENCE

4.1 Distributed Flow Migration

Figure 3 shows workflow of *NFActor*'s flow migration, which involves passing three request-response. Besides being fully distributed, the flow migration also guarantees two properties that (i) except for the migration



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

Figure 3: The flow migration process that migrates migration source actor running on migration source runtime to migration target runtime. (**MT**: Migration target actor. **MS**: Migration source actor. **CO**: Coordinator actor. **VS**: Virtual switch actor. **Dotted line**: Dataplane flow packets. **Dashed line**: Actor messages.)

target side buffer overflow or network packet reordering (which rarely happens in *NFActor*), no flow packets are dropped by the flow migration protocol, which we refer to as **loss-avoidance** property (this is slightly weaker that the loss-free property in OpenNF [24]) and (ii) the same **order-preserving** property as in OpenNF [24]. There has been a long understanding that providing good properties for flow migration would compromise the performance of flow migration [24]. *NFActor* breaks this misunderstanding using the novel distributed flow migration.

Runtimes can check the received cluster view list to obtain the contact address and load information of other runtimes, and select an appropriate target for flow migration and repli- cation.

The details of the three request-responses are summarized below.

- **1st req-rep:** The migration source actor sends its flow-5-tuple to the coordinator actor on the migration target runtime. The coordinator actor creates a migration target actor using the flow-5-tuple contained in the request, which returns a response back to the migration source actor. During the execution of the first request-response, migration source actor continues to process packet.
- **2nd req-rep:** The current flow actor sends its flow-5-tuple and the ID of the migration target runtime to the coordinator actor on the virtual switch. The coordinator actor uses the flow-5-tuple to find out the virtual switch actor and notifies it to change the destination runtime to migration target runtime. After changing the destination runtime, the virtual switch actor sends a response back to the migration source actor. The migration target actor starts to receive packets after the destination runtime of the virtual switch actor is changed and buffer all the received packets until it receives the third request. In the meantime, the migration source actor keeps processing the input packets until it receives the second response.
- **3rd req-rep:** the migration source actor sends its flow state to the migration target actor. After receiving the flow states, the migration target actor saves them, gives a response to the migration source actor and immediately start processing all the buffered packets. The migration source actor exits when it receives the response.

The Loss-Avoidance Property. Before the migration target actor receives the third request, it needs to buffer input packets indefinitely, which might lead to a buffer overflow if the third request takes a long time to arrive. *NFActor* simply drops additional flow packets after buffer overflow because *NFActor* needs to process packet at a high throughput rate and does not want to grow buffer indifinitely. In *NFActor*, a large collective buffer is used to buffer the packets for different migration target actors and the distributed flow migration process is extremely fast, so the buffer overflow rarely happens, even when migrating a huge number of flows. This is demonstrated in the evaluation section **??**.

Besides buffer overflow, the only step that might incur potential packet drop is in the third request-response. When the second response is received by the migration source actor, it must immediately send its flow state in the third request to the migration target actor. After sending the third request, there might be pending flow packets continuing to arrive at migration source actor. These pending packets are are sent out by the virtual switch actor before the destination runtime is changed. If this happens, the migration source actor has to discard these pending flow packets because it has already sent out the third request. Continuing to process these packets may generate inconsistent output packets.

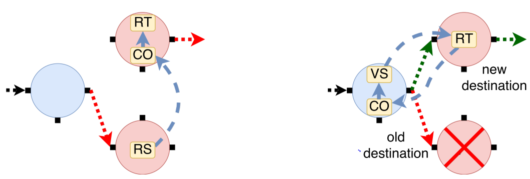
If the network doesn't reorder packet, which is a common case because *NFActor* is deployed over a L2 network, *NFActor*'s flow migration can eliminate the second cause of packet drop by transmitting second response in a network packet over the same network path as the data plane packets that are sent to the migration source actor. Recall that in Figure 2, the remote messages could be sent over input/output port of a runtime. The second response is encapsulated in a raw packet **??**, sent by the output port of the virtual switch and received by the input port of the migration source runtime, therefore sharing the same network path as the data plane packets that are sent to the migration source actor.

