/dev/stdpkt: A Service Chaining Architecture with Pipelined Operating System Instances in a Unix Shell

Motomu Utsumi
The University of Tokyo, Japan
motomu@hongo.wide.ad.jp

Hajime Tazaki
IIJ Research Laboratory, Japan
tazaki@iij.ad.jp

Hiroshi Esaki
The University of Tokyo, Japan
hiroshi@wide.ad.jp

ABSTRACT

Network Function Virtualization is being recognized as an important solution to satisfy a wide spectrum of user requirements, such as cost-effective and flexible service provisioning. Service Function Chaining is a key function in Network Function Virtualization to achieve rich and flexible pipelined network services with multiple virtual network functions.

We have investigated the implementation of function chain using a shell pipeline of Unix processes, which is the base technology contributing the practical success of the Unix philosophy.

In this paper, we evaluate the applicability of this idea in the function chaining scenario by conducting benchmarks in possible use cases. The evaluation is based on the prototype system which extends an existing userspace network stack, Linux Kernel Library, because it can use various mature network functions and has various performance optimization techniques. The evaluation confirms that, although the prototype system needs some performance improvements, the complex packet processing provided by functional chains of virtual nodes can be completely functional. Furthermore, we demonstrate up to 25% improvement on application goodput.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; *Network management*;

KEYWORDS

Service Chaining; Network Function Virtualization; Middlebox; Unix pipe

ACM Reference Format:
Motomu Utsumi, Hajime Tazaki, and Hiroshi Esaki. 2017. /dev/stdpkt: A Service Chaining Architecture with Pipelined Operating System Instances in a Unix Shell. In *AINTEC '17: AINTEC '17: Asian Internet Engineering Conference , November 20–22, 2017, Bangkok, Thailand.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3154970.3154972

1 INTRODUCTION

By fully utilizing the power of virtualization, Network Function Virtualization has the potential to solve not only the original motivation of replacing hard-to-upgrade facilities of network functions based on hardware appliances, but also to bring the flexibility of composing those functions by chaining/concatenating multiple functions. We believe that the flexibility of full recomposablility is a key to making the technology useful in practice.

We implemented function chaining using Unix pipelines in a standard Unix shell. Unix pipelines, first championed by McIlroy [6], and later expanded to network resources by Plan9 from Bell Labs [10], concatenates multiple programs into a stream to process data. Following the Unix philosophy, composing a simple program (or function) which *does a simple job well*, then chaining them to do larger jobs, we implemented Service Function Chaining in a Unix shell, providing modularity, simplicity, robustness, etc [12], making a well-working system. Prior work such as EtherPIPE [5] showed that this idea is useful for simple network packet processing. However, it did not function well and was difficult to implement, if the processing was complicated such as network address translation (NAT) or load balancing with stateful connection tracking.

While Unix pipeline permits the benefits mentioned above, there are multiple concerns with applying the idea to Service Function Chaining–lack of feature-rich functionality of network protocols in which a handcrafted Unix applications have, the performance shortcomings by repurposing the Unix pipe as a network channel, and the half-duplex nature of Unix pipe.

This paper explores the applicability of Unix pipelines for service function chaining and evaluates the idea with our prototype implementation, called `/dev/stdpkt` by extending Linux Kernel Library (LKL) [11]. The evaluation consists of a set of application goodput benchmarks and clarifies its benefits for creating function chains with rich network functions. Our contributions in this paper include:

- The design and implementation of `/dev/stdpkt` to provide network function platform with Unix pipes and feature-rich userspace network stack.
- The benchmarks with TCP/UDP goodput measurements in practical scenarios (§ 4.2) and the duration of instantiation of network functions (§ 4.3).
- We also envision the possible use cases of function chain under the Unix pipeline framework (§ 4.4).

2 REQUIREMENTS

Service function chaining can be divided into two components, network function and its interconnect. In a simple case, we can use an application running on a commodity OS over a virtual machine as a network function, and network devices such as tap and bridge as an interconnect since we do not face serious performance issue with this degree of complexity. However, when deploying a large number of network functions in a single physical node, there are new set of requirements that must be achieved for decent service.

