Title

MiddleNet: A Unified, High-Performance NFV and Middlebox Framework with eBPF and DPDK

Journal

IEEE Transactions on Network and Service Management, PP(99)

ISSN

1932-4537

Authors

Qi, Shixiong
Zeng, Ziteng
Monis, Leslie
et al.

Publication Date

2023

DOI

10.1109/tnsm.2023.3256891

MiddleNet: A Unified, High-Performance NFV and Middlebox Framework with eBPF and DPDK

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

***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 have a greater emphasis on functionality, take advantage of a full-fledged kernel-based system.**

L2/L3 NFs and L4/L7 middleboxes continue to be handled by distinct platforms on different nodes. This paper proposes MiddleNet that develops 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 the Data Plane Development Kit (DPDK) library 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 for processing, avoiding a dedicated protocol stack for each function. MiddleNet fully exploits the event-driven capabilities of 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 in L4/L7 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 the packet processing needed (Layers 2 - 7). Our experimental results show that MiddleNet achieves 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, thus avoiding having to deploy functions in purpose-built hardware appliances. This has broadened the networking capabilities provided by both the network and cloud platforms, offloading the burden from end-hosts that may have limited power and compute capability (*e.g.,* cell phones or IoT devices). With software-based network-resident functions, network services can be more agile. They can be deployed more dynamically on end-systems that house multiple services.[1]

But there continues to be a dichotomy in how various network-resident services are supported on software-based platforms. Layer 2 and Layer 3 (L2/L3) functions that seek to be transparent and act as a bump-in-the-wire are currently being supported with Network Function Virtualization (NFV) technologies. These focus on performance and are built with network functions (NFs) running in userspace supported by *kernel-bypass* technology such as Data Plane Development Kit (DPDK [2]). Primarily providing switching (demultiplexing and forwarding), they typically do not provide a full network protocol stack, and are exemplified by approaches such as OpenNetVM [3] and OpenvSwitch (OVS) [4].

On the other hand, middleboxes operating at Layer 4 through Layer 7 (L4/L7) require the *full network protocol stack's processing* (*e.g.,* for application layer functionality such as HTTP proxies), in addition to more complex stateful functionality in userspace, including storage and other I/O operations (*e.g.,* caching). Thus, flexibility and functionality are prominent concerns, with performance being a second (albeit important) consideration. A robust and proven kernel-based protocol stack is often desirable [5], as specialized userspace protocol stack implementations often do not support all possible corner cases.

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

Both NFV and middleboxes often have to build complex packet processing pipelines using function chaining. This helps ease development through the use of microservices, which can be independently scaled as needed to improve resource utilization. But the excessive overhead (*e.g.,* interrupts, data copies, context switches, protocol processing, serialization/deserialization) incurred within the data plane of current service function chains can be a deterrent. Even worse, the data plane overhead in current function chaining solutions increases with the function chain size, which significantly reduces their data transfer performance (see §II-C).

Using shared memory communication can help us achieve a more streamlined, efficient data plane design. Shared memory communication supports zero-copy packet delivery between network-resident functions, by having a shareable backend buffer to store packet data, avoiding unnecessary data plane overheads within a function chain.

Another dichotomy is in how the key building block for shared memory communication is designed. This relates to

[1] *This paper is an extended version of our previously published IEEE NetSoft 2022 [1] paper, with significant additions.*

how packets are moved between the NIC and the shared memory buffer, and *how packet descriptors are passed between functions in a function chain*. The first option is to exploit the ***event-driven*** networking subsystem provided by the extended Berkeley Packet Filter (eBPF [6]). eBPF offers extensive toolkits (*e.g.,* AF_XDP [7], `SKMSG` [8]) in support of zero-copy packet delivery. 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 does have size restrictions and must run to completion, requiring careful design [9]. A second alternative approach is to build the shared memory communication framework around ***polling-based*** DPDK, as has been used in many high-performance virtualized software-based networking environments, *e.g.,* OpenNetVM [3]. They provide zero-copy delivery into the userspace. Using poll-mode drivers (PMD) [10] and RTE RING [11], they avoid the deleterious effects of interrupt-based processing of network I/O (*e.g.,* receive-livelocks) under overload [12], making it possible to support complex function chaining at line rate. Nevertheless, dedicated polling continuously consumes significant CPU resources, and thus 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 functions.

In this work, we develop MiddleNet, a unified, high-performance NFV and middlebox framework. We take a somewhat unconventional approach by examining an event-driven eBPF design, and separately a polling-based DPDK design for supporting NFV and middlebox function chains with shared memory, and evaluating each design approach. We then arrive at the design of MiddleNet as the most suitable framework for a unified platform supporting both NFV and middlebox functionality. MiddleNet uses Single Root I/O Virtualization (SR-IOV [13]) to enable their co-existence.

MiddleNet makes the following contributions:

(1) We qualitatively discuss the usability of different data plane models for supporting NFV and middlebox capabilities. We carefully audit their data plane overheads and quantitatively assess the performance of each approach. We also look at how current data plane models support function chaining (§II).

(2) We then design the shared memory communication for MiddleNet both the NFV and middlebox (§III) functionality. We (qualitatively and quantitatively) examine the suitability of eBPF and DPDK in supporting different aspects of shared memory communication, including NIC-shared memory packet exchange and zero-copy I/O (*i.e.,* packet descriptor delivery) within the function chain (§IV and §V). This helps us understand the strengths and limitations of each option (DPDK's PMD, polling/interrupt-based AF_XDP in eBPF, DPDK's RTE RING, eBPF's `SKMSG`), and the root causes. MiddleNet chooses to leverage the strengths of polling-based DPDK for L2/L3 NFV, and takes advantage of event-driven eBPF for L4/L7 middleboxes, to strike the balance between performance and resource efficiency.

(3) For achieving a unified NFV/middlebox framework, we evaluate different alternatives: a hardware-based approach (via SR-IOV [13]) and a software-based approach (via virtual device interfaces, *e.g.,* virtio/vhost [14]). We assess the performance with SR-IOV and recommend its use for the unified design because of its minimum data plane overhead (§VI).

(4) MiddleNet supports function-chain-level isolation to address security concerns with shared memory communication. We create a private memory pool for each function chain to prevent unauthorized access from untrusted functions outside the chain. MiddleNet further enhances traffic isolation by applying packet descriptor filtering between functions (§VII).

II. Background and Motivation

We examine a number of virtualization frameworks and the networking support that can be provided for supporting network resident functions. We audit the data plane overheads for these different combinations of virtualization frameworks and networking approaches, and discuss their applicability for achieving a high-performance, lightweight, and unified NFV/middlebox framework.

A. Basic elements in supporting network resident functions

We identify four key elements for building NFV and middleware environments, including **virtualization frameworks**, the **virtual switch (vSwitch)**, the **protocol stack**, and the **virtual device interface**. Virtualization helps to multiplex compute resources, and can greatly improve resource efficiency, and reduce costs, while also providing isolation for building L2/L3 NFs and L4/L7 middleboxes. A vSwitch is typically used to provide L2 forwarding/L3 routing. The network protocol stack, often implemented in the OS kernel, provides protocol layer processing (*e.g.,* TCP/IP). It is necessary for L4/L7 middleboxes, but is less important for L2/L3 NFs. Virtual device interfaces are used to connect the virtualized function and its protocol stack (for L4/L7 middleboxes only) to the vSwitch, thus building a complete NF and middlebox environment. There are several alternatives for each of these elements, which we describe below.

Virtualization frameworks: Widely-adopted virtualization frameworks include virtual machines (VMs) and containers. VMs often depends on hardware-level virtualization supported by the Virtual Machine Monitor (VMM) or the hypervisor in the host that multiplexes the physical resources across multiple VMs. Each VM has its own OS layer (*i.e.,* guest OS). Unlike a VM, a container is built utilizing OS-level virtualization. Containers share a host's OS to access the underlying physical resources, instead of depending on the hypervisor. The host's OS utilizes Linux namespaces and *cgroups* to provide isolation between containers and restrict their access to system resources. Sharing the host's OS makes containers more lightweight. They can be provisioned more quickly compared to VMs [15].

Virtual switch (vSwitch): vSwitches can be broadly classified into kernel-based approaches (*e.g.,* in-kernel Open vSwitch and Linux bridge) and userspace approaches that bypass the kernel (*e.g.,* OVS-DPDK [16], and OVS-AF_XDP [17]). The kernel-based vSwitch runs within the host's OS kernel, using an in-kernel NIC driver to exchange packets with the physical NIC. The userspace vSwitch runs in the userspace of the host,

