Network Stack as a Service in the Cloud

Zhixiong Niu
City University of Hong Kong

Hong Xu
City University of Hong Kong

Dongsu Han
KAIST

Peng Cheng
Microsoft Research Asia

Yongqiang Xiong
Microsoft Research Asia

Guo Chen
Microsoft Research Asia

Keith Winstein
Stanford University

ABSTRACT

The tenant network stack is implemented inside the virtual machines in today's public cloud. This legacy architecture presents a barrier to protocol stack innovation due to the tight coupling between the network stack and the guest OS. In particular, it causes many deployment troubles to tenants and management and efficiency problems to the cloud provider. To address these issues, we articulate a vision of providing the network stack as a service. The central idea is to decouple the network stack from the guest OS, and offer it as an independent entity implemented by the cloud provider. This re-architecting allows tenants to readily deploy any stack independent of its kernel, and the provider to offer meaningful SLAs to tenants by gaining control over the network stack. We sketch an initial design called NetKernel to accomplish this vision. Our preliminary testbed evaluation with a prototype shows the feasibility and benefits of our idea.

1 INTRODUCTION

Virtual machines (VMs) have been a great abstraction for computing. In public clouds, it provides a clear boundary between the provider and the tenants, enabling them to innovate independently. For example, in networking, the conventional separation of concern is that the cloud provider is responsible for providing virtual NICs as the abstraction and tenants maintain their own network stacks. The tenant network stack is implemented inside the VM (as shown in Figure 1a).

While this provides strong isolation, when it comes to protocol stack evolution, the legacy architecture presents a barrier due to the tight coupling between the network stack and the guest operating system (OS). In particular, it causes many deployment problems to tenants and management and efficiency problems to the cloud provider.

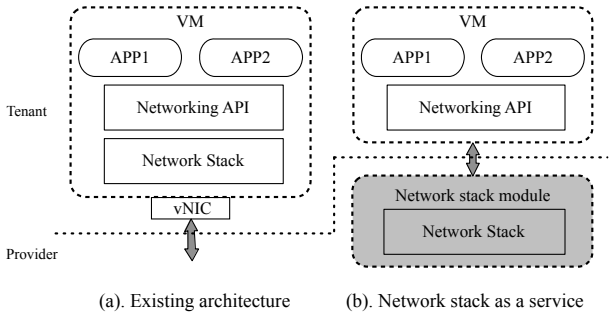

Figure 1: Today's VM network stack on the left, and network stack as a service on the right.

First, tenants in general find it painstaking to deploy or maintain new network stacks in public clouds. Since a network stack or a specific protocol is tied to some particular kernel, it cannot be directly used with other kernels. For example Google's recent BBR congestion control protocol is implemented in Linux [10]; Windows or FreeBSD VMs are then not able to use BBR directly. Porting a protocol to a different kernel requires tremendous efforts and is both time-consuming and error-prone, not to mention the fact that many tenants do not have the expertise to do it. Changing the guest kernel just because of the network stack is even less feasible, since it is extremely difficult to port all the existing applications to a new environment.

Even when a tenant's choice of network stack is available for her kernel, deploying and maintaining the stack usually incur high cost and performance penalty. Imagine that a tenant uses Linux VMs and wishes to deploy new userspace stacks, such as mTCP [22] or F-stack [5]. The tenant then needs to closely track the development of the project, merge or modify the code for her own environment, and test and debug to ensure compatibility and stability. Individually managing the stack further results in redundant efforts from many tenants.

Second, the cloud provider suffers from the inability to efficiently utilize her physical resources for networking because the tenant network stack is beyond her reach. It is difficult to meet or even define networking performance SLAs for tenant as we cannot explicitly provision or adjust resources just for the network stack. It is also impossible to deploy optimized

HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA

ACM ISBN 978-1-4503-5569-8/17/11…$15.00
https://doi.org/10.1145/3152434.3152442

network stacks with better efficiency in tenant VMs, although many such implementations are available in our community [24, 30, 35].