Using commodity OSes for the function chaining presents various benefits of reusability of an existing application, a rich set of supported network protocols, as well as the available options for development facilities such as tools and languages. However, those benefits become an obstacle (or generality tax [14]) when it comes to specializing the software stack such as high-performance network functions. This was pointed out by prior work (ClickOS [9], OSv [3], MirageOS [7]) on designing of network function with the fact that the packet processing overhead of commodity OSes and boot latency are large. On the other hand, several proposals on alternate interconnect such as VALE [13] also addressed the issue of general-purpose operating systems and proposed methods to completely bypass the kernel-based network stack in order to avoid low-speed code path of packet processing. These approaches are trying to achieve high-speed packet processing and quick boot, but at the cost of functional degradation. For instance, those proposals lack various network functionalities such as full TCP stack with various congestion avoidance algorithms, packet filtering and modification framework[1], packet encapsulation protocols and so on.

Our motivation is to fill the gaps of the drawbacks of prior work in order to satisfy the requirements of various users and services. The following lists the set of requirements toward our goal.

R1: Reusability of existing applications.
R2: Lightweight and quick instantiation.

[1]netfilter (http://www.netfilter.org/) in Linux for instance.

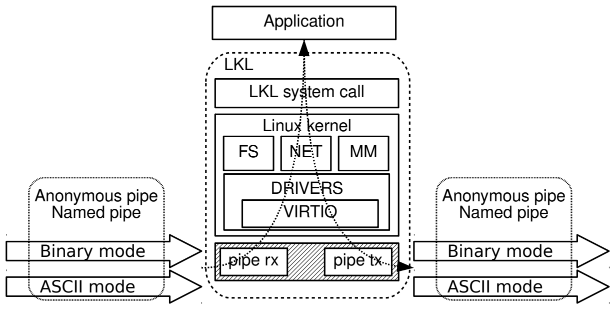


Figure 1: LKL application components and its extended modules (hatched area): pipe as a network channel.

R3: Reliable and feature-rich network functions.
R4: Development flexibility.
R5: Reasonable performance in function chaining.

The next section describes how we address the above requirements with our prototype implementation.

3 DESIGN AND IMPLEMENTATION

We design and implement a service function chaining architecture by extending LKL. Our prototype implementation, named `/dev/stdpkt`, uses LKL as a network function platform and uses a Unix pipe as an interconnect. LKL allows us to run unmodified Linux applications over Linux network stack (originally implemented for a kernel-space program) so that features such as packet filtering by netfilter can be easily accomplished. This also gives us development flexibility since plenty of Linux applications can be used as-is. Furthermore, the boot duration is reasonably quick (detailed in § 4.3) since an instantiation only involves a process invocation, which is often faster than instantiating an OS via a generic hypervisor such as QEMU.

In this section, we describe how our design addresses requirements (§ 2) and the challenges to the design for complex packet processing in a shell pipeline.

3.1 Challenges

Feature rich network function: To provide feature-rich network function, we initially thought that we could extend Linux virtual machines to run in a Unix shell and connect each virtual machine via a pipe. However, a Linux virtual machine is heavy and takes a time to boot (four seconds in our measurements § 4.3). This latency will spoil the shell usability and prevent us from providing multiple network function on demand. Additionally, since the required function in a Unix shell is only processing the packet, we do not need any other feature such as isolation of multiple processes or user

```
[DST_MAC] [SRC_MAC] [Eth_Type] XX XX XX ...
[DST_MAC] [SRC_MAC] [Eth_Type] XX XX XX ...
```

Figure 2: Packet format in ASCII mode.

and kernel space separation. Some prior work proposed lightweight operating systems for achieving quick boot and high performance [9] [3]. Though using these operating systems seems to meet our requirements (R2 and R5), these operating systems typically provide fewer functionalities than Linux.

Function chain in a shell pipeline: Unix pipeline is process chaining for a data stream, each process emits data to standard output which directs to the standard input of the next process via a pipe character "|". The output of a program can be used as the input of another program to trim, edit, or filter data. And the output can pass to other commands. While the pipeline in a shell is usually unidirectional, data goes through bidirectionally in our situation since packet flow is usually bidirectional.