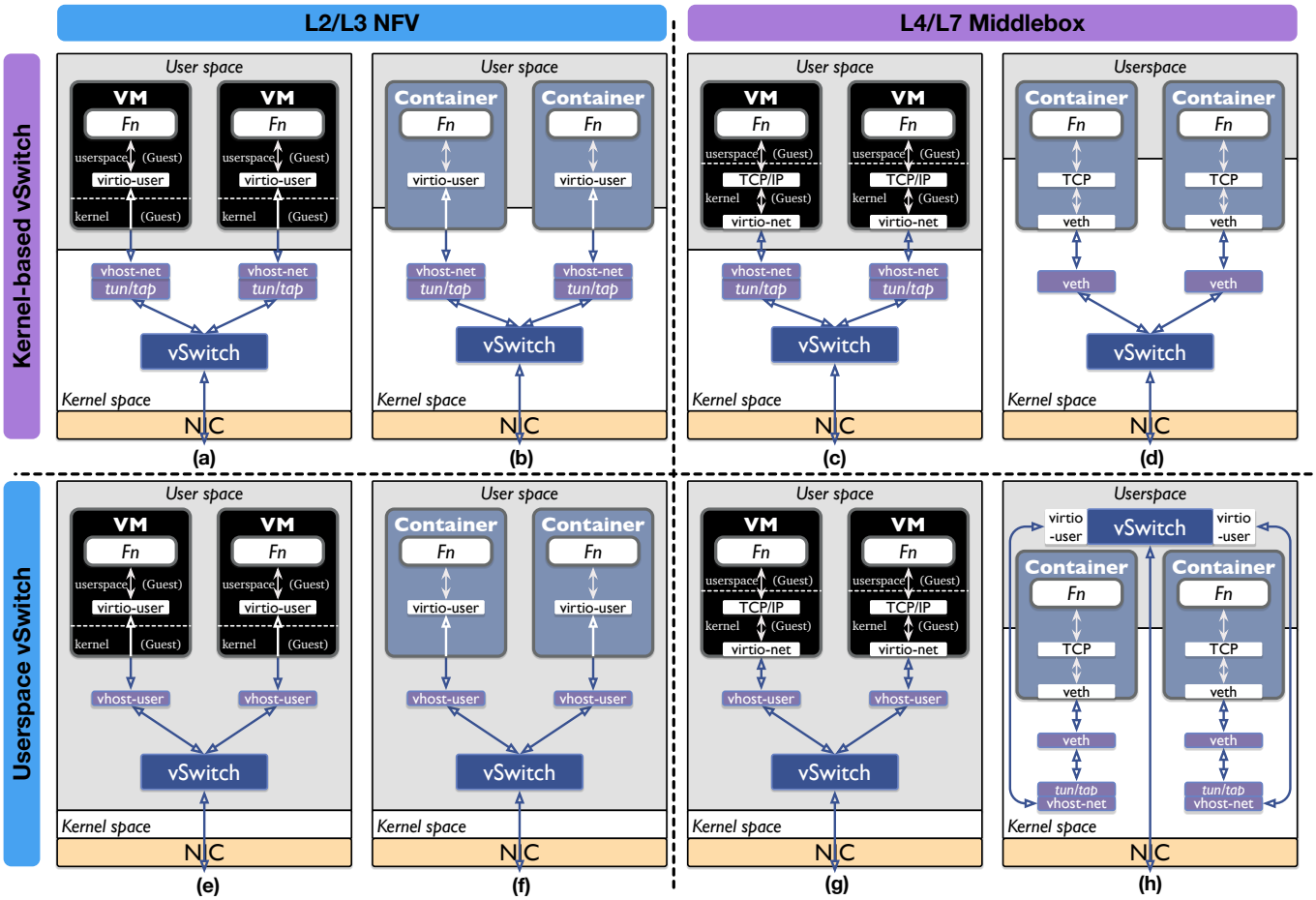


Fig. 1. Distinct data plane models for NFV and Middlebox, with different vSwitch options, virtual device interfaces, and virtualization frameworks: (a) kernel-based vSwitch + *virtio-user/vhost-net* & *TUN/TAP* + VM; (b) kernel-based vSwitch + *virtio-user/vhost-net* & *TUN/TAP* + container; (c) kernel-based vSwitch + *virtio-net/vhost-net* & *TUN/TAP* + VM; (d) kernel-based vSwitch + *veth* + container; (e) userspace vSwitch + *virtio-user/vhost-user* + VM; (f) userspace vSwitch + *virtio-user/vhost-user* + Container; (g) userspace vSwitch + *virtio-net/vhost-user* + VM; (h) userspace vSwitch + *virtio-user/vhost-net* & *TUN/TAP* + *veth* + container. **We assess (f) as the best solution for L2/L3 NFs and (d) as the best solution for L4/L7 middleboxes (§II-C).**

using a userspace NIC driver to exchange packets with the physical NIC.

The userspace vSwitch relies on kernel-bypass to exchange packets with the NIC. We consider two distinct, but widely adopted, kernel-bypass architectures: DPDK [2] and AF_XDP [7]. They both support zero-copy packet I/O between the NIC and userspace. However, they are fundamentally different in the way they are driven to execute. DPDK's kernel-bypass depends only on polling while the kernel-bypass in AF_XDP can be either event-driven (*i.e.,* triggered by each arriving packet) or polling. DPDK implements a Poll Mode Driver (PMD), polling the NIC for received packets and packet transmission completions. This facilitates high-performance packet I/O between the NIC and the userspace functions. However, this leads to high CPU usage even if there is no incoming packet. An additional, specialized kernel driver (*e.g.,* UIO driver or VFIO driver) is required to block interrupt signals from the NIC, which helps the userspace PMD to work properly through active polling. However, this requires the NIC to be dedicated to DPDK. The exclusivity of DPDK leads to compatibility problems between DPDK and the kernel stack; *e.g.,* the kernel stack now cannot access the NIC once DPDK has bound its kernel driver to the NIC. One solution is to use Single Root I/O Virtualization (SR-IOV [13]) to create multiple virtual Ethernet interfaces (these are called Virtual Functions, or VFs), and to dedicate DPDK's kernel driver to one of the VFs without disturbing the kernel stack (see §VI).

AF_XDP [7], is another kernel-bypass alternative to DPDK. The event-driven mode of AF_XDP makes it strict load-proportional. Event-driven AF_XDP executes only when a new packet arrives, thus it consumes *no* CPU cycle when there is no packet. This fundamentally makes event-driven AF_XDP more resource-efficient under light load compared to DPDK. The polling mode AF_XDP acts in a similar manner as DPDK. However, the polling mode of AF_XDP still introduces interrupt overhead due to the execution of the XDP program at the NIC driver, which results in lower performance compared to DPDK. We evaluate both polling-based and event-driven AF_XDP in §IV-D. In addition, AF_XDP (either polling or event-driven mode) does not require a specialized kernel driver to enable kernel-bypass, and thus it can work seamlessly with the kernel stack to support protocol processing for an L4/L7 middlebox. DPDK, on the other hand, requires SR-IOV support to share the physical NIC with the kernel stack. Compared to a purely kernel-based solution (*i.e.,* using the kernel stack for both L2/L3 NFs and L4/L7 middleboxes), AF_XDP achieves comparatively higher performance with zero-copy packet I/O between the NIC and userspace functions.

Network protocol stack: The protocol stack can be kernel-based or could be in userspace, using kernel-bypass for passing packets. The kernel-based network protocol stack (*e.g.,* Linux kernel protocol stack) provides a full-function, robust, and proven solution for protocol processing, often with better usability than userspace protocol stack solutions such as Microboxes [18] and mTCP [19], which provide limited support (*e.g.,* only TCP), thus limiting their usage. *We primarily focus on the kernel-based protocol stack in this work.*

Virtual device interfaces: Typical virtual device interfaces include *TUN/TAP*, *veth pairs*, and *virtio/vhost* devices. *TUN/TAP* operates as a data pipe (TUN for sending over L3 Tunnels, TAP for receiving L2 frames) that connects the kernel stack with userspace applications. *TUN/TAP* can work with *virtio/vhost* virtual device interfaces to connect VMs or containers to the kernel-based vSwitch (Fig. 1 (a) - (c)). The *virtio/vhost* interfaces execute as virtual NICs (vNICs) for VMs and containers. The *virtio* interface is in the VM/container, while the *vhost* interface is in the host as the backend of the *virtio* device. It is important to note that each has a userspace variant (*virtio-user*, *vhost-user*) as well as a kernel-based variant (*virtio-net*, *vhost-net*). The *virtio* variants and *vhost* variants can be freely combined, *e.g., virtio-user* can work with *vhost-net* (Fig. 1 (a), (b)); *virtio-net* can work with *vhost-user* (Fig. 1 (g)), *etc.* because they all follow the *vhost* protocol [14], having a consistent messaging APIs to work with different variants. *Veth pairs* are often used in container networking [20], working as data pipes between the container's network namespace and the host's network namespace. Unlike *virtio/vhost*, the *veth pair* works only in the kernel. It does not have a userspace variant, so it does not work directly with the userspace vSwitch (see Fig. 1 (h)).

B. Usability analysis of data plane models

Fig. 1 shows different variants for data plane connectivity for L2/L3 NFs and L4/L7 middleboxes by combining different options for virtualization, vSwitch, and virtual device interfaces. L2/L3 NFs do not require protocol layer processing, since they only offer an L2/L3 switch's forwarding capability, as in a vSwitch. L4/L7 middleboxes additionally require protocol stack processing. We first qualitatively evaluate the **usability** of different data plane models for L2/L3 NFs and L4/L7 middleboxes in Fig. 1, depending on whether the data plane model has a protocol stack or not.

The data plane models in Fig. 1 (a), (b), (e), (f) do not involve protocol layer processing and are suitable for L2/L3 NFs. The data plane models in Fig. 1 (c), (d), (g), (h), are all equipped with the kernel protocol stack and are suitable for L4/L7 middleboxes. Although data plane models for an L4/L7 middlebox (Fig. 1 (c), (d), (g), (h)) can also be used for an L2/L3 NF. The protocol processing however adds unnecessary overhead, as it is not required. In addition, we can extend the L2/L3 NF data plane models to support L4/L7 middleboxes by adding a userspace protocol stack; however, this approach is not favored by us for two reasons: (1) we want to use a full-function kernel protocol stack, and (2) having a separate userspace protocol stack in each middlebox function again adds to the memory footprint.

The use of the *virtio-user* interface helps an L2/L3 NF data plane to bypass protocol layer processing, acting as the vNIC driver in a VM/container's userspace, directly interacting with the userspace function. Depending on the vSwitch being used, the *virtio-user* device cooperates with different backend *vhost* devices to create a direct data pipe between the userspace function and the vSwitch (either kernel-based or in userspace) to exchange raw packets: the *vhost-net* device is used to connect with the kernel-based vSwitch through the *TUN/TAP* (Fig. 1 (a), (b)); the *vhost-user* device is used to connect with the userspace vSwitch (Fig. 1 (e), (f)).

When using containers to virtualize L4/L7 middleboxes (Fig. 1 (d), (h)), the key element to enable the network protocol stack is the *veth pair*. The container-side veth connects to the protocol stack in the container's network namespace (implemented in the host's kernel), for necessary protocol processing.[2] The host-side veth connects to host's network namespace, so it can seamlessly work with the kernel-based vSwitch (d). However, if we have to work with a userspace vSwitch (h), the packet needs to be injected from the userspace to the container's network namespace for protocol processing. To achieve this goal, the userspace vSwitch is connected to the kernel via the *virtio-user/vhost-net* and *TUN/TAP* device interfaces. The *TUN/TAP* interface is configured with a point-to-point link to the *veth pair*, which helps avoid duplicate L2/L3 processing in host's network namespace.

When using VMs to virtualize L4/L7 middlebox functions, the *virtio-net* device interface is used to utilize the protocol stack in VM's kernel. The *virtio-net* device operates as the in-kernel vNIC driver, interacting with the userspace function through VM's kernel stack. Just like the *virtio-user* device interface, the *virtio-net* interface can work with either a kernel-based vSwitch (Fig. 1 (c)) or a userspace vSwitch (Fig. 1 (g)) by cooperating with specific backend *vhost* device interface.

C. Auditing Overheads of data plane models

The data plane models in Fig. 1, with their selection of elements (*i.e.,* vSwitch, virtualization framework, virtual device interfaces) in constructing the data plane, may result in different data plane performance. Through a careful auditing of the overhead, we seek to identify the optimal data plane model for L2/L3 NFs and L4/L7 middleboxes. For this, we focus on the data plane overhead with a function chain.

For both L2/L3 NFs and L4/L7 middleboxes, function chains are mediated by the vSwitch to route packets between functions to be processed in the order they are configured in the chain. Additional protocol processing is required for the L4/L7 middlebox case. We only show the auditing results when using DPDK as the kernel-bypass architecture for the userspace vSwitch in this auditing.

