MiddleNet: A High-Performance, Lightweight, Unified NFV and Middlebox Framework

Ziteng Zeng, Leslie Monis, Shixiong Qi, K. K. Ramakrishnan
Dept. of Computer Science and Engineering, University of California, Riverside

2022 IEEE 8th International Conference on Network Softwarization (NetSoft) | 978-1-6654-0694-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/NetSoft54395.2022.9844083

***Abstract*—Traditional network resident functions (e.g., firewalls, network address translation) and middleboxes (caches, load balancers) have moved from purpose-built appliances to software-based components. However, L2/L3 network functions (NFs) are being implemented on Network Function Virtualization (NFV) platforms that extensively exploit kernel-bypass technology. They often use DPDK for zero-copy delivery and high performance. On the other hand, L4/L7 middleboxes, which usually require full network protocol stack support, take advantage of a full-fledged kernel-based system with a greater emphasis on functionality. Thus, L2/L3 NFs and middleboxes continue to be handled by distinct platforms on different nodes.**

This paper proposes MiddleNet that seeks to overcome this dichotomy by developing a unified network resident function framework that supports L2/L3 NFs and L4/L7 middleboxes. MiddleNet supports function chains that are essential in both NFV and middlebox environments. MiddleNet uses DPDK for zero-copy packet delivery without interrupt-based processing, to enable the 'bump-in-the-wire' L2/L3 processing performance required of NFV. To support L4/L7 middlebox functionality, MiddleNet utilizes a consolidated, kernel-based protocol stack processing, avoiding a dedicated protocol stack for each function. MiddleNet fully exploits the event-driven capabilities provided by the extended Berkeley Packet Filter (eBPF) and seamlessly integrates it with shared memory for high-performance communication in L4/L7 middlebox function chains. The overheads for MiddleNet are strictly load-proportional, without needing the dedicated CPU cores of DPDK-based approaches. MiddleNet supports flow-dependent packet processing by leveraging Single Root I/O Virtualization (SR-IOV) to dynamically select packet processing needed (Layer 2 to Layer 7). Our experimental results show that MiddleNet can achieve high performance in such a unified environment.

***Index Terms*—Middleboxes, NFV, DPDK, eBPF, service function chains.**

I. Introduction

Networks have increasingly become software-based, using virtualization to exploit common off-the-shelf (COTS) hardware to provide a wide array of network-resident functions, avoiding deploying functions in purpose-built hardware appliances. This has broadened the networking capabilities provided by the network and cloud platforms, thus offloading the burden from end-hosts that have limited power and compute capability (*e.g.,* cell phones or IoT devices). With softwarized network-resident functions, network services can be dynamically deployed across shared hosts.

However, there continues to be a dichotomy in how various network resident services can be supported on software-based platforms. Layer 2 and Layer 3 (L2/L3) functions that seek to be transparent bump-in-the-wire capabilities are being supported with Network Function Virtualization (NFV) designs. These focus on performance, are built around network functions (NFs) and chains in userspace supported by kernel-bypass technology such as DPDK [1]. Other than switching (demultiplexing and forwarding), they do not provide a full protocol stack and are exemplified by approaches such as OpenNetVM [2] and OpenvSwitch (OVS) [3]. The use of DPDK helps provide zero-copy packet delivery and shared memory to minimize overheads for data access within the chain, making it possible to support complex function chaining at line rate. Moreover, DPDK's poll-mode driver avoids undesirable behavior (*i.e.,* receive-livelocks) under overload [4]. Nevertheless, dedicated polling consumes significant CPU resources and is not load-proportional. While this may be reasonable in an NFV-only dedicated system, it is challenging for systems that host many services, including middlebox functionality.

Middleboxes require the full network protocol stack's processing (*e.g.,* application layer functionality such as HTTP proxies), in addition to more complex stateful functionality in userspace, such as storage and other I/O operations (*e.g.,* caching). Thus, flexibility and functionality are prominent concerns, with performance being a second (albeit important) consideration. It is often desirable to depend on a robust and proven kernel-based protocol stack [5], as specialized userspace protocol stack implementations often do not support all possible corner cases. However, depending on the kernel-based protocol stack often results in unnecessary duplicate protocol processing when multiple middlebox functions (MFs) are chained together. It is desirable to leverage the same design features of typical NFV platforms, such as zero-copy shared memory processing and the avoidance or mitigation of interrupt processing. This can recover some of the performance penalties of using a kernel protocol stack and ensure good overload behavior.

These distinct requirements for NFV and middlebox designs typically result in the need for different systems. However, networks require both types of functions to be supported concurrently for different flows, and in many cases, even for the same flow. This calls for supporting both types of functionality in a unified framework so that the functionality can be deployed on COTS hosts dynamically and flexibly.

For achieving this unified platform, we develop MiddleNet, a unified, high-performance NFV and middlebox framework. We take a somewhat unconventional approach by examining the design of two major alternatives in supporting NFV and middleboxes, and evaluating them. The first is to fully exploit the feature-rich networking subsystem provided by the extended Berkeley Packet Filter (eBPF [6]). Using eBPF's event-driven functionality, we examine its utility in supporting NFV and middlebox capabilities. Importantly, eBPF incurs negligible overhead in the absence of events (such as packet arrivals to a given function or even to the platform), making it an excellent fit for supporting a rich set of diverse, efficient network resident functions. An eBPF program has size restrictions and must run to completion, requiring careful design [7]. A second alternative approach is to build the unified framework around DPDK, as has been used in many high-performance virtualized environments, *e.g.,* OpenNetVM [2]. Such environments provide zero-copy delivery into userspace. With poll-mode drivers (PMD), they avoid the deleterious effects of interrupt-based processing of network I/O [4]. We implement these alternatives and evaluate their performance and resource usage for L2/L3 NFV and L4/L7 middlebox use cases. This helps us understand the strengths and limitations of each option, and understand their root causes. We then arrive at the design of MiddleNet as the most suitable framework for a unified platform for both capabilities. MiddleNet uses Single Root I/O Virtualization (SR-IOV [8]) to allow their co-existence. MiddleNet leverages the strengths of DPDK for L2/L3 NFV while taking advantage of eBPF for L4/L7 middleboxes.