3.2 Design

Our design choice, to overcome the challenges, is extending a userspace network stack derived from LKL, to harmonize with Unix shell pipeline. Figure 1 illustrates the detail of one LKL application component. An application linked with LKL works as a network function while an anonymous pipe (standard input, output) and a named pipe connected between other processes work as interconnects.

The interconnect is implemented as a typical virtio network device as other virtual network devices available in LKL. Therefore, the extension is relatively small[2] while retaining full functionalities such as hardware offload or configuration interfaces for LKL. The pipe-based virtual network device has two different modes of packet representation, binary mode and ASCII mode: the binary mode writes and reads the stream of raw packets to the device while the ASCII mode translates the raw data into the text-based strings, which the format is illustrated in Figure 2 (we use the same format as proposed in EtherPIPE [5]). This ASCII mode is useful if we use Unix commands such as `sed`, `awk`, or `grep` together with pipes.

Using a shell and pipes are enough to describe a network topology. For instance, we can connect two LKL network applications by a pipe, and can connect a group of LKL instances via a branch of a pipe by using `tee` command[3].

```
# LKLapp1 | tee named pipe1 | LKLapp2
```

[2]The modification is around 500 LoC.

[3]`tee` is the command to copy/duplicate the data from standard input to standard output.

```
# IF0="tap0" IF1="named pipe1|/dev/stdout"\
   LD_PRELOAD=liblkl-hijack.so            \
  ./nat-config.sh |  \   # iptables -t nat
 IF0="/dev/stdin|named pipe1" IF1="tap1"  \
   LD_PRELOAD=liblkl-hijack.so            \
   ./firewall-config.sh  # iptables -A DROP
```

Figure 3: Command: NAT and Firewall chaining. `LD_PRELOAD` used in this example is a way to dynamically link a library at runtime.

```
# LKLapp3 < named pipe1
```

3.2.1 *Feature rich network stack.*

Using LKL as a feature-rich network stack brings other benefits. We can reuse various Linux applications if LKL supports application programming interface (API) used by the applications. In addition to that, the mature development environment such as tools and languages are also available as-is, thanks to the compatible interface of LKL with typical Linux system. Finally, since an instantiation of network functions only involves a single process invocation, boot time is very quick compared to hypervisor-based virtual machines.

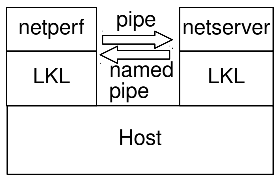
3.2.2 *Multiple pipes for bidirectional communication.*

To achieve bidirectional communication in a shell, we use an additional named pipe since anonymous pipes are unidirectional. By writing bytes to standard output, an LKL application can transmit packets via pipes, then the next process receives the packets at the other end of pipes from standard input. Based on the LKL framework of network interface backends (i.e., channels)[4], we extended LKL 1) to utilize pipes as a network channel and 2) to support multiple network interfaces in a single LKL application. As a result, an LKL application can be used as a single command to cooperate with other programs in a shell via pipes.

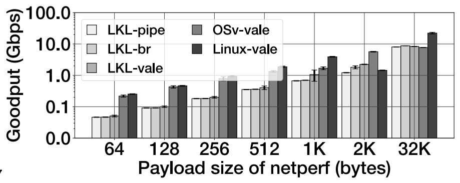
3.3 Usage

A set of commands in Figure 3 is an example service function chain with our proposal. A combination of `iptables` commands allows us to create a function chain of NAT and firewall, with the help of netfilter framework within LKL. The command line generates a network topology illustrated in Figure 5a: the (`nat-config.sh`) script contains `iptables` command to configure a source address translation at the outgoing interface (`stdout` of the first command) and the `firewall-config.sh` script, also described with `iptables` command, filters packets received from `stdin` based on the configured rules.

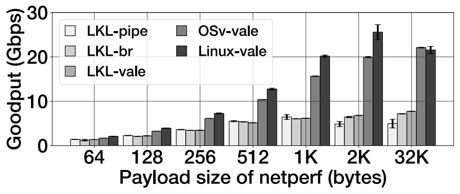
[4]Currently LKL supports tap devices, raw sockets, and Intel DPDK devices.



(a) Topology of directly connected measurement. (netperf | netserver)



(b) UDP application goodput.



