NFLambda: Functional Composition Architecture for Network Function Virtualization

ABSTRACT

We present *NFLambda*, a new "greenfield" framework for re-architecting NFV. NFLambda is designed using a novel actor framework, with the goal to fully take advantage of the *software* nature of NFV while exploiting special hardware and smart NIC capabilities such as DPDK. The goal is to simultaneously support high *scalability*, *availability*, *performance* and *velocity* required of NFV via modular and safe functional composition within and across multiple servers. In this paper we provide an overview of the overall architecture of NFLambda and outline the implementation of its key components. This is a *work in progress*.

1 INTRODUCTION

Inspired by server virtualization and cloud computing, the idea of *network function virtualization* (NFV) came from the desire to virtualize various "expensive" hardware "middleboxes" such as firewalls, NATs and load balancers, thereby reducing the capital expenditure of network operators. Many conventional hardware middlebox vendors have started offering software products that virtualize their existing hardware boxes. This can be viewed as the first wave of NFV.

Virtualization of network functions (NFs) that are traditionally performed by specialized hardware boxes introduces many new challenges such as slower packet processing and higher likelihood of software failures. Systems such as OpenNetVM, BESS and VPP [3, 4, 15] aim to speed up the software packet processing pipeline by leveraging new hardware or smart NIC features such as DPDK [5], whereas ParaBox and NFP [18, 21] attempt to exploit parallelism to accelerate the packet processing of a chain of virtualized NFs – *service function chain* (SFC). On the other hand, "softwarization" of NFs also affords the network operators with many new capabilities such as *elastic service provisioning* by dynamically ramping up or down VMs or software modules that implement various network functions on demand. This has led to calls for re-thinking and re-factoring NFV design [7, 16].

In this paper we advocate and advance a new "greenfield" framework for re-architecting NFV, referred to as *NFLambda*. It aims to fully take advantage of the *software* nature of NFV while allows for the flexibility to exploit special hardware and smart NIC capabilities such as DPDK that are increasingly being incorporated in "white box" switches and commodity servers. The goal is to simultaneously support high *scalability*, *availability* (fault tolerance), *performance* and *velocity* required by emerging and future cloud network operations via modular and safe functional composition within and across multiple servers. We believe in particular that *velocity* – the ability to quickly roll out new services or features by adapting existing or deploying new network functions [11] – is perhaps the most critical feature that the new wave of NFV architectures should strive to offer. In contrast to *match-action* operations in SDN which are in general stateless, most NFs of interest are *stateful*. A key challenge in NFV architecture design is to ensure composition of existing or new NFs can be done *dynamically* in a *safe* (*correct*) manner.

As network functions are naturally *reactive* systems [6, 14] that operate on packet streams. NFLambda is designed using an *actor framework* which provides not only a simple and yet powerful programming model, but also a natural way to compose and decompose NFs: conventional (*monolithic*) NFs are re-factored into a collection of *actors* (also denoted by *λNFs*), each of which is associated with one *behavior* (i.e., performs one function), maintains its own *state* and interacts with other actors via *message passing*. In a sense, actors (or λNFs) are "operators" (or mathematical functions) that react to input packet streams (or generally *messages*), perform certain transformations on them, and send them to other actors for further processing (or transmit them outside the system).

In this paper, we propose NFLambda which is a *strong-typed* system where each actor has a well-defined *behavior* operating on *typed* input and output streams. This ensures that actors (or λNFs) can be combined and composed in a *type-correct* manner. Unlike existing NFV frameworks [3, 15, 16, 18, 21], NFLambda is explicitly designed to support NFV and dynamic service chaining across multiple servers in a cluster environment. Put it in another way, instead of NFV service chains where traffic or data packets are steered through of a chain of vNFs that run on different physical or virtualized servers, in NFLambda, network functions are operators and/or (mathematical) functions that operate on (input or intermediate) data streams, conducting certain calculations on, or performing transformations of the data streams.

