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. Our main observation is that actor provides the unique benefits for light-weight, decentralized migration of network flow states. Based on *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 **Chuan:** complete the advantage brought by per-flow management. 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.**Chuan:** state whether they can or cannot be achieved by other NFV systems. 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 [28], OpenBox [18], CoMb [32], xOMB [17], Stratos [19], OpenNetVM [22, 34], ClickOS [26]. 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 [30, 33] and flow migration [20, 31, 23] capabilities simultaneously, both of which are of pivotal importance in practical NFV systems for resilience and scalability. **Cui:** add dpdk

Failure resilience is crucial for stateful NFs. Many NFs maintain important per-flow states. 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 [10] maintain TCP connection-related states by parsing TCP SYN/ACK/FIN packets for each flow. Some load-balancers [11] 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 [33].

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 con-

nection to exchange many requests and responses with a web server [8]; 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 [20]; 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. 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 [20][31]. This usually requires a huge amount of manual work to add up to thousands of lines of source code for one NF, *e.g.*, [20] reports that it needs to add 3.3K LOC for Bro [3] and 7.8K LOC for Squid caching proxy [16]. Realizing this difficulty, [23] uses 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.

Chuan: Modify this into an centralized SDN argument. Even if NF states can be correctly acquired, resilience operation, especially flow migration, needs to modify the flow state and redirect the flow to a new target. In existing systems, this is usually handled by a centralized SDN controller, which needs to initiate and coordinate the the entire migration process for each flow. Aside from compromised scalability due to the centralized control, to migrate losslessly migrate a flow, the controller has to perform complicated migration protocols that involves passing multiple message among SDN switches, migration source and migration target [20], which limits the applicability of the flow migration.

In this paper, we propose a software framework for building resilient NFV systems, *NFActor*, exploiting the actor framework for programming distributed services [1, 15, 27]. Our main observation is that actor provides the unique benefits for light-weight, decentralized migration of network flow states. *NFActor* tracks each flow's state with our high-performance flow actor, which transparently separates flow state from NF processing logic. *NFActor*'s flow migration is achieved by flow actors themselves without involving a centralized coordinator.

NFActor achieves transparent resilience, easy scalability and high performance in network flow processing based on the following technical highlights: **Chuan:** improve and add design highlights in the following

▷ Unlike existing work [20, 33] that patch resilience functionalities into NF software, *NFActor* achieves *transparent resilience* by providing a clean separation between important NF states and core NF processing logic in each NF module using a unique API, which makes extracting, serializing and transmitting important flow states an easy task. *NFActor* enforces a complete separation between resilience operations and NF module implementation. Using *NFActor*, programmers implementing the NF module only needs to focus on the core NF logic, they do not need to take care of the underlying resilient operations. On the other hand, flow actors can be transparently migrated and replicated as long as they load NF modules that are written with *NFActor* API. We refer to this as transparent resilience in *NFActor*.

▷ Fundamentally different from the existing systems, *NFActor* adopts a per-flow management principle. *NFActor* manages flows through a one-actor-one-flow abstraction. Packet processing of a flow are completely delegated to a unique actor that is specially created for that flow. *NFActor* creates a micro execution context for each flow using flow actors. Inside this execution context, a flow actor could actively exchange messages with other actors and transmit flow states without disturbing the normal NF processing. This serves as the basis for transparent resilience. **Chuan:** describe what the per-flow management is about and its advantages

▷ Resilience support in *NFActor* is provided in a fully distributed fashion, without directly involving a central controller, which distinguishes *NFActor* from the existing NFV systems [20]. Due to the message passing and decentralized nature of actor programming model, the flow management tasks are fully automated by actor scheduling and message passing. There is no need for the continuous monitoring of a centralized controller. Therefore the controller in NFActor framework is extremely light-weighted and failure resilient. The actor programming model only imposes a small overhead during service chain processing and flow management, improving the performance of NFActor. *NFActor*is implemented on top of high speed packet I/O library DPDK [9], which further improves the performance of *NFActor*.

Going beyond resilience, NFActor framework also enables several interesting new applications that existing NFV systems are hard to provide **Chuan:** explain why

hard. These applications utilize the feature of decentralized flow migration to reduce the output bandwidth consumption during deduplication and ensures correct MPTCP subflow processing. NFActor framework also provides live updates to NFs that process packets at the rate of millions packets per second with almost zero down time, due to the blazingly fast flow migration speed.