[2]Note: there is no L2/L3 processing in the container's network namespace. The reason is the container actually shares the same kernel with the host. As the L2/L3 processing is performed by the kernel-based vSwitch in the host's network namespace, packets enter into the protocol layer stack after being passed to the container's network namespace. Thus, no duplicate L2/L3 processing is performed inside the container. Each *veth pair* is assigned a unique IP address, which is used for L2/L3 forwarding across different containers' network namespaces. Applications with a container namespace share the same IP address and are differentiated by L4 port numbers.

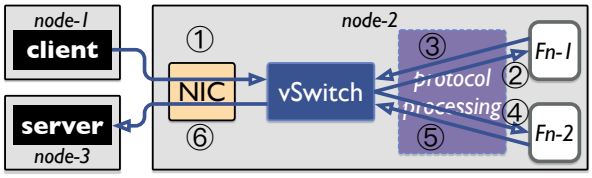


Fig. 2. A generalized data pipeline for an NFV/Middlebox chain. Note: we only show the client-to-server datapath; protocol processing is only available for L4/L7 middlebox.

We use the abstract function chain setup of two functions (Fig. 2) to represent the data pipeline for all cases. We assume functions in the same chain are placed on the same node so that there is no cross-node data transfer. The client sends packets to the backend server through an intermediate node (node-2 in Fig. 2) that implements the function chain. (①) A packet first arrives at the physical NIC and is then passed to the vSwitch. (②) The vSwitch routes the packet to the first function in the chain (Fn-1). (③) After the first function completes processing the packet, the packet is sent back to the vSwitch. (④) The vSwitch routes the packet to the next function in the chain (Fn-2). (⑤) The second function processes the packet and returns it to the vSwitch. (⑥) The vSwitch then routes the packet out through the NIC to the backend server.

Table I shows the overhead auditing for the L2/L3 scenarios (Fig. 1 (a), (b), (e), (f)). Table II shows the overhead auditing for the L4/L7 scenarios (Fig. 1 (c), (d), (g), (h)). We do not include the switching/routing overhead (*i.e.,* cycles spent on forwarding/routing table lookup), as it is a necessary operation to exchange packets between functions (either L2/L3 or L4/L7) and cannot be avoided. We have several key takeaways below drawn from our auditing of the packet flow.

Takeaway#1: *Using the **userspace vSwitch** in conjunction with **virtio-user/vhost-user** ((e) and (f)) saves a significant amount of overhead, and is preferred for L2/L3 NFs.*

The userspace vSwitch does not show a significant overhead difference compared to the kernel-based vSwitch when moving the packet between the vSwitch and the NIC (① and ⑥, see "Outside the chain" column in Table I). Compared to the userspace vSwitch (using DPDK for kernel-bypass), the kernel-based vSwitch incurs one additional interrupt when receiving packets from the NIC.

The advantage of the userspace vSwitch is the ability to work with userspace virtual device interfaces, *i.e., virtio-user/vhost-user*. Working in conjunction with *virtio-user/vhost-user*, the userspace vSwitch does not incur an interrupt or context switch when passing packets within the function chain (② to ⑤). On the other hand, the kernel-based vSwitch has to exchange the packet with the function in userspace through *virtio-user/vhost-net* & *TUN/TAP* ((a) and (b)), which incurs an interrupt and a context switch each time the packet crosses the kernel-userspace boundary (② to ⑤), a less desirable option. However, none of them avoid the data copies incurred when transmitting the packet within the chain (details below in **Takeaway#3**).

Takeaway#2: *Using the **kernel-based vSwitch** in conjunction with **veth** and container (d) incurs the least overhead for L4/L7 middleboxes.*

Just as with the L2/L3 NF use case, the use of different vSwitches in L4/L7 middlebox case to exchange packets between the NIC and middlebox (① and ⑥) does not have a significant difference. However, as L4/L7 middleboxes require kernel protocol processing, the kernel-based vSwitch has an advantage, as it can work seamlessly with the protocol stack in the host's kernel. Since containers share the host's kernel, it is ideal to follow the data plane model (d) and connect the kernel-based vSwitch with the container via the *veth pair*. As shown in Table II, each time when the packet is exchanged between the middlebox and the vSwitch (② to ⑤), (d) it saves 1 data copy and 1 context switch compared to (c), which also adopts the kernel-based vSwitch. As (c) uses *virtio-net/vhost-net* & *TUN/TAP* to connect VM and host's kernel, there is 1 data copy and 1 context switch involved.

The use of a userspace vSwitch along with the *virtio-*

TABLE I
OVERHEAD AUDITING OF L2/L3 NF DATA PLANE MODELS

Data pipeline No.			Outside the chain (NIC-vSwitch)		Within the chain (Fn-vSwitch-Fn)				total
			①	⑥	②	③	④	⑤	
# of copies	kernel-based vSwitch	(a)	0	0	1	1	1	1	4
		(b)	0	0	1	1	1	1	4
	userspace vSwitch	(e)	0	0	1	1	1	1	4
		(f)	0	0	1	1	1	1	4
# of interrupts	kernel-based vSwitch	(a)	1	0	1	1	1	1	5
		(b)	1	0	1	1	1	1	5
	userspace vSwitch	(e)	0	0	0	0	0	0	0
		(f)	0	0	0	0	0	0	0
# of context switch	kernel-based vSwitch	(a)	0	0	1	1	1	1	4
		(b)	0	0	1	1	1	1	4
	userspace vSwitch	(e)	0	0	0	0	0	0	0
		(f)	0	0	0	0	0	0	0

(a) kernel-based vSwitch + virtio-user/vhost-net & TUN/TAP + VM;
(b) kernel-based vSwitch + virtio-user/vhost-net & TUN/TAP + container;
(e) userspace vSwitch + virtio-user/vhost-user + VM;
(f) userspace vSwitch + virtio-user/vhost-user + container;

Note: Context switches may happen when two userspace processes (*e.g.,* the NF and the vSwitch) are placed on the same CPU core. However, in NFV scenario, NFs and the vSwitch are typically dedicated with a separate CPU core, owing to the need of high performance. We assume NFs and the vSwitch assigned with dedicated CPU core in the overhead auditing. *virtio-user* uses DPDK's PMD to send/receive packets. There is no interrupt involved.

TABLE II
OVERHEAD AUDITING OF L4/L7 MIDDLEBOX DATA PLANE MODELS

Data pipeline No.			Outside the chain (NIC-vSwitch)		Within the chain (Fn-vSwitch-Fn)				total
			①	⑥	②	③	④	⑤	
# of copies	kernel-based vSwitch	(c)	0	0	2	2	2	2	8
		(d)	0	0	1	1	1	1	4
	userspace vSwitch	(g)	0	0	2	2	2	2	8
		(h)	0	0	2	2	2	2	8
# of interrupts	kernel-based vSwitch	(c)	1	0	2	2	2	2	9
		(d)	1	0	2	2	2	2	9
	userspace vSwitch	(g)	0	0	2	2	2	2	8
		(h)	0	0	3	3	3	3	12
# of context switch	kernel-based vSwitch	(c)	0	0	2	2	2	2	8
		(d)	0	0	1	1	1	1	4
	userspace vSwitch	(g)	0	0	1	1	1	1	4
		(h)	0	0	2	2	2	2	8
# of protocol processing tasks	kernel-based vSwitch	(c)	0	0	1	1	1	1	4
		(d)	0	0	1	1	1	1	4
	userspace vSwitch	(g)	0	0	1	1	1	1	4
		(h)	0	0	1	1	1	1	4
# of serialization or deserialization (L7)	kernel-based vSwitch	(c)	0	0	1	1	1	1	4
		(d)	0	0	1	1	1	1	4
	userspace vSwitch	(g)	0	0	1	1	1	1	4
		(h)	0	0	1	1	1	1	4
# of L2/L3 processing tasks	kernel-based vSwitch	(c)	0	1	2	1	2	1	7
		(d)	0	1	1	0	1	0	3
	userspace vSwitch	(g)	0	1	2	1	2	1	7
		(h)	0	1	1	0	1	0	3

(c) kernel-based vSwitch + *virtio-net/vhost-net* & *TUN/TAP* + VM;
(d) kernel-based vSwitch + *veth* + container;
(g) userspace vSwitch + *virtio-net/vhost-user* + VM;
(h) userspace vSwitch + *virtio-user/vhost-net* & *TUN/TAP* + *veth* + container

user/vhost-net interface (h) is also less preferable than (d). (h) with the userspace vSwitch differs from (d) (which uses the kernel-based vSwitch) because packets have to be looped back from the vSwitch in userspace to the kernel for protocol processing. This incurs one more data copy, interrupt, and context switch compared to (d), as seen in Table II, resulting in poorer performance.

Using the userspace vSwitch and the *vhost-user* interface to work with a VM (g) is slightly better, as both the userspace vSwitch and the *vhost-user* interface work in the userspace, thus eliminating one context switch compared to using the *virtio-net/vhost-net & TUN/TAP* in (c). However, (g) still incurs an additional data copy because of the kernel-userspace boundary crossing within the VM. Moreover, as the packet has to traverse the entire VM's kernel stack in (c) and (g), there is unnecessary, duplicate L2/L3 processing involved in the VM's kernel in addition to the L2/L3 processing performed by the vSwitch in the host. This duplicate processing is avoided in (d) with the use of containers, which reuses the OS kernel from the host and avoids duplicate processing.

Takeaway#3: *Heavyweight service function chain for L2/L3 NFs and L4/L7 middleboxes.*

As shown in Table I and II, the major source of data plane overhead comes within the function chain (② to ⑤). Even with the best combination we identified for L2/L3 NFs (f) and L4/L7 middleboxes (d), there are excessive data copies within a service function chain with existing solutions. With the best L2/L3 solution (f), one data copy is incurred each time a packet is passed from the vSwitch to the NF (②, ④), and vice versa (③, ⑤). This also holds true for the best L4/L7 solution (d). The situation is worse for the L4/L7 case, as there are many additional overheads, including interrupts, context switches, protocol processing tasks, and serialization/deserialization tasks, that are incurred for the communication within the chain (② to ⑤).

Discussion: Containers share the host's kernel protocol stack, resulting in a smaller memory footprint than having a dedicated kernel stack in each VM. This becomes important with scale, as the number of NFs/middleboxes grows. The smaller footprint contributes to faster startup of containerized functions [15]. Containers also avoid duplicate L2/L3 processing for L4/L7 middleboxes (see **Takeaway#2**). For L2/L3 NFs, there is no significant difference in the data plane cost between VMs and containers (compare (e) and (f) in Table I). While we choose to work with containers, the design of MiddleNet is also generally applicable to a VM-based environment.

