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 [6]. 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 [20], OpenBox [10], CoMb [25], xOMB [9], Stratos [11], OpenNetVM [14, 27], ClickOS [18]) manage at middlebox level. Taking E2 [20] 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 [12, 24, 15]. 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 [18], 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 [4]. 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 [21]. 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 [6] 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 [18] 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[4] or netmap [5]. VNF instances then directly fetch packets from the mapped memory area, avoiding expensive context switches. Recent NFV systems [20, 13, 26, 18, 14] 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 systems[1, 8, 19]. 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 [8], Erlang [3], Orleans [7] 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 [8].

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 [2] 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. DESIGN

Figure 1 and 2 daemonstrate the basic architecture of NFActor framework. NFActor framework composes uniform runtime systems into a cluster. Within this cluster, fine-grained flow managmeent tasks could be quickly executed. This cluster is controlled by a lightweight controller for dynamic scaling and inititiating flow management tasks, including flow migration and fault tolerance.

The design of NFActor framework follows the following principles.

- **Low Overhead.** While providing compliated flow management tasks, the runtime system of NFActor framework must be able to process packets at high speed. Therefore, the execution context that NFActor created for each flow must be a lightweight abstraction, it should not compromise the processing speed of a NF. In NFActor, this execution context is constructed using actor programming model. We implement our own actor programming model to minimize the overhead associated with the execution context.

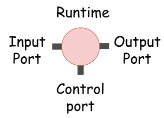
- **Efficiency.** To accomodate the need of high speed NFV systems, the flow management tasks of NFActor must be highlighy efficient. In high speed NFV systems, a NF may process millions of packets every second and handle tens of thousands of flows. To achieve high efficiency, NFActor completely abandoned using kernel networking stack to avoid the overhead of context switching. In NFActor, all the data, whethere it's data plane packet or remote actor messages, are transmitted through high-speed packet I/O (i.e. DPDK).

- **Scalability.** Modern NFV system must have good scalability, to accomodate varying network traffic. To provide good scalability, NFActor framework provides uniforms runtimes and connects multiple runtimes into a NFActor cluster. Within this cluster, the runtime system could manage output route for each flow that passes through it, to achieve dynamic load balancing. The uniform runtime design also faciliatates message passing among different runtimes, to achieve efficient flow management tasks.

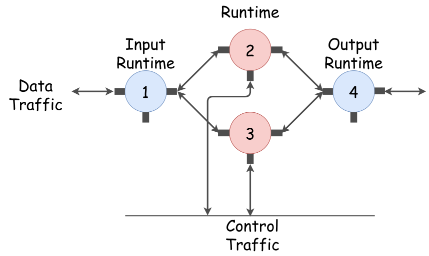
3.1 Runtime Cluster

Figure 1c gives an example of a running NFActor cluster. The NFActor cluster consists of multiple runtimes controlled by a light-weight controller through RPC.

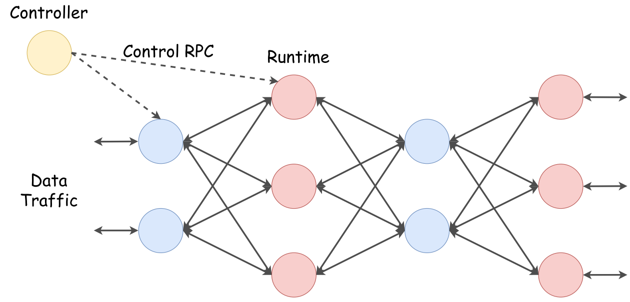
Figure 1a shows the basic struture of a runtime. A runtime consists of three ports. The input and output ports are used to receive and send dataplane packets. The control port is only used to transmit remote messages when flow actor executes flow management tasks. Both input and output ports could also be used to transmit remote messages, this is further illustrated in later chapters.



(a) A runtime with three ports.



(b) A minimal runtime connection.



(c) A NFActor runtime cluster consists of multiple lay, controlled by a controller.

