Pingmesh：一个用于测量和分析大规模数据中心网络延迟的系统

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

摘要

在一个大规模的数据中心网络中，我们是否可以在任意时间获取任意两台服务器之间的网络延迟？之后所收集的延迟数据可以用于解决一系列问题：告知应用程序感知的延迟问题是否是由网络引起的，用于定义和测量网络服务水平协议 (SLA) 以及自动网络故障排除。

为了肯定地解决上述问题，我们已经开发了用于测量和分析大规模数据中心网络延迟的系统——Pingmesh。Pingmesh 已经在微软的数据中心运行工作超过 4 年，每天收集数十 TB 的延迟数据。Pingmesh 不仅被网络软件开发人员和工程师广泛应用，还被应用程序和服务开发人员和运营商广泛使用。

CCS Concepts

•Networks→Network measurement; Cloud computing;

•Computer systems organization →Cloud computing;

关键词

数据中心网络　网络故障排除　静默丢包

1. 介绍

如今的数据中心中通常由数十万台服务器组成。这些服务器通过网卡 (NIC)、交换机、路由器、电缆和光纤等设备连接到一起，形成大规模的内部数据中心网络。随着云计算的快速发展，数据中心网络的范围还在继续扩大。在物理数据中心基础设施之上，构建了各种大规模分布式服务，例如：搜索引擎 [1]、分布式文件系统 [2]、存储 [3] 以及 MapReduce 计算 [4]。

这些分布式服务很庞大，有很多复杂的依赖并且产生了由多个组件组成的软件系统。所有的服务都是分布式的，这些服务的大多组件需要通过网络进行交互——在同一个数据中心内或跨越不同的数据中心。在这样的大型系统中，软件和硬件故障是常态而非异常。因此，网络团队会面临很多挑战。

第一个挑战是判断一个问题是否是一个网络问题。由于分布式系统的性质，许多故障表现为” 网络” 故障，例如：某些组件只能间接性地到达，或者端到端的延迟在第 99 百分位时突然增加，网络的吞吐量从每台服务器 20MB/s 下降到少于 5MB/s。我们的实验表明百分之五十的” 网络” 问题并不是网络所造成的。然而判断一个” 网络” 问题确实是由网络故障造成是十分困难的，反之亦然。

第二个挑战是定义和测量网络服务水平协议 (SLAs)。许多服务需要网络提供某些性能保证。例如，一个搜索请求可能会访问数十万台服务器，而这个搜索请求的性能是由最慢的服务器返回的最后一个请求所决定的。这些服务对网络延迟和丢包非常敏感，与网络 SLA 相关性很大。需要针对不同的服务单独测量和测量网络 SLA，因为这些服务可能使用不同的服务器集群和网络的不同部分。由于网络中的大量服务和用户，这成为一项具有挑战性的任务。

第三个挑战是网络故障排除。当由于不同的网络故障导致网络 SLAs 被破坏时，” 现场” 事件发生了。现场事件是指对客户、合作伙伴或收入产生影响的任何事件。现场事件需要被尽快检测、缓解和解决。但是数据中心网络拥有数十万到数百万台服务器、数十万台交换机以及数百万条电缆和光纤。因此检测问题发生的位置是十分困难的问题。

为了解决上述问题，我们设计和实现了 Pingmesh ——一个用于测量和分析大规模数据中心网络延迟的系统。Pingmesh 利用所有服务器启动 TCP 或 HTTP ping 以获取最大延迟测量覆盖

率。Pingmesh 生成了多个层次的完全图——在一个数据中心中, Pingmesh 使一个机架内的服务器形成一个完全图, 并且使用架顶式 (tOR) 交换机作为虚拟节点然后形成一个二级完全图。跨数据中心, Pingmesh 通过把每一个数据中心作为一个虚拟节点形成一个三级完全图。完全图和相关 ping 的参数由中央 Pingmesh 控制器控制。通过一个数据存储和分析管道收集和存储, 汇总和分析测量的延迟时间数据。通过这些延迟数据, 分别在宏观层次 (即: 数据中心级别) 和微观层次 (例如, 每台服务器和每个机架级别) 定义和测量网络 SLA。通过将服务和应用映射到它们使用的服务器来计算所有服务和应用的网络 SLA。