We thus put forth a vision of *network stack as a service* to facilitate bringing innovations to all layers of the stack. The central thesis of this vision is to *decouple* the VM network stack from the guest OS and offer it as an independent entity (say a VM) by the cloud provider. Figure 1(b) depicts the idea. We keep the application interfaces in the guest, such as BSD sockets for TCP/IP and Verbs for RDMA, intact and use them as the abstraction boundary instead of the virtual NICs. Packets are then handled *outside* the tenant VM in a network stack module (NSM) provided by the provider, whose design and implementation are transparent to tenants. Each VM has its own NSM whose resources are dedicated to providing networking services, and tenants pay for it in addition to the application VM charges.

We believe decoupling network stack from the guest OS, thereby enabling network stack as a service, fundamentally solves the problems discussed above. Tenants now can deploy any stack independent of its guest kernel without any effort or expertise. The provider can now offer meaningful SLAs to tenants by gaining control over the network stack. Overall the new architecture allows rapid and flexible deployment of more efficient network stacks, accelerating the innovation in the public cloud (§2).

Network stack as a service potentially opens up a line of inquiry on systems design and optimization, and certainly entails many possible solutions. We present in this paper an initial design called NetKernel that achieves this vision without radical change to existing tenant VMs (§3). We implement an initial prototype of NetKernel based on KVM (§4). We port the TCP/IP stack from Linux kernel 4.9, including the recent BBR protocol [10] as our NSM. Our preliminary evaluation on a testbed with 40 GbE NICs shows that NetKernel's design achieves comparable performance with the native stack implementation inside the guest kernel. We also show that a Windows VM running BBR is feasible using NetKernel as one of its new usecases (§4). We identify and discuss open research questions in §5, and summarize related work in §6 before concluding the paper.

2 NETWORK STACK AS A SERVICE

Separating the network stack from the guest OS marks a significant departure from the way networking is provided to VMs nowadays. In this section we elaborate why our vision of network stack as a service is a better architectural design. We also discuss the implications of the new architecture in a public cloud to defend our position.

2.1 Benefits

Network stack as a service offers three important benefits that are missing in today's architecture: deployment flexibility to tenants, efficiency to the provider, and faster innovation with better evolvability to the community.

First, tenants now have greater flexibility to choose and deploy any network stack independent of its guest OS. A Windows VM now can opt to use a Linux stack which is not feasible in current public clouds. They may also request a customized stack (say RDMA) for their applications. These benefits can be realized without any development or maintenance effort or expertise from the tenant side. The network stack is maintained by the provider transparent to tenants. Applications do not need to change since the classical networking APIs are preserved in our new architecture.

Second, providers can now offer meaningful SLAs to tenants and charge them accordingly. By gaining control over the stack, they can adopt various strategies to guarantee the networking performance delivered to the VMs (e.g. throughput, latency), while optimizing their resource usage. They can deploy an optimized stack to reduce the CPU overhead; dynamically scale up the network stack module with more dedicated cores; or scale out with more modules to support higher throughput to a large number of concurrent connections. They can also exploit the multiplexing gains by serving multiple tenant VMs with the same network stack module.

Finally, we believe innovation in networking can also be accelerated as a result of having network stack as a service. Architecturally any network stack can be used for tenant VMs without constraint on guest kernel, providing a path towards protocol stack evolution. Developers only need to focus on designing and implementing new protocols, not how to make them available on multiple OSes.

2.2 Implications

We discuss several implications as a result of providing network stack as a service.

Containers. Our discussion in this paper is centered around VM based virtualization that covers the vast majority of usage scenarios in a public cloud. Containers on the other hand are gaining tremendous popularity as a lightweight and portable alternative to VMs [4]. A container is essentially a process with namespace isolation. It thus relies on the network stack of its host, be it a bare-metal machine or a VM.

We find that in EC2 [2], Azure [3], and Google Container Engine [6], users have to launch containers in a VM. Therefore network stack as a service readily benefits containers running in VMs. For containers running directly on bare-metal machines, since they have to share the host machine's network stack, they suffer from the same deployment and efficiency problems mentioned in §1. The same arguments can be made here to move away from the legacy architecture and instead decouple the network stack from the OS, though the specific design may differ in many ways.