Figure 1: The flow migration performance of *NFActor*

A runtime system could not execute any load-balacning and flow management tasks by itself. It needs to be connected with other runtimes and collaborate with those runtimes. The three ports of a single runtime system could be connected to multiple runtimes. Figure 1b gives a minimal runtime connection that is able to achieve load-balancing and flow management tasks. In figure 1b, the input and output ports of runtime 2 and 3 are connected with the output port of runtime 1 and input port of runtime 4. From the perspective of runtime 2, runtime 1 is its input runtime and runtime 4 is its output runtime. Similarly, runtime 2 is the output runtime of runtime 1. In NFActor framework, a runtime could balance its workload among all of its output runtimes. This is why the dataplane traffic could enter from one end of the connection in figure 1b and exit from the other end.

From the perspective of runtime 2 and 3, they share the same input runtimes and output runtimes. These runtimes are classified into the same layer. The control ports of runtimes under the same layer are directly connected, so that flows could be quickly migrated and replicated among runtimes under the same layer.

As shown in figure 1c, we can construct a NFActor cluster by creating multiple layers of runtimes. This runtime cluster could be controlled by a controller, which monitors the workload on each runtime for dynamic scaling. This controller could also initiate flow management tasks by sending RPC requests to selected runtimes.

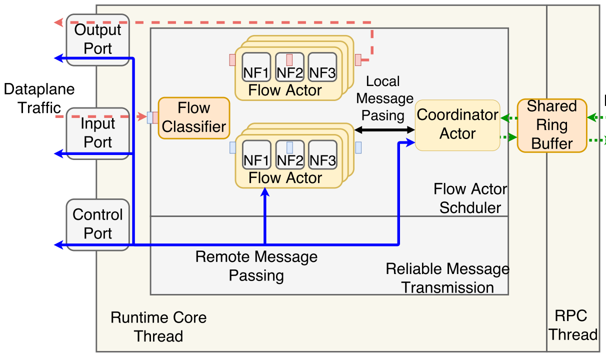


Figure 2: The internal architecture of a NFActor runtime system.

3.2 Runtime Architecture

Figure 2 demonstrates the internal architecture of a runtime, which consists of an RPC thread and a core thread. The RPC thread receives RPC requests from the controller and forward these requests to the core thread using a shared ring buffer. The core thread has a flow actor scheduler module and a reliable message transmission module. Both of these two modules are scheduled to run by a round-rubin module scheduler. The core thread employs a run-to-completion scheduling model. The core thread keeps processing the input packet keeps processing until it is either dropped, or sent out from the output port.

The core thread polls dataplane pacekts from the input port and forwards them to flow actor scheduler. The flow actor scheduler uses a flow classifier to classify packets into different flows. In our current implementation, the flow classifier uses tranditional flow-5-tuple (i.e. IP destination address, IP source address, transmission layer protocol, source port and destination port) to classify flows. Flow classifier creates an unique flow actor for each new flow, and forwards all the packets of a flow to its flow actor. The flow actor processes the packet by passing the packet through a pre-determined service chain at runtime initialization time. When the packet finishes processing, it is delivered to the output port. The flow actor scheduler also peoriodically schedules a coordinator actor, which is responsible for fetching RPC requests from the shared ring buffer and executes these requests.

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. In NFActor framework, we assign a unique ID for each runtime and each actor. When an actor sends a remote message, it only needs to specify the actor ID of the receiver and the runtime ID of the receiver. The message is then delivered reliably to remote side by the reliable message passing module.

The reason that a runtime has only one core thread is to improve resource utilization rate and minimize the overhead of using actor programming model. In our initial prototype implementation, we used LIBCAF [2] library to construct flow actors. LIBCAF library creates multiple worker threads and schedules flow actors to run onthese worker threads. In order to poll from the input port and send packet to flow actor, we have to create another polling thread. Under this design, we found out that the maximum throughput of a single runtime does not increase when the number of LIBCAF worker thread increases, because the polling thread has always been a bottleneck. Therefore, we abondan the multi-worker-thread design and use a single core thread to poll packets and schedule flow actors. This architecture 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. We show in evaluation section that this architecture could achieve satisfactory performance.

3.3 Control RPCs

The runtime exposes a serise of RPCs for controller to use. A controller could use these RPCs to control the connection between different runtimes. In terms of flow management tasks, the controller only needs to participate in the initiation phase using the exposed RPCs. There's no need for the controller to get involved in the execution of flow management tasks, because flow actors could complete them by themselves. The controller using these RPCs could be designed as a light-weight one, this is a key difference that distinguish NFActor framework from previous work, where controller must be involved in each flow management task, thereby limiting the scalability of the system. These RPCs could be divided into the following categories.