Because the second response are sent after the destination runtime of the virtual switch actor is changed and share the same network path as the data plane packets that are sent to the migration source actor, it also becomes a strong indication that no more input packets will be sent to the migration source actor. This is verified in our evaluation **??**.

Order-preserving Property. Since the second request-response eliminate the packet drop if the network doesn't reorder packets, flow packets could always be processed in the order that they are sent out from the virtual switch. The order-preserving property is therefore guaranteed.

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

4.2 Scalable Flow Replication



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

Figure 4: Flow replication that replicates the original actor running on the original runtime to replica runtime. (**RT**: Replication target actor. **RS**: Replication source actor. **CO**: Coordinator actor. **VS**: Virtual switch actor. **Dotted line**: Dataplane flow packets. **Dashed line**: Actor messages.)

The biggest difference of the *NFActor*'s replication method and existing works such as [38] is that NFActor framework replicates individual flow, not NF. This replication strategy is transparent to the NF modules and improves the scalability and resource utilization rate of *NFActor*. as flows could be directly replicated on another runtime, without the need for a dedicated backup server. In the mean time, this fine grained replication strategy provides a the same output-commit property as indicated in [38] with a desirable replication throughput and fast recovery time.

The detailed flow replication process is shown in figure 4. When a flow actor is created, it acquires its replica runtime by querying a round-robin list. If the flow actor has a valid replica runtime, whenever it finishes processing the packet, it sends a remote message, containing the current flow state and the packet, to the coordinator actor on the replication target runtime. The coordinator actor on the replication target runtime creates a replica flow actor using the same flow-5-tuple as the original flow actor to handle all the replication messages. The replica flow actor saves the flow state and sends the packet out from the output port of the replica runtime. Similar with [38], the receiver on the side of the output port of the replica runtime can only observe an output packet when the flow state has been replicated.

When a runtime fails, the coordinator sends recovery RPC requests **??** to all the replica runtime of the failed runtime. This RPC enables replica flow actor to send a request to the virtual switch actor, asking it to change the destination runtime to the replica runtime. When the response is received by the replica flow actor, the original flow is successfully restored on the replica runtime.

5. DYNAMIC SCALING

The dynamic scaling algorithm used by the controller is shown in Algorithm 1. The algorithm fully exploits the fast and scalable distributed flow migration to quickly resolve hot spot during scale-up and immediately shutdown idle runtime during scale-in.

The algorithm starts (line 2 in Algorithm 1) by polling the workload statistics from all the runtimes, containing the number of dropped packets on the input port, the current packet processing throughput and the current active flow number.

Since each runtime has a polling worker thread that keeps the CPU usage to 100% all the time, the controller can not decide whether the runtime is overloaded simply by reading the CPU usage. Instead, the controller uses the total number of dropped packets on the input port to determine overloaded. This is a very effective indicator in *NFActor* because when the runtime is not overloaded, it can not timely polls the all the packets from the input port, therefore increasing the number of the dropped packets. The controller keeps recording the maximum throughput during the previous overload for each runtime and uses that to identify idleness. If the current throughput is smaller than half of maximum throughput, then the controller identifies the runtime as idle.

Then algorithm decides whether to scale-out or scale-in (line 3-9 in Algorithm 1) and executes corresponding operations. To scale-up (line 10-15 in Algorithm 1), the runtime launches a new runtime and keeps migrating 500 flows from each overloaded runtimes, until all the hotspots are resolved. If the new runtime is overloaded during the migration, the algorithm continues to scale-up. To scale-in (line 16-20), the runtime selects a runtime with smallest packet throughput and migrate its flows to the rest of the runtime.