We implement *NFActor* on a real-world testbed. **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

2.1 Network Function Virtualization

A NFV system [13] typically consists of a controller and many VNF instances. Each VNF instance is a virtualized device running NF software. VNF instances are connected into service chains, implementing certain network services, *e.g.*, access service. Packets of a network flow go through the NF instances in a service chain in order before reaching the destination.

A VNF instance constantly polls a network interface card (NIC) for packets. Using traditional kernel network stack incurs high context switching overhead [26] and greatly compromise the packet processing throughput. To speed things up, hypervisors usually map the memory holding packet buffers directly into the address space of the VNF instances with the help of Intel DPDK[9] or netmap [12]. VNF instances then directly fetch packets from the mapped memory area, avoiding expensive context switches. Recent NFV systems [28, 21, 33, 26, 22] are all built using similar techniques.

Even though using DPDK and netmap to improve the performance of packet processing has become a new trend. Existing flow management systems are still using kernel networking stack to implement the communication channel. On contrary, NFActor completely abandons the kernel networking stack, by constructing a reliable transmission module using DPDK. Using this reliable transmission module does not incur any context switches, thereby boosting the message throughput to 6 million messages per second in our evalution.

2.2 Actor Model

The actor programming model has been used as the basic building block for constructing massive, distributed

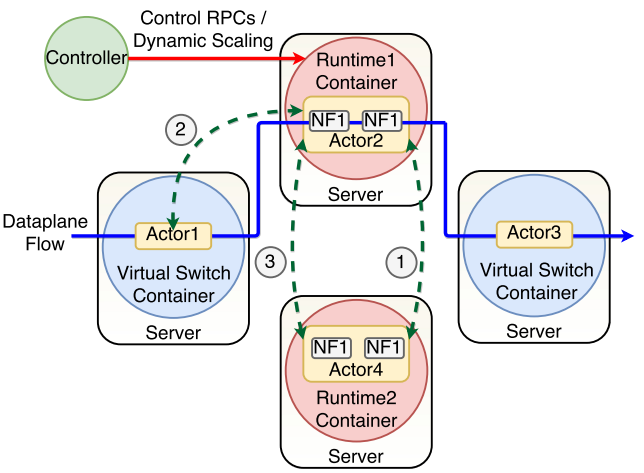


Figure 1: An overview of NFActor framework.

systems[1, 15, 27]. 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, status of peer actors), 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.

There are several popular actor frameworks, *i.e.*, Scala Akka [15], Erlang [5], Orleans [14] and C++ Actor Framework [4]. These actor frameworks have been used to build a broad range of distributed programs, including on-line games and e-commerce. For example, Blizzard (a famous PC game producer) and Groupon/Amazon/eBay (famous e-commerce websites) all use Akka in their production environment [15].

Actor model is a natural fit when buildling flow execution context. In a VNF instance, we can create one actor for one flow, and map the flow packet processing to actor message processing. In the mean time, the flow management tasks could be implemented as message handlers on the actor. However, none of the existing actor systems are optimzed for NFV envirnoment. In our initial prototype, we use C++ Actor Framework [4] to build NFActor, but the performance of that prototype turns out to be not satisfactory. This forces us to make a customized actor model for NFActor and greatly improves the performance.

3. NFACTOR OVERVIEW

Figure 1 demonstrates the basic architecture of *NFActor*, consisting of a light-weight controller, input and output virtual switches and several runtime systems (referred to as *runtime* in short). Both runtime and virtual switch are uniform platforms to run actors, they are executed inside containers for quick rebooting in case of

failure and elastic scaling in case of overload. **Cui:** Section 3 Overview: should give a clear definition on what a "runtime" is.

The incoming dataplane flows are first sent to the input virtual switch, which dispatches them to runtimes in a load-balanced fashion. Each runtime hosts a NF service chain that is determined during runtime boot phase. The runtime is designed with one-actor-one-flow principle, which decreases the overhead of passing messages among additional number of actors when processing the flow and eases the design of *NFActor*'s distributed flow migration and replication. **Cui:** Section 3 Overview: should briefly mention our design choice: "one-flow-one-actor". Therefore, when a runtime receives a new flow, it creates a new actor to process the flow. The actor loads all the required NF modules of the service chain and passes the received flow packet to these NF modules in sequence to achieve service chain processing. Once the service chain processing is finished, the actor sends the packet to an output virtual switch, where the packet is sent to its final destination.

