NetKernel: Network Stack as a Service in the Cloud *

Zhixiong Niu, Hong Xu, Dongsu Han*, Peng Wang, Libin Liu
NetX Lab, City University of Hong Kong; *KAIST
zx.niu@my.cityu.edu.hk, henry.xu@cityu.edu.hk, dongsuh@ee.kaist.ac.kr, {pewang4-c, libinliu-c}@my.cityu.edu.hk

1 Introduction

Multi-tenant data centers power many online applications and services for billions of users, by allowing applications to run on virtual machines (VMs) with a wide variety of operating systems and configurations. Networking plays a crucial role in delivering performance to VMs, and our community has proposed many new solutions at the end-host, including congestion control [2, 5], flow scheduling [3], and load balancing [6].

Despite the progress, the architecture of the VM network stack remains unchanged. Following traditional operating systems design, the VM network stack is part of the guest OS kernel and tightly coupled to it. Yet this architecture proves to be inefficient in terms of performance, and is increasingly becoming the barrier of innovation. First, most new networking protocols or services do not work with such an architecture, since the operator does not have total control over the network stack in the VMs. Second, solutions targeting multi-tenant data centers usually have to go with a hypervisor based implementation that does not involve modifying the VM network stack [5, 6]. Packets then need to go through additional processing and performance suffers. This also stresses the already-burdened hypervisor.

In this paper, we propose NetKernel, a new architectural design for VM network stack. NetKernel decouples the VM network stack from the guest kernel, by making it a customizable service whose implementation is independent of the VM. The operator or other third-parties can implement various transport protocols [2, 5], stack optimization [7, 8], and end-host network functions [4] as NetKernel modules for VMs to use. Essentially, NetKernel offers network stack as a service in the cloud.

We identify three important use cases of NetKernel. First, NetKernel gives tenants the flexibility to choose a combination of network protocols to maximize the performance according to their needs.

Second, NetKernel makes it easier for the operator to develop and deploy novel network protocols and services, by simply packaging them as NetKernel modules for VMs. The operator can optimize the stack for their hardware and network fabric using e.g. FPGA and RDMA without modifying the guest kernel.

Third, NetKernel also enables the operator to exert centralized control over individual tenants and coordinate their behavior globally, which is difficult to do with the current design. For example, the operator can easily adjust the flow priority thresholds in PIAS [3] by communicating to the NetKernel modules.

In addition, NetKernel can significantly reduce the efforts for the tenants to deploy and maintain new protocols. For example, MPTCP is updated by distributing source code and linux kernel packages. Developers have to recompile or reinstall the kernel to update their MPTCP VMs. Using NetKernel, MPTCP updates can be distributed as an updated image. Thus we believe NetKernel, and in general separating the network stack from the guest OS, is a better architectural design in multi-tenant data centers.

Note NetKernel differs from AC/DC [5] which still follows the conventional VM network stack. NetKernel also eliminates the overhead of TCP conversion at the hypervisor in AC/DC. Besides, NetKernel aims to cover the entire network stack instead of just transport protocols.

2 Design

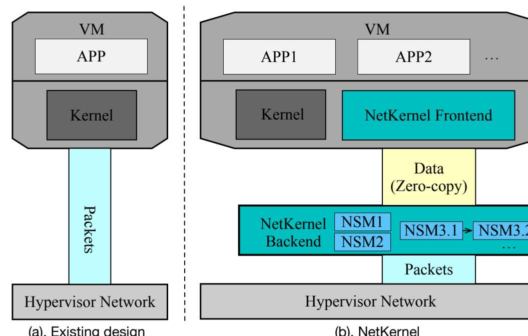

Figure 1: NetKernel design compared to existing VM network stack.

We now present the initial design of NetKernel. Figure 1.(a) shows the existing design: the network stack is

*Zhixiong Niu, Peng Wang, and Libin Liu are student authors.

located in the guest kernel, and packets are exchanged between the guest kernel and hypervisor network. In NetKernel, the applications in VMs are running on top of both the kernel and NetKernel frontend as in Figure 1.(b). The kernel is responsible for providing services as usual except for networking, which is taken care of by NetKernel. NetKernel frontend uses the same POSIX socket interface, so it is transparent to tenant applications. All the requests to the network stack are diverted by the NetKernel frontend and forwarded to the NetKernel backend. The backend consists of various Network Service Modules (NSMs), which are pre-built images providing different network services.

