NFActor: A Distributed Actor Framework for Building Resillient NFV Systems

ABSTRACT

The quick development of Network Function Virtualization (NFV) urges researchers to develop new functionalities for NFV system besides maximizing packet processing capacity. Among these new functionalities, resilience functionalities, such as flow migration and fault tolerance, are hard to tackle and yet very useful in production environment. However, implementing flow migration and fault tolerance requires manually modifying the source code of NF software and providing a control channel for message passing, which may be very tedious to implement and difficult to get right.

In this paper, we present NFActor framework, a framework for building transparently resilient NFV system using actor programming model. NFActor framework provides a set of APIs for constructing NF modules and NF modules written for NFActor framework are transparently resilient. This enables implementers to focus on the core logic design of NF modules without worrying about providing interfaces to implement resilience. Due to the use of actor framework, NFActor provides a very fast migration protocol and a lightweight flow replication protocol.

The evaluation result shows that: First, using NFActor does not incur a significant overhead when processing packet normally and NFActor framework scales well. Second, NFActor out-performs existing works on flow migration by more than 50% in flow migration completion time. Third, NFActor achieves a consistent recovery time even under increased workload.

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 [7]. With NFV, network operators no longer need to maintain complicated and costly hardware middleboxes. Instead, they can 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".

For a long period of time, middleboxes have been treated as a black box, which consume packets from ingress ports and generate output packets from egress ports. Usually, people do not concern on how packets are processed inside a middlebox. Based on this idea, most of the existing NFV management systems (i.e. E2 [24], OpenBox [11], CoMb [29], xOMB [10], Stratos [13], OpenNetVM [16, 31], ClickOS [20]) manage at middlebox level. Taking E2 [24] as an example, E2 builds a service graph to determine how the service chain are constructed and which physical server should a VNF instance be placed on. E2 also monitors the workload on each VNF instance to determine when to dynamically scale the system.

However, with the developement of NFV, researchers found out that managing at middlebox level could not satisfy the requirement of some applications. Some applications require direct management of a single network flow. A straightforwad example is flow migration. When migrating a flow, the NFV management system must transfer the state information associated with the flow from one middlebox to another, and redirecting the flow to the new middlebox in the mean time. Another example is fault tolerance of an individual flow. The NFV management system has to replicate flow's state on a replica and recovers flow's state on a new middlebox in case of the failure of the old middlebox.

There are some well known systems on managing individual network flows [14, 28, 17]. Even though these systems pave way for the future research, they have some limitations that compromise their applicability. First of all, in these systems, the flow management tasks are initiated from a central SDN controller. This architecture limits the scalability of the system. When the number of VNF instance and the traffic volume increase, this central SDN controller inevitably becomes the bottleneck in the system. Secondly, existing systems do not provide a uniform exeuction context for managing individual flow. Additional patch codes must be added to the middlebox software when using these systems, to acquire the state associated with the flow and to communicate with the centralized controller This makes adapting these systems tedious and hard. Finally, the communication channel, which is heavily used by these systems to transmit flow states, are not optimized for high speed NFV application. It is still based on the traditional kernel networking stack, which has been proved to be a performance bottleneck [20], thereby limiting the maximum packet throughput these systems can achieve.

Reliazing these limitations, we propose a new NFV management system in this paper, called NFActor. NFActor provides a distributed runtime environment, which could be controlled by a light-weight controller. Inside a runtime, we use actor programming model [1] to construct a uniform execution context for each network flow. The execution context is augmented with different kinds of message handlers for managing flow migration and fault tolerance. In the mean time, we provide a new interface for programming new NFs. This interface simply separates the core NF processing logic with the state of each flow. Finally, we make a simple yet efficient reliable transmission modle using the high-speed packet I/O functionality provided by DPDK [5]. This reliable tranmission module is used to pass all the messages during remote actor communication. All these parts are scheduled by a simple round-rubin scheduler inside the runtime.

The result of this architecture is the complete decoupling of flow management tasks from a centralized controller. Using its own execution context, each flow could migrate or replicate itself, without the coordination from a centralized controller. Even though new NF must be written specifically for NFActor architecture, it is not considered harmful [25]. The goold news is that programmers who write new NFs for NFActor only need to concentrate on the NF logic design. Once the NF is completed, it will be spontanesouly integrated with the flow execution context. The abstraction of flow execution context only incurs a small overhead when processing packet. Our evaluation results show that NFActor could achieve desirable packet throughput. The performance of flow migration and fault tolerance is also satasfactory according to the standard of modern high-performance NFV systems.