II. Related Work

NFV platforms use different implementation approaches and primarily operate at L2/L3. OpenNetVM [2], based on DPDK, uses the microservice paradigm with a flexible composition of functions and uses shared memory to achieve full line-rate performance. However, OpenNetVM lacks full-fledged protocol stack support, focusing on supporting L2/L3 NFs. Compared to OpenNetVM, MiddleNet supports processing across the entire protocol stack, including application support. Other NFV platforms take different approaches. Both ClickOS [9] and NetMap [10] use traditional kernel style processing and mapping of kernel-user space memory, using interrupts for notifications. The interrupt-based notification schemes of ClickOS and NetMap can be vulnerable to poor overload behavior because of receive-livelocks [4]. In contrast, the L2/L3 processing in MiddleNet uses polling, thus avoiding receive-livelocks. E2 [11] integrates all the NFs as one monolith to help improve performance but gives up some flexibility to build complex NF chains through the composition of independently developed functions. NFV designs have increasingly adopted the microservice paradigm for flexible composition of functions while still striving to achieve full line-rate performance. Supporting this, MiddleNet's disaggregated design offers the flexibility to build complex L2/L3 NF chains.

Network-resident middleboxes' functionality depends on having full kernel protocol processing, typically terminating a transport layer connection and requiring a full-fledged protocol stack. Efforts have been made to pursue a high-performance middlebox framework with protocol processing support [5], [12], [13]. However, each of these proposals has its difficulties. mOS [12] focuses on developing a monolithic middlebox, lacking the flexibility of a disaggregated design like MiddleNet. Microboxes [13] leverages DPDK and OpenNetVM's shared memory design to improve packet processing performance and flexible middlebox function chaining. However, it does not provide a full-fledged protocol stack (it only supports TCP). The CPU consumption of DPDK-based designs is a further deterrent in the L4/L7 use case, significantly when the chain's complexity increases. Establishing communication channels for a chain of middleboxes using the kernel network stack incurs considerable overhead. Every transfer between the distinct middleboxes typically involves full protocol stack traversals, which adds considerable overhead. It typically involves *two* data copies, context switches, protocol stack processing, multiple interrupts, and *one* serialization and deserialization operation. MiddleNet is designed to reduce these overheads by leveraging event-driven shared memory processing to minimize CPU consumption. StackMap [5] also leverages the feature-rich kernel protocol stack to perform protocol processing while bypassing the kernel to improve packet I/O performance. However, it is more focused on end-system support than middlebox function chaining. StackMap's capability may be complementary to the design of MiddleNet.

There has not been a significant effort to design a unified environment where L2/L3 NFV and L4/L7 middlebox environments co-exist. MiddleNet is designed to address this issue.

III. Design of MiddleNet: L2/L3 NFV

We first discuss the eBPF-based and DPDK-based networking alternatives for L2/L3 NFV support, given the performance requirement of operating at line rate and being capable of supporting service function chains. Since they operate at L2/L3, there is less emphasis on having a full-function protocol stack.

A. Overview

To minimize overheads, reduce resource consumption, and achieve the full line rate, MiddleNet needs to take full advantage of zero-copy packet delivery. For evaluating an eBPF-based L2/L3 NFV design, MiddleNet uses the kernel-bypass support offered by AF_XDP [14]. By DMAing packets into the userspace, MiddleNet saves CPU cycles spent on the data copying, protocol processing, and context switching overheads involved in kernel protocol stack processing. AF_XDP depends on the interrupts triggered by the event execution of the XDP program attached to the NIC driver (Fig. 2). This interrupt serves to notify the packet processing in the userspace. However, these interrupts have to be managed with care to avoid poor overload behavior when subjected to high packet rates [4]. For evaluating the DPDK-based alternative for MiddleNet, we adopt the architecture of OpenNetVM [2] for supporting L2/L3 NFs. An NF Manager mediates the delivery of packets to and from the network, running on a core that constantly polls the NIC and uses Receive (RX)/Transmit (TX) descriptor rings to communicate with the NFs. Function

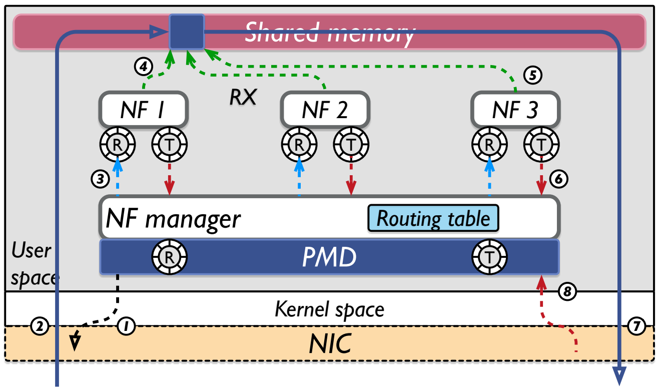

Fig. 1. Packet processing flow for DPDK-based L2/L3 NFV: RX and TX

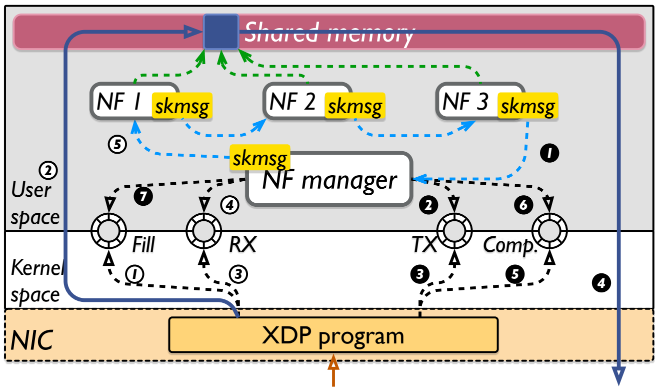

Fig. 2. Packet processing flow for eBPF-based L2/L3 NFV: RX and TX

chains are also mediated through the NF manager, which copies descriptor entries into the corresponding target NF, enabling lock-free communication. Data resides in shared memory (Huge Pages) so that there are no copies of the packet payload.