To optimize efficiency, NetKernel frontend and backend share data in memory using zero-copy. NSM images are configured and loaded to cater to application needs. Note that, more than one services or protocols can be built together as NSM. One can also employ multiple NSMs as a chain to form a more complicated network protocol stack as shown in Figure 1.(b). The NetKernel backend is bridged to the hypervisor network to transmit packets from/to the network.

NetKernel can also be deployed in a private cloud. In this case the operator can globally deploy a unique NetKernel network stack which is highly optimized for its network fabric.

3 Implementation and Preliminary Results

We have implemented a prototype of NetKernel to verify the feasibility of the idea. The prototype is implemented in user space in C and Python. The NetKernel frontend uses Linux command `LD_PRELOAD` to hijack the networking API calls from glibc, including `SOCKET()`, `CONNECT()`, `RECV()`, `SEND()`, `SETSOCKOPT()`, etc. to the backend. The backend consists of a tiny controller and various NSMs. The controller is written in Python to manage the NSMs. For example, if an application needs a particular service, the controller can load it instantly. The NSMs are pre-built KVM VMs.[1]

The frontend and backend communicate using a ring buffer based on IVSHMEM provided by KVM. When NetKernel frontend receives data from applications, it puts the data into the shared ring buffer. Then the backend NSM handles the data and sends it using its customized network stack.

Note that the current prototype is not optimized for performance. We plan to improve the implementation in many aspects, including a kernel space implementation, kernel bypass technologies such as DPDK, and container networking.

We performed a simple experiment to show the flexibility of NetKernel by switching different transport protocols on-the-fly, which corresponds to the first use case outlined in §1. Two NSMs are created with TCP Cubic and MPTCP, respectively. The TCP-NSM uses Linux 4.1, and MPTCP-NSM are based on Linux kernel MPTCP v0.91 [1]. There are two servers. One is the hypervisor running NetKernel. There are three VMs on this hypervisor: a sender, the TCP-NSM VM, and MPTCP-NSM VM. The other server is the receiver. The two servers are connected by two 1Gbps NICs.

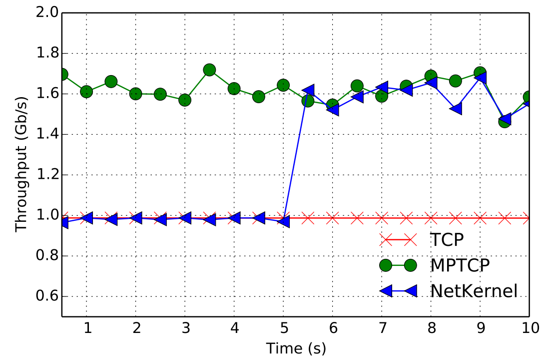

Figure 2: NetKernel allows dynamic switching of transport protocols on-the-fly. Legends: TCP and MPTCP show the throughput using existing VM network stack with TCP-Cubic and MPTCP, respectively. NetKernel shows throughput using the NetKernel prototype with a TCP-NSM first and MPTCP-NSM after 5s.

Figure 2 depicts the result. Without NetKernel, the sender VM uses either TCP or MPTCP in its kernel to send data. Now with NetKernel, data is sent using NetKernel NSMs. During 0s–5s, we use the TCP-NSM. At 5s, NetKernel switches the transport protocol by loading the MPTCP-NSM on-the-fly. Thus shortly after 5s, data is sent using MPTCP with ∼1.7Gbps throughput which is very similar to the performance achieved by running MPTCP inside the VM kernel. The application in the sender VM never needs to restart.

[1] The NSMs will be implemented by lightweight visualization technology, such as container, in the future.

4 References

[1] http://www.multipath-tcp.org.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[3] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. PIAS: Practical information-agnostic flow scheduling for data center networks. In *Proc. USENIX NSDI*, 2015.

[4] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End-host Network Functions. In *Proc. ACM SIGCOMM*, 2015.

[5] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *Proc. ACM SIGCOMM*, 2016.

[6] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford. CLOVE: How I Learned to Stop Worrying About the Core and Love the Edge. In *Proc. ACM HotNets*, 2016.

[7] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proc. SIGPLAN*, 2016.

[8] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proc. USENIX ATC*, 2016.