2. BACKGROUND

2.1 Network Function Virtualization

A NFV system [7] typically consists of a controller and many NF instances. Each NF instance is a virtualized device running NF software,which constantly fetches packets from an input port, processes the packets and then sends processed packets to the output port. NF 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 NF instance constantly polls a network interface card for packets. Using traditional kernel network stack incurs high context switching overhead [20]. To speed up packet processing, hypervisors usually map the memory holding packet buffers directly into the address space of the guest with the help of Intel DPDK[5] or netmap [6]. Guest can directly fetch packets from the mapped memory area, avoiding expensive context switches. Recent NFV systems [24, 15, 30, 20, 16] are built using similar techniques.

2.2 Actor Model

The actor programming model has been used as the basis for constructing massive, distributed systems[1, 9, 22] 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. The behavior of an actor is fully non-blocking and there is no need for actors to contend for a lock due to their message-passing nature.

There are several popular actor frameworks, *i.e.*, Scala Akka [9], Erlang [4], Orleans [8] and C++ Actor Framework [2]. 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 [9]. There has been no attempt to build NFV systems using the actor model. theirtheirtheirtheir

3. THE *NFActor* FRAMEWORK

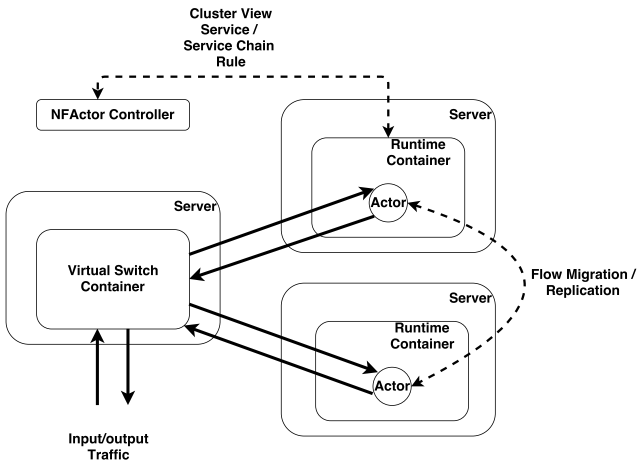


Figure 1: The architecture of *NFActor*

As illustrated in Fig. 1, *NFActor* consists of 3 critical components, a global controller, a global virtual switch and several runtime systems (referred to as *runtime* in short). The controller and the virtual switch are logical modules, which can be physically implemented on one or multiple servers or containers according to the workload. In our design, both the virtual switch and runtimes run inside containers, so that they can be quickly rebooted in case of failure, and elastically scaled in case of overload or underload.

In *NFActor*, incoming traffic flows are first sent to the virtual switch, which dispatches them to runtimes in a load-balanced fashion, by querying the load information collected by the global controller. The runtimes host NF service chains, and process the flows. When a runtime system receives a new flow, it constructs a new actor and delegates packet processing of the flow to that actor. The actor queries what NFs the flow should go through by looking for a matching service chain rule. Then the actor loads all the required NF modules to compose the service chain and passes the flow to the NF modules in sequence, before sending the processed flow back to the virtual switch. Finally, the virtual switch sends all the output flows to the respective destinations.

3.1 Controller

Similar with existing work [14, 24, 30], *NFActor* builds a central controller to manage the cluster of virtual switch and runtime systems. Nevertheless, the functionalities provided by this controller are all light-weight ones, including a view service, installing service chain rules and elastic scaling. Due to exploiting the actor model, *NFActor* controller is not directly involved in flow migration and fault tolerance. In contrast, the controllers in existing work are directly involved in supporting failure resilience (if the systems provide the support), by either initiating the flow migration procedure for a flow [14], or preparing replicas for NF instances [30].

The View Service identifies and maintains the states and loads of runtimes, using a heartbeat mechanism. Each runtime in *NFActor* regularly sends heartbeat messages to the controller, containing the current load information (*i.e.* CPU usage, memory usage) of the runtime. The controller gathers the information contained in the heartbeat messages into a cluster view list, which records the contact address, state (*running*, *leaving*, *fail*) and load information of all runtimes. The controller broadcasts the cluster view list to each runtime and the virtual switch if there are any state/load changes, so that runtimes and virtual switch have a local copy of this list. The views at different runtime do not have to be consistent because flow migration and fault tolerance protocol perform safety checking by exchanging requests and responses to eliminate the inconsistency.