Data plane models (f) "userspace vSwitch + *virtio-user/vhost-user* + container" and (d) "kernel-based vSwitch + *veth* + container" are the best solution for L2/L3 NFs and L4/L7 middleboxes, respectively, as they introduce the minimal amount of overhead and are most lightweight against other alternatives. However, even the optimal data plane models are too heavyweight to construct the function chain for L2/L3 NFs and L4/L7 middleboxes. In fact, the overhead in the current service function chain design builds as the size of the function chain increases, which can result in significant performance loss. Unnecessary packet processing overhead is introduced in the data transfer between vSwitch and functions, as well as expensive protocol processing (for L4/L7 only). All these factors make it difficult for us to achieve a high-performance NFV/middlebox framework.

III. Shared memory communication in MiddleNet

Shared memory communication can alleviate the data movement overheads of the data plane within a function chain by keeping the data in a userspace memory pool to be shared by different functions in the chain. Fig. 3 shows a generalized data pipeline using shared memory communication in MiddleNet. It is a chain, with two functions (either L2/L3 NFs or L4/L7 middlebox functions), both on the same host. Steps ① and ⑥ move the packets between the NIC and shared memory, while ② to ⑤ pass packet descriptors between functions to achieve zero-copy packet delivery within the function chain. An intermediate component (running in userspace) is used to provide forwarding/routing support within the function chain, which is similar to the vSwtich in Fig. 1. We call this intermediate component the "NF manager" in the L2/L3 scenario, or "message broker" in the L4/L7 scenario. The NF manager/message broker is responsible for moving packets between the NIC and the shared memory in steps ① and ⑥.

Three key elements enable shared memory communication for a function chain: (1) **NIC-shared memory packet exchange**. An incoming packet is moved into the userspace shared memory prior to processing by the function chain (either L2/L3 NF chain or L4/L7 middlebox chain); (2) **Zero-copy I/O within the function chain**. Instead of moving the data from one function to another, shared memory communication achieves zero-copy I/O within the function chain, by passing a pointer, which is the packet descriptor, to the data in shared memory. This substantially reduces overhead; (3) **Shared memory support**. A memory pool is initialized and mapped to each function in the chain before it can be accessed. There are multiple alternatives, with significant differences, for the "NIC-shared memory packet exchange" and "zero-copy I/O within the function chain" operations, which we now describe *qualitatively*.

1) NIC-shared memory packet exchange: There are two distinct options: one approach bypasses the kernel, the other is a kernel-based approach. The kernel-bypass approach DMA's the packet to shared memory without involving the kernel stack. *Exploiting kernel-bypass avoids heavyweight kernel processing and is better suited for building L2/L3 NFs as a 'bump-in-the-wire'.* As discussed in §II-A, the kernel-bypass approach can be further classified into a polling-based kernel-bypass (*i.e.,* with DPDK's PMD) and event-driven kernel-

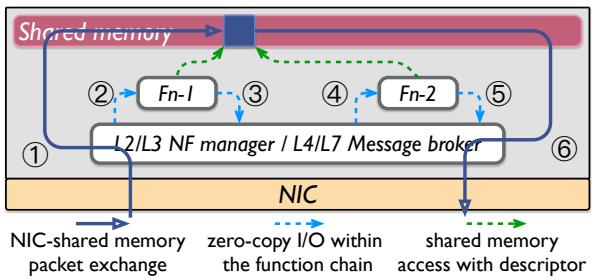


Fig. 3. A generalized shared memory communication data pipeline for a function chain in MiddleNet. Note: we only show the client-to-server datapath

bypass (*i.e.,* using AF_XDP). The NF manager (Fig. 3) works with these kernel-bypass alternatives to move packets between the NIC and shared memory (details in §IV-B and §IV-C).

The kernel-based approach, on the other hand, uses the kernel stack to pass packets between the NIC and the message broker in the userspace. The message broker exchanges packets with the kernel stack via the Linux socket interface. It then moves packets to shared memory for zero-copy processing within the function chain. This inevitably introduces overheads (*e.g.,* copy, context switch, etc) when a packet crosses the kernel-userspace boundary. It also incurs the overhead of kernel protocol layer processing, which is only useful for L4/L7 middleboxes. *The kernel-based approach is ideal for L4/L7 middleboxes, as it provides necessary processing using a full-function kernel protocol stack.*

2) Zero-copy I/O for function chaining: Zero-copy I/O for function chaining can also be broadly implemented using either: (1) polling-based zero-copy I/O, *e.g.,* DPDK's RTE RING [11]; or (2) event-driven zero-copy I/O, *e.g.,* eBPF's `SKMSG` [8]. It's important to understand the difference between these two options and their impact on performance.

eBPF's `SKMSG` is a socket-related eBPF program type, "`BPF_PROG_TYPE_SK_MSG`" [8]. `SKMSG` is attached to the socket of the function during its creation. It processes packets sent/received on the attached socket to/from the kernel. The execution of `SKMSG` is triggered by the arrival of a packet, which is strictly event-driven and is thus load-proportional. Working in conjunction with the eBPF socket map (`BPF_MAP_TYPE_SOCKMAP` [21]), which provides necessary routing information, `SKMSG` can deliver packet descriptors between functions. The other option, DPDK's RTE RING, is implemented as a circular FIFO queue, used for buffering packet descriptors. Dedicated for each function is a Receive (RX) and Transmit (TX) ring pair to pass packet descriptors using polling.[3] A function polls its own RX ring (using `rte_ring_dequeue()`) to receive packet descriptors and enqueue packet descriptors to its TX ring (using `rte_ring_enqueue()`) for transmission. A centralized routing component on the other side polls the TX ring of each function and moves queued packet descriptors to the RX ring of the destination function, based on its internal routing table.

3) Shared memory support: MiddleNet uses DPDK's multi-process support [22] to construct shared memory between functions within a service chain. We utilize a shared memory manager (running as a DPDK primary process[4]) to manage shared memory pools. During the initialization stage of MiddleNet, the shared memory manager in MiddleNet creates a private memory pool, with a unique "shared data file prefix" specified to isolate with other shared memory pools on the same node. The "shared data file prefix" is used by DPDK's EAL to create hugepage files (*i.e.,* actual file system objects for DPDK's memory pools) in the Linux file system. A DPDK process is allowed to access a hugepage file, only if the same file prefix was specified during its creation. Additional details are in Appendix-A, including shared memory support for VM-based functions. We leverage this feature to build a security domain for MiddleNet that enhances the security of using shared memory for communication between NFs (see §VII).

Each key element described is independent of the other, *e.g.,* using DPDK's multi-process doesn't require DPDK's PMD. So using DPDK's multi-process support to manage memory sharing between different functions incurs no polling overhead.

Overhead Auditing & Discussion: We perform overhead auditing of the function chain using shared memory communication. We consider two distinct approaches for both the L2/L3 NFs and L4/L7 middleboxes use cases: the polling-based approach (using DPDK's PMD and RTE RING), and the event-driven approach (using eBPF's AF_XDP and `SKMSG`).

To conserve space, we have summarized the main takeaways here. A detailed overhead auditing of function chains using shared memory can be found in Appendix-B. The overhead auditing clearly shows the advantage of using shared memory communication, to reduce the overhead in almost every dimension (*e.g.,* data copy, interrupt, context switch, etc). Thus, we factor it into our NFV/middlebox framework, MiddleNet. It is clear that L2/L3 MiddleNet should consider *kernel-bypass* NIC-shared memory packet exchange to facilitate high performance. L4/L7 MiddleNet adopts *kernel-based* NIC-shared memory packet exchange to provide the needed protocol processing. We understand the trade-off between a polling-based solution and an event-driven solution by implementing the alternatives, and evaluating their performance, to help us decide which to use for MiddleNet.

IV. Design of MiddleNet: L2/L3 NFV

We discuss the eBPF-based and DPDK-based 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

NIC-userspace kernel-bypass: MiddleNet takes full advantage of zero-copy packet delivery and kernel-bypass to move packets between the NIC and the userspace shared memory, so as to minimize overheads, reduce resource consumption, and achieve full line-rate L2/L3 packet processing (§III-1). We consider two kernel-bypass alternatives: polling-based DPDK's PMD and event-driven AF_XDP (§II-A).

Zero-copy I/O for function chaining: We evaluate two alternatives for L2/L3 MiddleNet, the *polling-based* approach and the *event-driven* approach. The polling-based alternative adopts DPDK's PMD for NIC-to-userspace delivery using kernel-bypass and DPDK's RTE RING for function chaining. The event-driven alternative adopts AF_XDP for NIC-to-userspace kernel-bypass and `SKMSG` for function chains. This helps us evaluate the trade-off between performance and resource efficiency when using a polling-based design or an event-driven design to achieve a 'bump-in-the-wire' L2/L3 NFV environment. Both of them use DPDK's multi-process support to manage the shared memory of L2/L3 MiddleNet (§III-3). We implement these two alternatives based

[3]Note: Polling the RTE ring does not require the simultaneous use of DPDK's PMD. It can be simply implemented as a `while` loop.

[4]The DPDK primary process has privileges, enabling it to initialize memory pools in huge pages.

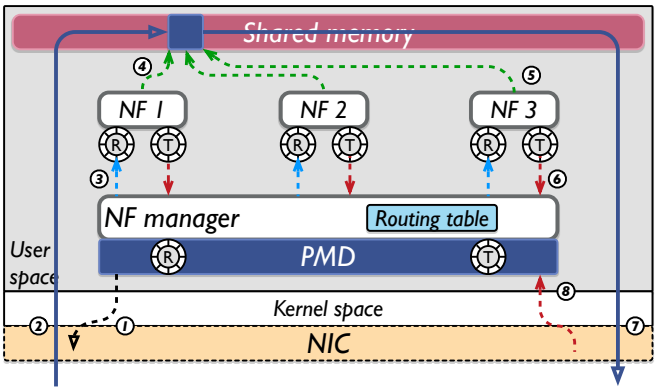


Fig. 4. Packet processing flow for DPDK-based L2/L3 NFV: RX and TX

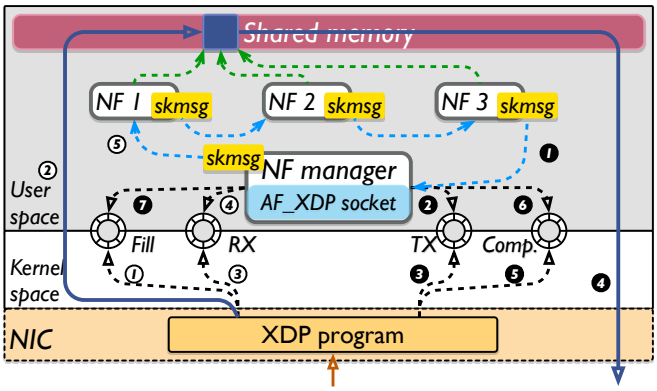


Fig. 5. Packet processing flow for eBPF-based L2/L3 NFV: RX and TX

on OpenNetVM's design [3], that is similar in principle to the design described in Fig. 3, §III.

