Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis*

Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz,
Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, Varugis Kurien†
Microsoft, †Midfin Systems
{chguo, lyuan, dxiang, yidang, rayhuang, dmaltz, zhaoyil, vinwang, bipang, stchen,
linzw}@microsoft.com, vkurien@midfinsystems.com

ABSTRACT

Can we get network latency between any two servers at any time in large-scale data center networks? The collected latency data can then be used to address a series of challenges: telling if an application perceived latency issue is caused by the network or not, defining and tracking network service level agreement (SLA), and automatic network troubleshooting.

We have developed the Pingmesh system for large-scale data center network latency measurement and analysis to answer the above question affirmatively. Pingmesh has been running in Microsoft data centers for more than four years, and it collects tens of terabytes of latency data per day. Pingmesh is widely used by not only network software developers and engineers, but also application and service developers and operators.

CCS Concepts

•Networks → Network measurement; Cloud computing; Network monitoring; •Computer systems organization → Cloud computing;

Keywords

Data center networking; Network troubleshooting; Silent packet drops

1. INTRODUCTION

In today's data centers there are hundreds of thousands of servers. These servers are connected via network interface cards (NICs), switches and routers, cables and fibers, which form large-scale intra and inter data center networks. The scale of the data center networks (DCNs) is growing even larger due to the rapid development of cloud computing. On top of the physical data center infrastructure, various large-scale, distributed services are built, e.g., Search [5], distributed file systems [17] and storage [7], MapReduce [11].

These distributed services are large and evolving software systems with many components and have complex dependencies. All of these services are distributed and many of their components need to interact via the network either within a data center or across different data centers. In such large systems, software and hardware failures are the norm rather than the exception. As a result, the network team faces several challenges.

The first challenge is to determine if an issue is a network issue or not. Due to the distributed systems nature, many failures show as "network" problems, e.g., some components can only be reached intermittently, or the end-to-end latency shows a sudden increase at the 99^{th} percentile, the network throughput degrades from 20MB/s per server to less than 5MB/s. Our experience showed that about 50% of these "network" problems are not caused by the network. However it is not easy to tell if a "network" problem is indeed caused by network failures or not.

The second challenge is to define and track network service level agreements (SLAs). Many services need the network to provide certain performance guarantees. For example, a Search query may touch thousands of servers and the performance of a Search query is determined by the last response from the slowest server. These services are sensitive to network latency and packet drops and they care about the network SLA. Network SLA needs to be measured and tracked individually for different services since they may use different set of servers and different part of the network. This becomes a challenging task because of the huge number of services and customers in the network.

The third challenge is network troubleshooting. When network SLAs are broken due to various network fail-

*This work was performed when Varugis Kurien was with Microsoft.

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787496

ures, "live-site" incidents happen. A live-site incident is any event that results in an impact to the customers, partners or revenue. Live-site incidents need to be detected, mitigated, and resolved as soon as possible. But data center networks have hundreds of thousands to millions of servers, hundreds of thousands of switches, and millions of cables and fibers. Thus detecting where the problem is located is a hard problem.

To address the above challenges, we have designed and implemented Pingmesh, a large-scale system for data center network latency measurement and analysis. Pingmesh leverages all the servers to launch TCP or HTTP pings to provide the maximum latency measurement coverage. Pingmesh forms multiple levels of complete graphs. Within a data center, Pingmesh lets the servers within a rack form a complete graph and also uses the top-of-rack (ToR) switches as virtual nodes and let them form a second complete graph. Across data centers, Pingmesh forms a third complete graph by treating each data center as a virtual node. The calculation of the complete graphs and related ping parameters are controlled by a central Pingmesh Controller.

The measured latency data are collected and stored, aggregated and analyzed by a data storage and analysis pipeline. From the latency data, network SLAs are defined and tracked at both the macro level (i.e., data center level) and the micro level (e.g., per-server and per-rack levels). The network SLAs for all the services and applications are calculated by mapping the services and applications to the servers they use.

Pingmesh has been running in tens of globally distributed data centers of Microsoft for four years. It produces 24 terabytes of data and more than 200 billion probes per day. Because of the universal availability of the Pingmesh data, answering if a live-site incident is because of the network becomes easier: If Pingmesh data does not indicate a network problem, then the live-site incident is not caused by the network.

Pingmesh is heavily used for network troubleshooting to locate where the problem is. By visualization and automatic pattern detection, we are able to answer when and where packet drops and/or latency increases happen, identify silent switch packet drops and black-holes in the network. The results produced by Pingmesh is also used by application developers and service operators for better server selection by considering network latency and packet drop rate.

This paper makes the following contributions: We show the feasibility of building a large-scale network latency measurement and analysis system by designing and implementing Pingmesh. By letting every server participate, we provide latency data for all the servers all the time. We show that Pingmesh helps us better understand data center networks by defining and tracking network SLA at both macro and micro scopes, and that Pingmesh helps reveal and locate switch packet drops including packet black-holes and silent random packet drops, which were less understood previously.

2. BACKGROUND

2.1 Data center networks

Data center networks connect servers with high speed and provide high server-to-server bandwidth. Today's large data center networks are built from commodity Ethernet switches and routers [1, 12, 2].

Figure 1 shows a typical data center network structure. The network has two parts: intra data center (Intra-DC) network and inter data center (Inter-DC) network. The intra-DC network is typically a Clos network of several tiers similar to the network described in [1, 12, 2]. At the first tier, tens of servers (e.g., 40) use 10GbE or 40GbE Ethernet NICs to connect to a top-of-rack (ToR) switch and form a Pod. Tens of ToR switches (e.g., 20) are then connected to a second tier of Leaf switches (e.g., 2-8). These servers and ToR and Leaf switches form a Podset. Multiple Podsets then connect to a third tier of Spine switches (tens to hundreds). Using existing Ethernet switches, an intra-DC network can connect tens of thousands or more servers with high network capacity.

One nice property of the intra-DC network is that multiple Leaf and Spine switches provide a multi-path network with redundancy. ECMP (equal cost multi-path) is used to load-balance traffic across all the paths. ECMP uses the hash value of the TCP/UDP five-tuple for next hop selection. As a result, the exact path of a TCP connection is unknown at the server side even if the five-tuple of the connection is known. For this reason, locating a faulty Spine switch is not easy.

The inter-DC network is to interconnect the intra-DC networks and to connect the inter-DC networks to the Internet. The inter-DC network uses high-speed, long haul fibers to connect data centers networks at different geolocations. Software defined networking (SWAN [13], B4 [16]) are further introduced for better wide area network traffic engineering.

Our data center network is a large, sophisticated distributed systems. It is composed of hundreds of thousands of servers, tens of thousands switches and routers, and millions of cables and fibers. It is managed by Autopilot [20], our home-grown data center management software stack, and the switches and NICs run software and firmware provided by different switch and NIC providers. The applications run on top of the network may introduce complex traffic patterns.

2.2 Network latency and packet drops

In this paper we use the term "network latency" from application's point of view. When an application A at a server sends a message to an application B at a peer server, the network latency is defined as the time interval from the time A sends the message to the time B receives the message. In practice we measure round-trip-time (RTT) since RTT measurement does not need to synchronize the server clocks.

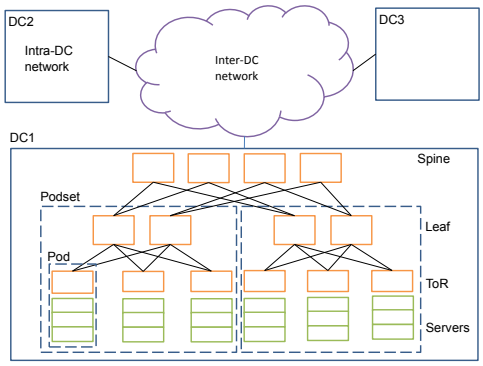


Figure 1: Data center network structure.