In the remainder of the paper, we first motivate the design of NFLambda using the actor framework 2. We then lay out

the design goals of NFLambda, and outline the NFLambda architecture and its key components 3. We then describe how we adapt the existing C++ actor framework (CAF) to achieve line-speed NF packet processing by incorporating DPDK and support actor-based NF compositions across multiple servers in a *seamless* fashion 4. The paper is concluded in Section 5.

2 MOTIVATION

A robust NFV framework must be able to support scalability, availability, performance and velocity for NF development and deployment. It should also take advantage of the true software nature of NFV while exploiting hardware capabilities that are available. The *statefulness* of NFs makes this difficult as the state has to be managed for scalability and fault tolerance. Earlier NFV research studies such as OpenNF [13] and Split-n-Merge [17] takes existing virtualized (monolithic) NFs as is, but exploits different types of NF state (*e.g.*, partitionable, non-partitionable) to address scaling and state migration issues. OpenNetVM and BESS [3, 20] takes advantage of DPDK to speed up packet processing pipeline with minimal or zero-copy. Parabox and NFP [18, 21] exploits parallelism to speed up SFC processing within a single multi-core server. E2 supports scaling by allowing multiple "NF pipelets" to run concurrently on a per-flow basis. S6 [19] introduces DSO (distributed shared object) to support elastic scaling. Openbox attempts to improve the performance by decomposing monolithic NFs into modules and combining a SFC into a graph of OpenBox module instances to enable code reuse (*e.g.*, packet header processing) and speed up SFC processing.

The proposed NFLambda framework is designed explicitly for a *multi-core, multi-server* (edge) cloud environment in which NFV typically operates to simultaneously attain the scalability, availability, performance and velocity required for network functions. It builds on top of a novel *actor framework* following a *functional reactive program* (FRP) paradigm, while leveraging special hardware and smart NIC capabilities such as DPDK for line-speed packet processing. The actor framework [1, 2, 9] provides a natural way to decompose a (monolithic) NF into *functional* modules, namely, *actors* (akin to "microservices" but equipped with a well-defined behavior and interacting with each other via well-defined interfaces ("mailbox")). It has a simple yet powerful programming model: an actor operates by receiving messages based on (a collection of) pre-specified message patterns, performing certain operations (transformations) on the received messages, and interacting with other actors by sending messages. Actors are lightweight processes that can be quickly spawned with minimal memory footprints and context switching overheads and can be executed *concurrently* on a single server or in a *distributed* fashion.

Because actors are loosely coupled and maintain their *own* states, they can be easily scaled up and out by adding new instances or new actors. The actor framework also has *built-in* monitoring and error-handling capabilities [1, 2, 9]. When an actor fails, given that the internal state is minimal, it can be recovered quickly using various recovery strategies. The ripple effect of an actor failure is limited and localized. With well-defined and strong-typed interfaces, actors can be composed dynamically while ensuring *safety*.

We use a (monolithic) load balancer as an example to illustrate how we can decompose it into actors with their own states, and how the (refactored) load balancer using the actor model can better support scalability, fault tolerance and velocity. A load balancer is an NF that maps `m` public destination IP addresses to `n` private destination IP addresses. In a monolithic load balancer as shown in Figure 1a, when a flow comes in, the flow classifier based on the rules classifies it, then the hash function produces a result which is looked up in the flow table. If a hit is found, the source IP is re-written and sent out using the forwarding table. However, if there is a miss in the flow table, a server is selected from the server pool and then the packet is forwarded. On the other hand, in the actor based load balancer (see Figure 1b), all the different functions are refactored into different actors which communicate via message passing. In addition to the various functions that actors perform, the load balancer supervisor keeps tabs on the flow mappers and informs the flow classifier actor if there is a failure. Similarly, the server monitor checks the servers and if there is any change, the server pool and forwarder's forwarding table is updated accordingly.

