VFP
A Virtual Switch Platform for Host SDN in the Public Cloud

DANIEL FIRESTONE

Daniel Firestone is the Tech Lead and Manager for the Azure Host Networking group at Microsoft. His team builds the Azure virtual switch, which serves as the datapath for Azure virtual networks, as well as SmartNIC, the Azure platform for offloading host network functions to reconfigurable FPGA hardware and Azure's RDMA stack. Before Azure, Daniel did his undergraduate studies at MIT. fstone@microsoft.com

The Virtual Filtering Platform (VFP) is a cloud-scale programmable virtual switch providing scalable SDN policy to one of the world's largest clouds, Microsoft Azure. It was designed from the ground up to handle the programmability needs of Azure's many SDN applications, the scalability needs of deployments of millions of servers, and to deliver the fastest virtual networks in the public cloud to Azure's VMs through hardware offloads.

We, the VFP team, describe here our goals and motivations in building VFP, VFP's design, and lessons we learned from production deployments. We also compare our design with that of other popular host SDN technologies such as OpenFlow [2] and Open vSwitch (OVS) [3] to show how our constraints in the public cloud can differ from those of popular open source projects. We believe these lessons can benefit the SDN community at large. More details of our design can be found in our recent NSDI paper [1].

The rise of public cloud workloads, such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform, has created a new scale of datacenter computing, with vendors regularly reporting server counts in the millions. These vendors not only have to provide scale and high density of VMs to customers, but must provide rich network semantics, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and ACLs, virtual routing tables, bandwidth metering, QoS, and more. This policy is sufficiently complex that it isn't feasible to implement at scale in traditional switch hardware.

Instead this is often implemented using Software-Defined Networking (SDN) on the VM hosts, in the virtual switch (vswitch) connecting VMs to the network, which scales well with the number of servers and allows the physical network to be simple, scalable, and very fast. As a large public cloud provider, Azure has built its cloud network on host-based SDN technologies. Much of the focus around SDN in recent years has been on building scalable and flexible network controllers and services—however, the design of the programmable vswitch is equally important. It has the dual and often conflicting requirements of a highly programmable dataplane, with high performance and low overhead. VFP is our solution to these problems.

Design Goals and Rationale

As a motivating example for VFP, we consider a simple scenario requiring four host policies used for O(1M) VM hosts in a cloud. Each policy is programmed by its own SDN controller and requires both high performance and SR-IOV offload support: the first is virtual networking, allowing a customer to define their own private network with their own IP addresses, despite running on shared multi-tenant infrastructure. Our virtual networks (VNETs) are based on the design from VL2 [4]. Second is an L4 (TCP/UDP connection) load balancer based on Ananta [5], which scales by running the load balancing NAT in the vswitch on end hosts, leaving the in-network load balancers stateless and scalable. We also include a stateful firewall and per-destination traffic metering for billing.

Originally, we built independent networking drivers for each of these host functions. As host networking became our main tool for virtualization policy, we decided to create VFP in 2011 because this model wasn't scaling. Instead, we created a single platform based on the Match-Action Table (MAT) model popularized by projects such as OpenFlow.

Original Goals

Our original goals for the VFP project were as follows:

1. *Provide a programming model allowing for multiple simultaneous, independent network controllers to program network applications, minimizing cross-controller dependencies.*

Implementations of OpenFlow and similar MAT models often assume a single distributed network controller that owns programming the switch. Our experience is that this model doesn't fit cloud development of SDN—instead, independent teams often build new network controllers and agents for those applications. This model reduces complex dependencies, scales better, and is more serviceable than adding logic to existing controllers. We needed a design that not only allows controllers to independently create and program flow tables, but enforces good layering and boundaries between them (e.g., disallows rules to have arbitrary GOTOs to other tables) so that new controllers can be developed to add functionality without old controllers needing to take their behavior into account.