The algorithm uses 500 flows as the basic migration number, which is a tunable value in *NFActor*. We use this value because 500 flows could be migrated within one millisecond in *NFActor* and the controller could gradually increases the workload during migration to evenly balance the workload.

6. IMPLEMENTATION

NFActor framework is implemented in C++. The core functionality of NFActor framework contains around 8500 lines of code. We use BESS [2][?] as the dataplane

Algorithm 1: The dynamic scaling algorithm used by *NFActor*'s controller.

```
while True do
    get the workload statistics of all the runtimes;
    state = null;
    if at least one runtime is overloaded then
        state = scale-out;
    else if the current throughput of all runtimes are
    smaller than half of the maximum throughput then
        state = scale-in;
    else
        state = null;
    if state == scale-out then
        launch a new runtime;
        while the new runtime is not overloaded &&
        the hotspots in overloaded runtimes are not
        resolved do
            foreach overloaded runtime do
                migrate 500 flows to the new runtime;
            update the workload statistics of all the
            runtimes;
    if state == scale-in then
        select a runtime with the smallest throughput
        to scale-in;
        notify the virtual switch to stop sending new
        flows to the selected runtime;
        while active flows on selected runtime is larger
        than 0 do
            migrate 500 flows to other runtimes in a
            round-robin way;
```

inter-connection tool to connect different runtimes and virtual switches. The three ports that are assigned to each runtime are zero-copy VPort in BESS, which is a high-speed virtual port for transmitting raw packets. BESS could build a virtual L2 ethernet inside a server and connect this virtual ethernet to the physical L2 ethernet. By connecting the virtual L2 ethernet with the ports of runtimes, We can connect different runtimes running on different servers together.

6.1 Reuse BEES Module System

The runtime needs to poll packets from the input port, schedule flow actors to run and transmit remote actor messages. To schedule these tasks efficiently, we decide to reuse BESS module systems. BESS module system is specifically designed to schedule packet processing pipelines in high-speed NFV systems, which is a perfect suit to NFActor runtime architecture. We port the BESS module system and BESS module scheduler to the runtime and implement all the actor processing tasks as BESS modules. These modules are connected into the following 5 pipelines.

- The first/second pipeline polls packets from the input/output port, runs actor scheduler on these packets and sends the packets out from the output/input port.
- The third pipeline polls packets from control ports, reconstruct packet stream into remote actor messages and send the actor messages to the receiver actors. (The first/second pipeline also carries out this processing because remote messages are also sent to input/output port **??**).
- The fourth pipeline schedules coordinator actor to execute RPC requests sent from the controller. In particular, coordinator actor updates the configuration information of other runtimes in the cluster and dispatches flow migration initiation messages to active flow actors in the runtime.
- When processing the previous four pipelines, the actors may send remote actor messages. These messages are placed into ring buffers **??**. The fifth pipeline fetches remote actor messages from these ring buffers and sends remote actor messages out from corresponding ports.

The runtime uses BESS scheduler to schedule these 5 pipelines in a round-rubin manner to simulate a time-sharing scheduling.

6.2 Customized Actor Library

To minimize the overhead of actor programming, we choose to implement our own actor library. In this actor library, due to the 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 mailbox []. For remote actor message passing, we assign a unique ID to each runtime and each actor. The sender actor only needs to specify the receiver actor's ID and runtime ID, then the reliable transmission module **??** could deliver the remote actor message to the receiver actor.

To schedule flow actors, we directly run a flow actor scheduler in the first three pipelines. The flow actor scheduler redirects both input flow packets and remote actor messages to corresponding flow actors, by looking up the flow actors using the flow identifier.

Even though the functionality implemented by our customized actor library is very simple compared with other mature actor libraries [15] [4], the simple architecture of our actor library decreases overhead associated with actor processing, enabling *NFActor*to satisfy the high-speed packet processing requirement of mordern NFV system.