Security. Most of the security protocols such as HTTPS/TLS work at the application layer. They can work as usual with network stack as a service. One exception is IPSec. Due to the certificate exchange issue, IPSec does not work directly in our design. However, in practice IPSec is usually implemented at

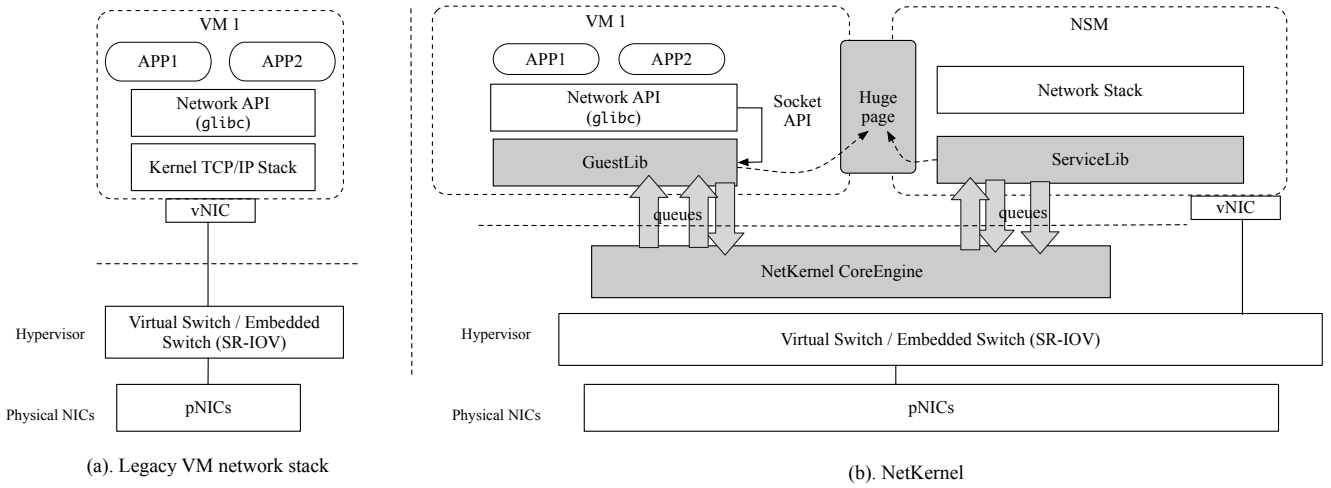

Figure 2: NetKernel design compared to existing VM network stack.

gateways instead of end-hosts. We believe the impact is not serious.

Removal of NIC in Guest. Due to the change in the abstraction boundary, the NICs no longer exist in the tenant VMs, which may lead to some changes for tenants. For example tunning the kernel TCP/IP stack via `sysctl` or tuning the NIC becomes infeasible in our architecture. We argue this is a sensible tradeoff since the provider now controls the stack and offers much better SLA for networking, minimizing the need of such performance tuning. Further network stack as a service is offered as an additional service; one can choose not to use it and still rely on the network stack inside her VM just like before.

3 DESIGN

We now outline the initial design of NetKernel, a framework that enables network stack as a service in today's cloud.

3.1 Overview

Figure 2(a) shows the legacy VM network stack as the baseline. The TCP/IP stack is usually implemented in the guest kernel. In some cases, the VM uses a userspace stack [1, 5, 22]. A virtual NIC (vNIC) usually connects the VM to a virtual overlay switch (vSwitch) in the hypervisor, such as OVS or Hyper-V Switch, which routes packets and performs accounting, monitoring, etc. In some cases the overlay switch is implemented as an embedded switch on special hardware so processing can be offloaded for high performance [13, 16, 20]. This way VM's traffic can bypass the host to the physical NIC (pNIC) by using SR-IOV for better performance.