The monolithic load balancer has various drawbacks. For example, an exogenous device needs to be added for scaling the load balancer and the required state the device needs for load-balancing is not immediately clear. Similarly, if we would like to change the logic of how servers are selected, then the code of the NF would have to be re-written and the NF would have to be re-started which affects the existing flows. In similar fashion, the effect of an increase in servers, server failure and recovery of the existing flow entries is that the affected and non-affected flows are complicated.

On the other hand, if we refactor the monolithic load balancer using the actor programming paradigm of NFLambda, we can get different actors that are loosely coupled. This allows us to scale up and out individual actors and run them on multiple cores or servers independent of each other. The flow classifier and dispatcher actor classifies the incoming flows based on rules which forms the control state of the actor. Then the flow id is sent to the flow mapper which checks if a mapping exists and sends the required information to the forwarder, which, based on the mapping in the control state, forwards the packets. If the mapping is not present in the flow

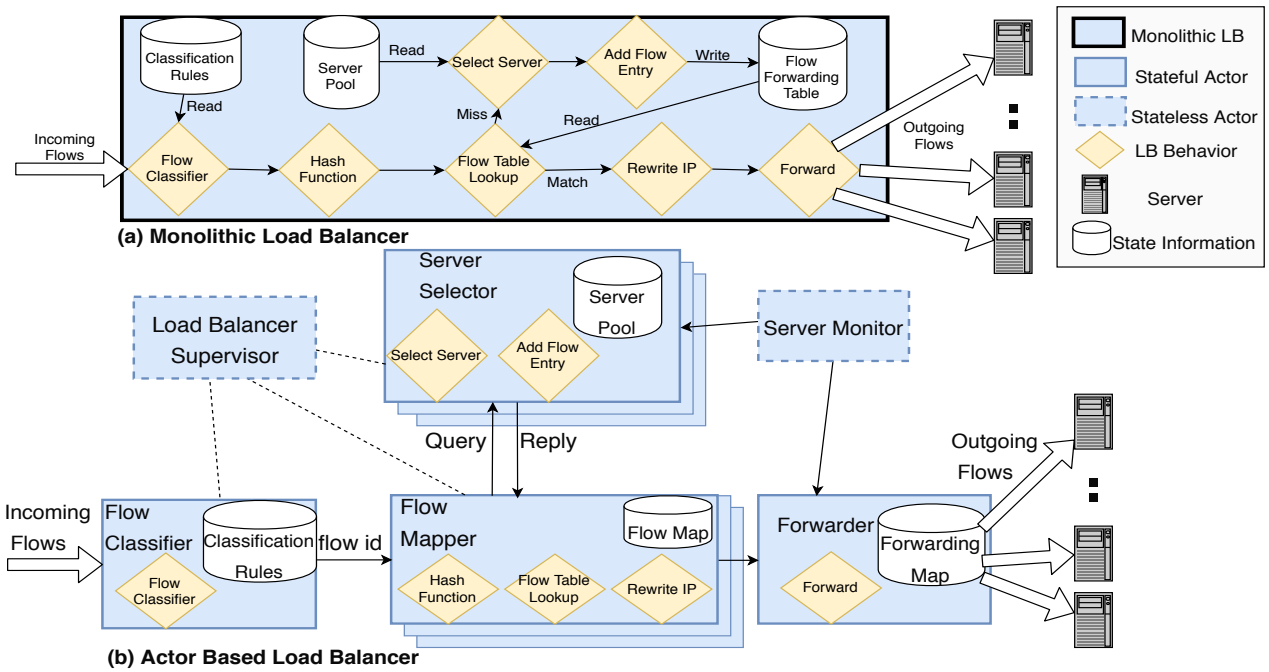

Figure 1: Monolithic vs. Actor based Load balancer

mapper, the server selector actor is invoked, which returns an appropriate server from the server pool.