(c) TCP application goodput.

Figure 4: Application goodput in function of payload size of transmitted packets with the standard deviation from 50 replications.

4 EVALUATION

The evaluations consist of three parts: a packet processing speed measurement of application goodput (§ 4.2), a boot time benchmark (§ 4.3), and a use case demonstration (§ 4.4). The goal of our evaluation is to understand the performance of our prototype both as network function and interconnect. We compared our implementation with other network function implementations, OSv [3] and typical Linux virtual machine over KVM, as well as Docker container. As for alternatives of interconnect to our pipe-based implementation, we used Linux bridge with tap devices, and VALE [13].

VALE is a Virtual Local Ethernet which focuses on quickly switching the packets. VALE is not suitable for our research purpose because applications have to use netmap API to connect to VALE. However, VALE is one of the candidates for realizing high-performance interconnect. Thus we extended LKL to use VALE as a network channel to compare the performance with Unix Pipe.

4.1 Experimental Setup

Our tests in this section were conducted on the same machine using a single-CPU Intel i7-3770K server with four cores at 3.50GHz and 32 GB RAM. In all cases, we used Ubuntu 16.04, Linux kernel 4.4.0 for host and guest OS, netperf 2.7.0[5], iptables 1.6.1, QEMU 2.9.93, Docker 1.12.6, VALE[6], OSv 0.24-425-g77b1f05, and our extended LKL[7].

4.2 Packet Processing Speed

Directly connected LKL applications: We measured the application goodput between two network function instances directly connected via a network channel as illustrated in Figure 4a. We used TCP_STREAM and UDP_STREAM mode of `netperf` to measure how fast the network function instance sends and receives packets. We changed the payload size of `netperf` and measured 50 times for each payload size.

[5]https://github.com/HewlettPackard/netperf
[6]https://github.com/luigirizzo/netmap code downloaded at Sep 22, 2017
[7]https://github.com/libos-nuse/lkl-linux-pipe

Table 1: Combination of instance and interconnect.

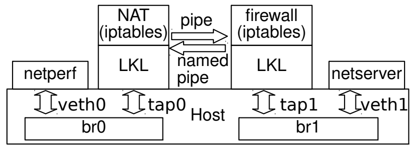
instance	interconnect	legend
LKL	Unix pipes	LKL-pipe
LKL	bridge	LKL-br
LKL	VALE	LKL-vale
Linux	VALE	Linux-vale
OSv	VALE	OSv-vale

In this measurement, to maximize the application goodput, we configured network function instances as follows. In the measurement with LKL and Linux, we used MTU size 60000 bytes so as not to fragment the packets, enabled checksum offload, TCP Segmentation Offload (TSO), and allowed merging receive buffers to reduce CPU overhead. In the measurement with OSv, we can not change MTU size so we used 1500 bytes. Pipe buffer size was 65536 bytes which is default size in Linux kernel 4.4.0. We used `taskset` command for CPU pinning to LKL application. We did not use the CPU pinning to Linux and OSv virtual machine because it did not improve the goodput while LKL did. Table 1 shows the combination of network function and interconnect we measured.

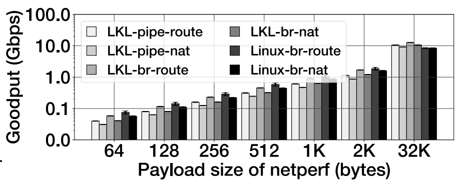
Figure 4b and 4c show the measured UDP and TCP application goodput in function of the payload size. The goodput increases along with the increase of payload size at most of the payload size, and Linux and OSv results outperform that of LKL in most of the cases. Note that the reason why the result of TCP is better than that of UDP is because TCP stack has a segmentation-offload feature while UDP does not.

We observed that the choice of interconnect with LKL does not present any significant differences while OSv and Linux do. This suggests the bottleneck during packet processing is inside LKL, not in an interconnect, thus using VALE with LKL, in this case, does contribute to the overall performance of LKL.