NetKernel aims to separate the network stack from the guest without requiring radical change to the tenant VM, so that it can be readily deployed. As depicted in Figure 2(b) the network API methods are intercepted by a NetKernel GuestLib in the guest kernel. The GuestLib can be readily deployed as a kernel patch and is the only change we make to the tenant VM. Network stacks are implemented by the provider on the same physical host as Network Stack Modules (NSMs). The NSM may take various forms: VM, container, or even a hypervisor module, with different tradeoffs in deployment, performance, etc. We intentionally keep the design general at this point and defer the specific realization of the NSM to future work. Inside the NSM, the ServiceLib interfaces with the network stack and GuestLib in the tenant VM. The NSM connects to the overlay switch, be it a virtual switch or a hardware one, and then the pNICs. Thus our design also supports SR-IOV. A NetKernel CoreEngine runs on the hypervisor and is responsible for setting up the NSM when a VM boots. It also facilitates the communication between GuestLib and ServiceLib.

To support fast communication between the tenant VM and the network stack in the NSM, NetKernel employs two communication channels based on shared memory that quickly move data around. The first is a small shared memory region between a tenant VM or a NSM and the CoreEngine on the hypervisor. This region consists of a set of queues, and are used to transmit event metadata and descriptors for application data across VM and NSM. The second is a huge page region shared between a VM and its corresponding NSM that is used to directly write or read actual data. Each pair of VM and NSM is allocated unique huge pages to ensure security and isolation.

3.2 Transport Service via Shared Memory

We now explain in detail how transport service is provided via shared memory in NetKernel as shown in Figure 3. We use TCP as the example.

All communication between GuestLib in VM and ServiceLib in NSM is done with the help of a data structure called NetKernel Queue Element, or `nqe`. It contains operation ID, VM ID, and fd for the VM, or operation ID, NSM ID, and connection ID (cID) for NSM. It also has a data descriptor if necessary, which is a pointer to the huge pages for data. Each `nqe` is copied between VM queues and NSM queues

by CoreEngine. It is small in size and copying incurs negligible overhead [21]. During the process, CoreEngine maps <VM ID, fd> to the corresponding <NSM ID, cID> and vice versa using a connection mapping table as shown in Figure 3. GuestLib, ServiceLib, and CoreEngine interact with queues using batched interrupts.

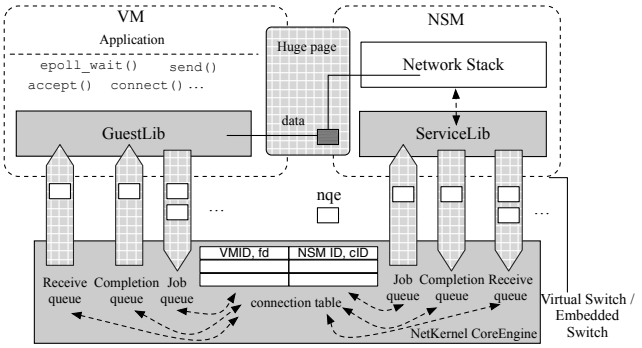

Figure 3: Transport service. Shaded areas indicate shared memory regions.

When an application registers a socket by invoking `socket()`, GuestLib intercepts the call and adds a `nqe` with the request into the VM job queue. CoreEngine is notified with a batched interrupt later about this new event among others in the VM *job queue*. It immediately assigns a new socket fd, wraps it with a new `nqe`, inserts it to VM *completion queue*, and notifies GuestLib using a batched interrupt. The application `socket()` is then returned. CoreEngine also independently applies for a new socket from the NSM by putting a new `nqe` into NSM job queue. ServiceLib gets a batched interrupt from the job queue later and forwards these requests to NSM's network stack. The response is again contained in `nqe`s and copied to NSM completion queue. CoreEngine then copies them into VM completion queue. It also adds a new entry into its connection mapping table for the new connection.

The `connect()` call is handled in largely the same way. GuestLib adds a `nqe` with the request into VM job queue. The application is returned right away. CoreEngine then copies the `nqe` to NSM job queue with connection mapping, and the NSM's network stack handles this request. The result is returned by ServiceLib putting a `nqe` into NSM completion queue and CoreEngine copying the `nqe` into the VM completion queue (with connection mapping). GuestLib returns the result of `connect()` by event notification (e.g. `epoll()`).