B. The DPDK-based L2/L3 NFV design

The DPDK-based approach can be 'expensive' in having dedicated CPU cores for polling. In addition to the NF manager that dedicates one CPU core for the PMD, for each NF of the L2/L3 function chain, one CPU core is used up for each function to poll its RTE RING. This can be wasteful if incoming traffic is low. Somewhat more complex NFV support, such as NFVnice [23], can be used to mitigate these overheads by sharing a CPU core across multiple NFs.

Fig. 4 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 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.

Service function chains: The NF manager utilizes destination information in the packet descriptor to support routing within an NF chain for the DPDK-based approach. The routing table in the NF manager is used to resolve that NF's ID, thus avoiding the need for each NF 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 NF 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

The NF manager in the eBPF-based L2/L3 MiddleNet opens a dedicated AF_XDP socket (*i.e.,* `XSK` [7]) 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. AF_XDP depends on interrupts triggered by the event execution of the XDP program attached to the NIC driver (Fig. 5). This interrupt notifies the packet processing component in userspace. However, these interrupts have to be managed with care to avoid poor overload behavior when subjected to high packet rates [12].

Fig. 5 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 DMA'ed 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). The system call by the NF manager (typically `sendmsg()`) notifies the kernel about the TX event (❸). The kernel then transmits the

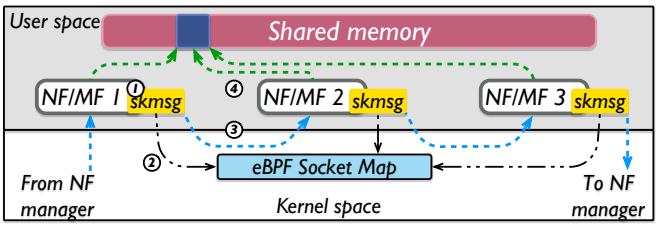


Fig. 6. Function chaining in MiddleNet: eBPF-based approach

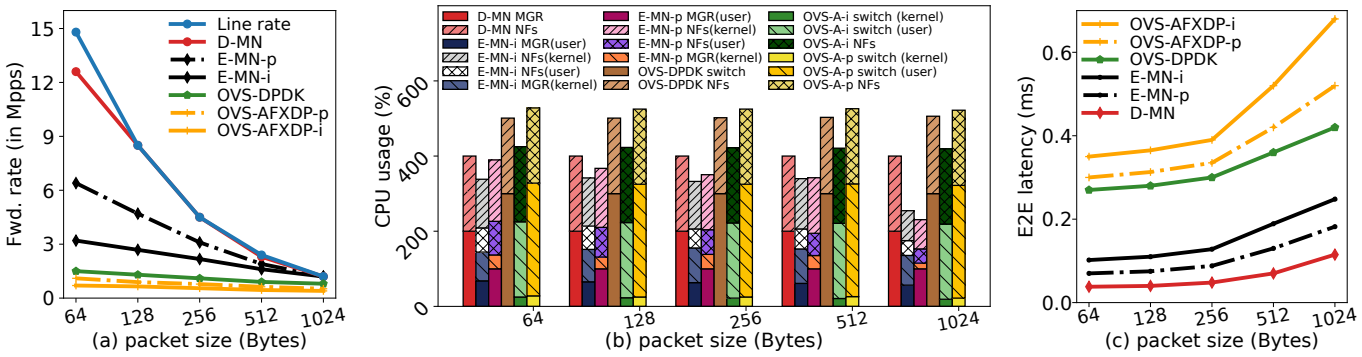


Fig. 7. Comparison between different L2/L3 alternatives: (a) Maximum loss free rate (MLFR) under different packet sizes, (b) CPU usage under MLFR under different packet sizes, (c) end-to-end latency under MLFR under different packet sizes. **Note:** *D-MN* refers to *D-MiddleNet*; *E-MN-i* refers to *E-MiddleNet* with interrupt-driven AF_XDP socket; *E-MN-p* refers to *E-MiddleNet* with polling-based AF_XDP socket; *OVS-A-i* refers to OVS-AF_XDP with interrupt-driven AF_XDP socket; *OVS-A-p* refers to OVS-AF_XDP with polling-based AF_XDP socket.

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).

Service function chains: The eBPF-based L2/L3 approach uses `SKMSG` to support NF chains. 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. 6, the NF creates a packet descriptor to be sent (①). The `SKMSG` performs a lookup in the socket map to determine the destination socket (②). It then redirects the packet descriptor to the next NF (③). That NF uses the descriptor to access data in shared memory (④) and passes the packet descriptor to the next NF through `SKMSG` after processing.

D. Performance evaluation

Experiment setup: We compare the performance of DPDK (*i.e.,* polling-based, hereafter referred to as *D-MiddleNet*) and eBPF (hereafter referred to as *E-MiddleNet*) 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. We use the data plane model (f) in §II-A as the primary baseline to compare with. For this, we choose two implementations of Open vSwitch as the kernel-bypass vSwitch in (f): OVS-DPDK [16] and OVS-AF_XDP [17]. We set up our experiments on NSF Cloudlab [24] with three nodes: the 1st node is configured with a *Pktgen* [25] load generator for L2/L3 NFV use case; the 2nd node is configured with MiddleNet alternatives (*D-MiddleNet*, *E-MiddleNet*) and OVS alternatives (OVS-DPDK, OVS-AF_XDP). 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 v21.11 [2] and *libbpf* [26] v0.6.0 for eBPF-related experiments.

To achieve the best possible performance for OVS-DPDK and OVS-AF_XDP baselines, we enable the "Multiple Poll-Mode Driver Threads" [27] feature in OVS. Each PMD thread runs on a dedicated CPU core and continually polls the physical NIC or the *vhost-user* (Fig. 1 (f)) to process incoming packets. OVS-AF_XDP uses polling to retrieve packets from the NIC by default. For this polling-based OVS-AF_XDP option (OVS-AF_XDP-p, Fig. 1 (f)), and OVS-DPDK, we create three PMD threads to achieve the highest performance. We additionally configure the AF_XDP socket in OVS-AF_XDP to run in the interrupt mode (*i.e.,* OVS-AF_XDP-i) [28].[5] This helps to move packets between NIC and userspace OVS in an event-driven manner. But, to achieve the optimal packet exchange performance between OVS-AF_XDP-i and NFs, we use polling to avoid interrupt overheads for packet exchanges between OVS and the NFs. Only a data copy overhead is incurred between OVS and the NFs when using polling on both sides. For this, we create two PMD threads to poll packets for getting packets to and from NFs (via *vhost-user*). For NFs in both the OVS-DPDK and OVS-AF_XDP setups, each *virtio-user* is dedicated with a CPU core to poll packets from OVS. We also configure the AF_XDP socket in *E-MiddleNet* to operate in polling mode (*E-MiddleNet*-p) and compare with the interrupt-based AF_XDP socket (*E-MiddleNet*-i).

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. We collect the average value measured across 5 repetitions. Each run is for 60 seconds.

Discussion: Fig. 7(a) shows the MLFR for different alternatives. *D-MiddleNet* achieves almost the line rate for different packet sizes. The exception is for packet sizes of 64Bytes,

[5]To enable the interrupt mode for AF_XDP, a user needs to specify the device type of the physical NIC as "afxdp-nonpmd" when attaching it to OVS.

achieving 12.6M packets/sec (84% of line rate) because of our limit on the number of CPU cores for the NF Manager and the PMD. Even with the limited CPU cores, *D-MiddleNet* outperforms both *E-MiddleNet*-i and *E-MiddleNet*-p. For a packet size of 64Bytes, *E-MiddleNet*-i is limited to a forwarding rate of 3.2 Mpps (only 25% of *D-MiddleNet*) while *E-MiddleNet*-p is limited to a forwarding rate of 6.3 Mpps (50% of *D-MiddleNet*). 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 [12]. The performance of *E-MiddleNet*-i is limited by its overheads, including a number of interrupts and context switches (Appendix-B). As we observe in Fig. 7(b), *E-MiddleNet*-i's NF manager and the NFs themselves spent most of the CPU time in the kernel (53% for the NF manager, 67% for NFs) to handle interrupts generated by AF_XDP socket or `SKMSG`, thus leaving fewer resources to perform the NF packet forwarding tasks. *E-MiddleNet*-p reduces interrupts by operating the AF_XDP socket in polling mode, which helps it achieve better throughput compared to *E-MiddleNet*-i. But, the performance of *E-MiddleNet*-p is still worse than *D-MiddleNet* as the execution of XDP program in the NIC driver is triggered by interrupts, in addition to the `SKMSG` overhead, all of which negatively impact the packet forwarding performance. Although devoting more resources to *E-MiddleNet*'s NF manager and the NFs may alleviate this overload, it only postpones the problem when 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. Focusing on the end-to-end packet latency, *D-MiddleNet* achieves a 2.6× improvement compared to *E-MiddleNet*-i, and is 1.8× better compared to *E-MiddleNet*-p (Fig. 7(c)).

Note that as the packet size increases, the CPU usage of both *E-MiddleNet*-i and *E-MiddleNet*-p is even lower compared to the other options. For example, at a packet size of 1024Bytes, the CPU usage of *E-MiddleNet*-i and *E-MiddleNet*-p are 63% and 58% of *D-MiddleNet*, respectively. Since *E-MiddleNet*-i and *E-MiddleNet*-p use *event-driven* shared memory communication, as the packet size increases and the packet rate decreases (bounded by the line rate of the NIC used in this experiment). The overhead for *E-MiddleNet*-i and *E-MiddleNet*-p, which is strictly proportional to the packet rate, diminishes. Thus the CPU overhead reduces for larger packet sizes for *E-MiddleNet*-i and *E-MiddleNet*-p, which makes the event-driven design attractive for larger packet sizes for L2/L3 NFs. However, the event-driven approach still suffers from poor performance and relatively high CPU usage in handling L2/L3 traffic with smaller packet sizes. On the other hand, *D-MiddleNet* maintains good performance across a range of packet sizes. Further, *D-MiddleNet* can utilize the scheduling principles in NFVnice [23] to reduce the CPU consumption by multiplexing a CPU core across multiple NFs.

Both *D-MiddleNet* and *E-MiddleNet* outperform OVS-DPDK and OVS-AF_XDP in terms of MLFR for receiving packets and latency. Looking at the CPU usage of OVS-DPDK, even though OVS-DPDK dedicates enough CPU resources (3 CPU cores for the OVS switch, one CPU core per NF) to achieve the best performance, the forwarding rate for it is worse than *E-MiddleNet*. This shows the negative impact of excessive data copies within the chain (§II-C). Even though *E-MiddleNet* also incurs interrupts and context switches in the data pipeline, as shown in Fig. 3, its exploitation of shared memory communication fundamentally improves the data plane performance of function chains, as discussed in Appendix-B. OVS-AF_XDP on the other hand performs poorly. Running OVS-AF_XDP in polling mode (OVS-AF_XDP-p) improves throughput and reduces latency compared to running OVS-AF_XDP in interrupt mode. This is because OVS-AF_XDP-i suffers the overhead of interrupts and context switches for moving packets between the NIC and userspace, just like *E-MiddleNet*-i. But the improvement of OVS-AF_XDP-p is limited, particularly because of the data copy overhead within the chain.