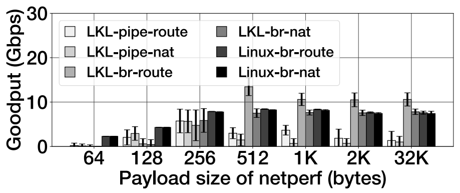
Connected via multi-hop nodes: Considering more practical use cases, we measured the network performance of the two network functions chaining. Figure 5a illustrates the topology of NAT and firewall LKL application chaining



(a) Topology of chaining use case measurement. (netperf | iptables | iptables | netserver)



(b) UDP application goodput.



(c) TCP application goodput.

Figure 5: Application goodput in function of payload size of transmitted packets with the standard deviation from 50 replications.

Table 2: Combination of instance, interconnect, and network function.

instance	interconnect	network function	legend
LKL	Unix pipe	routing x 2	LKL-pipe-route
LKL	Unix pipe	NAT, firewall	LKL-pipe-nat
LKL	bridge tap	routing x 2	LKL-br-route
LKL	bridge tap	NAT, firewall	LKL-br-nat
Linux	bridge tap	routing x 2	Linux-br-route
Linux	bridge tap	NAT, firewall	Linux-br-nat

with Unix pipe network channel. We employed IP routing, NAT, and firewall as network functions in a chain. In this measurement, NAT implemented by `iptables` converts the IP address of the outgoing interface. Firewall implemented by `iptables` with 500 DROP rules which do not match any packets between netperf and netserver. Other configurations are the same as the previous experiment. Table 2 shows the combination of the instance, interconnect, and network function we measured.

Figure 5b and 5c plot the UDP and TCP application goodput in function of the payload size. The UDP application goodput increased according to the increase of payload size. At most of the payload sizes, chaining of two routing obtained higher goodput than chaining of NAT and firewall: this is as expected since the packet processing of NAT and firewall is clearly heavier than pure IP routing.

It is worth noting that unlike the result of previous experiment (Figure 4), some of the goodput results with LKL (`LKL-br-route/nat`) is higher than one of Linux (`Linux-br-route/nat`) in both TCP and UDP: for instance, the TCP and UDP goodput of 32K bytes packet with LKL (`LKL-br-route/nat`) are 5% - 40% faster than with Linux (`Linux-br-route/nat`). This result suggests that LKL has a potential to process packets faster than Linux virtual machine *if* it works as a middlebox, not as an endpoint of communication. Although the result of `Linux-pipe-route/nat` does not perform as well as `Linux-br-route/nat` does, our pipe implementation can be updated with our private version of anonymous/named pipe.

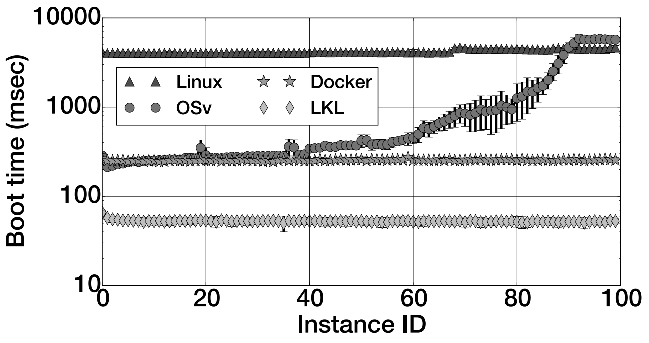


Figure 6: Boot time of each instance: the error bar indicates the standard deviation from 50 replications of experiment.

4.3 Boot time

In order to measure boot time, we collected elapsed time between the start of the commands and the reception of the first packet from an instance at the host. We measured this time with LKL, as well as with Docker container, Linux virtual machine, and OSv on KVM. We also measured how the numbers of network function instances on a single machine affects the boot time. For this measurement, we booted a large number of network function instances (100 instances) in sequence. We conducted this experiment for 50 times. In this measurement, LKL, Docker, and OSv called the sleep command after booting while Linux did not do anything after a boot completion.

Figure 6 shows the boot time of each network function instance with the instance ID, which we assigned to each instance. The boot time of LKL instance was 50 - 60 msec for all instance ID. This is the fastest type of instance–the Docker takes 240 msec at the fastest (5 times slower than LKL), OSv boots 400 msec (8 times slower), and Linux takes four seconds (80 times slower). Note that we also observed the boot time of OSv instances varies as the number of booted instances is increasing. This is due to the concurrent processing load of instances with OSv. In our measurement, single OSv instance consumes more CPU time than other instances at around 50 instances, and total CPU usage reached around 100% earlier than the others.