If the number of flows increases, then we can scale the number of flow mappers independent of each other. Therefore, we can ensure that the flow map fits within the memory, thus maintaining performance. There can be rules installed inside the flow dispatcher to send the flow id to the corresponding flow mapper for appropriate load balancing. Furthermore, we can introduce the LB supervisor which monitors the flow mappers and if a failure occurs, the supervisor informs the flow classifier which alters its control information to adapt to the failure of the flow mapper.

In a similar fashion, if the load on the server selector actor increases, we can scale out the server selectors independently of other actors to ensure in-memory operations for performance. While if the number of servers increases or if there is a failure, then the server monitor actor can update servers in the server selector actor and also update the forwarding maps. All of this happens while the flow mappers or any other actors in the system are not affected at all.

By refactoring the load balancer into the actor model, we can also accommodate different types of server selectors or flow mappers with diverse needs and instead of rewriting the whole NF, only selective actor(s) have to be modified while the rest of the system remains unchanged.

We evaluate the existing actor-frameworks like Akka [2], Erlang [1] and C++ Actor Framework(CAF) [8] for NFLambda architecture. The former actor-models run on top of a virtual-machine, which is not ideal for network-processing applications. CAF maintains connections to remote actors through TCP sockets. However, for virtualizing network functions as actors, the raw network traffic should be treated directly instead of dealing with TCP connection handlers. CAF handles I/O in a single event loop, which presents a bottleneck for network heavy applications. We have modified the CAF framework to work for a network-intensive platform. Our actor-framework also enables fast packet processing at line rate by integrating DPDK [5], to achieve user-space networking capabilities.

3 ARCHITECTURE

In this section, we present the system architecture and introduce the key components of NFLambda.

Realizing packet processing in a functional composition architecture is by no means straightforward. First, NFLambda should bake modularity, composability and type correctness into the system so that the application developers (*a la* NF programmers) do not have to write NFs from scratch and rather use different building blocks to build a desired system. Second, the efficiency and the throughput of the NFLambda is of considerable importance to the design. Third, NFLambda should abstract away the different complexities of the hardware; fault tolerance and scalability were some of the key challenges as these capabilities had to be baked into the system so that the complexity is hidden from the developers.

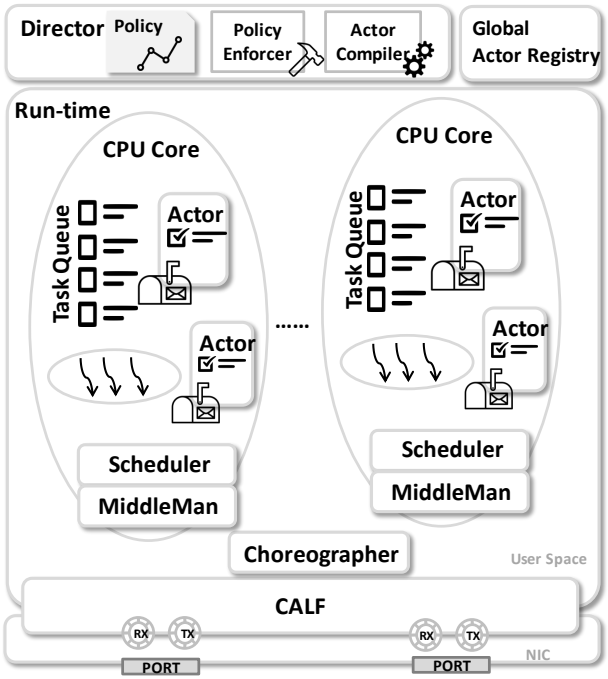

Figure 2: System Architecture of NFLambda.

Finally, a network centric view instead of the traditional box-centric view should be provided to better manage and control the whole system.

We schematically depict the overall system architecture of NFLambda in Figure 2. We assume that NFLambda has knowledge about the underlying network topology and link capabilities. Such information will be specified and provided by network operators as part of the input to NFLambda system. In the following we provide an overview of the key system components and their functionality.

