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

Motomu Utsumi
The University of Tokyo

Hajime Tazaki
IIJ Research Laboratory

Hiroshi Esaki
The University of Tokyo

1 INTRODUCTION

By fully utilizing the power of virtualization, Network Function Virtualization (or NFV) 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, to be fully recomposable, 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 [3], and later expanded to network resources by Plan9 from Bell Labs [6], 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 (SFC) in Unix, providing modularity, simplicity, robustness, etc [7], making a well-working system. Prior work such as EtherPIPE [2] showed that this idea was useful for simple network packet processing. However, it did not function well, or was difficult to implement, if the processing was complicated such as network address translation (NAT), or load balancing with stateful connection tracking.

We explore 1) the applicability of Unix pipelines for service function chaining with a feature-rich network stack, 2) study the performance with benchmarks, and 3) envision possible use cases of function chaining using the Unix pipeline framework addressing the goals of SFC. These preliminary contributions are the first steps of designing a SFC framework using Unix pipes in our prototype implementation, `/dev/stdpkt`, using a userspace network stack derived from Linux Kernel Library (LKL).

2 DESIGN AND IMPLEMENTATION

2.1 Challenges

Feature rich network function: We want to compose complex functions such as NAT or a stateful load balancer. To achieve such rich network functions, we extend a userspace network stack, the Linux Kernel Library (LKL), to harmonize with Unix shell pipeline.

APSys '17, September 2, 2017, Mumbai, India
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

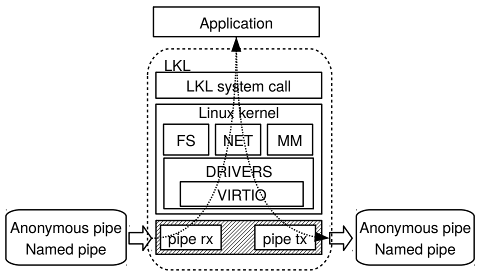

Figure 1: LKL application components and their extended modules (hatched area): pipe as a network channel.

Function chain in a shell pipeline: While a shell pipeline is usually unidirectional, packet flow is usually bidirectional. By using anonymous pipes (standard input, output) and named pipes together as a network channel, network functions implemented in the Linux kernel can interact within a shell command line.

2.2 Design

Figure 1 illustrates the detail of one LKL application component. Traditionally, service function chaining uses virtual machines to realize network functions and connects the virtual machine with network devices such as bridges and taps. In contrast, we use an LKL application instead of a virtual machine, and use pipes and named pipes instead of bridges or tap devices.

2.3 Usage

```
IF0="tap0" IP0="10.0.0.1" NETMASK0="24" \
 IF1="named pipe1|/dev/stdout" IP1="192.168.0.1" NETMASK1="24"\
 LD_PRELOAD=liblkl-hijack.so ./nat-config.sh | \
IF0="/dev/stdin|named pipe1" IP0="192.168.0.2" NETMASK0="24"\
 IF1="tap1" IP1="10.10.2.1" NETMASK1="24" \
 LD_PRELOAD=liblkl-hijack.so ./firewall-config.sh
```

The sequence above is an example of how we compose a NAT and a firewall. The NAT LKL application (`nat-config.sh`) has two interfaces, the first interface uses tap0 to send and receive packets, the second interface uses named pipe1 as a receive channel and standard output as a transmission channel. The Firewall LKL application (`firewall-config.sh`)

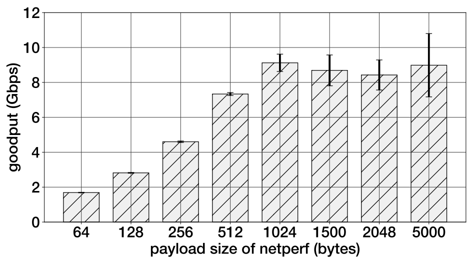

Figure 2: Application goodput as a function of packet payload size with the standard deviation from 100 replications.

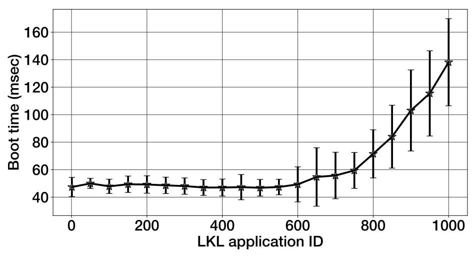

Figure 3: Boot time of each instance of the LKL application: the error bar indicates the standard deviation from 100 replications of the experiment.

also has two interfaces, the first receives standard input and named pipe1 as its output channel, the second interface uses tap1 as typical NICs.

3 EVALUATION

We evaluate packet processing speed as application goodput, and the time to boot. Our tests were conducted on a single machine with the following components: Intel i7-6700 (4 cores, 3.4 GHz) CPU, Intel 10-Gigabit X540-AT2 Network Card, 2x16G DDR4-2133 RAM, running a 64-bit Linux kernel 4.9.0, Netperf version 2.7.0, and LKL [1].

Packet Processing Speed: We measured the network performance of two LKL applications directly connected via a pipe and a named pipe. We used `netperf` TCP_STREAM mode to measure how fast the LKL applications sent and received packets. `netperf` transmits packets to stdout which is redirected to the stdin of the netperf server (`netserver`) via a pipe. `netserver` transmits packets to the named pipe and `netperf` receives packets from the named pipe. We varied the payload of `netperf` and measured 100 times for each payload size. To explore the maximum performance in this scenario, we used an MTU of 65500 bytes, enabled checksum offload, TCP Segmentation Offload (TSO), and allowed LKL to merge receive buffers. Figure 2 shows the application goodput as a function of payload size. When the payload size is less than 1024 bytes, goodput increases in accordance with the increase of payload size. Goodput reaches at most 9.1 Gbps with the 1024 bytes payload size.

Boot time: We measured the time from the start of the LKL application to its transmission of the first packet. We also measured how the numbers of LKL applications on a single machine affects the boot time. We instantiated 1000 LKL applications in sequence for 100 times, and measured the boot time of each LKL application.

Figure 3 shows the boot time of each LKL application instance with the LKL application ID we assigned to each instance. The boot time of the first instance was 47.4 msec, and it constantly increases if the ID is larger than 600, due to the concurrent processing load presented by many LKL instances. Past research also measured the boot time: ClickOS boots about 30 msec [5]. Jitsu unikernel, which is based on MirageOS, takes 20 msec on x86, 350 msec on ARM [4]. OSv with memcached takes 600 msec to start serving requests [1].

4 FUTURE WORK

A number of future directions are possible: 1) service chaining flows in our proposal are fixed, thus not able to dynamically update on each network flow request. This will be addressed by extending the current prototype to accept creation of new channels. 2) the base performance shown suggests much more optimization of several components of our prototype implementation.

ACKNOWLEDGMENTS

The authors thank to Randy Bush for his proofreading on this paper.

REFERENCES

[1] Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., and Zolotarov, V. OSv—Optimizing the Operating System for Virtual Machines. In *USENIX ATC* (2014), USENIX Association, pp. 61–72.

[2] 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.

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

[4] Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D., Mortier, R., Chaudhry, A., Singh, B., Ludlam, J., Crowcroft, J., and Leslie, I. Jitsu: Just-in-time summoning of unikernels. In *NSDI* (2015), USENIX Association, pp. 559–573.

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

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

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

[1]https://github.com/libos-nuse/lkl-linux-pipe