RTT is composed of application processing latency, OS kernel TCP/IP stack and driver processing latency, NIC introduced latency (e.g., DMA operations, interrupt modulation) [22], packet transmission delay, propagation delay, and queuing delay introduced by packet buffering at the switches along the path.

One may argue the latencies introduced by applications and kernel stack are not really from the network. In practice, our experiences have taught us that our customers and service developers do not care. Once a latency problem is observed, it is usually called a "network" problem. It is the responsibility of the network team to show if the problem is indeed a network problem, and if it is, mitigate and root-cause the problem.

User perceived latency may increase due to various reasons, e.g., queuing delay due to network congestion, busy server CPU, application bugs, network routing issues, etc. We also note that packet drops increase user perceived latency, since dropped packets need to be retransmitted. Packet drops may happen at different places due to various reasons, e.g., fiber FCS (frame check sequence) errors, switching ASIC defects, switch fabric flaw, switch software bug, NIC configuration issue, network congestions, etc. We have seen all these types of issues in our production networks.

2.3 Data center management and data processing systems

Next we introduce Autopilot [20] and Cosmos and SCOPE [15]. Data centers are managed by centralized data center management systems, e.g., Autopilot [20] or Borg [23]. These management systems provide frameworks on how resources including physical servers are managed, how services are deployed, scheduled, monitored and managed. Pingmesh is built within the framework of Autopilot.

Autopilot is Microsoft's software stack for automatic data center management. Its philosophy is to run software to automate all data center management tasks, including failure recovery, with as minimal human involvement as possible. Using the Autopilot terminology, a cluster, which is a set of servers connected by a local data center network, is managed by an Autopilot environment. An Autopilot environment has a set of Autopilot services including Device Manager (DM), which manages the machine state, Deployment Service (DS) which does service deployment for both Autopilot and various applications, Provisioning Service (PS) which installs Server OS images, Watchdog Service (WS) which monitors and reports the health status of various hardware and software, Repair Service (RS) which performs repair action by taking commands from DM, etc.

Autopilot provides a shared service mode. A shared service is a piece of code that runs on every autopilot managed server. For example, a Service Manager is a shared service that manages the life-cycle and resource usage of other applications, a Perfcounter Collector is a shared service that collects the local perf counters and then uploads the counters to Autopilot. Shared services must be light-weight with low CPU, memory, and bandwidth resource usage, and they need to be reliable without resource leakage and crashes.

Pingmesh uses our home-grown data storage and analysis system, Cosmos/SCOPE, for latency data storage and analysis. Cosmos is Microsoft's BigData system similar to Hadoop [3] which provides a distributed file system like GFS [17] and MapReduce [11]. Files in Cosmos are append-only and a file is split into multiple 'extents' and an extent is stored in multiple servers to provide high reliability. A Cosmos cluster may have tens of thousands of servers or more, and gives users almost 'infinite' storage space.

SCOPE [15] is a declarative and extensible scripting language, which is built on top of Cosmos, to analyze massive data sets. SCOPE is designed to be easy to use. It enables users to focus on their data instead of the underlying storage and network infrastructure. Users only need to write scripts similar to SQL without worrying about parallel execution, data partition, and failure handling. All these complexities are handled by SCOPE and Cosmos.

3. DESIGN AND IMPLEMENTATION

3.1 Design goal

The goal of Pingmesh is to build a network latency measurement and analysis system to address the challenges we have described in Section 1. Pingmesh needs to be always-on and be able to provide network latency data for all the servers. It needs to be always on because we need to track the network status all the time. It needs to produce network latency data for all the servers because the maximum possible network latency data coverage is essential for us to better understand, manage, and troubleshoot our network infrastructure.

From the beginning, we differentiated Pingmesh from various public and proprietary network tools (e.g., traceroute, TcpPing, etc.). We realized that network tools do not work for us because of the following reasons. First, these tools are not always-on and they only produce

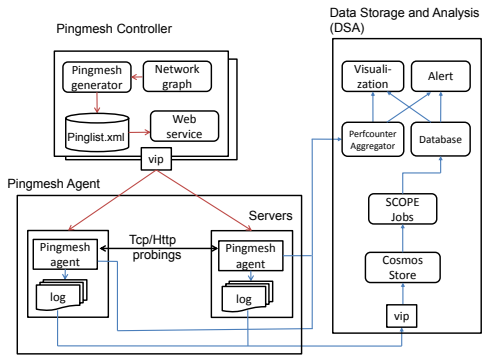


Figure 2: Pingmesh architecture.

data when we run them. Second, the data they produce does not have the needed coverage. Because these tools are not always-on, we cannot count on them to track the network status. These tools are usually used for network troubleshooting when a source-destination pair is known. This, however, does not work well for large-scale data center networks: when a network incident happens, we may not even know the source-destination pair. Furthermore, for transient network issues, the problem may be gone before we run the tools.

3.2 Pingmesh architecture

Based on its design goal, Pingmesh needs to meet the requirements as follows. First, because Pingmesh aims to provide the largest possible coverage and measure network latency from applications' point of view, a Pingmesh Agent is thus needed on every server. This has to be done carefully so that the CPU, memory, and bandwidth overhead introduced by Pingmesh Agent is small and affordable.

Second, the behavior of the Pingmesh Agent should be under control and configurable. A highly reliable control plane is needed to control how the servers should carry out network latency measurement.

Third, the latency data should be aggregated, analyzed, and reported in near real-time and stored and archived for deeper analysis. Based on the requirements, we have designed the architecture of Pingmesh as illustrated in Figure 2. Pingmesh has three components as we describe as follows.

Pingmesh Controller. It is the brain of the whole system, as it decides how servers should probe each other. Within the Pingmesh Controller, a Pingmesh Generator generates a pinglist file for every server. The pinglist file contains the list of peer servers and related parameters. The pinglist files are generated based on the network topology. Servers get their corresponding pinglist via a RESTful Web interface.

Pingmesh Agent. Every server runs a Pingmesh Agent. The Agent downloads the pinglist from the Pingmesh Controller, and then launches TCP/HTTP pings to the peer servers in the pinglist. The Pingmesh Agent stores the ping results in local memory. Once a timer times out or the size of the measurement results exceeds a threshold, the Pingmesh Agent uploads the results to Cosmos for data storage and analysis. The Pingmesh Agent also exposes a set of performance counters which are periodically collected by a Perfcounter Aggregator (PA) service of Autopilot.

Data Storage and Analysis (DSA). The latency data from Pingmesh Agents are stored and processed in a data storage and analysis (DSA) pipeline. Latency data is stored in Cosmos. SCOPE jobs are developed to analyze the data. SCOPE jobs are written in declarative language similar to SQL. The analyzed results are then stored in an SQL database. Visualization, reports and alerts are generated based on the data in this database and the PA counters.

3.3 Pingmesh Controller

3.3.1 *The pinglist generation algorithm*

The core of the Pingmesh Controller is its Pingmesh Generator. The Pingmesh Generator runs an algorithm to decide which server should ping which set of servers. As aforementioned, we would like Pingmesh to have as large coverage as possible. The largest possible coverage is a server-level complete graph, in which every server probes the rest of the servers. A server-level complete graph, however, is not feasible because a server needs to probe $n-1$ servers, where n is the number of servers. In a data center n can be as large as hundreds of thousands. Also a server-level complete graph is not necessary since tens of servers connect to the rest of the world through the same ToR switch.