The key feature that differentiates NFActor framework with existing works like [20] and [31] is that, resilience operations, *i.e.* flow migration and replication, is fully decentralized. Figure 1 demonstrates a flow migration process that migrates actor 2 on runtime 1 to runtime 2. The migration starts by actor 2 sending the first request to runtime 2. Runtime 2 launches actor 4, which is used as the migration target actor for accepting the flow packets and flow states of actor 2, before responding to actor 2. Actor 2 then sends the second request to the virtual switch, asking it to modify the output route to runtime 2. Finally, actor 2 sends its flow states in the third request to actor 4, completing the whole migration process. The details of our distributed flow migration is further illustrated in **??**. **Cui:** The third paragraph on describing flow migration is not clear. Should make it more clear, or make it the "Section 3.1 Example" section, or cut this paragraph.

A controller does exists in *NFActor*, which is a relatively light-weighted one. It's tasks are to monitor the workload of each runtime and control dynamic scaling. It's only involvement in flow management is during the initiation phase, when it uses control RPCs to tell the flows which runtime they should migrate to.

our main observration is that actor privdes the unique feature to implement a high throughput system due to its light-weight execution state and and

light-weight, decentralized migration of network flow states

Our main observation is that actor provides the unique benefits for light-weight, decentralized migration of network flow states.

4. NF MODULE API

To facilitate transparent resilience, *NFActor*needs to ensure a clean separation between the NF processing logic and the flow states associated with each flow. To enforce this criteria, we build a set of easy to use APIs that can be used to create a broad range of NF modules for *NFActor*.

Table 1 summarizes these APIs, which are provided as four public methods for each NF module. The allocation and deallocation methods enforce each NF module to allocate a flow state that is stored by a flow actor, therefore transferring the primary access right of the flow state to the flow actor. When the flow actor receives a input packet message, it calls the packet processing method to process the input packet passing along the flow state, so that any changes to the flow state when calling the third method is immediately visible to the flow actor when the flow actor finishes processing an input packet.

When the flow actor processes messages during resilience operations, it always has direct access to the latest flow state, which could be transmitted without disturbing the normal of NF processing. Therefore, the combination of the three methods servers as the basis for the transparent resilience in *NFActor*. These three methods are simple to use and properly generalize the structure of NFs that processes flow packets based on per-flow state. Using these three methods, we are able to create several representing NFs as shown in **??**. **Cui:** give a clearer description on why the four APIs can facilitate transparent resilience and build a rich set of NFs.

Finally, the flow actor could use the fourth method to check where the NF would like the actor to migrate to, by passing the current cluster configuration and the current flow state. This enables the flow actor to actively migrate itself instead of waiting for migration initiation command sends from the controller **??**, and sparkles several useful applications (i.e. decreasing the output bandwidth during deduplication and ensuring reliable MPTCP subflow processing **??**) that none of existing NFV systems can achieve.

Table 1: The public API methods that must be implemented for each NF modules in NFActor framework.

API	Usage
nf.allocate_new_fs()	Allocate a new flow state for a new flow actor.
nf.deallocate_fs(fs)	Deallocate the flow state when the flow actor expires or quits.
nf.process_pkt(input_pkt, fs)	Process the input packet using the current flow state associated with the flow actor.
nf.get_migration_target(cluster_cfg, fs)	Query the NF using the current cluster configuration and the current flow state about where the flow actor should be migrated to. This method enables each actor to do active migration.

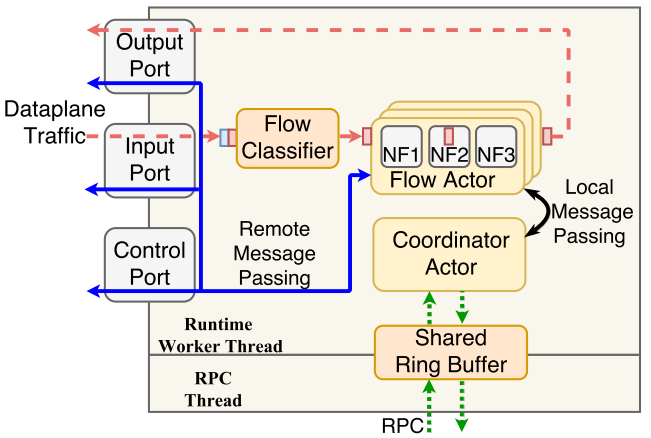


Figure 2: The internal architecture of a NFActor runtime system.