When the application sends data by `send()`, GuestLib intercepts the call and puts the data into the huge pages. Meanwhile it adds a `nqe` with a write operation to VM job queue along with the data descriptor. Then the application is returned, and the `nqe` is copied across queues as before. ServiceLib gets data from the huge page address and sends it to its network stack via the corresponding connection given in the `nqe`.

Packets are received by the NSM and go through the network stack for transport processing. The network stack is modified with callback functions to ServiceLib so NetKernel is involved when processing is done. When data is received ServiceLib puts data into the huge pages, and adds a `nqe` to NSM *receive queue* as shown in Figure 3. When an ACK to SYN/ACK is received a new `nqe` is also inserted to the NSM receive queue. GuestLib in turn signals the corresponding socket of the application for `epoll_wait()` and `accept()` for both cases. For an accept event CoreEngine generates a new socket fd on behalf of the VM for the new flow and inserts an entry to the connection mapping table. Finally for `recv()`, GuestLib simply checks and copies new data in VM receive queue if any.

We have described asynchronous operations so far. NetKernel can readily support synchronous operations, in which case the application is not returned by GuestLib until it obtains a `nqe` from the VM completion queue.

In addition, the job queues and completion queues can be implemented as priority queues to handle connection events and data events separately to avoid the head of line blocking.

4 PRELIMINARY RESULTS

We present our preliminary implementation and results now.

4.1 Prototype

We have implemented a simple prototype of NetKernel to verify the feasibility of our idea. We run our prototype on two servers each with Xeon E5-2618LV3 8-core CPUs clocked at 2.3 GHz, 192 GB memory, and Intel X710 40Gbps NICs. We use QEMU KVM 2.5.0 for the hypervisor and Ubuntu 16.04 with Linux kernel 4.9.0 for both the host and the guest OSes. For the Windows guest, we choose Windows Server 2016 (Datacenter). The NetKernel prototype is about 3000 lines of C code.

GuestLib. We implement GuestLib in userspace for fast prototyping. GuestLib uses `LD_PRELOAD` [8] to override the socket API calls from `glibc`, including `socket()`, `connect()`, `recv()`, `send()`, `setsockopt()`, etc. We defer the support of event based API such as `select()` and `epoll()` to future work. For Windows VMs we create a similar library to intercept Windows programs. GuestLib uses polling to process the queues for simplicity.

Queues and Huge Pages. The huge pages are implemented based on QEMU's IVSHMEM. The page size is 2 MB and we use 40 pages. The queues are ring buffers implemented as much smaller IVSHMEM devices.

ServiceLib. ServiceLib in the NSM continuously polls the queues to execute the operations from GuestLib via NetKernel CoreEngine. All the operations from the queues are diverted to the socket API of the network stack in the NSM. For example, if a connect operation arrives, ServiceLib calls the `connect()` backend function of the network stack in NSM. When packets arrive, ServiceLib now implements two callback functions: `nk_new_data_callback()` and `nk_new_accept_callback()` to process new data and new connections, respectively.

NSMs. We use KVM VMs as NSMs to host different network stacks with isolation in the prototype. We implement NSMs by porting the TCP/IP stack in Linux kernel 4.9 including Google's recent BBR [10]. Each NSM is assigned 1 CPU core, 1G RAM, and one virtual function (VF) of an Intel X710 40Gbps NIC with SR-IOV in our server.
NetKernel CoreEngine. The CoreEngine is implemented as a daemon on the KVM hypervisor.

4.2 Microbenchmarks

We report some microbenchmarks of NetKernel here. First we use the TCP Cubic NSM and compare it to running TCP Cubic natively in a VM. Figure 4 shows the result. We observe the NetKernel NSM achieves virtually same throughput with running TCP Cubic natively in the VM. Both can achieve line rate (~37 Gbps) when there are more than two flows.