We then come up with a design of multiple level of complete graphs. Within a Pod, we let all the servers under the same ToR switch form a complete graph. At intra-DC level, we treat each ToR switch as a virtual node, and let the ToR switches form a complete graph. At inter-DC level, each data center acts as a virtual node, and all the data centers form a complete graph.

In our design, only servers do pings. When we say a ToR as a virtual node, it is the servers under the ToR that carry out the pings. Similarly, for a data center as a virtual node, it is the selected servers in the data center that launch the probings.

At the intra-DC level, we once thought that we only need to select a configurable number of servers to participate in Pingmesh. But how to select the servers becomes a problem. Further, the small number of selected servers may not well represent the rest of the servers. We finally come up with the idea of letting *all* the servers participate. The intra-DC algorithm is: for any ToR-pair (ToR_x, ToR_y), let server i in ToR_x ping server i in ToR_y. In Pingmesh, even when two servers are in the pinglists of each other, they measure network latency separately. By doing so, every server can calculate its own packet drop rate and network latency locally and independently.

At the inter-DC level, all the DCs form yet another complete graph. In each DC, we select a number of servers (with several servers selected from each Podset).

Combining the three complete graphs, a server in Pingmesh needs to ping 2000-5000 peer servers depending on the size of the data center. The Pingmesh Controller uses threshold values to limit the total number of probes of a server and the minimal time interval of two successive probes for a source destination server pair.

3.3.2 Pingmesh Controller implementation

The Pingmesh Controller is implemented as an Autopilot service and becomes part of the Autopilot management stack. It generates Pinglist file for every server by running the Pingmesh generation algorithm. The files are then stored in SSD and served to the servers via a Pingmesh Web service. The Pingmesh Controller provides a simple RESTful Web API for the Pingmesh Agents to retrieve their Pinglist files respectively. The Pingmesh Agents need to periodically ask the Controller for Pinglist files and the Pingmesh Controller does not push any data to the Pingmesh Agents. By doing so, Pingmesh Controller becomes stateless and easy to scale.

As the brain of the whole Pingmesh system, the Pingmesh Controller needs to serve hundreds of thousands of Pingmesh Agents. Hence the Pingmesh Controller needs to be fault-tolerant and scalable. We use Software Load-Balancer (SLB) [14] to provide fault-tolerance and scalability for the Pingmesh Controller. See [9, 14] for the details of how SLB works. A Pingmesh Controller has a set of servers behind a single VIP (virtual IP address). SLB distributes the requests from the Pingmesh Agents to the Pingmesh Controller servers. Every Pingmesh Controller server runs the same piece of code and generates the same set of Pinglist files for all the servers and is able to serve requests from any Pingmesh Agent. The Pingmesh Controller can then easily scale out by adding more servers behind the same VIP. Also once a Pingmesh Controller server stops functioning, it is automatically removed from rotation by the SLB. We have setup two Pingmesh Controllers in two different data center clusters to make the controller even more fault tolerant geographically.

3.4 Pingmesh Agent

3.4.1 Pingmesh Agent design considerations

The Pingmesh Agent runs on all the servers. Its task is simple: downloads pinglist from the Pingmesh Controller; pings the servers in the pinglist; then uploads the ping result to DSA.

Based on the requirement that Pingmesh needs to be able to distinguish if a user perceived latency increase is due to network or not, Pingmesh should use the same type of packets generated by the applications. Since almost all the applications in our data centers use TCP and HTTP, Pingmesh uses TCP and HTTP instead of ICMP or UDP for probing.

Because we need to differentiate if a 'network' issue is because of the network or the applications themselves, Pingmesh Agent does not use any network libraries used by the applications. Instead, we have developed our own light-weight network library specifically designed for network latency measurement.

The Pingmesh Agent can be configured to send out and respond to probing packets of different lengths, other than the TCP SYN/SYN-ACK packets. As a result, the Pingmesh Agent needs to act as both client and server. The client part launches pings and the server part responds to the pings.

Every probing needs to be a new connection and uses a new TCP source port. This is to explore the multi-path nature of the network as much as possible, and more importantly, reduce the number of concurrent TCP connections created by Pingmesh.

3.4.2 Pingmesh Agent implementation

Though the task is simple, the Pingmesh Agent is one of the most challenging part to implement. It must meet the safety and performance requirements as follows.

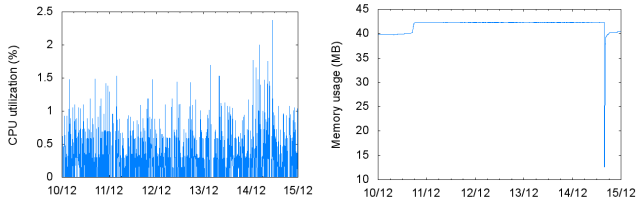
First, the Pingmesh Agent must be fail-closed and not create live-site incidents. Since the Pingmesh Agent runs on every server, it has the potential to bring down all the servers if it malfunctions (e.g., uses large portion of CPU and memory resources, generates large volume of probing traffic, etc.). To avoid bad things from happening, several safety features have been implemented into the Pingmesh Agent:

- The CPU and maximum memory usages of the Pingmesh Agent are confined by the OS. Once the maximum memory usage exceeds the cap, the Pingmesh Agent will be terminated.

- The minimum probe interval between any two servers is limited to 10 seconds, and the probe payload length is limited to 64 kilobytes. These limits are hard coded in the source code. By doing so, we put a hard limit on the worst-case traffic volume that Pingmesh can bring into the network.

- If a Pingmesh Agent cannot connect to its controller for 3 times, or if the controller is up but there is no pinglist file available, the Pingmesh Agent will remove all its existing ping peers and stop all its ping activities. (It will still react to pings though.) Due to this feature, we can stop the Pingmesh Agent from working by simply removing all the pinglist files from the controller.

- If a server cannot upload its latency data, it will retry several times. After that it will stop trying and discard the in-memory data. This is to ensure the Pingmesh Agent uses bounded memory resource. The Pingmesh Agent also writes the latency data to local disk as log files. The size of log files is limited to a configurable size.



(a) CPU usage (b) Memory usage

Figure 3: CPU and memory usages of Pingmesh Agent.

Second, a Pingmesh Agent needs to launch pings to several thousand of servers by design. But as a shared service, the Pingmesh Agent should minimize its resources (CPU, memory, and disk space) usage. It should use close to zero CPU time and as small memory footprint as possible, so as to minimize its interference with customers' applications.

In order to achieve the performance goal and improve Pingmesh's latency measurement accuracy, we use C++ instead of Java or C# to write Pingmesh Agent. This is to avoid the common language runtime or Java virtual machine overhead. We have developed a network library specifically for Pingmesh. The goal of the library is solely for network latency measurement, and it is designed to be light-weight and to handle a large number of concurrent TCP connections. The library is directly based on the Winsock API, and it uses the Windows IO Completion Port programming model for efficient asynchronous network IO processing. The library acts as both client and server, and it distributes the probing processing load to all the CPU cores evenly.

We have done extensive measurements to understand and optimize Pingmesh Agent's performance. Figure 3 shows the CPU and memory usages of the Pingmesh Agent on a typical server. During the measurement, this Pingmesh Agent was actively probing around 2500 servers. The server has 128GB memory and two Intel Xeon E5-2450 processors, each with 8 cores. The average memory footprint is less than 45MB, and the average CPU usage is 0.26%.

We note that the probing traffic generated by a Pingmesh Agent is small, typically tens of Kb/s. As a comparison, our data center network provides several Gb/s throughput between any two servers in a data center.