5. RUNTIME ARCHITECTURE

5.1 Runtime

Figure 2 demonstrates the internal architecture of a runtime. The runtime consists of a worker thread, which schedules actors to do service chain processing, and an RPC thread, which receives RPC requests from the controller and forwards them to the worker thread through a shared ring buffer. The runtime is configured with three ports, from which it can send and receive packets.

The worker thread polls dataplane packets from the input port and forwards the packets to a flow classifier, which uses traditional flow-5-tuple (i.e. IP destination address, IP source address, transmission layer protocol, source port and destination port) as the key to classify flows. For each new flow, the flow classifier creates an unique flow actor and forwards all the packets associated with the flow to that flow actor. The flow actor provides a micro execution context for the flow, which is configured several message handlers, including a handler for processing input packet and handlers for dealing with flow migration and replication. There is a coordinator actor who is responsible for executing the RPC requests sent from the coordinator and coordinate flow actors during flow management tasks. Flow actor and coordinator actor on the same runtime could directly pass messages with each other. Actors on different runtime use reliable message transmission module **??** to send remote messages over the three ports **??**.

The runtime is configured with a specific service chain during the boot phase and initializes all the NF modules as specified in the service chain. When a flow actor is created, it loads these modules and uses the flow state allocation method 1 to allocates all the To process flows across a service chain, during the initialization phase of the runtime, a service chain specifier is passed in to the runtime. The runtime then loads all the NF modules as indicated in the service chain specifier. When the flow classifier creates a new flow actor, the flow actor also loads these NF modules on the service chain and passes the input packet along the NF modules in sequence.

The reason that the runtime is designed as a single-worker-thread program is because the multi-worker-thread design may not bring significant performance gain. In our initial prototype implementation, we use LIBCAF [4] library to construct flow actors. LIBCAF library creates multiple worker threads and schedules flow actors to run on these worker threads. Because LIBCAF completely conceals the internal interfaces of the worker threads, we have to create a dedicated polling thread to poll packet from the input port. Under this design, we find that the maximum throughput of a runtime does not increase when the number of LIBCAF worker thread increases, because the polling thread has always been a bottleneck. Therefore, we abandon the multi-worker-thread design and use a single worker thread to poll packets and schedule flow actors. To our surprise, this architecture turns out to work very well because it allows us to perform aggressive optimization of actor programming model **??**. In the mean time, we can still maintain the scalability of the system by launching more runtimes.

5.2 Virtual Switch

The virtual switch is just a special runtime whose service chain is not configured during initialization time. Therefore, the flow actor created by the virtual switch does not need to do service chain processing and only performs virtual switching. Flow actors in the virtual switch are referred to as virtual switch actors throughout the rest of this paper.

When a virtual switch actor is created by the virtual switch, it selects one of the available runtimes as its destination in a round-rubin way. We use a round-rubin algorithm because the virtual switch must have good performance and round-rubin algorithm imposes the smallest overhead. Whenever the virtual switch actor receives an input packet, it replaces the destination MAC address of the input packet with the MAC address of the input port of the destination runtime, and modifies the source MAC address of the input packet into the MAC address of the output port of the virtual switch, it then sends the packet out from the output port.

The flow actor on the destination runtime could analyze the source MAC address of the packet and determine which virtual switch this packet comes from. This helps the flow actor to contact the virtual switch actor during flow migration and replication to change flow route **??**.

The virtual switch provides automatic load-balancing

inside a NFActor cluster as shown in figure 1, thereby improving the scalability of NFActor framework. When there is overload, the controller could launch a new runtime and the workload could be automatically balanced on that runtime. The architectural consistency of virtual switch and runtime also facilitates flow management tasks, as it enables direct communication between flow actors and virtual switch actors to dynamically change flow route **??** in a fully distributed fashion.

5.3 Controller and Control RPCs

The controller in NFActor is a light-weight one. It is responsible for monitoring the workload of each runtime, executes dynamic scaling. This is similar with NFV system controllers in E2 [28] and Stratos [19]. The most important difference of our controller is that it only needs to participate in the initiation phase of flow migration and replication. Due to the use of actor programming model, there is no need for the controller to coordinate flow migration and replication as in OpenNF [20] and Split/Merge [31].