B. The DPDK-based L2/L3 NFV design

The DPDK-based approach uses polling-based zero-copy packet delivery. Packets bypass the kernel stack and are delivered directly into the shared memory in the userspace. DPDK uses its Poll Mode Driver (PMD) to poll the RX ring to retrieve arriving packets constantly. While assuring packet processing performance, it comes at the cost of occupying one CPU core for polling. In addition, for each NF of the L2/L3 service function chain, one CPU core is used up for each function. Thus, DPDK can be 'expensive' in having dedicated CPU cores for each functioning component. This can be wasteful if incoming traffic is low. Somewhat more complex NFV support, such as NFVnice [15], can be used to mitigate these overheads by sharing a CPU core across multiple NFs.

Fig. 1 depicts the packet flow of DPDK-based L2/L3 NFs. In the RX path, PMD provides a packet descriptor for the NIC (①) to deliver the packet into the shared memory via DMA (②). The NF manager examines the packet, and moves the packet descriptor into the RX ring of the target NF (③), based on the routing table. The target NF obtains the packet descriptor by polling its RX ring and uses it to access the packet in shared memory (④). After the NF's packet processing is complete (⑤), the NF writes the descriptor to its TX ring (⑥). On the other side, the NF manager (as the DPDK primary process) continuously polls the NF's TX ring and sets up the packet transmission based on the descriptor in the ring (⑦). The PMD then completes the processing once the packet is transmitted, to clean up the transmit descriptor (⑧). Both TX and RX rings are polled by the PMD for RX and TX from/to the NIC, and NFs use polling to RX or TX packet descriptors. All of this inevitably increases CPU consumption. Methods to reduce CPU consumption have been explored [15], which this design can conveniently adopt.

Service function chains: To support routing between a chain of functions for the DPDK-based approach, the NF manager utilizes destination information in the packet descriptor to forward the packet to the next NF in the chain. The chaining is done by selecting the name of the next service NF. The routing table in the NF manager is used to resolve that NF's ID, thus avoiding the need for each function to maintain a private routing table. After the NF manager gets a packet descriptor from the TX ring of an NF, it parses the packet descriptor to look at the destination NF information. It then pushes a packet descriptor to the RX ring of the next function to transfer ownership of the shared memory frame (as pointed to by the descriptor). Ownership for write is based on the NF currently owning a descriptor to that frame in shared memory, thus ensuring a single writer and obviating the need for locks. Using the NF manager for 'centralized' routing mitigates contention when multiple NFs may forward to a downstream NF.

C. The eBPF-based L2/L3 NFV design

With the support of AF_XDP, packets processed by L2/L3 NFs are delivered to shared memory in userspace, also bypassing the kernel and avoiding data copies. NFs that utilize AF_XDP have a dedicated AF_XDP socket (*i.e.,* `XSK`) that serves as an interface to interact with the kernel to handle RX and TX for AF_XDP-based packet delivery. Each `XSK` is assigned a set of RX and TX rings to pass packet descriptors containing pointers to packets in shared memory. All `XSKs` share a set of 'Completion' and 'Fill' rings, owned by the kernel and used to transfer ownership of the shared memory frame between the kernel and userspace NFs. Fig. 2 depicts the zero-copy packet flow based on AF_XDP.

An XDP program works in the kernel space with the NIC driver to handle packet reception (and transmission). The NIC is provided a descriptor (①) pointing to an empty frame in shared memory. Upon reception, the packet is DMAed into shared memory (②), and a receive interrupt triggers an XDP_REDIRECT which moves the packet descriptor to the RX ring of the NF manager (③) before invoking it. In the interrupt service routine, the kernel notifies the NF manager about updates in its RX ring, which the NF manager then accesses via its `XSK` (④). The interrupt service routine is completed once the NF manager fetches the packet descriptor from the RX ring. The NF manager invokes the corresponding NF (⑤) and waits for NFs to complete processing.

After the NF completes packet processing, the NF manager is invoked to transmit the packet out of the node (❶). The descriptor is populated in the TX ring (❷), pointing to the packet frame in shared memory. The system call by the NF manager (typically `sendmsg()`) notifies the kernel about the TX event (❸). The kernel then transmits the packet based on the descriptor given in the TX ring (❹). If the packet is successfully transmitted, the kernel pushes the descriptor back to the 'Completion' ring (❺) to inform the NF manager that the frame can now be reused for the subsequent transmission. The NF manager fetches the packet descriptor from the 'Completion' ring (❻) and moves it to the 'Fill' ring for incoming packets (❼).

We implement the NF manager with three threads to manage the different rings without locks. We use one thread to handle the read of the RX ring (④) and another one to handle the transmit to the TX ring (❷). We use a third thread to coordinate between the 'Completion' ring and the 'Fill' ring. This thread watches for the kernel to move packet descriptors into the 'Completion' ring (❻) upon transmitting completions. The third thread then moves the packet descriptor from the 'Completion' ring to the 'Fill' ring (❼). The NF manager can access different rings without locks by using three different threads.

Service function chains: MiddleNet extensively uses eBPF's socket message to support service function chaining for L2/L3 NFs. A strong motivation for exploring eBPF's event-driven approach is its load-proportional overhead compared to the constant CPU consumption of the polling-based DPDK approach. We can still achieve zero-copy packet delivery capability for communication within a chain of functions.

To support flexible routing between functions, we utilize eBPF's socket map. The in-kernel socket map maintains a map between the ID of the target NF and the socket interface information. As shown in Fig. 3, the function creates a packet descriptor to be sent (①). The socket message program performs a lookup in the socket map to determine the destination socket (②). It then redirects the packet descriptor to the next function (③). That function uses the descriptor to access data in shared memory (④) and passes the packet descriptor to the next function through the socket message after processing.

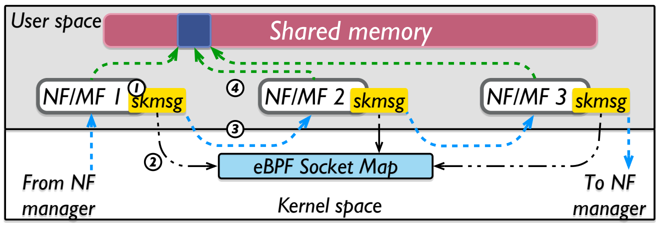

Fig. 3. Function chaining in MiddleNet: eBPF-based approach

Thus, communication within the function chain does not involve packet copies.