This view service enables the following: (1) A base for achieving distributed flow migration and fault tolerance. 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 replication. (2) Facilitating load balancing, as the virtual switch uses the view service to distribute flows to different runtimes. (3) Assisting the controller in making runtime scaling decisions, with the load information it provides.

Providing Service Chain Rules. In *NFActor*, the service chain of NFs is dynamically constructed inside the execution context of an actor, depending on the flow it is allocated to process. A service chain rule is analogous to an OpenFlow rule [21], installed on the controller to map flows to service chains on the go, to support dynamic service chain construction. It consists of a pattern to match packet headers and a service chain configuration to specify what NFs the matched packets should go through in order. A runtime uses service chain rule to determine what service chain a flow should use.

In *NFActor*, service chain rules are pre-configured at the controller and pushed to each runtime upon its startup. Pattern matching is implemented as longest prefix matching over the traditional 5-tuple (source IP, destination IP, application protocol type, source port and destination port) of a flow, while the service chain configuration is encoded using a 64-bit integer and every 4 bits of this integer represent a unique NF type.

Elastic scaling is triggered by analyzing the load information contained in the heartbeat messages. The controller creates new runtime systems when existing runtimes are overloaded (scale-out), or terminates idle runtimes (scale-in). Overload and idling of runtimes are detected according to thresholds on CPU usage.

Once a new runtime system is created, an overloaded runtime can migrate some of its flows to the new runtime until the overload is resolved (add reference). On the other hand, new flows entering *NFActor* tend to choose the new runtime as the target to put replicas (see Sec. 4). When the controller decides to shut down a runtime, it first turns its state into *leaving*. Then this runtime starts migrating all its flows to other runtimes and waits for all of its replicas to expire (Sec. 4). In the meantime, the virtual switch does not route new flows to the *leaving* runtime. The controller shuts down the *leaving* runtime when it becomes completely idle and removes it from the cluster view list.

3.2 Virtual Switch

NFActor uses a virtual switch as a gateway for flow dispatching. Previous work either rely on SDN switches [13, 14] to route the traffic, or build a customized dataplane for inter-connecting different NF instances [24].

Our virtual switch design results from the following considerations.

First, *NFActor* uses uniform runtime systems to process incoming flows and achieves good horizontal scalability. To fully unleash the scalability of *NFActor*, we must have a load-balancer to balance the load among different runtimes. Together with the view service, the virtual switch can easily balance the load. *Second*, we can build a customized communication module into the virtual switch, which enables us to design a simple distributed flow migration protocol, bypassing the overhead of communicating with a centralized SDN controller (see Sec. 5 for details). *Third*, the virtual switch can be efficiently implemented using an efficient hash table such as cuckoo hashing [23], and can be implemented in a scalable fashion, so that it will not render a bottleneck in the system.

The virtual switch maintains a switching hash table, where the key is the hash result of the 5-tuple of a packet, and the value is the MAC address of the selected runtime for handling the flow. Whenever the first packet of a new flow arrives, the virtual switch selects a runtime in the *running* state from the view service using a round-rubin algorithm. We choose a simple round-rubin algorithm because the virtual switch must run very fast and round-rubin algorithm introduces the smallest amount of overhead while providing satisfactory performance. , adds a key-value pair in the hash table, and forwards the packet to the selected runtime by replacing its destination MAC address by the MAC address of the runtime. Later packets from the same flow still needs to check the hash table. However, this overhead is greatly reduced by using a fast hash table like Cuckoo hash table [32].

3.3 Runtime

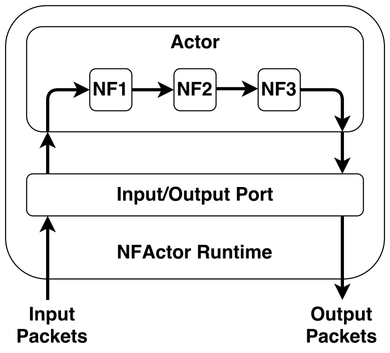


Figure 2: The architecture of a *NFActor* runtime system

The concept of a uniform runtime system (Fig. 2), as a basic flow processing and scaling unit in *NFActor*, does not appear in most existing work [11, 13, 24]. 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 a uniform runtime is to enable NF modules to achieve resilience automatically in the *NFActor* framework without manual intervention. As the uniform runtime system provides a network transparent abstraction for actors to communicate and exchange messages that are crucial to migration and replication. Especially, in a runtime system, 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 resilience on the basis of each micro execution context (Sec. 4). To be able to process multiple flows, the runtime system should be capable of handling multiple micro execution contexts concurrently. This is handled using a similar hash-table based flow classifier as mentioned in Sec. 3.2.