The controller communicates with runtime and virtual switches using a series of control RPCs. The use of control RPCs decouples the execution of the runtime with the execution of the controller and decreases the complexity in developing the controller. The controller uses control RPCs to poll workload information and notifies virtual switches and runtimes about the configuration information of each other. The configuration information includes mac address of the input/output/-control port and ID of runtime/virtual switch. Virtual switches and runtimes can use the configuration information to contact each other **??**.

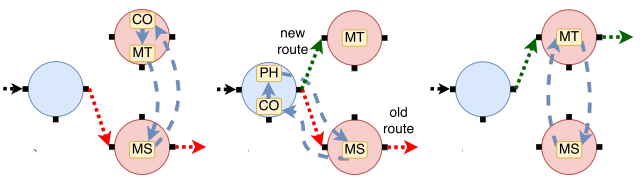
The following three RPCs enables the controller to initiate flow management tasks. The details of these RPCs are further illustrated in **??**.

- **Migrating Flows to Migration Target Runtime.** This RPC call sets up a migration target runtime for the RPC target and indicates the number of flows to migrate. When this call finishes, the RPC target should start migrating flows to the migration target.
- **Setting up Repliation Target Runtime.** This RPC call sets up a replication target runtime. When this call finishes, the RPC target should start replicating new flows to the replication target runtime.
- **Recovering Failed Runtime.** If the RPC target is the replication target runtime of the failed runtime, then flows replicated on the RPC target are directly recovered on the RPC target.

5.4 Flow Management

In NFActor, the flow management task is automatically executed by each flow actor, without the coordinator from a central controller. This feature provides good scalability when there are multiple runtimes in the cluster. Inside a NFActor cluster, each flow could do route selection by itself, therefore there is no need to rely on SDN switches and controllers. This improve the usablility and performance of NFActor framework, because SDN may not be available at all time and SDN incurs a high processing overhead when dynamically changing flow rules. Finally, the new NF modules APIs completely separate the flow state with NF processing logic. The flow actor could manipulate the flow states at any time, and the entire flow management tasks are completely transparent to the NFs. Any NF modules implemented on top of the APIs provided by NFActor could be seamlessly integrated with flow management tasks. From the perspective of NF module programmers, this feature helps them focus on the internal logic design when implementing NFs, instead of considering how to integrate their code with complicated flow management tasks. This greatly impiroves the applicability of NFActor framework. The following 2 sections give details about how flow migration and replication tasks are implemented in NFActor system.

5.4.1 Flow Migration



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

Figure 3: The three request-responses used during flow migration. (**MT**: Migration target actor. **MS**: Migration source actor. **CO**: Coordinator actor. **PH**: Flow actor on previous hop. **Dotted line**: Dataplane flow packets. **Dashed line**: Actor messages.)

As is shown in figure 3, when the current flow actor being migrated receives a migration command, it starts flow migration by executing the following three groups of request-responses.

- **First,** the current flow actor sends a request to the coordinator actor on the migration target runtime, containing the flow-5-tuple of the current flow actor. After receiving this request, the coordinator actor creates a migration target actor using the flow-5-tuple contained in the request. The migration taget actor then gives a response to the current flow actor.
- **Second,** the current flow actor sends another request to the coordinator actor of its previous hop runtime, containing the flow-5-tuple and the ID of the migration target runtime. The coordinator actor uses the flow-5-tuple to find out the flow actor

on the previous hop and notifies that flow actor to modify its output route to the migration target runtime. When the flow actor on previous hop finishes modifying the route, it gives a response back to the current flow actor. Also, after route modification, the migration target starts to receive data packets. The migration target actor buffers the data packets until the third group of request-response finishes.

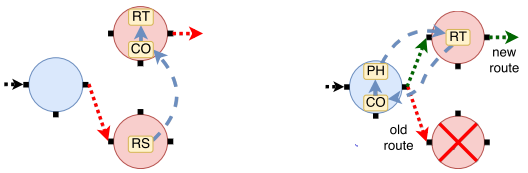
- **Third,** the flow actor sends its flow state to the migration target actor. After receiving the flow states, the migration target actor saves them, gives a response to the current flow actor and immediately start processing all the buffered packets. The current flow actor exits when receiving the response.