D. *Performance evaluation*

Experiment setup: We compare the performance of DPDK and eBPF approaches to support L2/L3 NFVs with a 'packet-centric' evaluation by comparing the Maximum Loss Free Rate (MLFR), the end-to-end latency, and CPU utilization at this MLFR for different packet sizes. The DPDK-based approach is based on the OpenNetVM design [2], and our eBPF-based approach is as described above. We set up our experiments on NSF Cloudlab [16] with three nodes: the 1st node is configured with a *Pktgen* [17] load generator for L2/L3 NFV use case; the 2nd node is configured with MiddleNet using each of the two alternatives; For this L2/L3 use case, the 3rd node is configured to return the packets directly back to the 1st node, to measure latency. Each node has a 40-core CPU, 192GB memory, and a 10Gbps NIC. We use Ubuntu 20.04 with kernel version 5.15. We use DPDK version 21.11 [1] and *libbpf* [18] version 0.6.0 for eBPF-related experiments. Each run is for 60 seconds. We collect the average value measured across 5 repetitions.

L2/L3 NFV Performance Evaluation: To compare the performance difference between the eBPF-based L2/L3 NFV approach and the DPDK-based L2/L3 NFV approach, we set up two NFs in a chain on the 2nd node: an L3 routing function followed by an L2 forwarding function. For the L3 routing function, MiddleNet updates the IP address of received packets, and the L2 forwarding function of a subsequent NF in the chain updates the MAC address of received packets and forwards it to the 3rd node.

Fig. 4(a) shows the MLFR for the DPDK-based and the eBPF-based approaches. The DPDK-based approach achieves almost the line rate for different packet sizes. The exception is for packet sizes of 64Bytes, achieving 12.6M packets/sec (84% of line rate) due to our limiting the number of CPU cores for the NF Manager and the PMD. Even with the limited CPU cores, the DPDK-based performance far exceeds the eBPF-based design performance. At a packet size of 64Bytes, the eBPF-based approach is limited to a forwarding rate of 3.2 Mpps (only 25% of the DPDK-based approach). Moreover, if the NFs have more complex processing or if the load were to be higher (*e.g.,* if there is bidirectional traffic), then we observe receive-livelock [4]. The performance of the eBPF-based L2/L3 NFV option is limited by its overheads, including a number of interrupts and context switches. For the two NFs setup in the eBPF-based approach, processing one packet requires 4 receive interrupts (1 receive interrupt incurred by AF_XDP, 3 interrupts incurred for communication within the NF chain). As we observe in Fig. 4(b), both the eBPF-based NF manager and the NFs spent most of the CPU time in the kernel (53% for the NF manager, 67% for NFs) to handle interrupts generated by `SKMSG`, thus leaving fewer resources to perform the NF packet forwarding tasks. Although devoting more resources to the eBPF-based NF manager and NFs may alleviate the overload, this only postpones the problem as

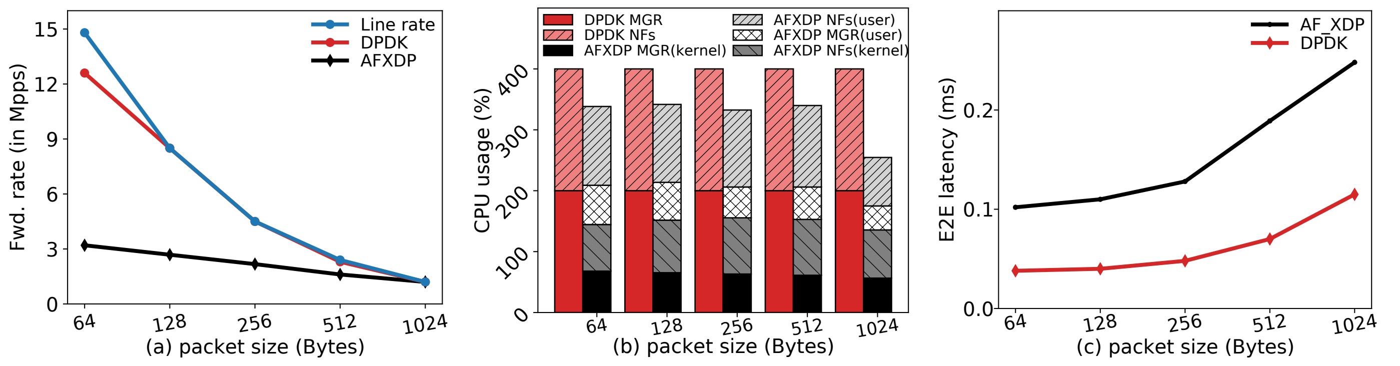

Fig. 4. Comparison between different L2/L3 NFV approaches: (a) Maximum loss free rate (MLFR) under different packet sizes, (b) CPU usage under MLFR under different packet sizes, (c) end-to-end latency under MLFR under different packet sizes.

the traffic load continues to increase. Moreover, using more resources to mitigate overload defeats the original intention of using eBPF-based event-driven processing since the goal of using it is for resource efficiency. For end-to-end packet latency, the DPDK-based approach achieves 2× improvement (Fig. 4(c)) compared to the eBPF-based approach.

The DPDK-based approach does constantly consume lots of CPU (one CPU core per NF, 2 CPU cores for the NF manager). While this is a concern, its much better performance makes it more attractive for L2/L3 NFs, since they have to act like a 'bump-in-the-wire'. An eBPF-based NFV implementation is less attractive because of its poor overload behavior.

IV. Design of MiddleNet: L4/L7 Middlebox

We discuss the corresponding eBPF-based and DPDK-based designs to support L4/L7 middleboxes. Since an L4/L7 middlebox relies heavily on protocol processing, we discuss optimizations, leveraging the kernel protocol stack processing, focusing on resource efficiency.

A. Overview

Packets pass through the kernel for protocol layer processing required by L4/L7 middleboxes. Incoming packets processed by the kernel network protocol stack are delivered through a socket to a message broker in userspace and inevitably incur data copying and context switch penalties. This penalty comes at a cost, but MiddleNet benefits significantly from a fully functional kernel protocol stack for such L4/L7 middleboxes. However, to eliminate subsequent, wasteful protocol processing within a chain of middlebox functions (MFs) and achieve the best possible performance, MiddleNet uses shared memory support for communication *within* an MF chain. This is done for both eBPF-based and DPDK-based approaches and avoids expensive data copies between MFs to achieve high-speed, scalable packet forwarding within the chain. In addition, since the MF operates directly on the payload, this obviates the need for a dedicated userspace protocol stack and makes the function more lightweight in terms of memory footprint. For the eBPF-based L4/L7 middlebox design, packets are forwarded between MFs using eBPF's SKMSG capability. We expect this alternative to be frugal in using compute resources given this event-driven functionality. For DPDK-based L4/L7 middlebox functionality, the message broker delivers descriptor entries to the ring of the target MF, with the payload being placed in the shared memory after protocol processing by the message broker. Thus, the design for L4/L7 middlebox processing is quite similar to the alternatives.