NFLambda Director provides a domain specific programming model and functional language for NF composition. Network function logic is composed in a reactive way, and an efficient tool-chain is implemented to compile NFLambda source code into deployable executables. NFLambda Director also analyzes policy requirement and enforces it to NFLambda Run-time. In the meantime, an abstraction of protocol metadata is streamed among operators.

NFLambda Global Actor Registry is like a "DNS" service for actor lookup. Whenever an actor is spawned, its identity is published to the Global Actor Registry. The registered actor is assigned with a globally unique actor-reference, and is discoverable by other actors directly rather than using 5 tuples to locate an actor in the traditional way.

NFLambda Run-time is implemented to deploy and evaluate NFLambda executables in a computing cluster. This runtime is composed of five subsystems: *C++ Actor Lambda Function (CALF)*, *Choreographer*, *Middleman*, *Scheduler*, *Actors*. *CALF* builds on top of DPDK [5] to provide efficient packet processing on multi-core systems. Based on the configuration and policies enforced by the director, CALF fetch packets from physical NIC and scatters packets into different queues, each of which maps to a *Middleman*. The *Middleman* is responsible for dequeuing packets from its queues and applies fine-grained developer-defined classification policies to redirect packets to the mailboxes of the respective actors. The runtime tags packets along various stages of the packet processing pipeline. Tags act as meta-information associated with the packets which may be used by actors along the pipeline to make decisions. *Actors* are assigned with a task and executed when an available worker thread is assigned by the scheduler. *Scheduler* is used to orchestrate, schedule and scale the evaluations of NFLambda modules. Note that *Middleman*, *Scheduler* and *Actors* are running on the same CPU cores for efficient packet processing without context switching. *Choreographer* serves as a local communication agent between the different components of the runtime and the Director. For instance, policies enforced by the Director is received by the Choreographer which then forwards it to the relevant runtime component.

In this paper, we mainly focus on discussing different components of NFLambda runtime system and its implementation details. We leave providing details about the Director and Global Actor Registry as part of future work.

4 RUNTIME IMPLEMENTATION

The actor-model was introduced in [10] and actor model frameworks such as Akka [2], Erlang [1] and C++ Actor Framework (CAF) [8] are increasingly becoming popular in building large-scale distributed applications. However, none of these frameworks are efficient when it comes to utilizing transport and network layer services. They all rely on the underlying TCP/IP stack to interact with other sub-systems. For instance, the issue with Akka remoting is that a single TCP pipe exists between two Akka nodes, and the singular pipe is shared between all the messaging that happen between two cluster nodes - which includes the system level communication, as well as the application messages. The message queue contention will thus result in lost messages and re-transmissions, or a back-pressure stream, which is not desirable in a NFV setup. Similarly, in CAF when an actor wants to communicate with a remote actor located on some remote CAF instance, a TCP socket connection is established before the communication begins. Systems such as BESS [3] and OpenNetVM [20], which run entirely on user-space, are able to process packets at a very fast pace mainly because they bind directly to the network interface using hardware accelerators such as Intel's DPDK. Doing so bypasses the kernel networking stack and thus avoids unnecessary copying

of packets and allows to process packets at line speed. For NFLambda, we design a domain-specific actor framework in C++ by incorporating DPDK while borrowing some ideas from CAF. We now describe the different components of NFLambda runtime.

4.1 Actors

Actors are concurrent and distributed compute primitives in NFLambda. They are isolated entities that interact with each other through message-passing. Actors do not share state and can thus be executed in parallel. This loosely-coupled data-sharing model enables the run-time system to distribute and execute the actors in multiple cores or distributed machines. The packet reference is wrapped in a NFLambda message structure with some meta-information about the run-time system. The message-passing in NFLambda is cheap as the actors work on the same packet buffer and only meta-information and packet reference is copied between actors. NFLambda utilizes the copy-on-write optimization provided by the CAF run-time system to allow the packet reference to be rightly used between the actors.