In *NFActor*, we exploit the actor programming model to implement the micro execution context. Each micro execution context is an actor. Flow processing by NFs in the service chain, flow migration and replication functionalities are all implemented as message handlers of the actor. The runtime system provides the basic runtime environment for all the actors it has created. In particular, it creates a new actor when a new flow is sent to it. The actor looks for a matching service chain rule for the flow and loads the corresponding NF modules specified in the rule. Then the actor processes the flow by passing the packet through each NF module in sequence, as well as performs flow migration and states replication in response to certain messages (Sec. 4 and Sec. 5).

Besides built-in resilience, additional benefits are brought by our uniform runtime system design. *First*, it simplifies service chain management. In previous work such as [24], the controller must create several different service chains on the data-plane. In *NFActor*, service chains are dynamically created inside each runtime in a very lightweight fashion without directly involving the controller. *Next*, with the help of a load-balancing virtual switch, our framework can easily scale out by adding more runtimes. In previous work [24], scaling-out involves modifying data-plane network paths, which is not trivial.

Each runtime can host one or multiple actors, 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 actors are running on the same runtime, the actors are scheduled to run on a worker thread whenever a message is sent to the actor's mailbox. The actor In addition, our design of the NF modules in the following section will show that passing pack-

```
class flow_state{
};

class network_function{
public:
  virtual void init() = 0;
  virtual flow_state* allocate_flow_state() = 0;
  virtual bool process_packet(rte_mbuf* pkt, flow_state* fs) = 0;
  virtual serialized_obj serialize_flow_state(flow_state* fs) = 0;
  virtual flow_state* deserialize_flow_state(serialized_obj* obj) = 0;
};
```

(a) An API for creating new NF module.

```
class state : public flow_state{
public:
  state():pkt_counter(0){};
  serialized_obj serialize() override{
    return serialized_obj(pkt_counter);
  }
  int pkt_counter;
};

class pkt_counter : public network_function{
public:
  void init() {};
  flow_state* allocate_flow_state() override{
    return new state();
  }
  bool process_packet(rte_mbuf* pkt, flow_state* fs) override{
    state* s = dynamic_cast<state*>(fs);
    if(!s){
      rte_pktmbuf_free(pkt);
      return false;
    }
    s->pkt_counter+=1;
    return true;
  }
};
```

(b) A example NF module created using the API.

Figure 3: The API and implementation of an example NF module.

ets to a NF for processing in an actor is essentially just a function call; only one copy of each NF module software needs to be loaded in a runtime, while multiple actors hosting service chains involving the NF can use it. The actor itself is a very lightweight one as millions of actors could be spawned in seconds [12]. To further reduce overhead of loading NF modules in the runtimes, the virtual switch can direct flows requiring a similar set of NFs to the same runtimes.

3.4 An API for Creating NF Module

With the micro execution context enabled by the runtime and the actor model, we need one last step towards achieving built-in resilience, which is to separate important NF states from the core processing logic of the NF module. With this separation, the actor can retrieve and serialize NF states for transmission whenever needed, without interfering with processing logic execution of the NF module. In *NFActor*, we achieve this separation by designing an API for implementing new NF modules in the actor model.

Fig. 3a illustrates this API. To implement a new NF class, programmer must use `network_function` as the base class and implement all 5 virtual methods. The `init` method is used by the runtime to initialize the NF module upon startup. When a new actor is created to handle a new flow, it first calls the `allocate_flow_state` method to create a local flow state object. Whenever the actor receives a new packet, the actor passes the received packet (represented as `rte_mbuf`) and the flow state object to the `process_packet` method for NF processing. The return value of `allocate_flow_state` indicates whether the packet should continue to be processed on the service chain. During flow migration and replication, the actor uses the last 2 methods to serialize flow state object for transmission and de-serialize `serialized_obj` to recover the flow states.

This API design turns a NF module into a state machine: NF processing is to feed the current event (input packet) and current state (flow state) into the state machine (NF module), generating a new state (updated flow state) and a new action (processed packet). This API design enforces the separation between flow states and processing logic of the NF, so that the actor can easily and correctly retrieve flow state for transmission during flow migration and replication.