2. *Provide a MAT programming model capable of using connections as a base primitive, rather than just packets—stateful rules as first-class objects.*

OpenFlow's original MAT model derives historically from programming switching or routing ASICs, and assumes that packet classification is stateless. However, we found our controllers required policies for connections, not just packets—for example, end users often found it more useful to secure their VMs using stateful access control lists (ACLs) (e.g., allowing outbound connections but not inbound ones) rather than stateless ACLs used in commercial switches. Controllers also needed NAT (e.g., Ananta) and other stateful policies. Stateful policy is more tractable in soft switches than in ASIC ones, and we believe a MAT model should take advantage of that.

3. *Provide a programming model that allows controllers to define their own policy and actions, rather than implementing fixed sets of network policies for predefined scenarios.*

Due to limitations of the MAT model provided by OpenFlow (historically, a limited set of actions, limited rule scalability, and no table typing), OpenFlow switches such as OVS have added virtualization functionality outside of the MAT model. For example, constructing virtual networks is accomplished via a virtual tunnel endpoint (VTEP) schema in OVSDB, rather than rules specifying which packets to encapsulate (encap) and decapsulate (decap) and how to do so.

We prefer instead to base all functionality on the MAT model, trying to push as much logic as possible into the controllers while leaving the core dataplane in the vswitch. For instance, rather than a schema that defines what a VNET is, a VNET can be implemented using programmable encap and decap rules matching appropriate conditions, leaving the definition of a VNET in the controller. We've found this greatly reduces the need to continuously extend the dataplane every time the definition of a VNET changes.

Later Goals Based on Production Lessons

Based on lessons from initial deployments of VFP, we added the following goals for VFPv2, a major update in 2013-14, mostly around serviceability and performance:

1. *Provide a serviceability model allowing for frequent deployments and updates without requiring reboots or interrupting VM connectivity for stateful flows, and strong service monitoring.*

As our scale grew dramatically to over O(1M) hosts, more controllers built apps on top of VFP, more engineers joined us, and we found more demand than ever for frequent updates, both features and bug fixes. In Infrastructure as a Service (IaaS) models, we also found customers were not tolerant of taking downtime for individual VMs for updates.

2. *Provide very high packet rates, even with a large number of tables and rules, via extensive caching.*

Over time we found more and more network controllers being built as the host SDN model became more popular, and soon we had deployments with large numbers of flow tables (10+), each with many rules, reducing performance as packets had to traverse each table. At the same time, VM density on hosts was increasing, pushing us from 1G to 10G to 40G and even faster NICs. We needed to find a way to scale to more policy without impacting performance and concluded we needed to perform compilation of flow actions across tables, and use extensive flow caching such that packets on existing flows would match precompiled actions without having to traverse tables.

3. *Implement an efficient mechanism to offload flow policy to programmable NICs, without assuming complex rule processing.*

As we scaled to 40G+ NICs, we wanted to offload policy to NICs themselves to support SR-IOV, which lets NICs indicate packets directly to VMs without going through the host. However, as controllers created more flow tables with more rules, we concluded that directly offloading those tables would require prohibitively expensive hardware resources for server-class NICs. Instead we wanted an offload model that would work well with

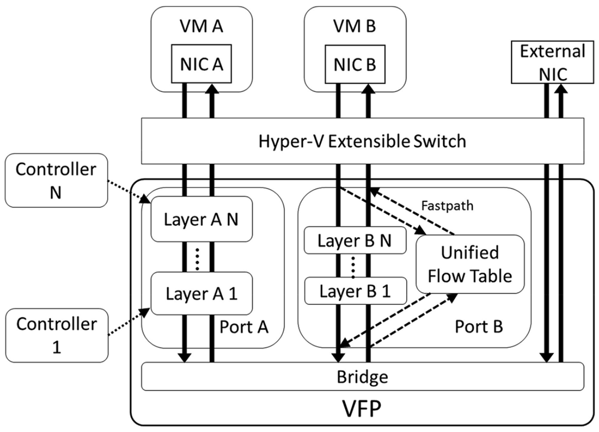


Figure 1: Overview of VFP design

our precompiled exact-match flows, requiring hardware to only support a large table of cached flows in DRAM and our associated action language.

VFP Overview

Figure 1 shows a model of the VFP design, which is described in subsequent sections. VFP operates on top of Hyper-V's extensible switch as a packet filter. Its programming model is based on layers, MATs that support a multi-controller model. VFP's packet processor includes a fastpath through Unified Flow Tables and a classifier used to match rules in the MAT layers.