Pingmesh 已经在微软的数十个全球分布式数据中心上运行了四年。它每天产生 24TB 的数据和超过 2000 亿个探针。由于 Pingmesh 数据的普遍可用性, 使得判断一个现场事件是否是由于网络故障造成的变得简单: 如果 Pingmesh 数据并未指示存在网络问题, 则现场事件不是由网络引起的。

Pingmesh 主要用于网络故障排除, 以找出问题所在。通过可视化和自动模式检测, 我们能够判断丢包和/或延迟增加发生的时间和位置, 识别网络中的交换机静默丢包问题和黑洞问题。应用程序开发人员和服务运营商也使用 Pingmesh 生成的结果, 根据网络延迟和丢包率更好地选择服务器。

本文包括以下内容: 我们通过设计和实现 Pingmesh 展示构建大规模网络延迟测量和分析系统的可行性。通过让每台服务器参与, 我们在任何时刻为所有服务器提供延迟数据。我们展示了 Pingmesh 通过在宏观和围观范围中定义和测量网络 SLA 来帮助我们更好地理解数据中心网络, 并且 Pingmesh 帮助揭示和定位交换机丢包, 包括数据包黑洞和静默随机丢包, 这在以前了解得很少。

2. 背景

2.1 数据中心网络

数据中心网络以高速连接服务器并提供高服务器到服务器带宽。如今的大型数据中心网络是由商用以太网交换机和路由器构建的 [5], [6], [7]。

图 1 展示了一个典型的数据中心网络结构。该网络有两个部分: 内部数据中心 (Intra-DC) 网络和中央数据中心 (Inter-DC) 网络。Intra-DC 网络通常是几层的 Clos 网络, 类似于 [5], [6] 和 [7] 中描述的网络。在第一层, 数十台服务器 (例如: 40 台) 使用 10GbE 或 40GbE 以太网 NIC 连接到架顶式 (ToR) 交换机, 形成一个 Pod。然后数十台架顶式交换机 (例如: 20 台) 连接到 Leaf 交换机的第二层 (例如: 2-8)。这些服务器、ToR 和 Leaf 交换机在一起形成一个 Podset。多个 Podsets 然后连接到 Spine 交换机 (几十到几百) 的第三层。使用现有的以太网交换机, 一个 intra-DC 网络可以连接数万个或更多具有更高网络容量的服务器。

Intra-DC 网络的一个不错的特性是由多个 Leaf 和 Spine 交换机提供具有冗余的多路径网络。ECMP(equal cost multi-path, 等价多路径路由) 用于对所有路径的流量进行负载均衡。ECMP 使用 TCP/UDP 五元组的哈希值进行下一跳选择。所以即使连接的五元组已知, TCP 连接的确切路径在服务器端也是未知的。因此, 很难对有故障的 Spine 交换机进行定位。

Inter-DC 网络用于连接 Intra-DC 网络并将 Inter-DC 连网络接到因特网 (Internet)。Inter-DC 网络使用高速, 长距离光纤连接在不同地方的数据中心网络。进一步引入软件定义网络 (SWAN [8], B4[9], Software defined networking) 以实现更好的广域网流量工程。

我们的数据中心网络是一个庞大而复杂的分布式系统。它由数十万台服务器, 数万台交换机和路由器以及数百万条电缆和光纤组成。并由我们自主研发的数据中心管理软件栈 Autopilot 管理, 交换机和 NIC 上运行着由不同交换机和 NIC 提供商提供的软件和固件。这些在网络上运行的应用程序可能会引入复杂的流量模式。

2.2 网络延迟 (Network Latency) 和丢包 (Packet Drops)

在本文中, 我们使用来自应用程序角度的术语”网络延迟”。当服务器上的应用程序 A 向对等服务器上的应用程序 B 发送了一个消息时, 网络延迟被定义为从 A 发出消息到 B 接收到消息的时间间隔。实际上, 我们测量的是往返时间 (Round Trip-Time, RTT), 因为测量 RTT 不需要同步服务器时间。