6.3 Reliable Message Passing Module

To reliably deliver remote actor messages, we build a customized reliable message passing module for NFActor framework. Unlike user-level TCP stack, where messages are inserted into a reliable byte stream and transmitted to the other end, the reliable message passing encodes messages into reliable packet streams.

The reliable message passing module creates one ring buffer for each remote runtime. When an actor sends a remote actor message, the reliable transmission module allocates a packet, copy the content of the message into the packet and then enqueue the packet into the ring buffer. A message may be splitted into several packets and different messages do not share packets. When the fifth pipeline is scheduled to run, the packets containing remote messages are dequeued from the ring buffer. These packets are configured with a sequential number, appended with a special header to differentiate them from normal data plane packets and sent to their corresponding remote runtimes. The remote runtime sends back acknowledgement packets. Retransmission is fired up 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 like [?] to implement the reliable message passing module. Because compared with our simple goal of reliably transmitting remote actor messages over an inter-connected L2 network, using a user-level TCP imposes too much processing overhead for reconstructing byte stream into messages. The packet-based reliable message passing provides additional benefits during flow management tasks. For instance, because the second response in the flow migration protocol is sent as a packet on the same path with the dataplane flow packet, it enables us to implement loss-avoidance migration with ease **??**. Also, during flow replication, we can directly send the output packet as a message to the replica, without the need to do additional packet copy.

6.4 Dedicated RPC Thread

As mentioned in **??**, the runtime has a dedicated RPC thread for receiving RPC request sent from the controller. In *NFActor*, the RPC are implemented with GRPC [8] and the RPC requests are sent over a reliable TCP connection. To avoid context switches, *NFActor*uses a dedicated RPC thread to receive the initial RPC requests and forward these requests to the worker thread through a shared ring buffer. This improves the performance of the worker thread by eliminating potential context switches caused by using kernel networking stack.

6.5 New Applications

Asides from NF processing, we build several new applications that is inspired by our light-weight and distributed flow migration. These applications use flow migration to achieve useful practical functionalities, including live NF update, reduce output bandwidth for deduplication NF and ensure reliable and safe MPTCP processing. We evaluate and demonstrate these new applications in our evaluation.

7. EVALUATION

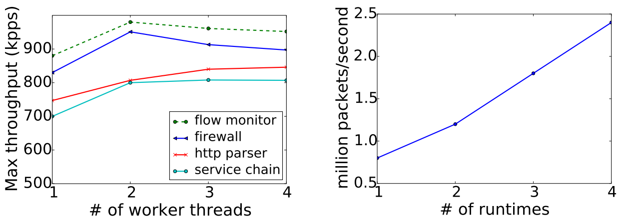
We evaluate *NFActor*framework using a Dell R430 Linux server, containing 20 logical cores, 48GB memory and 2 Intel X710 10Gb NIC. In our evaluation, we run the controller process, helper deamon process, virtual switch container and runtime containers on the same server.

To evaluate the performance of *NFActor*, we implement 3 customized NF modules using the API provided by *NFActor* framework, the 3 NF modules are flow monitor, firewall and HTTP parser. The flow monitor updates an internal counter when it receives a packet. The firewall maintains several firewall rules and checks each received packet against the rule. If the packet matches the rule, a tag in the flow state is flipped and later packets are automatically dropped. The firewall also records the connection status of a flow in the flow state. For the HTTP parser, it parses the received packets for the HTTP request and responses. The requests, responses and the HTTP method are saved in the flow state. Throughout the evaluation, we use a service chain consisting of "flow monitor→firewall→http parser" as the service chain. We generate evaluation traffic using the BESS's FlowGen module and we directly connect the FlowGen module to the external input port of the virtual switch.