3.5 Data Storage and Analysis

For Pingmesh data storage and analysis, we use the well established existing systems, Cosmos/SCOPE and Autopilot's Perfcounter Aggregator (PA) service, instead of reinventing the wheel.

The Pingmesh Agent periodically uploads the aggregated records to Cosmos. Similar to the Pingmesh Controller, the front-end of Cosmos uses load-balancer and VIP to scale out. At the same time, the Pingmesh Agent performs local calculation on the latency data and produces a set of performance counters including the packet drop rate, the network latency at 50^{th} the 99^{th} percentile, etc. All these performance counters are collected and aggregated and stored by the PA service of Autopilot.

Once the results are in Cosmos, we run a set of SCOPE jobs for data processing. We have 10-min, 1-hour, 1-day jobs at different time scales. The 10-min jobs are our near real-time ones. For the 10-min jobs, the time interval from when the latency data is generated to when the data is consumed (e.g., alert fired, dashboard figure generated) is around 20 minutes. The 1-hour and 1-day pipelines are for non real-time tasks including network SLA tracking, network black-hole detection, packet drop detection, etc. All our jobs are automatically and periodically submitted by a Job Manager to SCOPE without user intervention. The results of the SCOPE jobs are stored in a SQL database, from which visualization, reports, and alerts are generated.

In practice, we found this 20-minute delay works fine for system level SLA tracking. In order to further reduce response time, we in parallel use the Autopilot PA pipeline to collect and aggregate a set of Pingmesh counters. The Autopilot PA pipeline is a distributed design with every data center has its own pipeline. The PA counter collection latency is 5 minutes, which is faster than our Cosmos/SCOPE pipeline. The PA pipeline is faster than Cosmos/SCOPE, whereas Cosmos/SCOPE is more expressive than PA for data processing. By using both of them, we provide higher availability for Pingmesh than either of them.

We differentiate Pingmesh as an always-on service from a set of scripts that run periodically. All the components of Pingmesh have watchdogs to watch whether they are running correctly or not, e.g., whether pinglists are generated correctly, whether the CPU and memory usages are within budget, whether pingmesh data are reported and stored, whether DSA reports network SLAs in time, etc. Furthermore, the Pingmesh Agent is designed to probe thousands of peers in a light-weight and safe way.

All the Pingmesh Agents upload 24 terabytes latency measurement results to Cosmos per day. This is more than 2Gb/s upload rate. Though these look like large numbers, they are only a negligible fraction of the total capacity provided by our network and Cosmos.

4. LATENCY DATA ANALYSIS

In this section, we introduce how Pingmesh helps us better understand network latency and packet drops, define and track the network SLA, and determine if a live-site incident is because of network issues or not. All the data centers we describe in this section have similar network architecture as we have introduced in Figure 1, though they may vary in size and may be built at different times.

4.1 Network latency

Figure 4 shows the intra-DC latency distribution of two representative data centers DC1 in US West and DC2 in US Central. DC1 is used by distributed storage and MapReduce and DC2 is by an interactive Search service. Servers in DC1 are throughput intensive and the average server CPU utilization is as high as 90%. Servers in DC1 use the network heavily and transmit and receive several hundreds of Mb/s data on average all the time. DC2 is latency sensitive and servers have high fan-in and fan-out in that a server needs to communicate with a large number of other servers to service a Search query. The average CPU utilization in DC2 is moderate and the average network throughput is low but the traffic is bursty.

The CDFs in Figure 4 are calculated from latency data of one normal working day, when there were no network incidents detected and no live-site incidents because of the network. We track both intra-pod and inter-pod latency distributions, with and without TCP payload. If not specifically mentioned, the latency we use in the paper is the inter-pod TCP SYN/SYN-ACK RTT without payload.

Figure 4(a) shows the overall inter-pod latency distributions and Figure 4(b) shows the inter-pod distribution at high percentile. We once expected that the latency of DC1 should be much larger than DC2 since servers and the network in DC1 are highly loaded. But this turned out not the case for latencies at the 90^{th} or lower percentile.

But DC1 does have much higher latency at the high percentile as shown in Figure 4(b). At P99.9, the inter-pod latencies are 23.35ms and 11.07ms for DC1 and DC2, respectively. At P99.99, the inter-pod latencies become 1397.63ms and 105.84ms. Our measurement result shows it is hard to provide low latency (e.g., sub-milliseconds level) at three or four 9s, even when the servers and network are both light-loaded at macro time scale. This is because the server OS is not a real-time operating system and the traffic in our network is burst. We see 10^-5 packet drop rate for intra-pod communications (Section 4.2) even when average network utilization is low to moderate.

Figure 4(c) compares the intra-pod and inter-pod latency distributions, and Figure 4(d) studies the inter-pod latency with and without payload, all in DC1. For latency measurement with payload, after TCP connection setup, we let the client send a message (typically 800-1200 bytes within one packet). The client measures the payload latency once it receives the echoed back message from the server.

As shown in Figure 4(c), intra-pod latency is always smaller than inter-pod latency as expected. The 50^{th} (P50) and the 99^{th} (P99) intra-pod and inter-pod latencies are (216us, 1.26ms) and (268us, 1.34ms) for DC1. The differences at P50 and P99 are 52us and 80us, respectively. These numbers show that the network does introduce tens of microsecond latency due to queuing delay. But the queuing delay is small. Hence we can infer that the network provides enough network capacity.

Data center	Packet drop rate	
	Intra-pod	Inter-pod
DC1 (US West)	1.31×10^{-5}	7.55×10^{-5}
DC2 (US Central)	2.10×10^{-5}	7.63×10^{-5}
DC3 (US East)	9.58×10^{-6}	4.00×10^{-5}
DC4 (Europe)	1.52×10^{-5}	5.32×10^{-5}
DC5 (Asia)	9.82×10^{-6}	1.54×10^{-5}

Table 1: Intra-pod and inter-pod packet drop rates.

Figure 4(d) shows the latency difference with and without payload. With payload, the latency increases from 268us to 326us at P50, and from 1.34ms to 2.43ms at P99, respectively. The increase is mainly because of the increased transmission delay [1] and the user space processing overhead for the receiving servers to echo back the message. In most cases, the latency distributions with and without payload are similar. We introduced payload ping because it can help detect packet drops that are related to packet length (e.g., fiber FCS errors and switch SerDes errors that are related to bit error rate).

Based on the Pingmesh data, we are able to calculate not only the latency distributions of the data centers, but also the latency CDFs for all the applications and services. From these results, we are able to track network latency for all of them all the time.

4.2 Packet drop rate

Pingmesh does not directly measure packet drop rate. However, we can infer packet drop rate from the TCP connection setup time. When the first SYN packet is dropped, TCP sender will retransmit the packet after an initial timeout. For the rest successive retries, TCP will double the timeout value every time. In our data centers, the initial timeout value is 3 seconds, and the sender will retry SYN two times. Hence if the measured TCP connection RTT is around 3 seconds, there is one packet drop; if the RTT is around 9 seconds, there are two packet drops. We use the following heuristic to estimate packet drop rate:

$$\frac{probes\_with\_3s\_rtt + probes\_with\_9s\_rtt}{total\_successful\_probes}.$$

Note that we only use the total number of successful TCP probes instead of the total probes as the denominator. This is because for failed probes, we cannot differentiate between packet drops and receiving server failure. In the numerator, we only count one packet drop instead of two for every connection with 9 second RTT. This is because successive packet drops within a connection are not independent: the probability the second SYN is dropped is much higher if the first SYN is

[1]We have disabled cut-through switching at the switches in our data centers. This is to stop FCS errors from propagation.