RTT 由应用程序处理延迟, 操作系统内核 TCP/IP 协议栈和驱动程序处理延迟, NIC 加载延迟 (例如: DMA 操作, 中断调制)[10], 数据包传输延迟, 传播延迟和在路径上交换机的数据包缓冲产生的排队延迟组成。

有人可能会争辩说应用程序和内核协议栈产生的延迟并不是来自于网络。在实践中, 经验告诉我

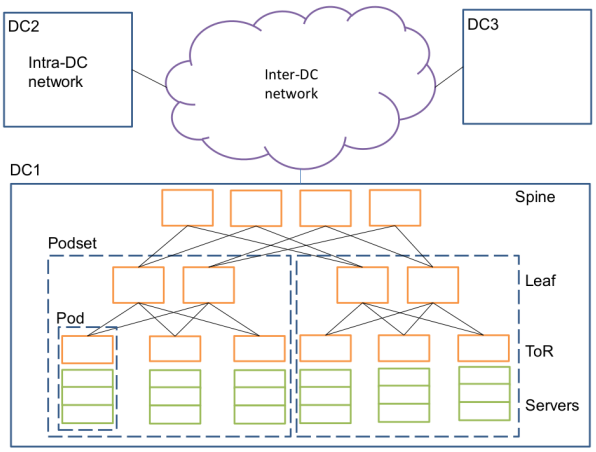


图 1. 数据中心网络结构

们, 用户和服务开发人员并不会关心这个。一旦观察到一个延迟问题, 通常都会被称为" 网络" 问题。网络团队有责任表明问题是否确实是网络问题, 如果是, 缓解并根除这个问题。用户感知延迟可能会由于不同的原因增加, 例如: 由于网络拥塞导致的排队延迟, 服务器 CPU 繁忙, 应用程序的漏洞, 网络路由问题等等。我们还注意到, 丢包会增加用户感知的延迟, 因为丢失的数据包需要重新传输。

2.3 数据中心管理和数据处理系统

接下来, 我们介绍 Autopilot[11]、Cosmos 和 SCOPE[12]。数据中心由集中式数据中心管理系统管理, 例如: Autopilot[11] 或 Borg[13]。这些管理系统提供了有关如何管理包括物理服务器在内的资源, 如何部署、调度、监控和管理服务的框架。Pingmesh 是在 Autopilot 的框架中构建的。

Autopilot 微软用于自动数据中心管理的软件栈。其理念是运行软件以自动化所有数据中心管理任务, 包括故障恢复, 并尽可能减少人为干预。使用 Autopilot 术语, 集群是由本地数据中心网络连接的一组服务器, 由 Autopilot 环境管理。一个 Autopilot 环境有一组 Autopilot 服务, 包括管理机器状态的 Device Manager(DM), 为 Autopilot 和不同应用程序进行服务部署的 Deployment Service(DS), 安装服务器操作系统镜像的 Provisioning Service(PS), 监控和报告不同硬件和软件健康状态的 Watchdog Service (WS), 通过从 DM 获取命令执行修复操作的 Repair Service(RS) 等。

Autopilot 提供共享服务模式 (a shared service mode)。共享服务是在每个 Autopilot 托管服务器上运行的一段代码。例如, Service Manager 是管理其他应用程序生命周期和资源使用情况的共享服务, Perfcounter Collector 是收集本地 perf 计数器并上传至 Autopilot 的共享服务。共享服务必须是轻量的, 占用很低的 CPU、内存和带宽资源, 并且需要是可靠的而不会造成资源泄露和崩溃。

Pingmesh 使用我们自主研发的数据存储和分析系统, Cosmos/SCOPE, 用于延迟数据存储和分析。Cosmos 是微软的大数据系统, 类似于 Hadoop[14], 它提供了像 GFS[2] 和 MapReduce[4] 这样的分布式文件系统。Cosmos 中的文件采用 append-only 方式, 文件被拆分为多个" 扩展区", 扩展区存储在多个服务器中以提供高可靠性。一个 Cosmos 集群可能拥有数万台或更多的服务器, 并为用户提供几乎" 无限" 的存储空间。