4. FAULT TOLERANCE

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

4.1 Replicating Controller

Since the controller is a single-threaded module that mainly collects the states of the runtimes, we persistently log these states and replicate them. The controller only needs to log the state of each runtime in the cluster view list. Whenever the controller needs to modify the state of a runtime, it logs the intended operation, modifies the state and logs a success mark for the intended operation.

The liveness of the controller is monitored by a guard process and the controller is restarted immediately in case of failure. On a reboot, the controller reconstructs the states in the cluster view list by replaying logs. Each runtime in the cluster monitors the connection status with the controller and reconnects to the controller in case of a connection failure.

4.2 Replicating Virtual Switch

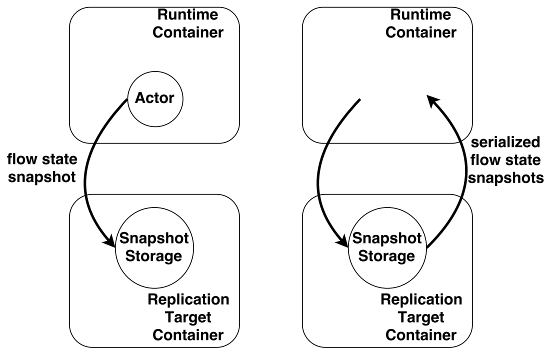


Figure 4: Replication process used by for *NFActor* runtime. The left side figure illustrates how actor stores its flow state snapshots to the snapshot storage. The right side figure illustrates how recovered runtime fetches the snapshot storage from another runtime.

The most important state of the virtual switch process is its switching hash table in memory. In order to replicate the virtual switch for failure resilience, we constantly check-point the container memory image of the virtual switch using CRIU [3], a popular tool for checkpointing/restoring Linux processes. One main technical challenge is that CRIU has to stop a process before checkpointing it, which may hurt the availability of the virtual switch.

We tackle this challenge by letting the virtual switch call a fork() periodically (by default, one minute), and then we use CRIU to checkpoint the child process. Therefore, the virtual switch can proceed without affecting the system performance.

4.3 Replicating Runtime

To perform lightweight runtime replication, we leverage the actor abstraction and state separation to create a lightweight flow state replication strategy. In a runtime, important flow states associated with a flow is owned by a unique actor. The runtime can replicate each actor independently without incurring the overhead of check-pointing the entire container images [30, 27]. In *NFActor*, each actor replicates its state by choosing another runtime to store its flow state snapshots. This replication strategy avoids the need for using dedicated back-up servers [30] and achieves very good scalability, as newly created runtimes after scaling-out could also be used to store flow state snapshots.

This primary-backup replication approach can tolerate the failure of one runtime (between runtimes that the actor and its replica are residing in. We think this fault-tolerance guarantee is sufficient because the chance for both runtimes (usually on two server machines) failing at the same time is extremely low.

Finding a Replication Target: When an actor is created, it selects a runtime in the *running* state with the smallest workload as its replication target. Then the actor negotiates with the replication target about whether the replication target can accept this actor's flow state snapshot. In case that replication target refuses to store the actor's state snapshot, the actor tries to select another replication target.

Snapshot Storage: Each runtime maintains several snapshot storage for each runtime in the cluster except itself. The snapshot storage stores all the flow state snapshots sent from the same runtime. Each snapshot storage is managed by an independent thread for fast storing and retrieving.

Flow State Replication (Fig. 4): After determining the replication target, the actor performs flow state replication. For every fixed number of packets that the actor has processed, the actor creates a snapshot of the flow states of all NF modules and sends the snapshot to the snapshot storage on the actor's replication target. Note that the flow state replication is independent with NF processing and it ensures built-in fault tolerance. The snapshot storage keeps saving the newly received snapshot and discarding the old one. Note that this replication strategy only ensures weak consistency because once the runtime fails, the actors on the failed runtime can only be recovered to its old state. For strong consistency, NFActor could be easily extended to a similar framework as in [30]. However, as we can see from Sec. 7.3, even using replication with weak consistency imposes a significant overhead on the performance of the NFActor runtimes.

Recovering Failed Runtime (Fig. 4): In case that a runtime fails, the controller will immediately detect the failure and reboot the failed runtime. The restarted runtime first performs recovery process by sending a recovery message to every other runtime in the cluster, asking for content of the snapshot storage of the recovered runtime. The recovered runtime then uses these snapshot storage to reconstruct its flow states before failure. The reconstruction process could be paralleled on different threads because each snapshot storage could be used independently. When the runtime finishes recovery, it re-joins the cluster and resumes normal flow processing.