B. The eBPF-based L4/L7 middlebox design

Fig. 5 depicts the packet flow for the eBPF-based L4/L7 middlebox processing. For inbound traffic, after the payload is moved into shared memory by the message broker (①), a packet descriptor containing information about the location of the data in shared memory is sent to the target MF via eBPF's SKMSG mechanism (②). The MF then uses the descriptor to access the data in shared memory (③). For outbound traffic, once the MF has finished processing the packet (④), the MF uses the SKMSG to inform the message broker (⑤), which then fetches the packet in shared memory (⑥) and transmits it on the network via the kernel protocol stack.

Function chain support: With the eBPF-based L4/L7 approach, MiddleNet utilizes the eBPF's SKMSG and socket map for delivering packet descriptors within the function chain (similar to what we described for L2/L3 NFV with eBPF), as shown in Fig. 3. Although the eBPF-based L4/L7

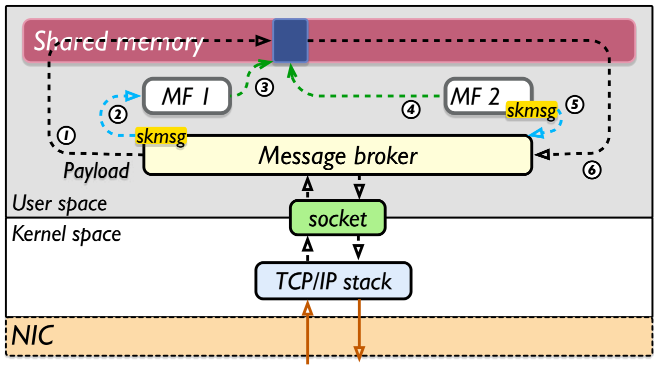

Fig. 5. Packet processing flow for eBPF-based L4/L7 middleboxes

approach still executes in a purely interrupt-driven manner, since the kernel protocol stack is involved, it often uses a flow-controlled transport protocol. This potentially avoids overloading the receiver, and therefore, receive-livelocks are less of a concern. Interrupt-based processing does not use up a CPU like polling, so it is more resource-efficient and benefits the L4/L7 use case. We further mitigate the impact of interrupts with batching.

Adaptive batching of `SKMSG` Processing: Since the eBPF-based approach uses `SKMSG` for packet descriptor transmission, bursty traffic can cause a large number of `SKMSG` transfers. We consider an adaptive batching mechanism to reduce the overhead of frequent `SKMSG` transfers. For each interrupt generated by `SKMSG`, instead of reading only one packet descriptor present in the socket buffer, we read multiple (up to a limit) packet descriptors available in the socket buffer. Thus, we can reduce the total number of interrupts, even for frequent `SKMSG` transfers, and mitigate overload behavior.

C. The DPDK-based L4/L7 middlebox design

We base our DPDK-based L4/L7 middlebox design on the overall design of OpenNetVM [2]. To leverage the kernel protocol stack, we restructure the NF manager of the L2/L3 use case (Fig. 1) into a message broker. The message broker is assigned a Linux socket interface to receive payloads from the kernel, as shown in Fig. 6. The message broker writes the received payload to shared memory (①), then, consulting the routing table, pushes the packet descriptor to the RX ring of the target MF (②). The MF keeps polling its RX ring for arriving packets. The MF uses the received packet descriptor to access the packet in shared memory and processes it (③). Once the processing is complete (④), the MF pushes the packet descriptor to its TX ring. On the other side, the message broker polls the TX ring of MFs for the packet descriptor (⑤), then accesses the shared memory and sends the packet out through the kernel protocol stack (⑥).

Function chain support: The function chain support in the DPDK-based L4/L7 middlebox design is the same as the

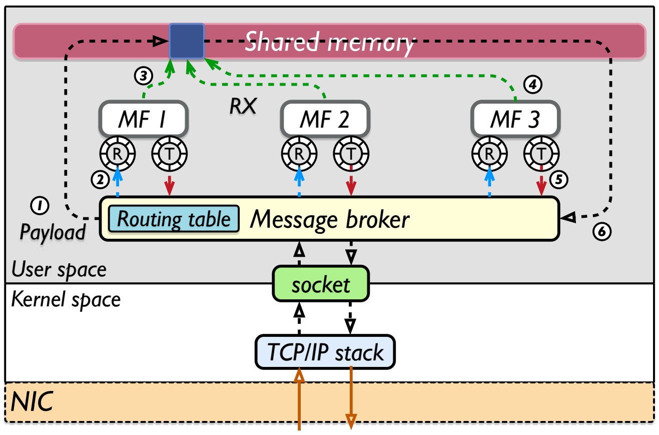

Fig. 6. Packet processing flow for DPDK-based L4/L7 middleboxes

service function chain support in the DPDK-based L2/L3 NFV use case (§III-B). Here, the message broker performs the (same) tasks to transfer packet descriptors between MFs.

D. Performance Evaluation of L4/L7 middleboxes

Experiment Setup: We now study the performance differences between the eBPF-based L4/L7 implementation option for MiddleNet (Fig. 5, hereafter referred to as *MN-E*) and the DPDK-based L4/L7 MiddleNet implementation (Fig. 6, hereafter referred to as *MN-D*). As a third alternative, we use an NGINX proxy to study the impact of the loosely-coupled design of MiddleNet. The NGINX proxy acts as a non-virtualized proxy to perform functions via internal function calls, which avoids introducing context switches or interrupts to provide good dataplane performance with a static, monolithic function implementation. We reuse most of the experiment setup described in §III-D.