SCOPE[12] 是一种声明式和可扩展的脚本语言, 它构建在 Cosmos 之上, 用于分析海量数据集。SCOPE 采用易于使用的设计。它使用户能够更专注于他们的数据而不是底层存储和网络基础设施。用户只需要编写类似于 SQL 的脚本, 而不必担心并行执行、数据分区和故障处理。所有这些复杂的事务都交由 SCOPE 和 Cosmos 处理。

3. 设计和实现

3.1 设计目标

Pingmesh 的目标是构建一个网络延迟测量和分析系统, 用来解决我们在第一节中提到的挑战。Pingmesh 需要一直处于运行状态并且为所有的服务器提供网络延迟数据。因为我们需要一直跟踪网络状态, 所以 Pingmesh 需要始终处于运行状态。而 Pingmesh 之所以需要为所有的服务器都提供网络延迟数据, 是因为最大可能的网络延迟数据覆盖对我们更好地理解, 管理和排除网络基础设施架构故障至关重要。

从一开始, 我们就将 Pingmesh 与各种公共和专有网络工具 (例如: traceroute, TcpPing 等) 区分开来。鉴于以下的原因, 我们意识到这些网络工具对于我们而言没有作用。首先, 这些工具并不是一直运行的, 只有当我们运行它们时才会产生数据。其次, 它们产生的数据没有我们所需的覆盖范围。由于这些工具不是一直运行的, 我们无法对它们进行计算以跟踪实时的网络状态。这些工具通常用于在源-目标已知时进行网络故障排除。然而, 这对于大规模数据中心网络并不适用: 当网络事件发生时, 我们甚至可能不知道源-目标对。此外, 对于瞬时的网络问题, 在

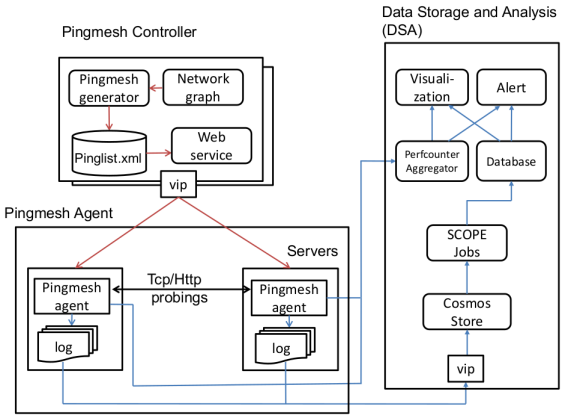


图 2. Pingmesh 体系架构

运行工具之前, 问题可能已经消失。

3.2 Pingmesh 体系架构

基于 Pingmesh 的设计目标, 它需要满足以下的需求。首先, 因为 Pingmesh 旨在提供尽可能大的覆盖范围并测量应用程序角度的网络延迟, 因此每台服务器都需要安装 Pingmesh Agent。我们必须小心地引入 Pingmesh Agent, 以保证 Agent 所占用的 CPU, 内存和带宽资源是少且合理的。其次, Pingmesh Agent 的行为应该是可控和可配置的。需要高度可靠的控制面板来控制服务器应该如何执行网络延迟测试。

第三, 应该近乎实时地汇总、分析和报告延迟数据, 并存储和存档已进行更深入的分析。基于上述需求, 我们设计了如图 2所示的 Pingmesh 体系架构。Pingmesh 有三个组件, 如下所述。

Pingmesh Controller(Pingmesh 控制器)。它是整个系统的核心, 决定了服务器之间应该如何相互探测。在 Pingmesh Controller 中, Pingmesh Generator 为每个服务器生成一个 pinglist 文件。pinglist 文件中包含对等服务器列表和相关参数。pinglist 文件基于当前的网络拓扑生成。服务器通过 RESTful Web 接口获取相应的 pinglist 文件。