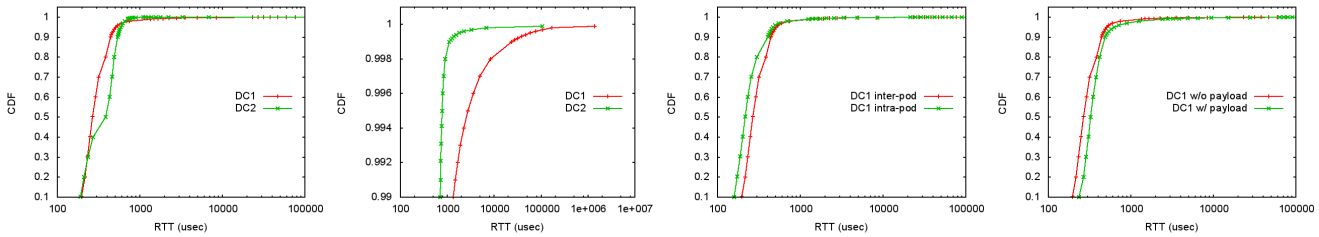


Figure 4: (a)Inter-pod latency of two data centers. (b) Inter-pod latency at high percentile. (c) Intra-pod and inter-pod latency comparison. (d) Latency comparison with and without payload.

dropped. We have verified the accuracy of the heuristic for a single ToR network by counting the NIC and ToR packet drops.

In our network, SYN packets are treated the same as other packets. Hence the drop rate of SYN packets can be considered representative drop rate of the other data packets in normal condition. This assumption, however, may not be true when packet drop rate is related to packet size, e.g., due to FCS errors. We did see packets of larger size may experience higher drop rate in FCS error related incidents. In what follows, the results we present are when the networks were in normal condition.

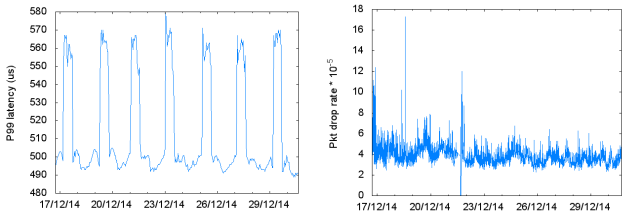
Our network does not differentiate packets of different IP protocols (e.g., TCP vs. UDP). Hence, our packet drop calculation holds for non-TCP traffic as well.

Table 1 shows the packet drop rates of five data centers. We show both the intra-pod and inter-pod packet drop rates. For intra-pod packet drops, those are drops at ToR switch, NIC, and end-host network stack. The inter-pod packet drops may come from the Leaf and Spine switches and the corresponding links, in addition to the ToR, NIC, and end-host stack.

From Table 1, several observations can be made. First, the packet drop rates are in the range of $10^{-4} - 10^{-5}$. We track the packet drop rates for all our data centers every day and we find the drop rate is in this range unless network incidents happen. Second, the inter-pod packet drop rate is typically several times higher than that of intra-pod. This indicates most of the packet drops happen in the network instead of the hosts. Third, the intra-pod drop rate is around 10^{-5}, which is larger than we have expected.

Our experience tells us packet drops may occur due to many reasons, e.g., switch buffer overflow, NIC receiving buffer overflow, optical fiber FCS errors, switching ASIC malfunction, etc. Though our measurement results suggest that the packet drop rate at normal condition is around $10^{-4} - 10^{-5}$, we are still at the early phase in understanding why it stays in this range.

Many data center applications, e.g., Search, may use hundreds or even thousands of TCP connections simultaneously. For these applications, high latency tail therefore becomes the norm due to the large number of connections used. Applications have introduced several application level tricks to deal with packet drops [10].



(a) The 99^{th} percentile latency (b) Packet drop rate

Figure 5: The 99^{th} network latency and packet drop rate metrics for a service.

From the per server latency data, we can calculate and track network SLAs at server, pod, podset, and data center levels. Similarly, we can calculate and track network SLA for individual services.

4.3 Is it a network issue?

In large distributed data center systems, many parts may go wrong. When a live-site incident happens, it is not easy to identify which part causes the problem. There are occasions that all the components seem fine but the whole system is broken. If the network cannot prove it is innocent, the problem will then be called a "network problem": I did not do anything wrong to my service, it must be the fault of the network.

The network team is then engaged to investigate. A typical procedure is as follows. The network on-call engineer asks the service which is experiencing issues for detailed symptoms and source-destination pairs; he then logs into the source and/or destination servers and runs various network tools to reproduce the issue; he may also look at the switch counters along the possible paths for anomaly; if he cannot reproduce, he may ask for more source-destination pairs. The procedure may need several rounds of iterations.

The above approach does not work well for us, since it is a manual process and does not scale. If the issue turns out not to be caused by the network, the service owners waste their time in engaging with the wrong

team. If the issue is indeed because of the network, the manual process causes long time-to-detect (TTD), time-to-mitigate (TTM), and time-to-resolve (TTR).

Pingmesh changed the situation. Because Pingmesh collects latency data from all the servers, we can always pull out Pingmesh data to tell if a specific service has network issue or not. If Pingmesh data does not correlate to the issue perceived by the applications, then it is not a network issue. If Pingmesh data shows it is indeed a network issue, we can further get detailed data from Pingmesh, e.g., the scale of the problem (e.g., how many servers and applications are affected), the source-destination server IP addresses and TCP port numbers, for further investigation.

We define network SLA as a set of metrics including packet drop rate, network latency at the 50^{th} percentile and the 99^{th} percentile. Network SLA can then be tracked at different scopes including per server, per pod/podset, per service, per data center, by using the Pingmesh data. In practice we found two network SLA metrics: packet drop rate and network latency at the 99^{th} percentile are useful for telling if an issue is caused by the network or not. Figure 5 shows these two metrics for a service in one normal week. The packet drop rate is around 4×10^{-5} and the 99^{th} percentile latency in a data center is 500-560us. (The latency shows a periodical pattern. This is because this service performs high throughput data sync periodically which increases the 99^{th} percentile latency.) If these two metrics change significantly, then it is a network issue.

We currently use a simple threshold based approach for network SLA violation detection. If the packet drop rate is greater than 10^{-3} or the 99^{th} percentile latency is larger than 5ms, we will categorize this as a network problem and fire alerts. 10^{-3} and 5ms are much larger than the normal values. We keep Pingmesh historical data for 2 months, and we run various data analysis on top of the Pingmesh data to track the network SLAs for different data centers and customers. There are huge opportunities in using data mining and machine learning to get more value out of the Pingmesh data.

In Section 5, we will study one specific packet drop in detail: switch silent packet drops.

5. SILENT PACKET DROP DETECTION

In this section, we introduce how Pingmesh helps detect switch silent packet drops. When silent packet drops happen, the switches for various reasons do not show information about these packet drops and the switches seem innocent. But applications suffer from increased latency and packet drops. How to quickly identify if an ongoing live-site incident is caused by switch silent packet drops therefore becomes critical.

In the past, we have identified two types of switch silent packet drops: packet black-hole and silent random packet drops. Next, we introduce how we use Pingmesh to detect them.

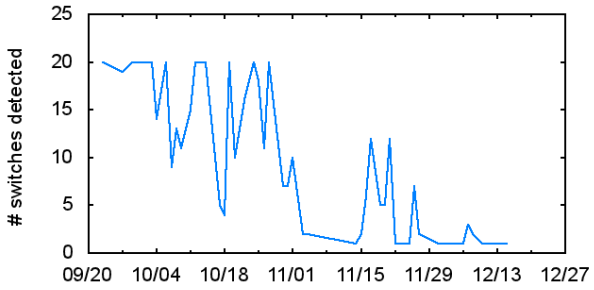


Figure 6: The number of switches with packet black-holes detected.