4.2 CALF

CALF is a domain specific distributed-actor framework for network functions (NFs). One of the main goals while designing CALF is to efficiently distribute network packet processing across multiple cores (if available) in the system. With modern DPDK-enabled NICs natively supporting Receive Side Scaling (RSS) [12] capability, we are now able to create multiple receive (RX) queues, and cores can be assigned to them. By creating relevant policies catering to application needs, incoming traffic can be distributed across such RX queues. In other words, CALF is able to leverage hardware-capabilities to distribute and guide incoming traffic to different cores. Another important feature of CALF is to give the ability to application developer to specify packet templates which allows CALF to process different packet formats. CALF creates separate queues to process packets with different templates. Filter policies are used to steer incoming traffic to relevant queues. For each packet type, we can further create multiple queues. For example, let us consider an application developer wants to handle two packet types: i) control packets (used to configure or change the internal state of the application behavior), ii) data packets (user-traffic). CALF creates separate queues for control and data packets. Data packets can further be steered into different data-packet specific queues. Such division of queues (running on separate cores) helps parallelize different services or functions associated with NFs. It also allows to prioritize different packet types. Since control messages are less resource intensive tasks which may affect the behavior of the application, queues processing control packets could be assigned higher priority.

match	tag	middlemanID
dstnw=10.0.1.0/24	web	1
dstnw=10.0.2.0/24	stream	2

Table 1: Examples of Filter Policies used to Tag Incoming Packets and assign Middleman# for Packet Processing

In addition to distributing traffic across multiple cores, CALF also allows tagging of incoming packets using packet format, type, packet data, etc. Table 1 shows few examples of how packets are tagged by matching them against destination IP network and assign them to an instance of middleman running on the cluster. Such tags can then act as meta-information which can further be used in the processing pipeline (*e.g.* by the Middleman). For example, all packets whose destination IP belongs to `10.0.1.0/24` network are set with a tag called `web` and are assigned to be processed by `middlemanID` 1. Both, *match* $\rightarrow$ *tag* and *tag* $\rightarrow$ *middlemanID* have a many to one mapping. Entries in such tables are dynamically configured by the Director.

4.3 Middleman

Middleman subsystem is responsible for managing the message-passing between the actors in the NFLambda run-time system. It delivers the messages to the correct actor mailbox, and interface the CALF subsystem with NFLambda architecture. In a distributed environment, middleman spawns and maintains proxy-actors for the remotely identified actors. This gives a high-level system abstraction in such a distributed deployment. The middleman subsystem overrides the functionality from the CAF actor-framework. It is remodeled to have a wider bandwidth by allowing an array of backend I/O middleman-actors. A middleman-actor is tied to a single unique processing core in the system and handle the incoming traffic parallely. Each middleman-actor will have a single producer-single consumer queue to wait-on the incoming traffic. The middleman will blocking listen at these queues and direct the traffic to the subscribed actors as mapped by the director service.

NFLambda aims to exploit the fast buffering nature of DPDK with the CALF subsystem and the loosely-coupled actors running on multiple cores to achieve faster network packet processing. When a NF service chain is described, the first actor in the chain - the generator or the source actor will subscribe to a traffic pattern. The middleman will compile this pattern and install this traffic rule as tags at the director and mapping it with the actor-reference of the source actor. One source actor can subscribe to multiple different tags by specifying traffic rules separated by logical operators. Middleman steers the incoming traffic to the correct source actor

based on the meta-information in the packets. Middleman handles directing the communication between different actors in the service chain.