Past research also conducted the similar experiment: ClickOS boots in about 30 msec [9]. Jitsu unikernel, which is based on MirageOS [7], takes 20 msec on x86 and 350 msec on ARM [8]. OSv with memcached takes 600 msec to start serving requests [3]. In our measurement, the boot time of the first OSv instance was 400 msec. Though the boot time can vary depending on how "boot time" is defined and measured, our results are comparable with the result of [3] and confirm that LKL boots as quickly as other past research. OSv developer [3] mentioned this boot time can be optimized further using ramfs instead of ZFS. This optimization is also applicable to LKL since we can change the configuration of LKL to support other features available in Linux.

4.4 Use case

In this section, we will demonstrate a couple of example use cases of our proposal. A command line in Figure 7 represents port mirroring example by tee command to investigate packet flows. The tee command captures packets at the middle, duplicating them to the named pipe2. pktparser is our original tiny packet parser, which reads the Ethernet packet binary stream from standard input and generates the hex dump to standard output. By using existing commands such as text2pcap[8] and tcpdump command[9] to filter and decode packets, it can easily inspect packets generated by applications.

```
# IF0="named pipe1|stdout"              \
    LD_PRELOAD=liblkl-hijack.so         \
    ./app1-config.sh |                  \
    tee named pipe2  |                  \
  IF0="stdin|named pipe1"               \
    LD_PRELOAD=liblkl-hijack.so         \
    ./app2-config.sh;
# ./pktparser < named pipe2 |           \
   text2pcap - - |                      \
   tcpdump -r -;
```

Figure 7: Command: port mirroring.

A command line in Figure 8 conducts packet filtering by grep command. In this case, grep command blocks the packet whose source or destination mac address is '00:11:22:33:44:55'.

[8]text2pcap is the command to convert ASCII hex dump to packet capture format.

[9]tcpdump is the command to read packet capture format and print out a description of the contents of packet.

```
# IF0="named pipe1|stdout"              \
    IFTYPE0="ascii"                     \
    LD_PRELOAD=liblkl-hijack.so         \
    ./app1-config.sh |                  \
  grep --line-buffered -v "001122334455" |\
  IF0="stdin|named pipe1"               \
    IFTYPE0="ascii"                     \
    LD_PRELOAD=liblkl-hijack.so         \
    ./app2-config.sh
```

Figure 8: Command: packet filtering.

Moreover, as we showed in § 4.2, LKL can provide NAT and firewall implemented by iptables. Naturally, we can use any combination of network function above.

5 DISCUSSION

So far, we have focused on the functionality and its applicability of Unix pipeline to Service Function Chaining. However, through our prototype implementation and benchmarks, we can see a couple of possible bottlenecks. The rest of this section describes the detail of possible bottlenecks, unsuitable applications, and suitable applications of our current prototype implementation.

Bottleneck The first one is LKL. As shown in § 4.2, in the measurement of directly connected LKL applications, choice of interconnect with LKL has a small impact on application goodput. However, OSv and Linux with VALE obtained better goodput than LKL with VALE. This suggests the bottleneck is not the interconnect but LKL.

In contrast, LKL with bridge obtained better goodput than Linux with bridge in the measurement of multi-hop nodes, Although we have not found the exact reason for this, one possible reason is position of LKL instances in packet communication. In the measurement of directly connected LKL applications, we executed userland application which sends/receives packets and LKL instances work as an endpoint, while in the measurement of multi-hop nodes, packet handling was done in kernel space and LKL instances work as a middlebox. Thus overhead of packet handling in LKL would depend on the function of LKL instance. As future work, we would profile more deeply and explore the possibility of better performance with LKL.

Unix pipes are the next bottleneck. We are using pipes for interconnects without any modifications, thus any packets passed through LKL via pipes are copied at least twice: when the left LKL application sends a packet, it copies packet data from user space to kernel space during a write(2) system call of the pipe, and then copied again at the right application

Table 3: Characteristics of various Service Function Chaining architectures.