We consider a typical HTTP workload for the L4/L7 middlebox scenario and examine application-level metrics, including request rate, response latency, and CPU usage, where the middlebox acts as a reverse proxy for web servers. The 1st node is configured to run the Apache Benchmark [19] to generate HTTP workloads. The 2nd node is configured with the MiddleNet system. On the 3rd node, we configure two NGINX [20] instances as web servers. We enable adaptive batching for the eBPF-based approach to minimize the overhead incurred by frequent `SKMSG` interrupts between MFs and the message broker at high concurrency. We use a chain with two MFs. The first is a reverse proxy function that performs round-robin load balancing between the two NGINX web server backends on the 3rd node. The second function is a URL rewrite function that helps perform redirection for static websites.

We also compare the scalability of *MN-D* and *MN-E*, when the number of MFs in a linear chain increases. To evaluate the impact of CPU-intensive tasks on the network performance of MF chains, we let MFs perform prime number generation (based on the sieve-of-Atkin algorithm [21]) when a request is received. Each MF is assigned one dedicated CPU core to perform tasks, including RX/TX of requests and the prime number generation. We set the concurrency level (*i.e.,* the number of clients sending HTTP requests concurrently) of Apache Benchmark to 512 to generate sufficient load.

Evaluation: Fig. 7 compares the RPS, response latency, and CPU usage of the different alternatives. *MN-E* depends on in-kernel `SKMSG` to pass packet descriptors between MFs, which inevitably generates context switches and interrupts, leading to slightly worse latency and throughput than *MN-D*. When the concurrency is between 1 and 32, there is a throughput difference between *MN-D* and *MN-E*, ranging from 1.09× to 1.3×. The throughput of NGINX's monolithic implementation is in-between *MN-D* and *MN-E*. At the lowest concurrency level of 1, *MN-E* consumes 37% of the CPU, which is a 10× reduction compared to *MN-D* (404%, *i.e.,* 4 CPU cores). Since *MN-D* uses polling to deliver packet descriptors, it continuously consumes CPU resources even when the traffic

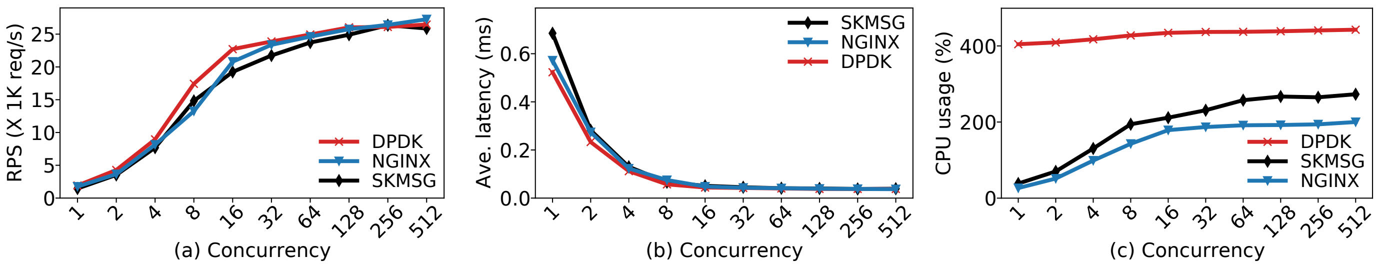

Fig. 7. RPS (a), latency (b) and CPU usage (c) comparison between different L4/L7 middlebox approaches.

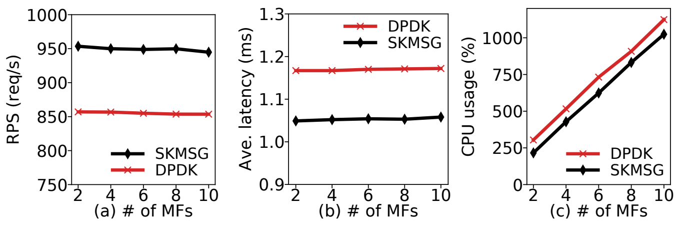

Fig. 8. RPS (a), latency (b) and total CPU usage (c) comparison with increasing number of CPU-intensive MFs in the chain.

load is low, resulting in wasted CPU resources. Although *MN-D* achieves 1.3× better RPS and latency compared to the *MN-E* at a concurrency of 1, *MN-E*'s resource efficiency more than makes up for its lower throughput (which is likely not the goal when using a concurrency of 1, in any case) compared to *MN-D*'s constant usage of CPU. Thus, it is more desirable to use the lightweight *MN-E* approach for these light loads.

When the concurrency level increases and the load is higher, the adaptive batching of the *MN-E* approach amortizes the interrupt and context switch overheads. The performance gap between *MN-E* and the others reduces to be within 1.05× for concurrency levels higher than 64. With adaptive batching, `SKMSG` can pass a set of packet descriptors, incurring only one context switch and interrupt, saving substantial CPU cycles, reducing latency, and improving throughput.

Compared to a monolithic NGINX as a middlebox, the *MN-E* approach exhibits slightly worse throughput and latency performance (1.04× less RPS due to 1.04× higher response delay) because of the overhead of function chaining, `SKMSG`, and virtualization. NGINX's internal function calls have slightly lower overhead (25% less on average) than middlebox function chaining with `SKMSG`, which has additional context switches and interrupts. However, running a set of middlebox functions as microservices improves flexibility and resiliency, allowing us to scale better, according to traffic load, especially with heterogeneous functions. Moreover, it allows functions to be shared between different middlebox function chains to improve resource utilization. With orchestration engines, *e.g.,* Kubernetes, intelligent scaling and placement policies can be applied with MiddleNet to improve resource efficiency further while still maintaining performance very close to a monolithic middlebox design.

Fig. 8 evaluates the scalability of *MN-D* and *MN-E* with CPU-intensive MFs. Both *MN-D* and *MN-E* show good scalability as the number of MFs increases. Surprisingly, *MN-E* performs even better than *MN-D* with CPU-intensive tasks in MFs, with a 10% improvement in RPS and a 10% reduction in latency. This is because with the prime number generation being CPU-intensive, it can quickly saturate the assigned CPU core and contend for CPU with the polling-based RX tasks of *MN-D*'s MF. But for *MN-E*, the RX of requests is triggered by interrupts, which is strictly load-proportional and avoids CPU contention. Since the prime number generation is performed within *MN-E*'s MFs, it is able to fully utilize the assigned CPU core, improving its performance. To improve *MN-D*'s performance, more CPU resources need to be assigned to the MFs, meaning that we are using resources inefficiently. In addition, for the combined CPU usage of the message broker and MFs, *MN-D* always needs one more CPU core than *MN-E* (Fig .8(c)). The extra CPU usage of *MN-D* is due to the RX polling in the message broker to receive requests from the MF. Since prime number generation is time-consuming, it results in a lower request rate. This means that the CPU devoted to handling RX of requests is used inefficiently. This reiterates the fact that *MN-D* uses resources inefficiently for this case, when dealing with CPU-intensive functions.