5. DISTRIBUTED FLOW MIGRATION

In this section, we present a lightweight, distributed flow migration protocol for *NFActor*, designed to circumvent inefficiencies observed for flow migration in exising NFV systems.

5.1 Main Idea

In most existing work [14, 28], flow migration is tightly coupled with implementation of the NF software. To migrate flows from a NF, not only large amounts of patch code needs to be added for extracting and transmitting NF states, but also the centralized controller is heavily involved, leading to a scalability issue. Migration of a specific flow has to be initiated by the controller [14]; during the migration process, the controller has to exchange messages with migration source, migration target and the SDN switch.

We build the flow migration functionalities as message handlers of the actors, and provide flow migration as a basic operation in *NFActor*. Flow migration is transparent to implementation of the NF module, eliminating the need of patch code. On the other hand, flow migration is initiated by the actor that processes the flow, and only involves 3 passed of request-response messages in sequence, making the entire process lightweight and scalable.

Based on the actor model, flow migration can be regarded as a transaction between a source actor and a target actor, where the source actor delivers its entire state and processing tasks to the target actor. Flow migration is successful once the target actor has completely taken over packet processing of the flow. In case of unsuccessful flow migration, the source actor can fallback to regular packet processing and instruct to destroy the target actor.

5.2 Distributed Flow Migration Protocol

We next present the details of our flow migration procedure.

Initiate Flow Migration. In *NFActor*, flow migration is primarily used to resolve hot spot (overloaded runtimes), or shut down idle runtimes. Each runtime keeps monitoring its CPU and memory usage. If thresholds on resource consumption are exceeded (Sec. 6), the runtime starts migrating flows to other runtimes with a smaller load. The runtime keeps a local copy of the workload of other runtimes through the view service. Whenever runtime would like to migrate a actor, it selects a target runtime with a smaller workload and notifies the actor about the target runtimeIf the controller detects an idle runtime, it will turn the state of the idle runtime into "*leaving*". Then the idle runtime starts migrating all its flows to other runtimes before the controller shuts it down. Idle runtime rejects all the migration requests from other actors to keep other actors from migrating to it.

To notify a flow to migrate to another runtime (the migration target runtime system), the current runtime sends the ID of the migration target runtime to the actor handling the flow. Then the actor starts migrating the flow by itself, using the flow migration steps described below.

Create Target Actor (Fig. 5a): The source actor sends a 'create target actor' message to the target runtime. This message also contains the flow identifier of the actor handling the flow to be migrated (*i.e.*, source actor). Upon receiving the message, the target runtime creates a target actor, configures the NF modules, and registers the target actor with the flow identifier, so that the target actor can correctly receive forwarded packets of the flow from the virtual switch. The target runtime sends an 'ok' message back to the source actor.

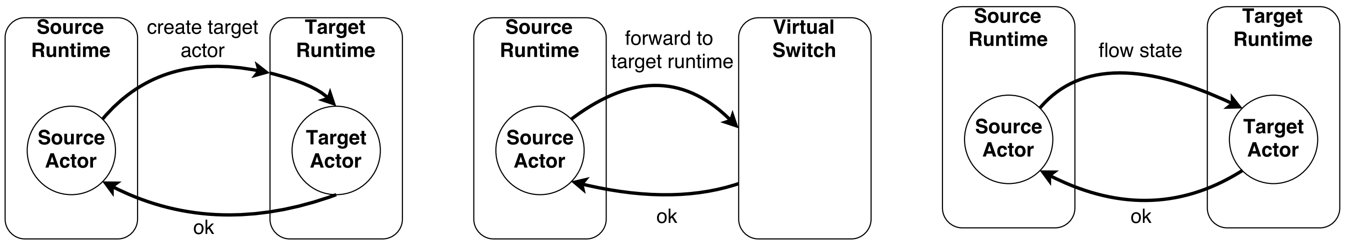
Contact Virtual Switch (Fig. 5b): After finishing the first pass of request-response, the source actor then sends a 'forward to target runtime' message to the virtual switch, carrying the flow identifier of the source actor and the ID of the target runtime. After receiving this message, the virtual switch updates its switching hash table by changing the MAC address associated with the flow 5-tuple to the MAC address of the target runtime. Then the virtual switch sends an 'ok' message back to the source actor. Instead of being a control plane message, this 'ok' message is carried in a data plane packet with the same flow identifier (that the source actor is handling), whose content is a global unique magic number . We use a unique magic number to prevent the content of the flow packet from interfering the migration protocol. When the source actor receives this data plane response packet, it knows that there are no more data-plane packets of the flow coming to it and it can safely proceed to the final pass of request-response. The use of a data plane response packet ensures lossless flow migration [14] without incurring more message passing overhead . The data plane response is sent after the virtual switch updates its hash table. Because data plane packet comes in order (re-order may happen, but rarely), whenever the migration source actor receives the data plane response, it can ensure that it will not receive any more data plane packets so that it can safely continue without missing any data plane packet.