The rest of the section tries to answer the following questions. *First,* what is the packet processing capacity of *NFActor* framework? (Sec. 7.1) *Second,* how well is *NFActor* scales, both in terms of the number of worker threads used by a runtime and the number of runtimes running inside the system? (Sec. 7.1) *Third,* how good is the flow migration performance of *NFActor* framework when compared with existing works like OpenNF? (Sec. 7.2) *Fourth,* what is the performance overhead of flow state replication and does the replication scale well? (Sec. 7.3)

7.1 Packet Processing Capacity

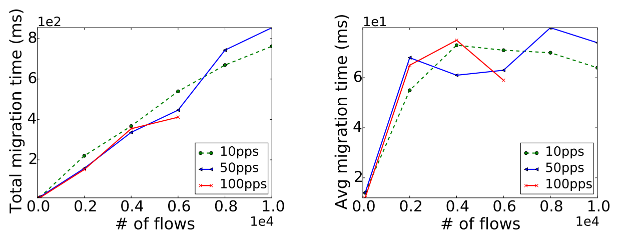
Figure 5 illustrates the normal case performance of running *NFActor* framework. Each flow in the generated traffic has a 10 pps (packet per second) per-flow packet rate. We vary the number of concurrently generated flows to produce varying input traffics. In this evaluation, we gradually increase the input packet rate to the *NFActor* cluster and find out the maximum packet rate that the *NFActor* cluster can support without dropping packets. In figure 5a, the performance of different NF modules and the service chain composed of the 3 NF modules are shown. Only one *NFActor* runtime is launched in the cluster. It is configured with



(a) Packet processing capacity of a single *NFActor* runtime system running with different number of worker threads.

(b) Aggregate packet processing capacity of several *NFActor* runtimes.

Figure 5: The performance and scalability of *NFActor* runtime, without enabling flow migration



(a) The total time to migrate different numbers of flows.

(b) The average flow migration time of a single flow when migrating different number of flows.

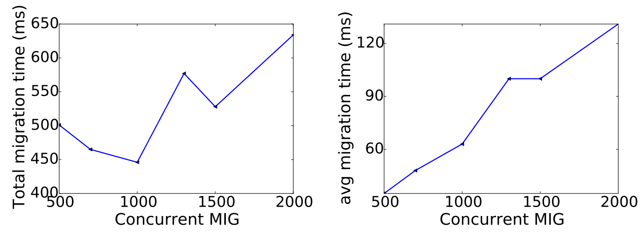
Figure 6: The flow migration performance of *NFActor*

different number of worker threads. In figure 5b, we create different number of *NFActor* runtimes and configure each runtime with 2 worker threads. Then we test the performance using the entire service chain.

From figure 5a, we can learn that the packet throughput decreases when the length of the service chain is increased. Another important factor to notice is that the *NFActor* runtime does not scale linearly as the number of worker threads increases. The primary reason is that inside a *NFActor* runtime, there is only one packet polling thread. As the number of input packets increases, the packet polling thread will eventually become the bottleneck of the system. However, *NFActor* runtime scales almost linearly as the total number of *NFActor* runtimes increases in the cluster. When the number of runtimes is increased to 4 in the system, the maximum packet throughput is increased to 2.4M pps, which confirms to the line speed requirement of NFV system.

7.2 Flow Migration Performance

We present the evaluation result of flow migration in this section. In order to evaluate flow migration performance, we initialize the cluster with 2 runtimes running with 2 worker threads and then generate flows to one of the runtimes. Each flow is processed by the service chain consisting of all the 3 NF modules. We generate



(a) The total time to migrate all the flows when changing the maximum concurrent migrations.

(b) The average flow migration time of a single flow when changing the maximum concurrent migrations.

Figure 7: The flow migration performance of *NFActor* when changing the maximum concurrent migrations.

different number of flows, each flow has the same per-flow packet rate. In order to see how the evaluation performs under different per-flow packet rate, we also tune the per-flow packet rate with 10pps, 50pps and 100pps. When all the flows arrive on the migration source runtime. The migration source runtime starts migrating all the flows to the other runtime in the cluster. We calculate the total migration time and the average per-flow migration time. In order to control the workload during the migration, the runtime only allows 1000 concurrent migrations all the time. The result of this evaluation is shown in figure 7.