Lossless Migration. Even though the three request-responses are seemingly trivial, they actually achieve lossless migration as defined in OpenNF. If the three request-responses are successfully completed, the flow being migrated will not miss processing a single packet. The key reason is that when the flow actor being migrated receives the second response, it will not receive any more data plane packet sent to it anymore. This is because the second response is actually delivered by the same network path as the data plane packets. Recall that in figure 2, the remote messages could also be sent over input/output ports of a runtime. The second response is actually sent by the output port of the previous hop runtime and received by the input port of the current runtime, thereby sharing the same network path as the data plane packets. If the network does not re-order any packets (the network could indeed re-order packets, but the possibility is extremely low and there is no known method to fight agaist this kind of error) , then the current flow actor receives no more dataplane packets because the route has been changed prior to the previous hop actor sending out the second response. Therefore, no packet is missed during the migration operation.

Error Handling. The three request-responses may not always be successfully executed. In case of request failure, the current flow actor is responsible for restoring the modified route (if it happens) and resumes normal packet processing. The migration target actor is deleted after a timeout.

5.4.2 Flow Replication

The biggest difference of the replication method used by NFActor and existing works such as [33] is that NFActor framework repliates individual flow, not NF. The reason that NFActor is able to replicate at such a fine granularity is because NFActor provides a fast execution context built using actor programming model



(a) Flow replication. **(b)** Flow recover

Figure 4: Flow replication and recover. (**RT**: Replication target actor. **RS**: Replication source actor. **CO**: Coordinator actor. **PH**: Flow actor on previous hop. **Dotted line**: Dataplane flow packets. **Dashed line**: Actor messages.)

and NFActor directly stores flow states inside each flow actor.

This replication strategy improves the scalability and resource utilization rate of the system. In NFActor, the flows could be directly replicated to another runtime within the same layer, without the need for a dedicated backup server. In the mean time, this fine grained replication strategy provides a strong replication consistency as indicated in [33] with a desirable performance.

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-rubin list. If the flow actor has a valid replica runtime, whenever it finishes processing the packet, it sends a remote message, containing the flow state and the packet, to the coordinator actor on the replication target runtime. The coorinator 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 [33], the receiver on the side of the output port of the replia runtime can only observe an output packet when the flow state has been replicated, which guarantes strong replication consistency.

When a runtime fails, the controller sends recovery RPC requests to all the replica runtime of the failed runtime. This RPC enables replica flow actor to send a route modification request to the previous hop flow actor. The previous hop flow actor then changes the output route to the replica runtime and gives a response back to the replica flow actor. When this request-response finishes, the original flow is recovered on the replica runtime.

5.5 Alternative Design Option

A runtime in NFActor framework is configured with a single service chain. Each flow is handled by a unique actor, which carries out all the processing on the configured service chain. Alternative design options do exist, however, they may not fully achieve our design goal to achieve low overhead and high scalability.

There are two alternatives to the one-flow-one-actor design. First of all, using a single flow actor to process multiple flows compromise the efficiency of flow migration protocol, especially when multiple flows come from different virtual switch actors. Under this situation, the flow actor must synchronize the responses sent from different virtual switch actors, therefore decreasing the performance of migration. Secondly, chaining several flow actors together to process the same flow imposes unnecessary overhead for flow processing. Therefore, the one-flow-one-actor design achieves a sweet point in minimizing the actor processing overhead and improving the efficiency of flow migration protocol design.

The alternative design to one-runtime-one-service-chain is to dynamically configure multiple service chains on a single runtime. Then due to the one-flow-one-actor design, we need to do an additional service chain selection, based on some pre-defined rules. This adds additional overhead to the flow actor processing and increases the complexity when managing the NFActor cluster, because the controller must populates the service chain rule dynamically to each runtime. With the one-runtime-on-service-chain design, if another service chain is needed, the system administrator could launch a new NFActor cluster and configure a different service chain to use.

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 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 coordinate these tasks, 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 implement our own actor library. Due to the single-worker-thread design, when actor transmits local messages, there is no need to use a mailbox [4] [15] protected by synchronization primitives to receive the message. The local message transmission are directly implemented as a function call, therefore eliminate the overhead of enqueuing and dequeuing the message from the 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 is able to access the high-speed hash maps for storing flow-key to actor mapping and actor-id to actor mapping in the flow classifier **??**. The flow actor scheduler directly indexes the hash map using the key contained in the incoming actor messages and redirect the message to the actor. The coordinator actor is scheduled by the fourth pipeline. The coordinator actor also has accesses to the hash maps in the flow classifier, therefore it is able to forward messages to other flow actors in the runtime.