Throughout these experiments, *MN-E* has significant resource savings at different concurrency levels compared to *MN-D*, while having comparable throughput. Further, *MN-E* can even achieve better performance than *MN-D* when it executes CPU-intensive functions even when it uses resources more frugally. It also achieves close to the same performance as a highly optimized, monolithic application like NGINX. The resource efficiency benefits of the event-driven capability of eBPF, in conjunction with `SKMSG` to support shared memory processing, is a highly desirable way of building L4/L7 middlebox functionality in software.

V. A Unified Design based on SR-IOV

Based on the understanding from studying the alternative approaches and their performance characteristics, we now develop the overall architecture of MiddleNet that supports the co-existence of network resident NFV and middlebox capabilities in a unified framework running on a single system.

SR-IOV [8] allows multiple Virtual Functions (VFs) on a shared NIC, as depicted in Fig. 9. A VF acts as a distinct logical interface on the PCIe that offers direct access to the physical NIC resources that are shared across multiple VFs. It still achieves close to the single physical NIC's performance. By dividing the hardware resources available on the physical NIC into multiple VFs, we can dedicate a VF for each L2/L3 NFV and L4/L7 middlebox functions without having any one take up the entire physical NIC. The aggregate **NIC** performance will still be at the line rate. MiddleNet uses the Flow Bifurcation mechanism [22] for splitting traffic within the physical NIC in a flow or state-dependent manner. Since each VF is associated with different IP and MAC addresses, MiddleNet dynamically selects the packet processing layer (based on the VF it is attached to) from L2 to L7, providing a rich set of network-resident capabilities.

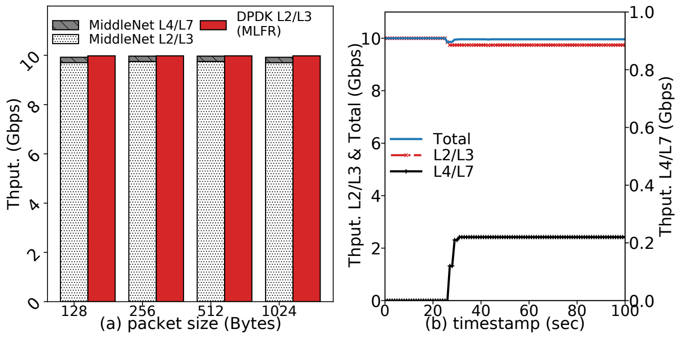

Fig. 10. (a) Aggregate throughput for various packet sizes. For L2/L3 NFV, we use Maximum loss free rate (MLFR) (b) Time series of throughput for L2/L3 NFV and total (left Y-axis) and L4/L7 middlebox (right Y-axis).

A. Flow and State-dependent packet processing using SR-IOV

MiddleNet attaches flow rules to the packet classifier in the physical NIC to support flow (and possibly state) dependent packet processing. Once a packet is received, the packet classifier parses and processes it based on its IP 5 tuple (*i.e.,* source/destination IPs, source/destination ports, protocol), which helps differentiate between packet flows.
(1) For a packet that needs to be handled by L2/L3 NFs, the classifier hands it to the VF bound to DPDK. The VF DMA's the raw packet to the shared memory in userspace. On the other side, the NF manager obtains the packet descriptor via the PMD and processes the packet in shared memory.
(2) For a packet that needs to be handled by L4/L7 MFs, the packet classifier hands the packet to the kernel TCP/IP stack through the corresponding VF. Since L4/L7 MFs require transport layer processing, MiddleNet utilizes the full-featured kernel protocol stack.

Because SR-IOV allows multiplexing of physical NIC resources, the split between the DPDK path and Linux kernel protocol stack path can be easily handled. L2/L3 NFs and L4/L7 MFs can co-exist on the same node in MiddleNet.

Using SR-IOV in a simple design, however, would result in these two frameworks co-existing as two distinct and separate functions providing services for distinct flows. The NIC switch feature of SR-IOV [23] can be used to bridge between different VFs within the NIC[1]. For both L2/L3 NFs and L4/L7 MFs to operate on the same flow, we use the NIC switch to have packets pass through the kernel protocol stack in or out of the L4/L7 layer to the L2/L3 NF. But this approach inevitably introduces extra overhead and may compromise the performance gain achieved by L2/L3 kernel bypass. Instead, since the packet payload (including header and all) is in MiddleNet's shared memory, an ideal way is to directly transfer packets between L2/L3 NFs and L4/L7 MFs without looping it back through SR-IOV's NIC switch. One possible approach is using the Kernel NIC Interface (KNI [24]) to perform userspace protocol processing by directly leveraging the kernel networking protocol stack while still keeping packets in shared memory. We speculate that MiddleNet could utilize KNI to remap the address space of the userspace shared memory into kernel address space, thus eliminating expensive system calls, interrupts and data copies, and can interact with the kernel protocol stack to provide much higher performance. A small overhead would be incurred for context switching and converting each packet's control information between the *rte_mbuf* and *sk_buff* formats. KNI also provides the flexibility of configuring the number of kernel threads that can be used for processing and setting the core affinity for each kernel thread. This flexibility will allow us to dynamically control the usage of CPU resources based on different types of workloads [15].

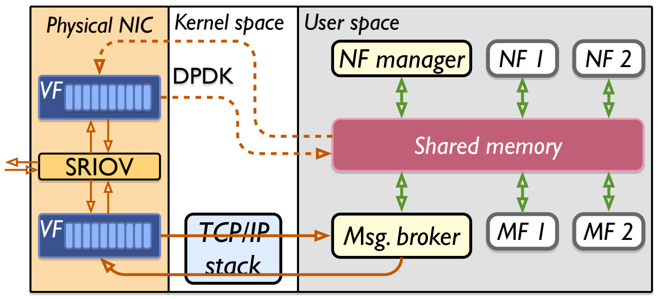

Fig. 9. The overall architecture of MiddleNet: A Combination of DPDK and eBPF via SR-IOV.