5.1 Packet black-hole

Packet black-hole is a special type of switch packet drops. For a switch that is experiencing packet black-holes, packets that meet certain 'patterns' are dropped deterministically (i.e., 100%) by the switch. We have identified two types of packet black-holes. In the first type of black-hole, packets with specific source destination IP address pairs get dropped. The symptom is as following: server A cannot talk to server B, but it can talk to servers C and D just fine. All the servers A-D are healthy.

In the second type of black-hole, packets with specific source destination addresses and transport port numbers are dropped. Note that for this type of black-hole, packets with the same source destination address pair but different source destination port numbers are treated differently. For example, Server A can talk to Server B's destination port Y using source port X, but not source port Z.

The first type of black-holes is typically caused by TCAM deficits (e.g., parity error) in the switching ASIC. Some TCAM entries in the TCAM table may get corrupted, and the corruption causes only packets with certain source and destination address patterns been dropped. (Since only destination address is used for next-hop lookup for IP routing, on may wonder why source IP address plays an role. Our guess is that a TCAM entry includes not only destination address but also source address and other meta data.)

We know less about the root causes of the second type of black-hole. We suspect it may be because of errors related to ECMP which uses source and destination IP addresses and port numbers to decide the next forwarding hop.

Based on our experience, these two types of packet black-holes can be fixed by reloading the switch. Hence the question becomes how to detect the switches with black-holes.

We have devised a ToR switch black-hole detection algorithm based on Pingmesh data. The idea of the algorithm is that if many servers under a ToR switch experience the black-hole symptom, then we mark the

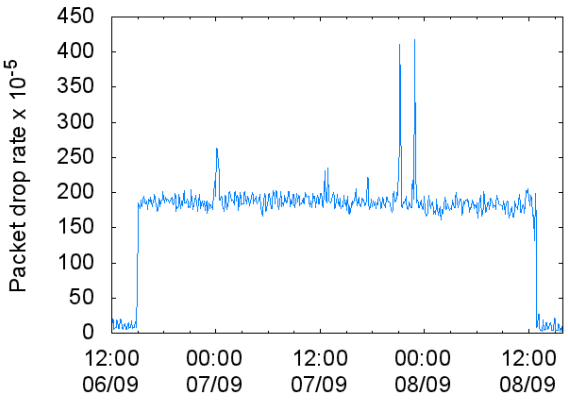


Figure 7: Silent random packet drops of a Spine switch detected by Pingmesh during an incident.

ToR switch as a black-hole candidate and assign it a score which is the ratio of servers with black-hole symptom. We then select the switches with black-hole score larger than a threshold as the candidates. Within a podset, if only part of the ToRs experience the black-hole symptom, then those ToRs are blacking hole packets. We then invoke a network repairing service to safely restart the ToRs. If all the ToRs in a podset experience the black-hole symptom, then the problem may be in the Leaf or Spine layer. Network engineers are notified to do further investigation.

Figure 6 shows the number of ToR switches with black-holes the algorithm detected. As we can see from the figure, the number of the switches with packet black-holes decreases once algorithm began to run. In our algorithm, we limit the algorithm to reload at most 20 switches per day. This is to limit the maximum number of switch reboots. As we can see, after a period of time, the number of switches detected dropped to only several per day.

We would like to note that the TCP source port of the Pingmesh Agent varies for every probing. With the large number of source/destination IP address pairs, Pingmesh scans a big portion of the whole source/destination address and port space. After Pingmesh black-hole detection came online, our customers did not complain about packet black-holes anymore.

5.2 Silent random packet drops

The higher the tier a switch is located in the network, the more severe impact it will have when it begins to drop packets. When a Spine switch drops packets silently, tens of thousands of servers and many services will be impacted and live-site incidents with high severity will be triggered.

Here we introduce how Pingmesh helped locate silent random packet drops of a Spine switch. In one incident, all the users in a data center began to experience increased network latency at the 99^{th} percentile. Using Pingmesh, we could confirm that the packet drops in that data center has increased significantly and the drops were not deterministic. Fig. 7 shows the packet drop rate change of a service. Under normal condition, the percentage of latency should be at around $10^{-4} - 10^{-5}$. But it suddenly jumped up to around 2×10^{-3}.

Using Pingmesh, we could soon figure out that only one data center was affected, and the other data centers were fine. Packet drops at ToR and Leaf layers cannot cause the latency increase for all our customers due to the much smaller number of servers under them. The latency increase pattern shown in Figure 8(d) pointed the problem to the Spine switch layer.

But we could not find any packet drop hint (FCS errors, input/output packet discards, syslog errors, etc.) at those switches. We then suspected that this is likely a case of silent packet drops. The next step was to locate the switches that were dropping packets.

Again, by using Pingmesh, we could figure out several source and destination pairs that experienced around 1%-2% random packet drops. We then launched TCP traceroute against those pairs, and finally pinpointed one Spine switch. The silent random packet drops were gone after we isolated the switch from serving live traffic. The postmortem analysis with the switch provider revealed that the packet drops were due to bit flips of a fabric module of that switch.

The above case is one of the first silent random packet drop cases we met and it took us long time to resolve. After that we ran into more cases and we have improved both Pingmesh data analysis and other tools for better automatic random silent packet drop detection. Our experiences told us that random silent packet drops may be because of different reasons, e.g., switching fabric CRC checksum error, switching ASIC deficit, linecard not well seated, etc. These types of switch silent packet drops cannot be fixed by switch reload and we have to RMA (return merchandise authorization) the faulty switch or components.

Compared to packet drops due to network congestion and link FCS errors, packet black-holes and silent random drops are new and less understood to us. Due to the whole coverage and always-on properties of Pingmesh, we are able to confirm that switch silent packet drops do happen in real-world and categorize different silent packet drop types, and further locate where the silent packet drops happen.

6. EXPERIENCES LEARNED

Pingmesh is designed to be scalable. We understand that not all the networks are of our size. We believe that the lessons we learned from Pingmesh are beneficial to networks of both large and small scales. One of the lessons we learned is the value of trustworthy latency data of full coverage. If the data is not trustworthy, then the results built on top of it cannot be trusted. Our experience told us that not all SNMP

data are trustworthy. A switch may drop packets even though its SNMP tells us everything is fine. We trust Pingmesh data because we wrote the code, tested and ran it. When there are bugs, we fixed them. After several iterations, we knew we can trust the data. Because of the full coverage and trustworthy of its latency data, Pingmesh could carry out accurate black-hole and silent packet drop detection. As a comparison, simply using switch SNMP and syslog data does not work since they do not tell us about packet black-holes and silent drops.

In what follows, we introduce several additional lessons we have learned from building and running Pingmesh, which we believe can be applied to networks of different scales as well.

6.1 Pingmesh as an always-on service

From the beginning of the project, we believed that Pingmesh needs to cover all the servers and be always-on. But not everyone agreed. There were arguments that latency data should only be collected on-demand; that we should only let a few selected servers participate in latency measurement, so as to reduce the overhead. We disagree with both of them.

In its essence, the first argument is always-on vs on-demand. One may argue that it is a waste of resource if the always-on latency data is not used, hence we should only collect latency data when it is needed.This argument has two issues. First, we cannot predict when the latency data will be needed since we do not know when a live-site incident will happen. When a live-site incident occurs, having network latency data readily at hands instead of collecting them at that time is a much better choice. Second, when something bad happens, we typically do not know which network devices caused the trouble, hence we do not even have the source destination pairs to launch latency measurement.