	R1:reusability	R2: boot latency	R3: feature richness	R4: development flexibility	R5: chaining performance
EtherPIPE [5]	++	+	–	++	-
OpenNetVM [1]	-	+	-	-	++
ClickOS [9]	+	+	+	-	+
Jitsu [8]	-	+	-	-	N/A
OSv [3]	+	+	+	+	N/A
/dev/stdpkt (this paper)	**++**	**+**	**++**	**++**	**+**

during a `read(2)` system call. Those memory copies can be avoided if multiple processes share the memory resource by implementing a private pipe which reduces the number of data copies.

Another possible bottleneck is process scheduling. Pipe and named pipe have a writer and the reader processes, and every pipe has a buffer, so if the writer process writes faster than the reader, the writer cannot write to the pipe until the reader reads the packet. This bottleneck would be solved by introducing a process scheduler which consider the buffer occupancy. We plan to analyze this internal behavior more deeply in future work.

Unsuitable application. The current implementation of pipes is not dynamically configurable, we have to design the connectivity in advance and then build the chains. Therefore, it is difficult to apply our proposal when chains are dynamically updated or chaining path depends on each flow. Extending the current prototype implementation to accept new channel creation would address this.

Suitable application. LKL application boots quickly (less than 70 msec). Therefore, this idea is suitable for a short-lived network function, we can deploy and destroy LKL network functions in an instant on-demand. This enables operators to use resources efficiently.

6 RELATED WORK

We have explored both network function and interconnect in previous work. However, this work does not satisfy the requirements we presented in § 2. Table 3 shows the characteristics of prior work. The rest of this section describes the detail of related work.

EtherPIPE [5] proposes a character device driver for a network interface card to utilize Unix commands for packet processing. From the context of Service Function Chaining, this research uses Unix commands as network functions and Unix pipes as interconnect. Their motivation came from the fact that Unix commands (`cat`, `grep`, `sed`, `awk`) are sufficient for simple packet processing (R1). The technique of processing packets on UNIX shell is similar to our research. However, the network functions offered by EtherPIPE are limited to simple packet processing (R3), while we often need more complicated packet processing at the middlebox such as connection tracking and NAT with a stateful inspection. Their achieved goodput was upto 1.5 Gbps with five MAC address filtering rules at 64 byte frame size. Their implementatin was a proof of concept and performance was future work for their research (R5).

NetVM [1] is a platform for running network function on KVM. OpenNetVM [15] is based on NetVM architecture and run the network function as a native process or a process inside Docker containers to realize lightweight, quick boot, and isolation (R2). In their proposal, network function is actually the callback function written in C language. Interconnect is shared memory with Intel's Data Plane Development Kit [2]. Network function manager provides zero-copy packets routing between network functions. As a result, OpenNetVM achieved 40 Gbps with 5 network function chaining (R5). However, we cannot run existing Linux application on their platform (R1). If we want to implement the complicated network application such as stateful firewall or proxy which terminates a TCP connection, it requires a reasonable amount of re-implementation effort and it is hard to provide equal stability of commodity OS (R3).

ClickOS [9] is a platform for middlebox processing based on Xen hypervisor. They choose Click modular router [4] as a programming abstraction of middlebox packet processing (R1). In their proposal, network function is minimal operating system optimized for running Click and interconnect is ClickOS Switch which is based on VALE [13] and shared memory with Xen Memory Grant. They achieved quick boot (about 30 msec) and 21.7 Gbps throughput with two directly connected network function (R2, R5). However, if we need a new module, we have to develop within Click framework (R4). Moreover, unlike LKL which has full Linux kernel functionality, ClickOS does not have a complete TCP stack (R3).

Jitsu [8] proposed architecture for providing an application on demand with MirageOS [7], type-safe unikernel. MirageOS can be used as a network function and Jitsu showed the possibility of provisioning a required application with quick boot operating system on demand, indicating boot latencies of 350ms on ARM and 30ms on X86 (R2). However, MirageOS focuses on running OCaml application. Therefore, we cannot use existing applications and other languages to develop (R1, R4).