The core VFP model assumes a switch with multiple ports that are connected to virtual NICs (VNICs). VFP filters traffic from a VNIC to the switch, and from the switch to a VNIC. All VFP policy is attached to a specific port. From the perspective of a VM with a VNIC attached to a port, ingress traffic to the switch is considered to be "outbound" traffic from the VM, and egress traffic from the switch is considered to be "inbound" traffic to the VM. VFP's API and its policies are based on the inbound/outbound model.

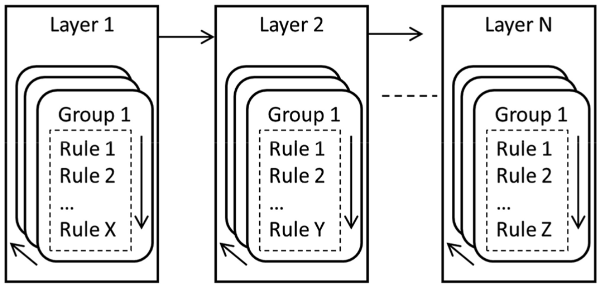


Figure 2: VFP objects: layers, groups, and rules

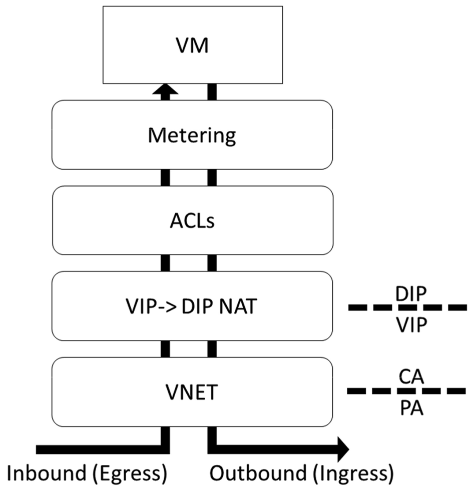


Figure 3: Example VFP layers with boundaries

Programming Model

VFP's core programming model is based on a hierarchy of VFP objects that controllers can create and program to specify their SDN policy, with ports containing layers of policy made up of groups of rules.

Layers

VFP divides a port's policy into layers. Layers are the basic Match Action Tables that controllers use to specify their policy. They can be created and managed separately by different controllers. Logically, packets into a VM go through each layer one by one, matching rules in each based on the state of the packet after the action performed in the previous layer, with returning packets coming back in the opposite direction.

Figure 3 shows layers for our SDN deployment example. A VNET layer creates a customer address (CA) / physical address (PA) boundary by having encapsulation rules on the outbound path and decapsulation rules on the inbound path. In addition, an ACL layer for a stateful firewall sits above our Ananta NAT layer. The security controller, having placed it here with respect to those boundaries, knows that it can program policies matching dynamic IP addresses (DIPs) of VMs in CA space. Finally, a metering layer used for billing sits at the top next to the VM, where it can meter traffic exactly as the customer in the VM sees it.

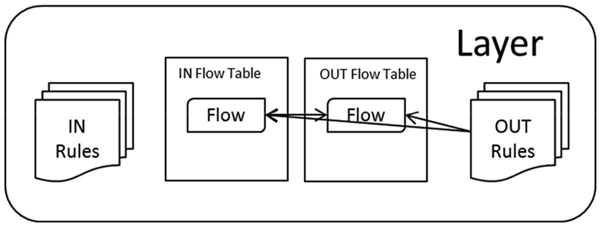
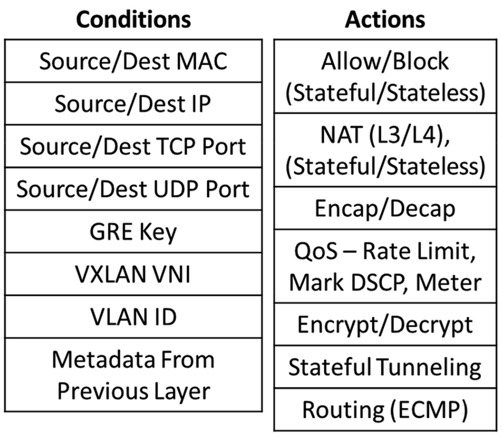