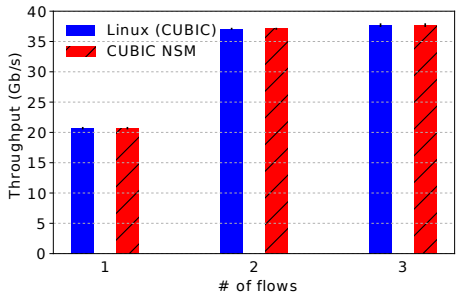

Figure 4: Throughput of TCP Cubic and NetKernel TCP Cubic NSM. The chunk size for the huge page operations is 8 KB.

Chunk Size	64B	512B	1KB	2KB	4KB	8KB
Latency	8ns	64ns	117ns	214ns	425ns	809ns

Table 1: Memory copying latency in NetKernel.

We then measure the communication overhead within NetKernel. NetKernel has two types of transmissions for `nqes` and data chunks. A `nqe` is copied between VM and NSM via CoreEngine. The cost of this is ~12ns per event. For data chunks, Table 1 shows the latency of memory copying between GuestLib and ServiceLib with random address reads. We find that even a large chunk of 8KB costs less than 0.81μs to copy. At the same time, NetKernel can achieve ~64Gbps (64B) and ~81Gbps (8KB) between GuestLib and ServiceLib for each core. This demonstrates that NetKernel is unlikely to be the bottleneck in data transmission.

4.3 Flexibility

To demonstrate the flexibility benefit of providing network stack as a service, we conduct the following experiment. We use a Windows VM with the NetKernel BBR NSM, so traffic from the Windows VM actually uses the BBR congestion control. We also use a Windows VM running its default C-TCP in kernel as well as a Linux VM running Cubic and BBR (without NetKernel) for comparison. The TCP server is located in Beijing, China, and the client is in California, USA. The uplink bandwidth of the server is 12 Mbps and the average RTT is 350 ms.

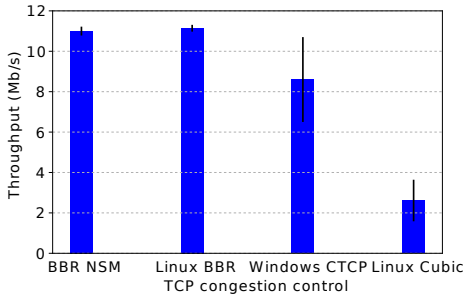

Figure 5: A Windows VM utilizes BBR by NetKernel, achieving similar throughput with original Linux BBR.

Figure 5 depicts the throughput results averaged over 10s. The Linux VM with default TCP Cubic obtains the lowest throughput of 2.61 Mbps, and the Windows VM with default C-TCP achieves 8.60 Mbps. Using the NetKernel BBR NSM the Windows VM has a much better result of 11.12 Mbps, consistent with the performance of the unmodified BBR in the Linux VM (11.14 Mbps). This shows that NetKernel has the potential to allow VMs to flexibly use different network stacks with close-to-native throughput performance.

5 RESEARCH AGENDA

Decoupling network stack from the guest OS can potentially open up many new avenues of research. Here we discuss a few important areas that require immediate attention and may lead to more future work.
NSM form. The NSMs can take various forms as mentioned in §3. They may be (1) full-fledged VMs with a monolithic kernel, which is used in our NetKernel prototype; (2) light-weight unikernel-based [12, 33] VMs with a minimal set of libraries to run the network stack; or (3) even containers or modules running on the hypervisor. Each choice implies vastly different tradeoffs. For example, VM based NSMs is the most flexible and can readily support existing network stacks from various OSes. It also provides good isolation. On the other hand VMs consume more resources and may not offer best performance due to various overheads. A container or a module based NSM consumes much less resources and can offer better performance. Yet it is much more challenging to port a complete network stack to a container or a hypervisor module, while achieving memory isolation at the same time [29]. A thorough investigation into the design choices and their tradeoffs for NSM is more than necessary to achieve the vision of network stack as a service.
Centralized management and control. Network stack as a service (NSaaS) facilitates centralized network management and control. Since the network stack is maintained by the provider, management protocols such as failure detection [17]

and monitoring [28] can be deployed readily as NSMs. Meanwhile, some new protocols such as Fastpass [31] and pHost [14] require coordination among end-hosts and are deemed infeasible for public clouds. They can now be implemented as NSMs and deployed easily for all tenants. In this regard, NSaaS removes the hurdle of managing the tenant network stack in the cloud, enabling new research to quickly develop in this area.