Using only a small number of selected servers for latency measurement limits the coverage of Pingmesh data, and poses challenges on which servers should be chosen. As we have demonstrated in the paper, letting all the servers participate gives us the maximum possible coverage, and easily balance the probing activity among all the servers. As we have demonstrated in the paper, the CPU, memory and bandwidth overhead introduced by Pingmesh is affordable.

Having latency data that is always-on brings benefits that we did not recognize in the beginning. After experiencing a few live-site incidents due to packet black-hole and switch silent packet drops, we found that we could use the Pingmesh data to automatically detect these types of switch failures, because of the whole coverage and always-on nature of Pingmesh data (Section 5).

6.2 Loosely coupled components help evolvement

Pingmesh benefits from a loosely coupled system design. Pingmesh Controller and Pingmesh Agent interact only through the pinglist files, which are standard XML files, via standard Web API. Pingmesh Agent provides latency data as both CSV files and standard performance counters.

Due to its loosely coupled design, Pingmesh could be built step by step in three phases. In the first phase, we focused on Pingmesh Agent. We built a simple Pingmesh Controller which statically generates pinglist files using a simplified pinglist generation algorithm. The latency data was simply put into Cosmos without automatic analysis. This phase demonstrated the feasibility of Pingmesh. At the end of this phase, the latency data was already used for network SLA calculation.

In the second phase, we built a full fledged Pingmesh Controller which automatically updates pinglists once network topology is updated or configuration is adjusted. The new version of Pingmesh Controller is also of higher capacity and more fault tolerant by setting up multiple controllers in geo-distributed data centers.

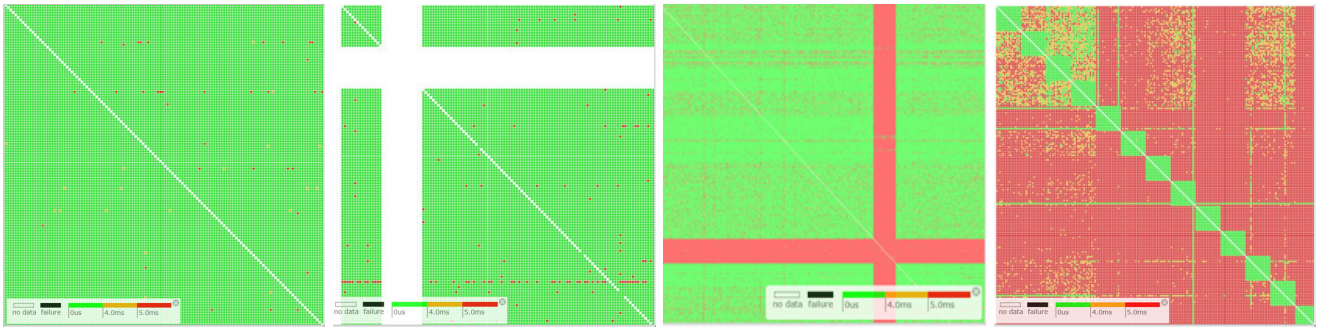
In the third phase, we focused on data analysis and visualization. We built a data processing pipeline which automatically analyzes the collected latency data in every 10 minutes, one hour, one day, respectively. The processed results are then stored in database for visualization, report and alert services.

The major tasks of these three phases were finished in June 2012. After that, many new features were added into Pingmesh:

Inter-DC Pingmesh. Pingmesh originally worked for intra-DC. However, extending it to cover Inter-DC is easy. We extended the Pingmesh Controller's pinglist generation algorithm so as to select a set of servers from every data center and let them carry out Inter-DC ping and the job was done. There is no single line of code or configuration change of the Pingmesh Agent. We did add a new inter-DC data processing pipeline though.

QoS monitoring. After Pingmesh was deployed, network QoS was introduced into our data center which differentiates high priority and low priority packets based on DSCP (differentiated service code point). Again, we extended the Pingmesh Generator to generate pinglists for both high and low priority classes. In this case, we did need a simple configuration change of the Pingmesh Agent to let it listen to an additional TCP port which is configured for low priority traffic.

VIP monitoring. Pingmesh was originally designed to measure network latency of physical networks. In our data centers, load-balancing and IP address virtualization is widely used. Address virtualization exposes a logical Virtual IP address (VIP) to users, and the VIP is mapped to a set of physical servers. The physical IP addresses of these servers are called DIP (destination IP). In our load-balancing system, there is a control plan maintains the VIP to DIP mapping and a data plan that delivers packets that target for a VIP to the DIPs via packet encapsulation. When Pingmesh got deployed, a natural extension is let Pingmesh to monitor the availability of the VIPs. This again is done by extending the Pingmesh Generation algorithm to cover



(a) Normal (b) Podset down (c) Podset failure (d) Spine failure

Figure 8: Network latency patterns through visualization.

the VIPs as the target, without touching the rest of the Pingmesh pipeline.

Silent packet drop detection. As we have discussed in Section 5, we have been using Pingmesh for silent packet drop detection. Since the latency data is already there, we only need to figure out the detection algorithm and implement the algorithm in the DSA pipeline without touching other Pingmesh components.

Network metrics for services. Two Pingmesh metrics have been used by service developers to design and implement better services. The Pingmesh Agent exposes two PA counters for every server: the 99^{th} latency and the packet drop rate. Service developers can use the 99^{th} latency to get better understanding of data center network latency at server level. The per-server packet drop rate has been used by several services as one of the metrics for server selection.

For the above extensions, only inter-DC Pingmesh and QoS monitoring were by design, the rest three just happened out of our expectation. Thanks to Pingmesh's loosely coupled design, all these features were added smoothly without adjusting its architecture.

6.3 Visualization for pattern discovery

We have invested heavily in Pingmesh data analysis and visualization. Our happy findings are that data speaks for themselves and that visualization helps us better understand and detect various latency patterns.

Figure 8 shows several typical visualized latency patterns. In the figure, a small green, yellow, or red block or pixel shows the network latency at the 99th percentile between a source-destination pod-pair. Green means the latency is less than 4ms, yellow means the latency is between 4-5ms, and red is for latency larger than 5ms. A white block means there is no latency data available.

Figure 8(a) shows an (almost) all-green pattern, which means the network works fine. Though looks straightforward, this all-green pattern is one of the most widely used feature of Pingmesh. Using this pattern, we can easily tell the global healthy status of the network.

Figure 8(b) shows a pattern of a white-cross. The width of the white-cross corresponds to a Podset, which contains around 20 pods. This pattern shows a Podset-down scenario. Podset-down typically is due to the loss of power of the whole Podset.

Figure 8(c) shows a pattern of a red-cross. The width of the red-cross again corresponds to a Podset. The red-cross shows high network latency from and to the Podset. This pattern shows there is a network issue within the Podset, since the network latency of other Podsets are normal. There may be several causes of the Podset red-cross. If both Leaf and ToR switches are all L3 switches, then at least one of the Leaf switches is dropping packets. If the whole Podset is a L2 domain, then it is possibly caused by broadcast storm, e.g., due to some switches loss their configuration.

Figure 8(d) shows a pattern of red-color with green-squares along the diagonal. Here each small green-square is a Podset. It shows that the network latencies within the Podsets are normal, but cross-Podset latency are all out of network SLA. It shows a network issue at the Spine switch layer.

The success of the visualization is beyond our expectation. It has become a habit for many of us to open the visualization portal regularly to see if the network is fine. The visualization portal has been used not only by network developers and engineers, but also by our customers to learn if there is a network issue or not. We also observed an interesting usage pattern: When the visualization system was first put into use, it was typically used by the network team to 'prove' to our customers that the network was fine. Now our customers usually use the visualization to show that there is indeed an on-going network issue. This is a usage pattern change that we are happy to see.