Figure 4: A layer with a stateful flow



Conditions	Actions
Source/Dest MAC	Allow/Block (Stateful/Stateless)
Source/Dest IP	NAT (L3/L4), (Stateful/Stateless)
Source/Dest TCP Port	Encap/Decap
Source/Dest UDP Port	QoS – Rate Limit, Mark DSCP, Meter
GRE Key	Encrypt/Decrypt
VXLAN VNI	Stateful Tunneling
VLAN ID	Routing (ECMP)
Metadata From Previous Layer	

Figure 5: Example conditions and actions

Layering also gives us a good model on which to implement stateful policy. We keep flow state on a layer with a hash table tracking all TCP, UDP, or RDMA connections in either direction. When a stateful rule is matched, it creates both an inbound and outbound flow in the layer flow tables, with appropriate actions in each direction (e.g., NAT or ACL).

Rules

Rules are the entities that perform actions on matching packets in the MAT model. Per original goal 3, rules allow the controller to be as expressive as possible while minimizing fixed policy in the dataplane. Rules are made up of two parts: a condition list, specified via a list of conditions, and an action. Example conditions and actions are listed in Figure 5.

Rules can be organized into groups for purposes of doing transactional update/replace operations, or to split a port into sub-interfaces (e.g., allow creation of independent policies for multiple Docker-style containers behind a single port).

Packet Processor and Flow Compiler

A primary innovation in VFPv2 was the introduction of a central packet processor. We took inspiration from a common design in network ASIC pipelines e.g.,—parse the relevant metadata from the packet and act on the metadata rather than on the packet, only touching the packet at the end of the pipeline once all decisions have been made. We compile and store flows as we see packets. Our just-in-time flow compiler includes a parser, an action language, an engine for manipulating parsed metadata and actions, and a flow cache.

Unified FlowIDs

VFP's packet processor begins with parsing. One each of an L2/L3/L4 header (as defined in Table 1) form a header group, and the relevant fields of a header group form a single FlowID. The tuple of all FlowIDs in a packet is a Unified FlowID (UFID)—the output of the parser.

Header	Parameters
Ethernet (L2)	Source MAC, Dest MAC
IP (L3)	Source IP, Dest IP, ToS (DSCP+ EC)
Encapsulation (L4)	Encapsulation Type Tenant ID, Entropy (Optional)
TCP/UDP (L4)	Source Port, Dest Port, TCP Flags (note: does not support Push/Pop)

Table 1: Valid parameters for each header type

Action	Notes
Pop	Remove this header.
Push	Push this header onto the packet. All header parameters for creating the new header are specified.
Modify	Modify this header. All header parameters needed are optional, but at least one is specified.
Ignore	Leave this header as is.

Table 2: Header Transposition actions

Header	NAT	Encap	Decap	Encap+NAT
Outer Ethernet	Ignore	Push (SMAC, DMAC)	Pop	Push (SMAC, DMAC)
Outer IP	Modify (SIP, DIP)	Push (SIP, DIP)	Pop	Push (SIP, DIP)
GRE	Not Present	Push (Key)	Pop	Push (Key)
Inner Ethernet	Not Present	Modify (DMAC)	Ignore	Modify (DMAC)
Inner IP	Not Present	Ignore	Ignore	Modify (SIP, DIP)
TCP/ UDP	Modify (SPt, DPt)	Ignore	Ignore	Modify (SPt, DPt)

Table 3: Example Header Transposition

Header Transpositions

Our action primitives, Header Transpositions (HTs), so called because they change or shift fields throughout a packet, are a list of paramaterizable header actions, one for each header. Actions (defined in Table 2) are to *Push* a header (add it to the header stack), *Modify* a header (change fields within a given header), *Pop* a header (remove it from the header stack), or *Ignore* a header (pass over it). Table 3 shows examples of a NAT HT used by Ananta, and encap/decap HTs used by VL2.