Container. Though we focus on VMs here, containers can also benefit greatly from NSaaS. A critical limitation of the current container technology is that containers have to use the host's network stack. There are many cases where it is actually better to use different stacks for containers running on the same host. A container running a Spark task may use DCTCP for its traffic, while a web server container may need BBR or CUBIC. Enabling NSaaS for containers is one of our ongoing work.

Pricing model and accounting CPU and RAM. Network stack as a service can spur research on new pricing models. One may charge tenants based on the number of NSM instances or number of cores, even CPU and memory utilization on average per instance used for example. One may also use SLA based pricing, based on for example the maximum number of concurrent connections supported, maximum throughput allowed, etc.

Resource efficiency and optimization. Finally, much can be done in various aspects to optimize the system design for NSaaS. Take NetKernel as an example. The control and data flows between the tenant VM and NSM require frequent communication via the hypervisor (done by CoreEngine in NetKernel). We use polling for fast prototyping now. More efficient soft interrupts (with batching) or hypercalls can provide low latency while saving precious CPU cycles here. Also the performance overhead of VM-NSM communication is very small as shown in §4.2, but may still penalize applications especially when SR-IOV is used for host bypassing. The latency overhead may also affect the scalability of handling many concurrent short connections [24]. It is possible to use more efficient IPC mechanisms or even implement NetKernel's CoreEngine in embedded hardware switch [13] to reduce this overhead.

The resource allocation and scheduling of the NSMs also needs to be strategically managed and optimized when we use a NSM to serve multiple VMs concurrently while providing QoS guarantees.

6 RELATED WORK

We survey three lines of work closely related to ours.

There are many novel network stack designs that improve performance. The kernel TCP/IP stack continues to witness optimization efforts in various aspects [24, 30, 35]. On the other hand, since mTCP [22] userspace stacks based on high performance packet I/O have been quickly gaining momentum [1, 7, 26, 27, 32, 36, 37]. Beyond transport layer, novel flow scheduling [9] and end-host based load balancing schemes [18, 23] are developed to reduce flow completion times. These proposals are designed to solve specific problems of the stack with targeted applications or scenarios. This paper focuses on a broader and fundamental issue: how can we properly re-factor the VM network stack, so that tenants can readily use a network stack of their choice and enjoy meaningful performance SLAs, without worrying about maintenance or deployment? We advocate to decouple network stack from the guest OS as a promising answer in this paper. These solutions can be potentially deployed as different NSMs on NetKernel.

There is some recent work on enforcing a uniform congestion control logic across tenants without modifying the VMs [11, 19]. The differences between this line of work and ours are clear: First these approaches require packets to go through two different stacks, one in the guest kernel and another in the hypervisor, leading to performance and efficiency loss. NetKernel does not suffer from these problems. Second they focus on congestion control while our work targets the entire network stack.

Lastly, in a broader sense, our work is also related to the debate on how an OS should be architected in general, and microkernels [15] and unikernels [12, 25] in particular. Microkernels take a minimalist approach and only implement address space management, thread management, and IPC in the kernel. Other tasks such as file systems and I/O are done in userspace [34]. Unikernels [12, 25] aim to provide various OS services as libraries or modules that can be flexibly combined to construct an OS. Different from these works that require radical changes to the OS, we seek to flexibly provide the network stack as a service without re-writing the existing guest kernel or the hypervisor which are largely monolithic kernels. In other words, our approach brings some key benefits of microkernels and unikernels without a complete overhaul of existing virtualization technology.

7 CONCLUSION

We have advocated a vision of network stack as a service in public cloud in this position paper. Decoupling the network stack from the guest OS provides flexibility and efficiency benefits for tenants and providers, and accelerates innovation without being constrained by the network stack in the guest VM. We also sketched our initial NetKernel design as a step to realize this vision and presented preliminary implementation results to demonstrate the feasibility of our idea. We showed that network stack as a service opens up new design space and identified many research challenges along the way. Going forward, we hope more discussion and effort can be stimulated in the community to fully accomplish this vision.