D-MiddleNet does constantly consume considerable CPU (one CPU core per NF, 2 CPU cores for the NF manager). While this is a concern, its superior performance makes it more attractive for L2/L3 NFs, since they have to act like a 'bump-in-the-wire'. *E-MiddleNet* is less attractive because of its poor overload behavior.

V. 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

Protocol processing support: Unlike L2/L3 NFs, packets pass through the kernel for the required protocol layer processing for L4/L7 middleboxes. L4/L7 MiddleNet uses a message broker (Fig. 8) to leverage the protocol processing in the kernel stack. Incoming packets processed by the kernel network protocol stack are delivered through a socket to a message broker in userspace. This comes at a cost, but MiddleNet benefits significantly from a fully functional in-kernel protocol stack for L4/L7 middleboxes. For a more detailed discussion of kernel protocol processing costs, refer to Appendix-B.

Zero-copy I/O for function chaining & shared memory support: We follow a similar methodology as in §IV to evaluate what is the most suited zero-copy I/O capability for function chains in L4/L7 MiddleNet. For the eBPF-based L4/L7 middlebox design, packets are forwarded between middlebox functions (hereafter referred to as MFs) using

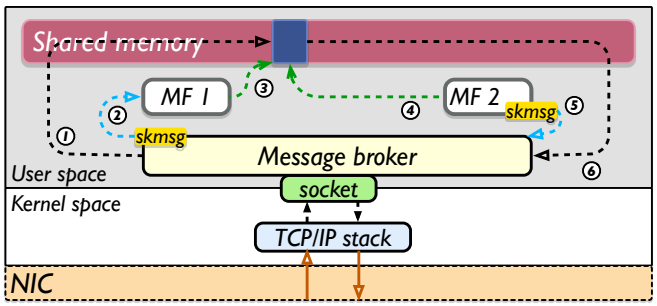


Fig. 8. Packet processing flow for eBPF-based L4/L7 middleboxes

eBPF's SKMSG capability. For DPDK-based L4/L7 middlebox functionality, the message broker delivers descriptor entries to the ring of the target MF, with the payload in shared memory, after protocol processing by the message broker.

B. The eBPF-based L4/L7 middlebox design

Fig. 8 depicts the packet flow for the eBPF-based L4/L7 MiddleNet. For inbound traffic, after the payload is moved into shared memory by the message broker (①), a packet descriptor is sent to the target MF via SKMSG (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), it uses 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: The eBPF-based L4/L7 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. 6. Although the eBPF-based L4/L7 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 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

To leverage the kernel protocol stack, we restructure the NF manager of the L2/L3 use case (Fig. 4) into a message broker in the DPDK-based L4/L7 MiddleNet. 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).

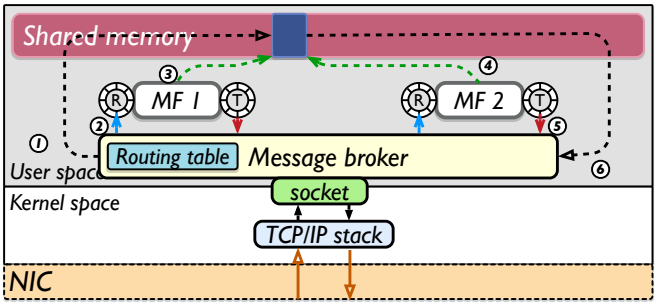


Fig. 9. Packet processing flow for DPDK-based L4/L7 middleboxes

Function chain support: The function chain support in the DPDK-based L4/L7 MiddleNet is the same as in the DPDK-based L2/L3 NFV use case (§IV-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 MiddleNet (Fig. 8, hereafter referred to as *E-MiddleNet*) and the DPDK-based L4/L7 MiddleNet implementation (Fig. 9, hereafter referred to as *D-MiddleNet*). As a third alternative, we use an NGINX proxy to study the impact of the loosely-coupled function chain (thus supporting a microservices paradigm) design in 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 achieve good data plane performance with a static, monolithic function implementation. We also use the data plane model in Fig. 1 (d) (hereafter referred to as *K-vSwitch*), as an additional alternative to compare with. We choose the Linux bridge as the implementation of the kernel-based vSwitch in Fig. 1 (d). While the in-kernel OVS bridge could be another option, the Linux bridge offers all the functionality of a vSwitch for our evaluation purposes and is natively supported in Linux. In addition, the performance difference between Linux bridge and the in-kernel OVS bridge is not considered to be significant [29], [30]. It has also been noted that the in-kernel OVS bridge has difficulties being maintained as a separate project in addition to Linux kernel [17]. We reuse most of the testbed setup described in §IV-D.

We consider a typical HTTP workload (Apache Benchmark [31]) 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 generate HTTP workloads. The 2nd node is configured with the MiddleNet system. On the 3rd node, we configure two NGINX [32] instances as web servers. We enable adaptive batching for *E-MiddleNet* to minimize the overhead incurred by frequent SKMSG interrupts within the chain 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 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 *D-MiddleNet* and *E-MiddleNet*, 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 [33]) 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

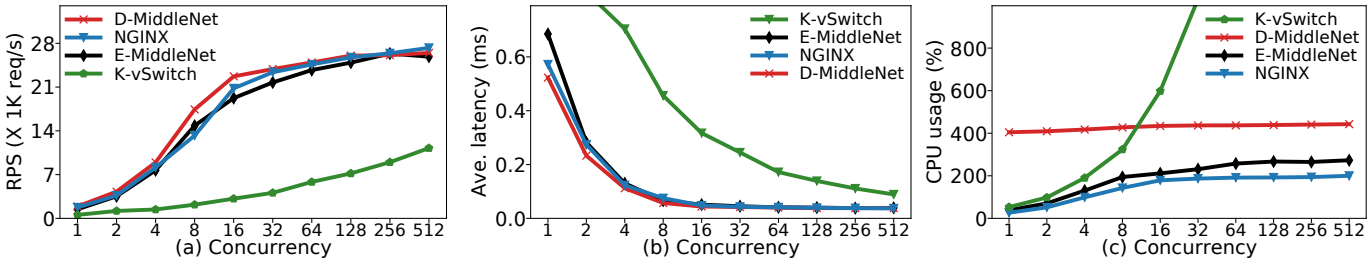


Fig. 10. RPS (a), latency (b) and CPU usage (c) comparison between different L4/L7 middlebox approaches. Note: The CPU usage of the data plane model (d), in Fig. 1, exceeds 10 CPU cores at concurrency level 32 and consumes 30 CPU cores at concurrency level 512.

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. 10 compares the RPS, response latency, and CPU usage of the different alternatives. *K-vSwitch* has the lowest performance and highest CPU usage compared to the others. At a concurrency level of 512, the RPS of *K-vSwitch* is only ∼42% of the others, while its latency is ∼2.3× higher. The CPU usage of *K-vSwitch* is even higher than *D-MiddleNet* for concurrency levels greater than 16. This demonstrates the heavyweight nature of the service function chain as discussed in §II-C and demonstrates the benefit of having a zero-copy function chain of the MiddleNet alternatives.

The use of `SKMSG` in *E-MiddleNet* leads to slightly worse latency and throughput than *D-MiddleNet*. When the concurrency is between 1 and 32, there is a throughput difference between *D-MiddleNet* and *E-MiddleNet*, ranging from 1.09× to 1.3×. At the lowest concurrency level of 1, *E-MiddleNet* consumes 37% of the CPU, which is a 10× reduction compared to *D-MiddleNet* (404%, *i.e.,* 4 CPU cores). Since *D-MiddleNet* uses polling to deliver packet descriptors, it continuously consumes CPU resources even when the traffic load is low, resulting in wasted CPU resources. Although *D-MiddleNet* achieves 1.3× better RPS and latency compared to the *E-MiddleNet* at a concurrency of 1, *E-MiddleNet*'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 *D-MiddleNet*'s constant usage of CPU. Thus, it is more desirable to use the lightweight *E-MiddleNet* approach for these light loads.

When the concurrency level increases and the load is higher, the adaptive batching of the *E-MiddleNet* approach amortizes the interrupt and context switch overheads. The performance gap between *E-MiddleNet* 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.

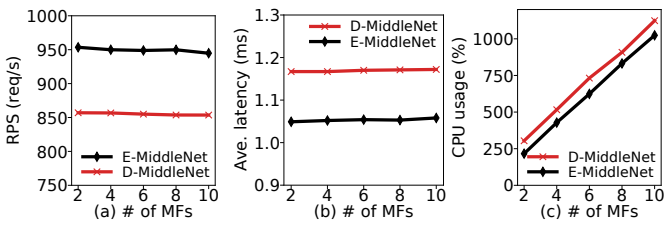


Fig. 11. RPS (a), latency (b) and total CPU usage (c) comparison with increasing number of CPU-intensive MFs in the chain.

Compared to a monolithic NGINX as a middlebox, the *E-MiddleNet* 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 `SKMSG`, which has additional context switches and interrupts. However, running a set of middleboxes 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 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. 11 evaluates the scalability of *D-MiddleNet* and *E-MiddleNet* with CPU-intensive MFs. Both *D-MiddleNet* and *E-MiddleNet* show good scalability as the number of MFs increases. Surprisingly, *E-MiddleNet* performs even better than *D-MiddleNet* with CPU-intensive 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 *D-MiddleNet*'s MF. But for *E-MiddleNet*, 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 *E-MiddleNet*'s MFs, it is able to fully utilize the assigned CPU core, improving its performance. To improve *D-MiddleNet*'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, *D-MiddleNet* always needs one more CPU core than *E-MiddleNet* (Fig. 11(c)). The extra CPU usage of *D-MiddleNet* 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 *D-MiddleNet* uses resources inefficiently for this case, when dealing with CPU-intensive functions.

Throughout these experiments, *E-MiddleNet* has significant resource savings at different concurrency levels compared to *D-MiddleNet*, while having comparable throughput. Further, *E-MiddleNet* can even achieve better performance than *D-*

MiddleNet 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.

VI. 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 [13] allows multiple Virtual Functions (VFs) on a shared NIC, as depicted in Fig. 12. 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 MiddleNet and L4/L7 MiddleNet without having anyone take up the entire physical NIC. The aggregate **NIC** performance will still be at the line rate. MiddleNet uses the Flow Bifurcation mechanism [34] 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.

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 middlebox functions (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.