6.4 Pingmesh limitations

During the period of running Pingmesh, we have uncovered two limitations of Pingmesh. First, though Pingmesh is able to detect which tier a faulty network

device is located in, it cannot tell the exact location. In our network, there are tens to hundreds of switches at the Spine layer. Knowing the Spine layer is experiencing some issue is good but not enough. We need methods to locate and isolate the faulty devices as fast as possible. This is a known limitation of Pingmesh from beginning. As described in Section 5.2, we combine Pingmesh and TCP traceroute to address this issue.

The second limitation comes from Pingmesh's current latency measurement. Though the Pingmesh Agent can send and receive probing messages of up to 64 KB, we only use SYN/SYN-ACK and a single packet for single RTT measurement. Single packet RTT is good at detecting network reachability and packet-level latency issues. But it does not cover the case when multiple round trips are needed. We recently experienced a live-site incident caused by TCP parameter tuning. A bug introduced in our TCP parameter configuration software rewrote the TCP parameters to their default value. As a result, for some of our services, the initial congestion window (ICW) reduced from 16 to 4. For long distance TCP sessions, the session finish time increased by several hundreds of milliseconds if the sessions need multiple round trips. Pingmesh did not catch this because it only measures single packet RTT.

7. RELATED WORK

Our experiences running one of the largest data center networks in the world taught us that all the components including applications, OS kernel, NIC, switching ASIC and firmware, and fibers may cause communication failures. See [4] for a summary of various failures that may cause network partition.

[21] and [6] studied traffic and flow characteristics of different types of data centers, by collecting network traces. Pingmesh focuses on network latency and is complementary to these works.

Both Pingmesh and [18] are designed to detect packet drops in the network. Both use active probing packets and are capable of covering the whole network. The approaches, though, are different. [18] uses RSVP-TE base source routing to pinpoint the routing path of a probing packet. It hence needs to create the routing paths and maps in advance. It also means that the probing packets are traversing the network in LSPs (label switched paths) different from those used by the non-probing packets. Second, RSVP-TE is based on MPLS, which, though is widely used for WAN traffic engineering, is not used within the data centers. Pingmesh can be used for both intra-DC and inter-DC networks. Using source routing does provide an advantage: [18] can directly pinpoint the switches or links that drop packets. We have shown in Section 5.2 Pingmesh can localize faulty devices together with traceroute.

Cisco IPSLA [8] also uses active packets for network performance monitoring. IPSLA is configured to run at Cisco switches, and is capable of sending ICMP, IP, UDP, TCP, and HTTP packets. IPSLA collects network latency, jitter, packet loss, server response time, and even voice quality scores. The results are stored locally at the switches and can be retrieved via SNMP or CLI (command-line interface). Pingmesh differs from IPSLA in several ways. First, Pingmesh uses server instead of switches for data collection. By doing so, Pingmesh becomes network device independent whereas IPSLA works only for Cisco devices. Second, Pingmesh focuses on both measurement and latency data analysis. To achieve its goal, Pingmesh has not only Pingmesh Agent for data collection, but also a control plane for centralized control and a data storage and analysis pipeline. IPSLA does not have such a control plane and data storage and analysis pipeline.

NetSight [19] tracks packet history by introducing postcard filters at the switches to generate captured packet events called postcard. Several network troubleshooting services, nprof, netshark, netwatch, ndb, can be built on top of NetSight. Compared with NetSight, Pingmesh is server-based in that it does not need to introduce additional rules into the switches. Further Pingmesh is capable of detecting switch silent packet drops. It is not clear how silent packet drop rules can be written for NetSight, since it is not known in advance which type of packets may be dropped.

ATPG [25] determines a minimal set of probing packets that cover all the network links and forwarding rules. Pingmesh does not try to minimize the number of probings. As long as the overhead is affordable, we prefer to let Pingmesh run all the time. Further it is not clear how ATPG can deal with packet black-holes where the rules for black-holes cannot be determined in advance.

Pingmesh focused on physical network and it uses active probings by installing the Pingmesh Agent in the servers. For third party VMs and virtual networks, however, installing the Pingmesh Agent may not be feasible. In this case passive traffic collection as explored by VND [24] may be used.

8. CONCLUSION

We have presented the design and implementation of Pingmesh for data center network latency measurement and analysis. Pingmesh is always-on and it provides network latency data by all the servers and for all the servers. Pingmesh has been running in Microsoft data centers for more than four years. It helps us answer if a service issue is caused by the network or not, define and track network SLA at both macro and micro levels, and it has become to be an indispensable service for network troubleshooting.

Due to its loosely coupled design, Pingmesh turned out to be easily extensible. Many new features have been added while the architecture of Pingmesh is still the same. By studying the Pingmesh latency data and learning from the latency patterns via visualization and data mining, we are able to continuously improve the

quality of our network, e.g., by automatically fixing packet black-holes and detecting switch silent random packet drops.

9. ACKNOWLEDGEMENT

We thank Lijiang Fang, Albert Greenberg, Wilson Lee, Randy Kern, Kelvin Yiu, Dongmei Zhang, Yongguang Zhang, Feng Zhao, the members of the Wireless and Networking Group of Microsoft Research Asia for their support at various stages of this project. We thank our shepherd Sujata Banerjee and the anonymous SIGCOMM reviewers for their valuable and detailed feedback and comments.

10. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. SIGCOMM*, 2008.

[2] Alexey Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://code.facebook.com/posts/360346274145943/, Nov 2014.

[3] Hadoop. http://hadoop.apache.org/.

[4] Peter Bailis and Kyle Kingsbury. The Network is Reliable: An Informal Survey of Real-World Communications Failures. *ACM Queue*, 2014.

[5] Luiz Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, March-April 2003.

[6] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Internet Measurement Conference*, November 2010.

[7] et.al Brad Calder. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *SOSP*, 2011.

[8] Cisco. IP SLAs Configuration Guide, Cisco IOS Release 12.4T. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipsla/configuration/12-4t/sla-12-4t-book.pdf.

[9] Citrix. What is Load Balancing? http://www.citrix.com/glossary/load-balancing.html.

[10] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *CACM*, Februry 2013.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[12] Albert Greenberg et al. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, August 2009.

[13] Chi-Yao Hong et al. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

[14] Parveen Patel et al. Ananta: Cloud Scale Load Balancing. In *ACM SIGCOMMM*. ACM, 2013.

[15] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB'08*, 2008.

[16] Sushant Jain et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *ACM SOSP*. ACM, 2003.

[18] Nicolas Guilbaud and Ross Cartlidge. Google Backbone Monitoring, Localizing Packet Loss in a Large Complex Network, Feburary 2013. Nanog57.

[19] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *NSDI*, 2014.

[20] Michael Isard. Autopilot: Automatic Data Center Management. *ACM SIGOPS Operating Systems Review*, 2007.

[21] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, 2009.

[22] Rishi Kapoor, Alex C. Snoeren, Geoffrey M. Voelker, and George Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *ACM CoNEXT*, 2013.

[23] Cade Metz. Return of the Borg: How Twitter Rebuilt Google's Secret Weapon. http://www.wired.com/2013/03/google-borg-twitter-mesos/all/, March 2013.

[24] Wenfei Wu, Guohui Wang, Aditya Akella, and Anees Shaikh. Virtual Network Diagnosis as a Service. In *SoCC*, 2013.

[25] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic Test Packet Generation. In *CoNEXT*, 2012.