B. Performance evaluation

We investigate the performance of a unified L2/L3 NFV and L4/L7 middlebox and examine the interaction between the two, using SR-IOV to split the traffic. To mitigate interference between the load generators for L2/L3 (Pktgen [17]) and L4/L7 (Apache Benchmark [19]), we deploy Pktgen on the 1st node and Apache Benchmark on the 3rd node. We configure two NGINX servers on the 3rd node as the L4/L7 traffic sink. We configure two VFs on the 2nd node with SR-IOV and bind L2/L3 MiddleNet (DPDK) and L4/L7 MiddleNet (eBPF) to separate VFs. We use the same NFs (L3 routing and L2 forwarding) and MFs (reverse proxy and URL rewrite) on the

[1]A SR-IOV enabled NIC must include the internal hardware bridge to support forwarding and packet classification between VFs on the same NIC.

2nd node as described in §III-D and §IV-D. We modify the NFs and MFs to perform hairpin routing: L2/L3 NFs return traffic to the 1st node, and L4/L7 MFs return traffic to the 3rd node. Thus, we eliminate the interference that occurs between the two traffic generators. For L2/L3 traffic, we keep the sending rate at the MLFR. For L4/L7 traffic, we use a concurrency of 256 with the Apache Benchmark.

We study whether there is interference by checking the aggregate throughput as well as the throughput for the L2/L3 traffic processed by NFV and the L4/L7 processed by the middlebox, as shown in Fig. 10(a). The aggregate throughput of L2/L3 NFs and L4/L7 MFs remains close to 10Gbps, with negligible performance loss across various packet sizes. We also study the impact of adding L4/L7 flows when L2/L3 traffic (128Bytes packets) goes through MiddleNet at line rate (10 Gbps link). As shown in Fig. 10(b), at the 25th second, the Apache Benchmark starts to generate L4/L7 traffic (0.22Gbps), and the throughput of L2/L3 NFs correspondingly drops to 9.78Gbps. Thus, our unified design in MiddleNet for the co-existence of DPDK-based L2/L3 NFs and eBPF-based MFs provides both flexibility and performance.

VI. Conclusion

We presented MiddleNet, a unified environment supporting L2/L3 NFV functionality and L4/L7 middlebox functions. In MiddleNet, we chose the high-performance packet processing of DPDK for L2/L3 NFs and the resource efficiency of eBPF for L4/L7 middlebox functions. MiddleNet leverages shared memory processing for both use cases to support high-performance function chains. Experimental results demonstrated the performance benefits of using DPDK for L2/L3 NFV. MiddleNet can achieve full line rate for almost all packet sizes given adequate CPU resources provided to MiddleNet's NF manager. Its throughput outperforms an eBPF-based design that depends on interrupts by $4\times$ for small packets and has a $2\times$ reduction in latency. For the L4/L7 use case, the performance of our eBPF-based design in MiddleNet is close to the DPDK-based approach, getting to within $1.05\times$ at higher loads (large concurrency levels). In addition, the eBPF-based approach has significant resource savings, with an average of $3.2\times$ reduction in CPU usage compared to a DPDK-based L4/L7 design. Using SR-IOV on the NIC, MiddleNet creates a unified environment with negligible impact on performance, running the DPDK-based L2/L3 NFV service chains and eBPF-based L4/L7 middlebox function chains on the same node. This can bring substantial deployment flexibility.

Acknowledgment

We thank US National Science Foundation for their generous support through grants CRI-1823270 and CSR-1763929. We also thank our shepherd, Prof. Barbara Martini, and the anonymous reviewers for their valuable suggestions and comments.

References

[1] DPDK Project, "Data Plane Development Kit," https://www.dpdk.org/, 2022, [ONLINE].

[2] Zhang et al., "Opennetvm: A platform for high performance network service chains," in *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ser. HotMIddlebox '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 26–31.

[3] Ben et al., "The design and implementation of open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 117–130.

[4] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.

[5] Kenichi et al., "StackMap: Low-Latency networking with the OS stack and dedicated NICs," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 43–56.

[6] The Linux Foundation, "eBPF," https://ebpf.io/, 2022, [ONLINE].

[7] Miano et al., "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[8] Dong et al., "High performance network virtualization with sr-iov," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–10.

[9] Martins et al., "ClickOS and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 459–473.

[10] L. Rizzo, "Netmap: A novel framework for fast packet i/o," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 9.

[11] Palkar et al., "E2: A framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 121–136. [Online]. Available: https://doi.org/10.1145/2815400.2815423

[12] Jamshed et al., "mOS: A reusable networking stack for flow monitoring middleboxes," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 113–129. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/jamshed

[13] Liu et al., "Microboxes: High performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 504–517.

[14] The kernel development community, "AF_XDP," https://www.kernel.org/doc/html/latest/networking/af_xdp.html, 2022, [ONLINE].

[15] Kulkarni et al., "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 639–652, 2020.

[16] Dmitry et al., "The design and operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1–14.

[17] "Pktgen - Traffic Generator powered by DPDK," https://github.com/pktgen/Pktgen-DPDK, 2022, [ONLINE].

[18] "libbpf," https://github.com/libbpf/libbpf, 2022, [ONLINE].

[19] The Apache Software Foundation, "ab - Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2022, [ONLINE].

[20] F5 Networks, Inc., "NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy," https://www.nginx.com/, 2022, [ONLINE].

[21] "Sieve of atkin," https://en.wikipedia.org/w/index.php?title=Sieve_of_Atkin&oldid=1048307934, 2021, [ONLINE].

[22] DPDK Project, "Flow Bifurcation How-to Guide," https://doc.dpdk.org/guides-19.02/howto/flow_bifurcation.html, 2022, [ONLINE].

[23] Viviano, Amy, "NIC Switches," https://docs.microsoft.com/en-us/windows-hardware/drivers/network/nic-switches, 2022, [ONLINE].

[24] DPDK Project, "DPDK Kernel NIC Interface," https://doc.dpdk.org/guides/prog_guide/kernel_nic_interface.html#figure-pkt-flow-kni, 2022, [ONLINE].