- **Adding Input(Output) Runtime.** This RPC call adds input(ouput) runtime to the RPC target(the runtime being called). Before calling this RPC, the controller must guarantee that the input(output) port of the RPC target is connected with the output(input) port of the input(output) runtime.

- **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. The migration target must stay in the same layer as the RPC 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.

The replication target must stay in the same layer as the RPC target.

- **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.

3.4 NF Module

Each NF used by NFActor is implemented as a loadable module. The runtime system could select which NF module to load and use. In order to achieve the service chain processing as indicated in figure 2, we pass in an argument indicating the composition of the service chain before initializing a runtime. The runtime guarantees that each flow actor processes the packet through each NF as indicated in the service chain. The modular NF design is similar to that in NetBricks [21], however, NFActor modifies the interface exposed by the NF module to achieve efficient flow management.

When executing flow management tasks, flow actor must be able to extract and transmit the flow states of all the NFs on the service chain, without disturbing the normal service chain processing. To speed up this process, we propose to separate the flow state with the processing logic of the NF module, and store the flow state inside the flow actor.

We summarize the core APIs used by NFActor to construct new NF modules in figure 3. Using these APIs, the flow actor could acquire a new flow state when it is created. When the flow actor processes a packet, the flow actor passes the current flow state and the input packet together into the processing logic of a NF module. The processing logic could directly update the flow state according to the input packet. Since the flow state is stored by the flow actor, the flow actor could directly manipulate its flow state when executing flow management tasks, without disturbing the NF processing logic.

```
new_state = nf.acquire_new_flow_state()

(output_pkt, updated_state) =
nf.processing_logic(input_pkt, current_state)
```

Figure 3: The core APIs used by NFActor to construct new NF modules.

Even though this design facilitates flow management tasks, it has its own limitation. Using this design, it is hard to extract and tranismit shared states. However, it is a complicated tasks to guaratee the consistency of shared states when managing a single flow. It may require multiple synchronizations, thereby affecting the processing speed of a single flow management task. Sice our primary goal when designing NFActor is to provide a high performance exeuction context, we do not aim to synchronize shared state. Instead, when flow management affects the consistency of the shared state, we could explicitly notify the NF about the result of flow management tasks and let NF module to handle the inconsistecy.

3.5 Route Selection

In a NFActor cluster, the existence of multiple layers of runtimes require us to do route selection for each flow. This functionality is directly assigned to each flow actor on the runtime. When a new flow actor is created, the flow actor select an output runtime as its destination using round-rubin algorithm. Before flow actor sends an input packet out, it replaces the destination MAC address of the packet with the MAC address of the input port of the destination runtime, and modifies the source MAC address as the MAC address of the output port of the current runtime. The flow actor also analyzes the source MAC address of the input packet to determine which input runtime the flow comes from.

We can see from later section that, when executing flow management tasks, the route of the flow needs to be dynamically changed. Since each flow actor has route selection functionality, the flow actor could directly send messages to the flow actors on its previous hop and next hop to modify the route.

3.6 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.

3.6.1 Flow Migration

As is shown in figure ??, 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 reorder 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.

3.6.2 Flow Replication

3.7 Dynamic Scaling

4. IMPLEMENTATION

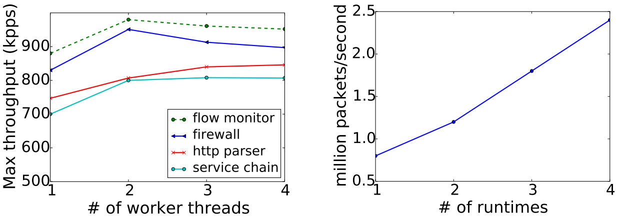
The implementation of the core functionalities of NFActor framework consists of 9921 lines of C/C++ code, excluding the implementation of 3 customized NF modules and miscellaneous helper codes. In NFActor, each runtime is containerized using Docker. The data plane of NFActor is inter-connected using BESS [13], which is a virtual switch for implementing high performance NFV system. The control plane of NFActor is interconnected using OpenVSwitch [22]. 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 [4]. 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.