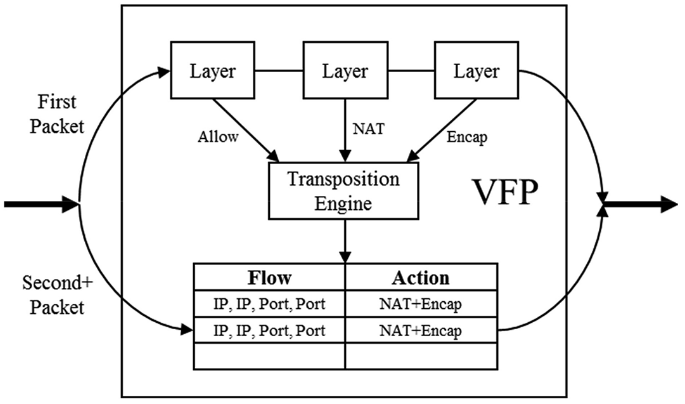


Figure 6: VFP Unified Flow Table

VFP creates an action for a UFID match by composing HTs from matched rules in each layer. For example, a packet passing the example Ananta NAT layer and the VL2 VNET encap layer may end up with the composite Encap+NAT transposition in Table 3.

Unified Flow Tables and Caching

The intuition behind our flow compiler is that the action for a UFID is relatively stable over the lifetime of a flow—so we can cache the UFID with the resulting HT from the engine. The resulting flow table where the compiler caches UFs is called the Unified Flow Table (UFT).

With the UFT, we segment our datapath into a fastpath and a slowpath. On the first packet of a TCP flow, we take a slowpath, running the transposition engine and matching at each layer against rules. On subsequent packets, VFP takes a fastpath, matching a unified flow via UFID and applying a transposition directly. This operation is independent of the layers or rules in VFP.

Operationalizing VFP

As a production cloud service, VFP's design must take into account serviceability, monitoring, and diagnostics. During update, we first pause the datapath, then detach VFP from the stack, uninstall VFP (which acts as a loadable kernel driver), install a new VFP, attach it to the stack, and restart the datapath. This operation looks like a brief connectivity blip to VMs, while the NIC stays up. To keep stateful flows alive across updates, we support serialization and deserialization for all policy and state in VFP on a port. VFP also supports live migration of VMs. During the blackout time of the migration, the port state is serialized out of the original host and deserialized on the new host.

VFP implements hundreds of performance counters and flow statistics, on per port, per layer, and per rule bases, as well as extensive flow statistics. This information is continuously uploaded to a central monitoring service, providing dashboards on which we can monitor flow utilization, drops, connection resets, and more, either on a VM or aggregated on a cluster/host/VNET basis. VFP also supports remote debugging and tracing for rules and policies as part of its diagnostics suite.

Hardware Offloads and Performance

VFP has long used standard stateless offloads (VXLAN/NVGRE encapsulation, QoS bandwidth caps, and reservations for ports, etc.) to achieve line rate with SDN policy. But to enable added goal 3 of full SR-IOV offload and host bypass, we built logic to directly offload our unified flows. These are exact-match flows representing each connection on the system, so they can be implemented in hardware via a large hash table, typically in inexpensive DRAM. In this model, the first packet of a new flow goes through software classification to determine the UF, which is then offloaded.

We've used this mechanism to enable SR-IOV in our datacenters with VFP policy offload on custom FPGA-based SmartNICs we've deployed on all new Azure servers. As a result we've seen bidirectional 32Gbps+ VNICs with near-zero host CPU and <25μs end-to-end TCP latencies inside a VNET.

Experiences

We have deployed 22 major releases of VFP since 2012. VFP runs on all Azure servers, powering millions of VMs, petabits per second of traffic, and providing load balancing for exabytes of storage, in hundreds of datacenters in over 30 regions across the world. In addition, we are releasing VFP publicly as part of Windows Server 2016 for on-premises workloads, as we have seen it meet all of the major goals listed above in production.