4.4 Scheduler

The NFLambda run-time system is designed to efficiently utilize a multi-core environment and maximize the performance throughput. The lambda-actors are the workhorses of the NFLambda architecture, which are scheduled to run parallely on the worker-threads tightly bound to multiple cores. The lambda-actors are asynchronous tasks that have contained-state for their execution and execute till completion in the thread. The number of actors in the application in general will be greater in number than the number of worker-threads available for execution. The run-time system dynamically assigns actors to the available set of worker-threads. The task to be executed could be sending of messages to other actors or execution of some lambda functions at the actors. The task-scheduling is handled with the Scheduler subsystem. The Scheduler is flexibly designed to take a custom policy to override the task-scheduling on the threads. NFLambda works on a multi-level priority based scheduling algorithm. The internal system control messages always have higher priority than the user-defined actor messages. For example, when an actor is monitoring another actor on failure, the event message handling will be taking higher priority than an application actor's execution. The scheduler is implemented over synchronized multiple producer and multiple consumer queues, where the actors are emplaced. The worker-threads dequeue from the queues when they are ready to execute.

4.5 Current Status

We have developed CALF which bypasses traditional OS kernel and is designed specially for vNF that can potentially handle packets at line-speed. CALF utilizes part of CAF, but re-designs most of its I/O (communication layers) such as middleman, actor registry, etc. to leverage DPDK for kernel-passing and customized NF packet processing pipelines and support parallel and distributed NF operations in a multi-core server cluster. CALF also borrows insights from BESS and OpenNetVM. However, unlike BESS, CALF supports multiple (concurrent/parallel) packet processing pipelines (that are associated with the same DKDP-capable NIC), and flexible packet layouts on top of DPDK to handle different network traffic types; and unlike Open NetVM, CALF implements a traffic scatter to distribute network traffic into different DPDK memory pools which makes network traffic available to the upper layer application (i.e., middleman). As next step, we have been exploiting the integration between CALF and middleman, global actor registry and director implementation.

5 CONCLUSION

We have presented NFLambda which fundamentally changes the composition and deployment of NFs through the novel use of the actor programming framework. By using the actor framework, we are able to support diverse NF functionality, flexibility, scalability, fault tolerance and have a high velocity for NFs. Furthermore, the use of packet templates allows for type correctness in SFCs at compile time whereas CALF allows for high-performance packet processing across multiple CPUs in multiprocessor systems. Therefore, NFLambda provides a powerful framework for the development and deployment of future NFs.

REFERENCES

[1] Erlang. https://www.erlang.org/.

[2] Scala Akka. http://www.akka.io/.

[3] BESS. http://span.cs.berkeley.edu/bess.html, 2017.

[4] Cisco's Vector Packet Processing. https://wiki.fd.io/view/VPP, 2017.

[5] DPDK. http://dpdk.org/, 2018.

[6] G. Berry. Real time programming : special purpose or general purpose languages. In *Research Report*. 1989.

[7] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proc. SIGCOMM*, 2016.

[8] D. Charousset, R. Hiesgen, and T. C. Schmidt. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, 2016.

[9] D. C. *et al.*. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *SPLASH*, 2013.

[10] H. C. *et al.*. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, 1973.

[11] M. D. *et al.*. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (Proc. NSDI)*, 2018.

[12] M. S. *et al.*. Receive side coalescing for accelerating tcp/ip processing. 2006.

[13] A. Gember-Jacobson et al. OpenNF: Enabling Innovation in Network Function Control. In *Proc. SIGCOMM*, 2014.

[14] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems. Springer-Verlag New York, Inc., 1985.

[15] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. NSDI*, 2014.

[16] S. Palkar et al. E2: A Framework for NFV Applications. In *Proc. SOSP*, 2015.

[17] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proc. NSDI*, 2013.

[18] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu. NFP: Enabling Network Function Parallelism in NFV. In *Proc. SIGCOMM*, 2017.

[19] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker. Elastic scaling of stateful network functions. In *Proc. NSDI)*, 2018.

[20] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proc. of HotMIddlebox*, 2016.

[21] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proc. SOSR*, 2017.