OSv [3] proposed new operating system designed especially for virtual machine environment. They redesigned many functions including network stack and achieved 24%-25% higher performance of netperf TCP STREAM (single-stream throughput) than Linux virtual machine at the expense of functionalities [3]. Furthermore, OSv achieved fast boot, 600 ms (R2). Unlike other research and similar to our research, one of their goals is running existing Linux executables by allowing application to call Linux ABI (R1). However, LKL can support many more applications which are written for Linux because LKL provides equivalent functionality such as netfilter (R3).

7 CONCLUSION AND FUTURE WORK

This paper is the first attempt to design a Service Function Chaining by applying Unix pipeline in a standard Unix shell. We have proven the applicability of the Unix pipeline for Service Function Chaining by showing performance benchmarks and use cases. We have extended LKL to use Unix pipes as network channel by taking several challenges to overcome poor features of packet processing in Unix shell, and half-duplex nature of Unix pipe. Though optimizations are required, directly connected LKL applications reach 8Gbps goodput with UDP and 6.5 Gbps goodput with TCP at most. Furthermore, NAT and firewall LKL application chains obtain 10.5 Gbps goodput with UDP and 5.6 Gbps goodput with TCP at most. The quick boot time of LKL application (-70 msec) allows us on-demand instantiation for resource-efficient operations.

A couple of future directions are possible: 1) service chaining flows in our proposal are fixed thus not able to dynamically update upon each network flow request. This shall be addressed by extending the current prototype to accept new channel creation. 2) the base performance shown in this paper suggests optimization to several components of our prototype implementation. Our observations on the possible bottleneck are listed and will be analyzed to provide more detailed performance profiling in our future work.

8 ACKNOWLEDGMENTS

The authors thank to Matthieu Coudron for his comments on this work and Zachary S. Bischof for his proofreading on this paper.

REFERENCES

[1] Hwang, J., Ramakrishnan, K. K., and Wood, T. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *USENIX NSDI'14* (Berkeley, CA, USA, 2014), pp. 445–458.

[2] Intel. Intel DPDK: Data Plane Development Kit. http://dpdk.org. (Accessed Sep 26th 2017).

[3] Kivity, A., et al. OSv—Optimizing the Operating System for Virtual Machines. In *USENIX ATC '14* (June 2014), pp. 61–72.

[4] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (Aug. 2000), 263–297.

[5] Kuga, Y., Matsuya, T., Hazeyama, H., Cho, K., Meter, R. V., and Nakamura, O. A packet I/O architecture for shell script-based packet processing. *China Communications 11*, 2 (Feb 2014), 1–11.

[6] Laboratories, B. THE UNIX ORAL HISTORY PROJECT . http://www.princeton.edu/~hos/Mahoney/expotape.htm. (Accessed Jun 2nd 2017).

[7] Madhavapeddy, A., et al. Unikernels: Library operating systems for the cloud. In *ACM ASPLOS '13* (New York, NY, USA, 2013), pp. 461–472.

[8] Madhavapeddy, A., et al. Jitsu: Just-in-time summoning of unikernels. In *USENIX NSDI'15* (Oakland, CA, May 2015), pp. 559–573.

[9] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. Clickos and the art of network function virtualization. In *USENIX NSDI'14* (Berkeley, CA, USA, 2014), pp. 459–473.

[10] Presotto, D. L., and Winterbottom, P. The Organization of Networks in Plan 9. In *USENIX Winter.* (1993).

[11] Purdila, O., Grijincu, L. A., and Tapus, N. R. I. C. R. . t. LKL: The Linux kernel library. In *RoEduNet, 2010* (2010), pp. 328–333.

[12] Raymond, E. S. *The art of Unix programming.* Addison-Wesley Professional, 2003, ch. Basics of the Unix Philosophy.

[13] Rizzo, L., and Lettieri, G. VALE, a Switched Ethernet for Virtual Machines. In *ACM CoNEXT '12* (New York, NY, USA, 2012), pp. 61–72.

[14] Williams, D., and Koller, R. Unikernel monitors: Extending minimalism outside of the box. In *USENIX HotCloud '16* (2016).

[15] Zhang, W., et al. OpenNetVM: A Platform for High Performance Network Service Chains. In *ACM HotMIddlebox '16* (2016), pp. 26–31.