5. EVALUATION

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

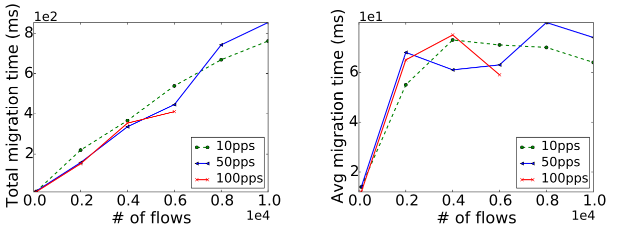
To evaluate the performance of *NFActor*, we implement 3 customized NF modules using the API provided by *NFActor* framework, the 3 NF modules are flow monitor, firewall and HTTP parser. The flow monitor updates an internal counter when it receives a packet. The firewall maintains several firewall rules and checks each



(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 4: 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 5: The flow migration performance of *NFActor*

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. 5.1) *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. 5.1) *Third,* how good is the flow migration performance of *NFActor* framework when compared with existing works like OpenNF? (Sec. 5.2) *Fourth,* what is the performance overhead of flow state replication and does the replication scale well? (Sec. 5.3)

5.1 Packet Processing Capacity

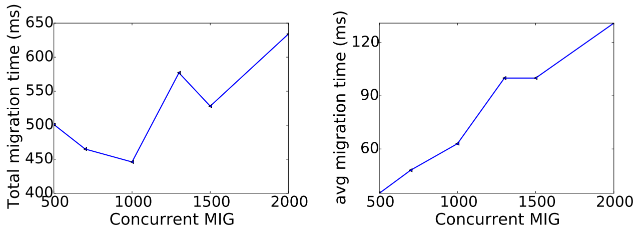
Figure 4 illustrates the normal case performance of running *NFActor* framework. Each flow in the generated traffic has a 10 pps (packet per second) per-flow packet rate. We vary the number of concurrently generated flows to produce varying input traffics. In this evaluation, we gradually increase the 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 4a, 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 4b, 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 4a, 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.

5.2 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 6.

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 con-



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

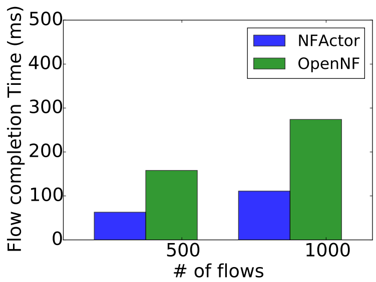


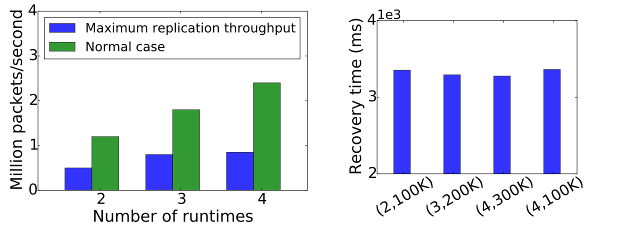
Figure 7: The flow migration performance of *NFActor*. Each flow in *NFActor* runtime goes through the service chain consisting of the 3 customzied NF modules. OpenNF controlls PRADS asset monitors.

trols 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 4a.

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 6. As we can see from fig 6b, 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 5b, 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 [12]. We generate the same number of flows to both *NFActor* runtimes and NFs controlled by OpenNF and calculate the total time to migrate these flows. The evaluation result is shown in



(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 8: The flow migration performance of *NFActor*

figure 7. 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 6a. All of these factors contribute to the improved flow migration performance of *NFActor* framework.

5.3 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 8.

In figure 8a, 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 8b 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.

6. 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 [11, 20], to improving the performance of NFV software [14, 13, 18, 21], to migrating flows among different NF instances [24, 15, 12], and to replicating NF instances [23, 26]. 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 [16] 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 [12] and Split/Merge [24] 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 checkpointing the entire process image and making a back-up for the created process image [26], 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 [26] 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 [3, 8, 7, 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 [3, 8, 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 [17] 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.

7. 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.

8. REFERENCES

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

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

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

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

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

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

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.
[22] 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.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.