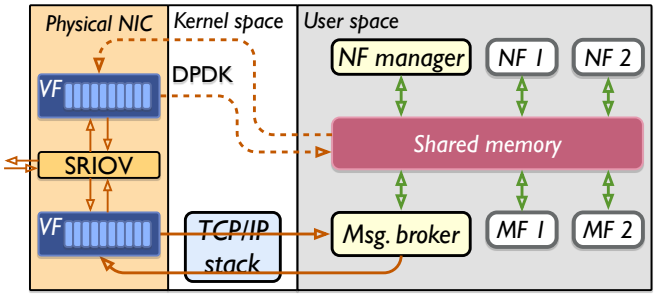


Fig. 12. The overall architecture of MiddleNet: A Combination of DPDK and eBPF via SR-IOV.

TABLE III
Overhead auditing of unified designs

	NIC switch in SR-IOV	*virtio-user/vhost-net & TUN/TAP*
# of interrupts	2	2
# of copies	1	2
# of context switch	1	2

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. There are two options for bridging the L2/L3 MiddleNet and L4/L7 MiddleNet: (1) A hardware-based approach that utilizes the NIC switch feature offered by SR-IOV [35] to connect different VFs within the NIC;[6] (2) A software-based approach that uses *virtio-user/vhost-net* & *TUN/TAP* device interfaces to connect L2/L3 MiddleNet to the kernel stack (see Fig. 1 (b)), which is then connected to L4/L7 MiddleNet.[7]

Table III compares the overhead generated by different alternatives. We only audit the datapath overhead between the NF manager in L2/L3 and the message broker in L4/L7, as they are the entry point of L2/L3 and L4/L7 MiddleNet. The hardware-based approach seamlessly works with the kernel-bypass in L2/L3 MiddleNet and moves the packet from the L2/L3 MiddleNet to the NIC via DMA. The NIC switch forwards the packet to the VF attached to the kernel stack without incurring any CPU overhead. All the overhead in the hardware-based approach is caused by passing the packet from the kernel stack to the message broker, however, is still less than software-based approach. The software-based approach inevitably introduces extra overhead and may compromise the performance gain achieved by L2/L3 kernel bypass. Based on the overhead auditing, we decide to 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, for both L2/L3 NFs and L4/L7 MFs to operate on the same flow.

B. Performance evaluation of unified design

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 [25]) and L4/L7 (Apache Benchmark [31]), 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

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

[7]DPDK's Kernel NIC Interface (KNI [36]) is another software-based approach that provides equivalent functionality as *virtio-user/vhost-net* & *TUN/TAP*. However, KNI lacks several important features compared to *virtio-user/vhost-net* & *TUN/TAP*, such as multi-queue support, checksum offloading, *etc.* This makes the performance of KNI not as comparable as *virtio-user/vhost-net* & *TUN/TAP* [37].

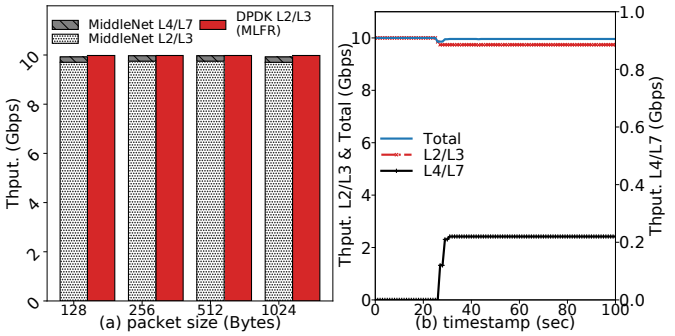


Fig. 13. (a) Aggregate throughput for various packet sizes. For L2/L3 NFV, we use Maximum loss free rate (MLFR) (b) Time series of throughput for L2/L3 NFV and total (left Y-axis) and L4/L7 middlebox (right Y-axis).

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 2nd node as described in §IV-D and §V-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. 13(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. 13(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.

VII. Isolation and Security Domains in MiddleNet

The use of shared memory raises concerns as it may weaken the isolation/security boundary between the functions that share the same memory region. Our trust model assumes that only functions in MiddleNet trust each other. Functions in MiddleNet (NFs or MFs), which run as DPDK secondary processes, share the same private memory pool by using the same "shared data file prefix" (specified by the shared memory manager (§IV-A)) during their startup. We 'admission control' functions by validating the creation of a MiddleNet function that is authenticated and uses the correct file prefix. We additionally apply inter-function packet descriptor filtering to prevent unauthorized access to the data in shared memory, through the virtual address in the packet descriptor. In accordance with the way packet descriptors are passed, these are different for L2/L3 (with DPDK's RTE ring) MiddleNet versus L4/L7 (with eBPF's `SKMSG`) MiddleNet.

Descriptor filtering for L2/L3 NFs: We leverage the NF manager in L2/L3 MiddleNet to perform packet descriptor filtering. Once the NF manager polls a new packet descriptor from an NF's TX ring, it queries its internal filtering map and checks whether the packet descriptor is authorized to be sent to the target NF based on matched rules. Unauthorized packet descriptors are dropped by the NF manager.

Descriptor filtering in L4/L7: Since the L4/L7 MiddleNet uses `SKMSG` to pass packet descriptors between functions (§V-B), it is natural to exploit eBPF's extensibility to filter packet descriptors. We add an additional eBPF map to the `SKMSG` program to store filtering rules. Each time a packet descriptor arrives, the `SKMSG` program parses the destination of the packet descriptor and uses it as the key to lookup the filtering rule. The packet descriptor is passed to the destination if allowed; otherwise, the descriptor is recognized as unauthorized and discarded.

VIII. Related Work

NFV platforms use different implementation approaches and primarily operate at L2/L3. OpenNetVM [3], 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 [38] and NetMap [39] 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 [12]. In contrast, the L2/L3 processing in MiddleNet uses polling, thus avoiding receive-livelocks. E2 [40] 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 NFs. 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], [18], [41]. However, each of these proposals has its difficulties. mOS [41] focuses on developing a monolithic middlebox, lacking the flexibility of a disaggregated design like MiddleNet. Microboxes [18] leverages DPDK and OpenNetVM's shared memory design to improve packet processing performance and achieve flexible middlebox 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 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 shared memory processing, in the meanwhile, adopting eBPF-based event-driven 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.

eBPF-based NFV/Middlebox: [42]–[44] explore the use of eBPF to implement NFV/Middlebox functions. These eBPF-based functions reside in the kernel, running as a set of eBPF programs attached at various eBPF hooks, *e.g.,* eXpress Data Path (XDP), and Traffic Control (TC). This avoids expensive context switches, as packet processing always remains within the kernel. In addition, since the packet payload is retained in the kernel buffers. Only the packet metadata,[8] which contains packet descriptor, is passed between different eBPF-based functions, thus achieving zero-copy packet delivery in the kernel. Compared to MiddleNet, [42]–[44] focus on the affinity in the kernel. In contrast, L2/L3 MiddleNet relies on DPDK, which uses SR-IOV to achieve a unified design. [42]–[44] can seamlessly work with the kernel protocol stack for protocol processing. However, the eBPF-based functions in [42]–[44] are triggered using kernel interrupts, thus potentially suffering from poor overload behavior [12]. Thus, their approach can perform poorly compared to L2/L3 MiddleNet, which leverages DPDK to achieve line-rate performance. Additionally, the eBPF-based functions can only be used to support L2/L3/L4 use cases within the kernel. Since L7 middleboxes not only require protocol processing, but have application code that typically run in userspace, approaches as in [42]–[44] result in expensive packet transfers between the kernel performing packet processing and the L7 userspace application. The shared memory design in L4/L7 MiddleNet avoids this overhead, thus achieving better data plane performance for a unified L4/L7 environment.

IX. Conclusion

We presented MiddleNet, a unified environment supporting L2/L3 NFV functionality and L4/L7 middleboxes. 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 NF chains and eBPF-based L4/L7 middlebox chains on the same node. This can bring substantial deployment flexibility.

[8]The packet metadata is represented as a "xdp_md" data structure when using the XDP hook, and is in the form of a "sk_buff" data structure when using TC hook.

Acknowledgments

We thank US National Science Foundation for their generous support through grants CRI-1823270 and CSR-1763929.

Appendix A
Details of DPDK's shared memory support

After the DPDK primary process (*i.e.,* shared memory manager) initializes the memory pools, it writes the memory pool information (*e.g.,* base virtual address, the allocated huge pages) into a configuration file through DPDK's EAL (Environment Abstraction Layer [45]). The DPDK secondary processes (*i.e.,* functions, L2/L3 NF manager, L4/L7 message broker) read the configuration file during startup and use DPDK's EAL to map the same memory regions allocated by the DPDK primary process. This ensures all the DPDK secondary processes share the same memory pools, thereby facilitating shared memory communication between functions.

When VMs are used, they rely on the emulated PCI to access physical memory in the host. This requires multiple address translations (*i.e.,* Guest Virtual Address to Guest Physical Address and then to Host Virtual Address). This adds a burden while sharing memory across different VMs, since they have different virtual address mappings to the host. It requires the hypervisor (as it knows the virtual address mappings of different VMs) to remap the base virtual address in the packet descriptor, which adds additional processing latency. In contrast, a container shares the same virtual memory address, which means that its virtual address can be interpreted by other containers without an additional translation. This facilitates memory sharing between different functions implemented in containers and makes it straightforward to build shared memory for function chains using existing tools such as DPDK's multi-process support.

Appendix B
Overhead auditing of function chains using shared memory

To *quantitatively* understand the benefit of shared memory communication and the difference between alternatives, we now perform an auditing of the overheads for the function chain in Fig. 3.

(1) L2/L3 NF use case: For the L2/L3 NF use case, we study two alternatives: first is (α) NIC-shared memory packet

exchange with *polling*-based kernel-bypass (using DPDK's PMD) + *polling*-based zero-copy I/O for function chaining (using DPDK's RTE RING); second is (β) NIC-shared memory packet exchange with *event-driven* kernel-bypass (using eBPF's AF_XDP) + *event-driven* zero-copy I/O for function chaining (using eBPF's `SKMSG`). We skip the *kernel-based* NIC-shared memory packet exchange in this auditing, as it is apparently unsuitable for L2/L3 NFs.

Table IV shows the overhead auditing of L2/L3 NF scenario for both ((α) and (β)). Compared to the optimal L2/L3 data plane model (f) discussed in §II-C, the polling-based shared memory communication approach (α) avoids any data copy, interrupt, and context switch, throughout the entire data pipeline (from ① to ⑥ of Fig. 3). The event-driven alternative (β) eliminates all the data copies as well. However, the use of AF_XDP and `SKMSG` introduces additional interrupts and context switches. In particular, every packet transfer within the chain incurs one interrupt and context switch, which is a non-negligible overhead, especially if the chain grows in scale.