After the hash table update at the virtual switch, the packets of the flow are now forwarded to the target runtime, which dispatches them to the target actor. The target actor buffers the received flow packets without actually processing it, until the flow migration is completed.

Migrate Flow States (Fig. 5c): The source actor serializes all the flow states using the API provided by each NF module (Fig. 3a). Then the source actor sends the serialized flow states to the target actor. After receiving the serialized flow states, the target actor immediately sends back an 'ok' message to indicate migration success. Then it processes all the buffered packets and resumes normal flow packet processing. After receiving the 'ok' message from the target actor, the source actor notifies the source runtime about the successful migration, performs cleanups and quits.

5.3 Controlling the Maximum Number of Concurrent Migrations

The runtime controls the maximum number of con-



(a) Create target actor **(b)** Contact virtual switch **(c)** Migrate flow states

Figure 5: Distributed Flow Migration Protocol

current migrations that is allowed to perform in the system. This is because too many concurrent migrations may quickly overloads the migration target runtime as the traffic is now rescheduled to the migration target runtime. The runtime thus closely monitors the number of concurrent migrations. If the number surpasses a threshold, the runtime stops any further migrations.

5.4 Failure Handling

Failures may happen during the execution of the above flow migration steps. Before the source actor receives the final acknowledgement from the target actor in the last step (Fig. 5c), any failure will terminate the migration process and flow processing on the source actor should be properly resumed. We next discuss how the flow migration protocol handles failure.

Messages lost in the network. In each of the three steps (Fig. 5a to 5c), if either the request message or the response message is lost, the source actor will be interrupted by a timeout and will terminate its flow migration process. This further leads to a timeout on the target actor, which will terminate the target actor. Step 2 (Fig. 5b) and step 3 (Fig. 5c) involve changing packet forwarding path and migrating flow states. Therefore, before terminating the migration process, the source actor also sends a request to the virtual switch to change the forwarding path back to itself.

Runtime failure or virtual switch failure. During the migration process, the source (target) runtime keeps monitoring the liveness of the target (source) runtime and the virtual switch. In case that the target runtime or the virtual switch fails, the source actor receives a notification from its runtime, immediately terminates the migration process and sends a request to the virtual switch (after its recovery if it fails) to change the forwarding path back to itself. In case that the source runtime fails, the target actor sends the request to the virtual switch to change the forwarding path back to the source actor (which will be recovered by the fault tolerance mechanism). Since the migration process is not logged, after the source runtime and the source actor are recovered, the source actor resumes processing the flow without knowing its previous migration attempt.

In any case of a migration failure, the source runtime will select another migration target runtime for the flow, and run the flow migration protocol again.

6. IMPLEMENTATION

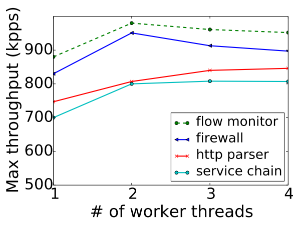
The implementation of the core functionalities of NFActor framework consists of 9921 lines of C/C++ code, excluding the implementation of 3 customized NF modules and miscellaneous helper codes. In NFActor, each runtime is containerized using Docker. The data plane of NFActor is inter-connected using BESS [15], which is a virtual switch for implementing high performance NFV system. The control plane of NFActor is inter-connected using OpenVSwitch [26]. The actor runtime is implemented using libcaf [2], which is a C++ actor programming framework.

The internal implementation of NFActor runtime is separated into 2 parts, which are a packet polling thread and several actor worker threads. The packet polling thread polls the input queue created by the BESS for packets and fetches the packets directly from the huge page memory area [5]. Then the packet polling loop sends the packet to a actor as an actor message. All the actors are scheduled to run on the worker threads. When the actor gets it's schedule to run, it processes as many received messages as possible. When the actor finishes processing a packet, it sends the packet back to the packet polling loop through a lockless multi-producer queue. The packet polling loop in turn sends the packet to the outside world.