Over six years of developing and supporting VFP, we learned a number of lessons of value:

- **L4 flow caching is sufficient**. We didn't find a use for multitiered flow caching such as OVS megaflows. The two main reasons: being entirely in the kernel allowed us to have a faster slowpath, and our use of a stateful NAT created an action for every L4 flow and reduced the usefulness of ternary flow caching.
- **Design for statefulness from day 1.** The above point is an example of a larger lesson: support for stateful connections as a first-class primitive in a MAT is fundamental and must be considered in every aspect of a MAT design. It should not be bolted on later.
- **Layering is critical.** Some of our policy could be implemented as a special case of OpenFlow tables with GOTOs chaining them together, with separate inbound and outbound tables. But we found that our controllers needed clear layering semantics or else they couldn't reverse their policy correctly with respect to other controllers.

- **GOTO considered harmful.** Controllers will implement policy in the simplest way needed to solve a problem, but that may not be compatible with future controllers adding policy. We needed to be vigilant in not only providing layering but enforcing it. We see this layering enforcement not as a limitation compared to OpenFlow's GOTO table model but, instead, as the key feature that made multi-controller designs work for multiple years running.
- **IaaS cannot handle downtime.** We found that customer IaaS workloads cared deeply about uptime for each VM, not just their service as a whole. We needed to design all updates to minimize downtime and provide guarantees for low blackout times.
- **Design for serviceability.** Serialization is another design point that turned out to pervade all of our logic—in order to regularly update VFP without impact to VMs, we needed to consider serviceability in any new VFP feature or action type.
- **Decouple the wire protocol from the dataplane.** We've seen enough controllers/agents implement wire protocols with different distributed systems models to support O(1M) scale that we believe our decision to separate VFP's API from any wire protocol was a critical choice for VFP's success. For example, bandwidth metering rules are pushed by a controller, but VNET required a VL2-style directory system (and an agent that understands that policy comes from a different controller than pulled mappings) to scale.
- **Everything is an action.** Modeling VL2-style encap/decap as actions rather than tunnel interfaces was a good choice. It enabled a single table lookup for all packets—no traversing a tunnel interface with tables before and after. The resulting HT language combining encap/decap with header modification enabled single-table hardware offload.
- **Design for end-to-end monitoring.** Determining network health of VMs despite not having direct access to them is a challenge. We found many uses for in-band monitoring with packet injectors and auto-responders implemented as VFP rule actions. We used these to build monitoring that traces the E2E path from the VM-host boundary. For example, we implemented Pingmesh-like [6] monitoring for VL2 VNETs.
- **Commercial NIC hardware isn't ideal for SDN**. Despite years of interest from NIC vendors about offloading SDN policy with SR-IOV, we have seen no success cases of NIC ASIC vendors supporting our policy as a direct offload. Instead, large multicore NPUs are often used. We used custom FPGA-based hardware to ship SR-IOV in Azure, which we found was lower latency and more efficient.

Conclusions and Future Work

We introduced the Virtual Filtering Platform (VFP), our cloud scale vswitch for host SDN policy in Microsoft Azure. We discussed how our design achieved our dual goals of programmability and scalability. We discussed concerns around serviceability, monitoring, and diagnostics in production environments, and provided performance results, data, and lessons from real use. Future areas of investigation include new hardware models of SDN and extending VFP's offload language.

References

[1] D. Firestone, "VFP: A Virtual Switch Platform for Host SDN in the Public Cloud," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-firestone.pdf.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review,* vol. 38, no. 2 (April 2008): http://ccr.sigcomm.org/online/files/p69-v38n2n-mckeown.pdf.

[3] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, M. Casado, "The Design and Implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf.

[4] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *Proceedings of the ACM Conference on Data Communication (SIGCOMM '09)*, pp. 51–62: https://www.researchgate.net/publication/234805283_VL2_A_Scalable_and_Flexible_Data_Center_Network.

[5] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, N. Karri, "Ananta: Cloud Scale Load Balancing," in *Proceedings of the ACM Conference on Data Communication (SIGCOMM '13),* pp. 207–218: http://conferences.sigcomm.org/sigcomm/2013/papers/sigcomm/p207.pdf.

[6] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, V. Kurien, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *Proceedings of the ACM Conference on Data Communication (SIGCOMM '15)*: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/pingmesh_sigcomm2015.pdf.