(2) L4/L7 middlebox use case: For the L4/L7 middlebox use case, we study two alternatives: (γ) *kernel-based* NIC-shared memory packet exchange + *polling*-based zero-copy I/O for function chaining (using DPDK's RTE RING); (δ) *kernel-based* NIC-shared memory packet exchange + *event-driven* zero-copy I/O for function chaining (using eBPF's `SKMSG`). We skip the *kernel-bypass* NIC-shared memory packet exchange in this auditing, as L4/L7 middleboxes depend on the kernel stack for protocol processing.

Table V shows the overhead auditing of L4/L7 middlebox options ((γ) and (δ)). Compared to the optimal L4/L7 data plane model (d) in §II-C, the polling-based (γ) and event-driven (δ) shared memory communication approaches avoid any data copy within the function chain (② to ⑤ in Fig. 3), because of the zero-copy I/O. However, moving a packet from the NIC to shared memory (① in Table V) incurs two data copies, and vice versa (⑥ in Table V). One data copy comes from the packet exchange between the NIC and the message broker (Fig. 3), where the kernel stack needs to copy the packet from the kernel to the message broker in userspace, after protocol processing. The message broker then moves the packet into shared memory, which introduces the second copy. With the middlebox chain of two functions, using shared memory communication ((γ) or (δ)) shows no significant benefit compared to optimal L4/L7 data plane model (d) because of the data copy incurred when moving packets between the NIC and shared memory. They all introduce 4 data copies throughout the entire data pipeline (from ① to ⑥ in Fig. 3 and Fig. 2). The shared memory communication for the L4/L7 middlebox scenario ((γ), (δ)) shows its advantages of saving on data copies (due to the zero-copy I/O) compared to the L4/L7 data plane model (d) only when the size of the chain grows. In comparison, the data copy overhead in (d) will increase as the chain increases.

Another essential asset of shared memory communication is that it completely eliminates protocol processing, serialization, and deserialization overheads within the chain. These tasks are performed before the packet is moved to shared memory by the message broker, and vice versa (① and ⑥ in Table V). No matter the size of the chain, the total # of protocol processing tasks or serialization/deserialization tasks incurred when using shared memory communication is always *two*. On the other hand, these overheads in the data plane model (d) increase as the chain scales, indicating poor scalability.

The event-driven approach (δ), which uses `SKMSG` to implement the zero-copy I/O, incurs one interrupt and one context switch for each transmission within the function chain (② to ⑤ in Fig. 3). This inevitably has a higher latency compared to using DPDK's RTE RING. With DPDK's RTE RING, different functions exchange packet descriptors entirely in userspace and avoid expensive context switches. For the I/O latency going from one function to the next, eBPF's `SKMSG` needs $\sim$20 microseconds to send each packet descriptor. On the other hand, DPDK's RTE RING only needs $\sim$0.5 microseconds. This penalty with `SKMSG`'s kernel interrupts and context switching overheads makes the low-latency DPDK's RTE RING ideal for building high-performance function chains, desirable for latency-sensitive workloads. However, DPDK's RTE RING comes at the cost of constant polling and thus resource consumption. From a resource efficiency standpoint, `SKMSG` 's event-driven nature makes it more efficient, because it does *not* consume CPU cycles when there is no traffic. This is similar to AF_XDP, as they both belong to the eBPF system of Linux. The latency of `SKMSG` is less of a concern if there are other dominant latencies masking it. This is often true for L4/L7 middleboxes, where application-level latency and kernel protocol processing latency dominate the total request delay. It requires further optimization on the use of `SKMSG`, *e.g.,* having packet descriptors directly routed between functions without being mediated by the message broker (details in §V-B), which

TABLE IV
OVERHEAD AUDITING OF L2/L3 NF CHAIN USING SHARED MEMORY COMMUNICATION

Data pipeline No.		NIC-shared memory		Within the chain				total
		①	⑥	②	③	④	⑤	
# of copies	(α) polling	0	0	0	0	0	0	0
	(β) event-driven	0	0	0	0	0	0	0
# of interrupts	(α) polling	0	0	0	0	0	0	0
	(β) event-driven	2	1	1	1	1	1	7
# of context switch	(α) polling	0	0	0	0	0	0	0
	(β) event-driven	1	1	1	1	1	1	6

(α) *polling*-based kernel-bypass (using DPDK's PMD) + *polling*-based zero-copy I/O for function chaining (using DPDK's RTE RING);
(β) *event-driven* kernel-bypass (using eBPF's AF_XDP) + *event-driven* zero-copy I/O for function chaining (using eBPF's `SKMSG`).

TABLE V
OVERHEAD AUDITING OF L4/L7 MIDDLEBOX CHAIN USING SHARED MEMORY COMMUNICATION

Data pipeline No.		NIC-shared memory		Within the chain				total
		①	⑥	②	③	④	⑤	
# of copies	(γ) polling	2	2	0	0	0	0	4
	(δ) event-driven	2	2	0	0	0	0	4
# of interrupts	(γ) polling	2	1	0	0	0	0	3
	(δ) event-driven	2	1	1	1	1	1	7
# of context switch	(γ) polling	1	1	0	0	0	0	2
	(δ) event-driven	1	1	1	1	1	1	6
# of protocol processing tasks	(γ) polling	1	1	0	0	0	0	2
	(δ) event-driven	1	1	0	0	0	0	2
# of serialization or deserialization (L7)	(γ) polling	1	1	0	0	0	0	2
	(δ) event-driven	1	1	0	0	0	0	2

(γ) *kernel-based* NIC-shared memory packet exchange + *polling*-based zero-copy I/O for function chaining (using DPDK's RTE RING);
(δ) *kernel-based* NIC-shared memory packet exchange + *event-driven* zero-copy I/O for function chaining (using eBPF's `SKMSG`).

can considerably reduce the amount of interrupt and context switch generated by SKMSG.

REFERENCES

[1] Z. Zeng, L. Monis, S. Qi, and K. K. Ramakrishnan, "Middlenet: A high-performance, lightweight, unified nfv and middlebox framework," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 180–188.

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

[3] 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 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.

[4] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "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.

[5] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "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] The kernel development community, "AF_XDP," https://www.kernel.org/doc/html/latest/networking/af_xdp.html, 2022, [ONLINE].

[8] Red Hat, Inc., "Understanding the eBPF networking features in RHEL," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel_configuring-and-managing-networking, 2022, [ONLINE].

[9] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "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.

[10] "Poll Mode Driver," https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html, 2023, [ONLINE].

[11] "DPDK RTE Ring," https://doc.dpdk.org/guides/prog_guide/ring_lib.html, 2022, [ONLINE].

[12] 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.

[13] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–10.

[14] Martín, Eugenio Pérez, "Deep dive into Virtio-networking and vhost-net," https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net, 2022, [ONLINE].

[15] V. Jain, S. Qi, and K. K. Ramakrishnan, "Fast function instantiation with alternate virtualization approaches," in *2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2021.

[16] Giller, Robin, "Open vSwitch with DPDK Overview," https://www.intel.com/content/www/us/en/developer/articles/technical/open-vswitch-with-dpdk-overview.html, 2022, [ONLINE].

[17] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff, "Revisiting the open vswitch dataplane ten years later," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 245–257.

[18] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "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.

[19] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a highly scalable user-level TCP stack for multicore systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502.

[20] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 656–671, 2021.

[21] Cloudflare, Inc. , "SOCKMAP - TCP splicing of the future," https://blog.cloudflare.com/sockmap-tcp-splicing-of-the-future/, 2022, [ONLINE].

[22] "DPDK Multi-process Support," https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html, 2022, [ONLINE].

[23] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 639–652, 2020.

[24] 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.

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

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

[27] "Multiple Poll-Mode Driver Threads," https://docs.openvswitch.org/en/latest/intro/install/dpdk/#multiple-poll-mode-driver-threads, 2022, [ONLINE].

[28] William Tu, "netdev-afxdp: Add interrupt mode netdev class," https://patchwork.ozlabs.org/project/openvswitch/patch/1582837038-31955-1-git-send-email-u9012063@gmail.com/, 2020, [ONLINE].

[29] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *2016 International Conference on Computing, Networking and Communications (ICNC)*, 2016, pp. 1–7.

[30] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34.

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

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

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

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

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

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

[37] J. Tan, C. Liang, H. Xie, Q. Xu, J. Hu, H. Zhu, and Y. Liu, "Virtio-user: A new versatile channel for kernel-bypass networks," in *Proceedings of the Workshop on Kernel-Bypass Networks*, ser. KBNets '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 13–18.

[38] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "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.

[39] 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.

[40] 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*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015.

[41] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "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.

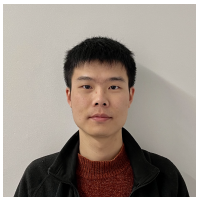
[42] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for ebpf-based network functions in an era of microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, 2021.

[43] M. Abranches, O. Michel, and E. Keller, "Getting back what was lost in the era of high-speed software packet processing," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 2022, pp. 228–234.

[44] N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Accelerating virtual network functions with fast-slow path architecture using express data path," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1474–1486, 2020.

[45] "Environment Abstraction Layer," https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html, 2022, [ONLINE].

Shixiong Qi is a Ph. D. student in Department of Computer Science and Engineering at the the University of California, Riverside. He received the B.Sc. degree in Electronic and Information Engineering from the Nanjing University of Posts & Telecommunications, China, in 2015. In 2018, he obtained his M.Sc. degree in Communication and Information Systems from Xidian University, China. His current research interests focus on cloud computing, 5G, and Network Function Virtualization.



Ziteng Zeng received his B.Sc. degree in Computer Science from Zhejiang University, China, in 2020. In 2022, he obtained his MS degree in Computer Science from University of California, Riverside. His research interests focus on cloud computing and Network Function Virtualization. He is currently working as a Software Engineer at Google.

Leslie Monis received his B.Tech. degree in Computer Science and Engineering from NITK, India, in 2019. In 2022, he obtained his MS degree in Computer Science from University of California, Riverside. His research interests focus on cloud computing and Network Function Virtualization. He is currently working as a Software Engineer at NVIDIA.

K. K. Ramakrishnan Dr. K. K. Ramakrishnan is Professor of Computer Science and Engineering at the University of California, Riverside. Previously, he was a Distinguished Member of Technical Staff at AT&T Labs-Research. Prior to 1994, he was a Technical Director and Consulting Engineer in Networking at Digital Equipment Corporation. Between 2000 and 2002, he was at TeraOptic Networks, Inc., as Founder and Vice President. K. K. is an ACM Fellow, an IEEE Fellow and an AT&T Fellow, recognized for his fundamental contributions on communication networks, including his work on congestion control, traffic management and VPN services. He has published nearly 300 papers and has 183 patents issued in his name. K. K. received his MTech from the Indian Institute of Science (1978), MS (1981) and Ph.D. (1983) in Computer Science from the University of Maryland, College Park, USA.