This simple actor programming could not achieve perfect message matching and complete separation of the

internal actor state, as other mature actor frameworks do [15] [4]. However, due to its simple architecture, it only imposes a small overhead when doing actor processing, therefore it is able to satisfy the high-speed packet processing requirement of modern 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.

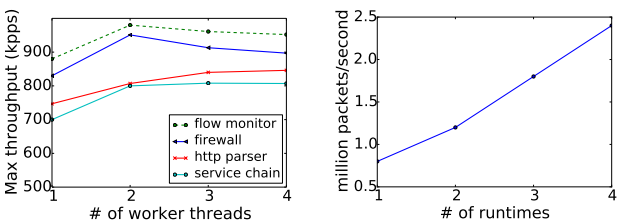
The reliable message passing module creates one ring buffer for each remote runtime. When an actor sends a remote message, the reliable transmission module allocates a packet, copy the content of the message into the packet and then enqueue the packet into the ring buffer. A message may be splitted into 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 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 [**?**] 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 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 lossless 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.

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



(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.

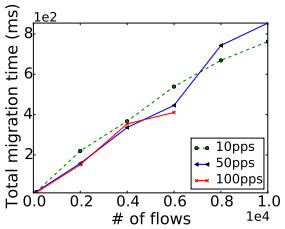
Figure 5: The performance and scalability of *NFActor* runtime, without enabling flow migration

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.

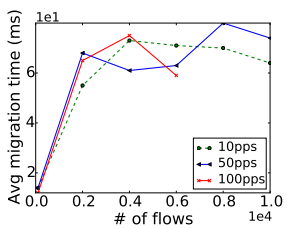
The rest of the section tries to answer the following questions. *First,* what is the packet processing capacity of *NFActor* framework? (Sec. 7.1) *Second,* how well is *NFActor* scales, both in terms of the number of worker threads used by a runtime and the number of runtimes running inside the system? (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 different number of worker threads. In figure 5b, we

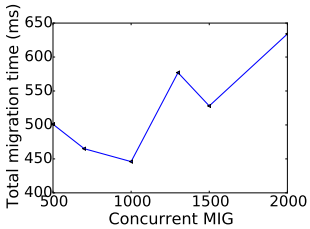


(a) The total time to migrate different numbers of flows.

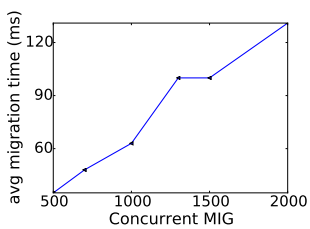


(b) The average flow migration time of a single flow when migrating different number of flows.

Figure 6: The flow migration performance of *NFActor*



(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.

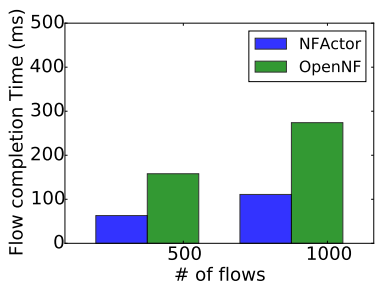


Figure 8: The flow migration performance of *NFActor*. Each flow in *NFActor* runtime goes through the service chain consisting of the 3 customzied NF modules. OpenNF controlls PRADS asset monitors.

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

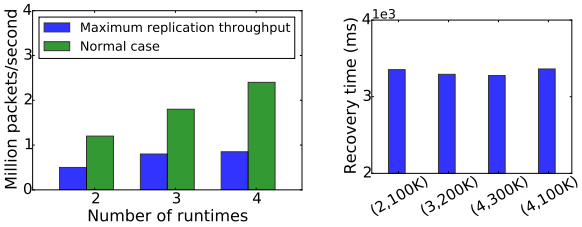
We present the evaluation result of flow migration in this section. In order to evaluate flow migration performance, we initialize the cluster with 2 runtimes running with 2 worker threads and then generate flows to one of the runtimes. Each flow is processed by the service chain consisting of all the 3 NF modules. We generate 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 [20]. We generate the same number of flows to both *NFActor* runtimes and NFs controlled by OpenNF and calculate the total time to



(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*

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. RELATED WORK

Network Function Virtualization (NFV). NFV is a new trend that advocates moving from running hardware middleboxes to running software network function instances in virtualized environment. The literature has developed a broad range of NFV applications, from scaling and controlling the NFV systems [19, 28], to improving the performance of NFV software [22, 21, 26, 29], to migrating flows among different NF instances [31, 23, 20], and to replicating NF instances [30, 33]. However, none of the above mentioned systems provide a uniform runtime platform to execute network functions. Most of the NF instances are still created as a standalone software running inside virtual machine or containers. Even though modular design introduced by ClickOS [24] simplifies the way of how NF functions are constructed, however, nowadays there are new demands for NFV system, which require advanced control functionality to be integrated even into the NF softwares.