We can see that as the number of migrated flows increase, the migration completion time increases almost linearly. This is because the average flow migration time remains almost a constant value and the runtime controls the maximum number of concurrent migrations. Note that when the system is not overloaded at all (100 flows), the average flow migration completion time is as small as 636us.

When the per-flow packet rate is 100pps, the maximum number of flows that we use to evaluate the system is 6000. Continuing the evaluation with 8000 and 10000 flows just overloads the runtime as shown in figure 5a.

Since we control the number of concurrent migrations, we also want to see what happens if we change the number of concurrent migrations. We generate 6000 flows, each with 50 pps per-flow packet rate, and change the the number of concurrent migrations. The result of this evaluation is shown in fig 7. As we can see from fig 7b, increasing the maximum concurrent migrations increase the average flow migration completion time. However, whether the total flow migration completion time increased depends on the total number of flows that wait to be migrated. From the result of fig 6b, the choice of 1000 concurrent migrations sits in the sweat spot and accelerates the overall migration process.

Finally, we compare the flow migration performance of *NFActor* against OpenNF [24]. We generate the same number of flows to both *NFActor* runtimes and NFs

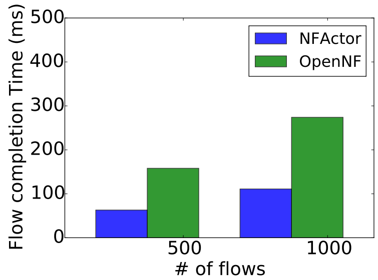
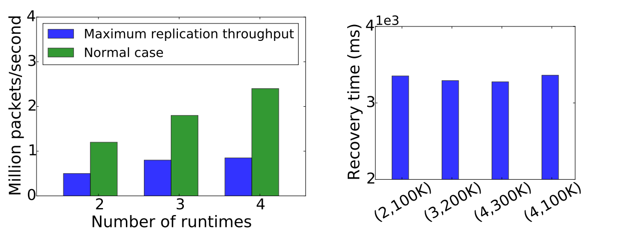


Figure 8: The flow migration performance of *NFActor*. Each flow in *NFActor* runtime goes through the service chain consisting of the 3 customzied NF modules. OpenNF controlls PRADS asset monitors.



(a) The packet throughput of a *NFActor* cluster when replication is enabled. The throughput is compared against the throughput when replication is disabled.

(b) The recovery time of a failed runtime 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 9: The flow migration performance of *NFActor*

controlled by OpenNF and calculate the total time to migrate these flows. The evaluation result is shown in figure 8. Under both settings, the migration completion time of *NFActor* is more than 50% faster than OpenNF. This performance gain primarily comes from the simplified migration protocol design with the help of actor framework. In *NFActor*, a flow migration process only involves transmitting 3 request-responses. Under light workload, the flow migration can complete within several hundreds of microseconds. Under high workload, *NFActor* runtime system controls the maximum number of concurrent migrations to control the migration workload, which may increase the migration performance as indicated in figure 7a. All of these factors contribute to the improved flow migration performance of *NFActor* framework.

7.3 Replication Performance

In this section, we present the flow state replication evaluation result. In our evaluation, the actor creates a flow snapshot for every 10 flow packets that it has processed. Then it sends the flow state snapshot to the replica storage. In this evaluation, we first generate flows to the *NFActor* cluster to test the maximum throughput of a *NFActor* cluster when enabling replication. Then we calculate the recovery time of failed *NFActor* runtime. The recovery time is the from time that the controller detects a *NFActor* runtime failure, to the time that the recovered *NFActor* finishes replaying all of its replicas and responds to the controller to rejoin the cluster. Through out this evaluation, the runtime uses the service chain consisting of the 3 NF modules to process the flow. The result of the evaluation is shown in figure 9.