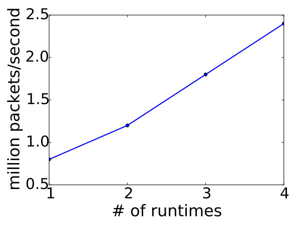
7. EVALUATION

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

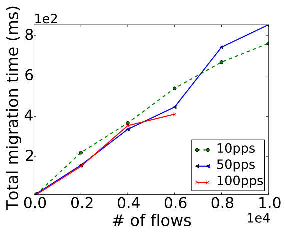


(a) Packet processing capacity of a single *NFActor* runtime system running with different number of worker threads.

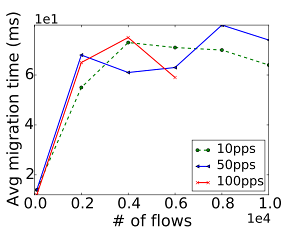


(b) Aggregate packet processing capacity of several *NFActor* runtimes.

Figure 6: 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 7: The flow migration performance of *NFActor*

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

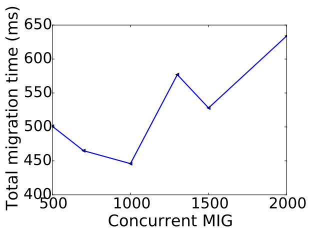
Figure 6 illustrates the normal case performance of running *NFActor* framework. Each flow in the generated traffic has a 10 pps (packet per second) per-flow packet rate. We vary the number of concurrently generated flows to produce varying input traffics. In this evaluation, we gradually increase the 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 6a, 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 6b, 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 6a, 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.

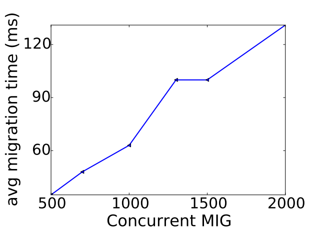
7.2 Flow Migration Performance

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 8.

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



(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 8: The flow migration performance of *NFActor* when changing the maximum concurrent migrations.

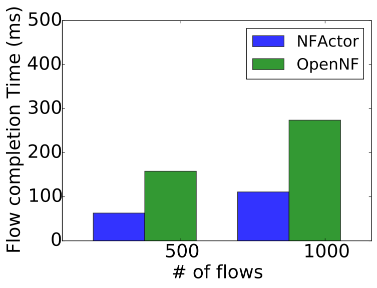


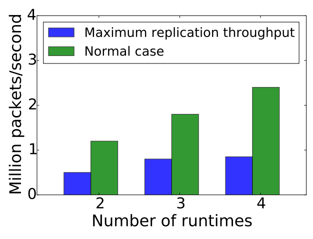
Figure 9: The flow migration performance of *NFActor*. Each flow in *NFActor* runtime goes through the service chain consisting of the 3 customzied NF modules. OpenNF controlls PRADS asset monitors.

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.

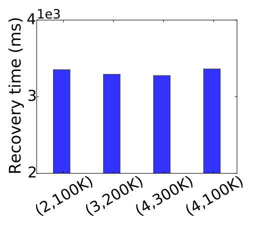
When the per-flow packet rate is 100pps, the maximum number of flows that we use to evaluate the system is 6000. Continuing the evaluation with 8000 and 10000 flows just overloads the runtime as shown in figure 6a.

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 8. As we can see from fig 8b, 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 7b, 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 [14]. 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 10: The flow migration performance of *NFActor*

migrate these flows. The evaluation result is shown in figure 9. 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 8a. 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 10.

In figure 10a, 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 10b 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 [13, 24], to improving the performance of NFV software [16, 15, 20, 25], to migrating flows among different NF instances [28, 17, 14], and to replicating NF instances [27, 30]. 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 [18] 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 [14] and Split/Merge [28] 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 framework 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 check-pointing the entire process image and making a back-up for the created process image [30], 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 [30] 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 [4, 9, 8, 2]. 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 [4, 9, 2], 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 [19] 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] C++ Actor Framework. `http://actor-framework.org/`.

[3] CRIU. `https://criu.org/Main_Page`.

[4] Erlang. `https://www.erlang.org/`.

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

[6] Netmap. `info.iet.unipi.it/~luigi/netmap/`.

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

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

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

[10] 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.

[11] 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.

[12] D. Charousset, R. Hiesgen, and T. C. Schmidt. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, 45:105–131, April 2016.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[22] 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.

[23] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[24] 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.

[25] 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.

[26] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of open vswitch. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2013.