8 ACKNOWLEDGMENT

The project is supported in part by the Hong Kong RGC GRF-11216317, GRF-11202315, CRF-C7036-15G, CityU SRG grant 7004677, and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (No.2015-0-00164).

REFERENCES

[1] [n. d.]. http://www.seastar-project.org/. ([n. d.]).

[2] [n. d.]. Amazon EC2 Container Service. https://aws.amazon.com/ecs/details/. ([n. d.]).

[3] [n. d.]. Azure Container Service. https://azure.microsoft.com/en-us/pricing/details/container-service/. ([n. d.]).

[4] [n. d.]. Docker community passes two billion pulls. https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/. ([n. d.]).

[5] [n. d.]. F-Stack: A high performance userspace stack based on FreeBSD 11.0 stable. http://www.f-stack.org/. ([n. d.]).

[6] [n. d.]. Google Container Engine. https://cloud.google.com/container-engine/pricing. ([n. d.]).

[7] [n. d.]. Introduction to OpenOnload-Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. http://www.moderntech.com.hk/sites/default/files/whitepaper/V10_Solarflare_OpenOnload_IntroPaper.pdf. ([n. d.]).

[8] [n. d.]. Linux Programmer's Manual LD.SO(8). http://man7.org/linux/man-pages/man8/ld.so.8.html. ([n. d.]).

[9] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. PIAS: Practical Information-Agnostic Flow Scheduling for Data Center Networks. In *Proc. USENIX NSDI*.

[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-based Congestion Control. *Commun. ACM* 60, 2 (February 2017), 58–66.

[11] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. 2016. Virtualized Congestion Control. In *Proc. ACM SIGCOMM*.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proc. ACM SOSP*.

[13] D. Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *Proc. NSDI*.

[14] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and S. Shenker. 2015. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *Proc. ACM CoNEXT*.

[15] David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. 1992. Microkernel operating system architecture and Mach. In *Proc. the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*.

[16] A. Greenberg. [n. d.]. SDN in the Cloud. Keynote, ACM SIGCOMM 2015. ([n. d.]).

[17] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proc. ACM SIGCOMM*.

[18] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proc. ACM SIGCOMM*.

[19] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Jason Gu, Wes Felter, John Carter, and Aditya Akella. 2016. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *Proc. ACM SIGCOMM*.

[20] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. 2015. mSwitch: A highly-scalable, modular software switch. In *Proc. ACM SOSR*.

[21] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. USENIX NSDI*.

[22] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. USENIX NSDI*.

[23] Naga Katta, Mukesh Hira, Aditi Ghag, Changhoon Kim, Isaac Keslassy, and Jennifer Rexford. 2016. CLOVE: How I Learned to Stop Worrying About the Core and Love the Edge. In *Proc. ACM HotNets*.

[24] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proc. ASPLOS*.

[25] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. In *Proc. ASPLOS*.

[26] Ilias Marinos, Robert NM Watson, and Mark Handley. 2014. Network stack specialization for performance. In *Proc. ACM SIGCOMM*.

[27] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proc. ACM SIGCOMM*.

[28] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and precise triggers in data centers. In *Proc. SIGCOMM*.

[29] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proc. USENIX OSDI*.

[30] Sharvanath Pathak and Vivek S Pai. 2015. ModNet: A Modular Approach to Network Stack Extension.. In *Proc. USENIX NSDI*.

[31] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proc. ACM SIGCOMM*.

[32] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proc. USENIX OSDI*.

[33] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. 2011. Rethinking the library OS from the top down. In *Proc. ACM ASPLOS*.

[34] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proc. USENIX FAST*.

[35] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proc. USENIX ATC*.

[36] Tianlong Yu, Shadi Abdollahian Noghabi, Shachar Raindel, Hongqiang Liu, Jitu Padhye, and Vyas Sekar. 2016. FreeFlow: High Performance Container Networking. In *Proc. ACM HotNets*.

[37] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proc. ACM SIGCOMM*.