In figure 9a, we can see that there is an obvious overhead to enable replication on *NFActor* runtimes. The overall throughput when replication is enabled drops around 60%. This is due to the large amount of replication messages that are exchanged during the replication process. Internally, the replication messages are sent over Linux kernel networking stack, which involves data copy and context switching, thus increasing the performance overhead of using replication. However, the overall throughput when replication is enabled could scale to 850K pps when 4 runtimes are used, which is enough to use in some restricted settings.

Finally, figure 9b shows the recovery time of *NFActor* runtime when replication is enabled. We found that the recovery time remains a consistent value of 3.3s, no matter how many runtimes are used or how large the input traffic is. The reason of this consistent recovery time is that the *NFActor* runtime maintains one replica on every other *NFActor* runtimes in the cluster. During recovery, several recovery threads are launched to fetch only one replica from another runtime. Then each recovery thread independently recovers actors by replaying its own replica. In this way, the recovery process is fully distributed and scales well as the number of replica increases. Note is that the average time it takes for a recovered runtime to fetch all the replicas and recover all of its actors is only 1.2s. So actually around 2.1s is spent in container creation and connection establishment.

8. DISCUSSION

Even though *NFActor*provides transparent resilience for stateful NFs, *NFActor*focuses on handling per-flow state. Currently, *NFActor*could not correctly handle shared states, *i.e.*, the states shared by a bunch of flows. Even though the NF API in *NFActor*achieves a clean separation between per-flow state and NF processing logic, it can not correctly separate shared state. Therefore, migrating and replicating flows that share states with other flows may cause un-predicted errors in *NFActor*. A potential solution to this limitation is to enforce the programmer to write a handler that explicitly deals with the inconsistency during resilience operation. We leave this to our future work.

Another limitation of *NFActor*is that *NFActor*may incorrectly handle flows with packet encapsulation. *NFAc-*

*tor*uses the flow-5-tuple to differentiate flows. However, different flows may share the same flow-5-tuple if their flow packets are encapsulated. This is a common for flows that are sent over the same VxLAN tunnel. In that case, those flows are handled by the same flow actor, resulting in incorrect flow processing. If *NFActor*knows what kind of encapsulation the input packet uses, *NFActor*could add a decapsulation function in the virtual switch to correctly extract different flows. This is also left in our future work.

9. CONCLUSION

In this work, we present a new framework for building resilient NFV system, called NFActor framework. Unlike existing NFV system, where NF instances run as a program inside a virtual machine or a container, NFActor framework provides a set of API to implement NF modules which executes on the runtime system of NFActor framework. Inside the NFActor framework, packet processing of a flow is dedicated to an actor. The actor provides an execution context for processing packets along the service chain, reacting to flow migration and replication messages. NF modules written using the API provided by NFActor framework achieves flow migration and state replication functionalities in a transparent fashion. The implementer of the NF module therefore only needs to concentrate on designing the core logic. Evaluation result shows that even though the NFActor framework incurs some overhead when processing packets, the scalability of NFActor runtime is good enough to support line-speed requirement. NFActor framework outperforms existing works by more than 50% in flow migration completion time. Finally, the flow state replication of NFActor is scalable and achieves consistent recovery time.

10. 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] Erlang. `https://www.erlang.org/`.

[6] FFMPEG. `https://ffmpeg.org/`.

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

[8] GRPC. `http://www.grpc.io/`.

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

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

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

[12] Linux Virtual Server. `www.linuxvirtualserver.org/`.

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

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

[15] Scala Akka. `akka.io/`.

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

[17] The NFActor Project. `http://` 2017.

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

[19] 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.

[20] 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.

[21] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: Enabling Innovation in Middlebox Applications. In *Proc of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox'15)*, 2015.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.