Among the advanced control functionality, flow migration and fault tolerance are definitely the two of the most important features. Existing work such as OpenNF [20] and Split/Merge [31] requires direct modification to the core processing logic of NF softwares, which is tedious and hard to do. On the other hand, existing work rely on SDN to carry out migration protocol, thereby increasing the complexity of the migration protocol. Finally, the migration process is fully controlled by a centralized SDN controller, which may not be scalable if there are many NF instances that need flow migration service. The proposed NFActor frame-

work overcomes most of the above mentioned obstacles by providing a uniform runtime system constructed with actor framework. The actors could be migrated by themselves without the coordination from a centralized controller. The framework provides a fast virtual switch to substitute the functionality of a dedicated SDN switch. With the help of the actor framework and the customized virtual switch, the migration protocol only needs to transmit 3 request-responses. Finally, the NFActor achieves transparent migration without the need for manual modification of the NF software. This greatly simplifies the the required procedures for using migration service.

Another important control functionality lies on replication. The replication process usually involves checkpointing the entire process image and making a back-up for the created process image [33], which may halt the execution of the NF software, leading to packet losses. NFActor framework is able to check-point of the state of the flow, which is relatively lightweight to do and does not incur a high latency overhead. Similar with migration process, NF modules written using NFActor framework could be transparently replicated. Existing work like [33] rely on automated tools to extract important state variables for replicating.

Actor Programming Model. The actor programming model has been widely used to construct resilient distributed software [5, 15, 14, 4]. The actors are asynchronous entities that can receive and send messages as if they are running in a dedicated process. The actors usually run on a powerful runtime system [5, 15, 4], enabling them to achieve network transparency. It greatly simplifies programming with actor model. Even though actor programming model is widely used in both the industry and academic worlds, we have not found any related work that leverage actor programming model to construct NFV system, even though there is a natural connection among actor message processing and NF flow processing. Reliazing this problem, we are the first one to introduce actor programming model into NFV system and shows that using actor programming model can really bring benefits for designing NFV applications.

Lightweight Execution Context. There has been a study on constructing lightweight execution context [25] in kernel. In this work, the authors construct a light weight execution context by creating multiple memory mapping table in the same process. Switching among different memory tables could be viewed as switching among different lightweight execution contexts. NFActor provides a similar execution context, not for kernel processes, but for network functions. Each actor inside NFActor framework actually provides a lightweight execution context for processing a packet along a service chain. Being a lightweight context, the actors do not introduce too much overhead as we can see from the experiment session. On the other hand, packet processing is fully monitored by the execution context, thereby providing a transparent way to migrate and replicate flow states.

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] HTTP Keep Alive. https://en.wikipedia.org/wiki/HTTP_persistent_connection.
[9] Intel Data Plane Development Kit. http://dpdk.org/.
[10] iptables. https://en.wikipedia.org/wiki/Iptables.
[11] Linux Virtual Server. www.linuxvirtualserver.org/.
[12] Netmap. info.iet.unipi.it/~luigi/netmap/.
[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] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xomb: extensible open middleboxes with commodity servers. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 49–60. ACM, 2012.

[18] A. Bremler-Barr, Y. Harchol, and D. Hay. Openbox: Enabling innovation in middlebox applications. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 67–72. ACM, 2015.

[19] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual middleboxes as first-class entities. *UW-Madison TR1771*, 2012.

[20] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review*, 44(4):163–174, 2015.

[21] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[22] 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.

[23] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for nfv: simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 239–253, 2016.

[24] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[25] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight contexts: An os abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[26] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[27] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 38. ACM, 2016.

[28] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[29] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, GA, Nov. 2016. USENIX Association.

[30] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 1. ACM, 2013.

[31] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, 2013.

[32] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336, 2012.

[33] 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 *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 227–240. ACM, 2015.

[34] 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 *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization. ACM*, 2016.