Pingmesh Agent。每个服务器都运行一个 Pingmesh Agent。Agent 从 Pingmesh Controller 中下载 pinglist 文件, 然后对 pinglist 文件中的对等服务器发起 TCP/HTTP ping 测试。Pingmesh Agent 在本地内存中保存 ping 的结果。一旦计时器超时或测量结果的大小超过阀值, Pingmesh Agent 会将结果上传至 Cosmos 进行数据存储和分析。Pingmesh Agent 还开放了一组性能计数器, 这些计数器的数据由 Autopilot 的 Perfcounter Aggregator(PA) 服务定期收集。

Data Storage and Analysis(DSA, 数据存储和分析)。通过一个数据存储和分析管道存储和处理网络延迟数据。网络延迟数据存储在 Cosmos 中。使用多个 SCOPE 任务来分析数据。SCOPE 任务采用类似于 SQL 语言的声明式语言编写。然后将分析后的结果存储在一个 SQL 数据库中。基于此数据库和 PA 计数器中的数据进生成可视化, 报告和警报。

3.3 Pingmesh Controller

3.3.1 pinglist 生成算法

Pingmesh Controller 的核心是 Pingmesh Generator。Pingmesh Generator 运行一个算法来决定哪个服务器应该 ping 哪组服务器。如上所述, 我们希望 Pingmesh 有尽可能大的覆盖范围。最大可能的覆盖范围是服务器层次的完全图, 在这个图中每个服务器会探测所有剩下的服务器。然而, 一个服务器层次的完全图是不可行的, 因为假设服务器的总数是 n, 那么一个服务器需要探测 n-1 个服务器。在数据中心, n 的取值可以大到数十万。此外数十个服务器通过相同的架顶式交换机 (ToR) 连接到网络, 因此服务器层次的完全图并不是必要的。

于是我们提出了一个多层次完全图的设计。在 Pod 中, 我们让同一个架顶式交换机下所有的服务器形成一个完全图。在 intra-DC 层, 我们把每个 ToR 交换机看作一个虚拟节点, 然后把所有的 ToR 交换机形成一个完全图。在 inter-DC 层, 每个数据中心充当一个虚拟节点, 然后所有的数据中心形成一个完全图。

在我们的设计中, 只有服务器进行 ping 操作。当我们说一个 ToR 交换机是一个虚拟节点时, 是这个 ToR 交换机下的服务器执行 ping 操作。同样, 对于作为虚拟节点的数据中心, 是从数据中心选取的服务器发起探测。

在 intra-DC 层, 我们曾经认为只需要选取一个可配置数量的服务器来参与 Pingmesh。但是如何选择服务器成为一个问题。进一步, 少量选定的服务器可能不能很好地代表其余的服务器。我们最后决定让所有的服务器都参与其中。intra-DC 的算法是: 对于任意的 ToR-pair(ToRx, ToRy), 用 ToRx 中的服务器 i ping ToRy 中的服务器 i。在 Pingmesh 中, 即使当两台服务器位于彼此的 pinglist 中, 也会分开测量网络延迟。通过这样做, 每台服务器可以独立得在本地计算自己的丢包率和网络延迟。

在 inter-DC 层, 所有的 DC 形成了另一个完全

图。在每个 DC 中, 我们选择多个服务器 (从每个 Podset 中选择多个服务器)。

结合这三个完全图并根据数据中心的大小, Pingmesh 中的一个服务器需要 ping2000 ~ 5000 对服务器。Pingmesh Controller 使用阀值控制服务器的探测总数以及对源目标服务器的两个连续探测的最小时间间隔。

3.3.2 Pingmesh Controller 的实现

Pingmesh Controller 实现为一个 Autopilot 服务并成为 Autopilot 管理栈的一部分。通过运行 Pingmesh 生成算法, Pingmesh 为每个服务器生成 Pinglist 文件。然后将这些文件存储在 SSD 中, 并通过 Pingmesh Web 服务提供给服务器。Pingmesh Controller 为 Pingmesh Agent 提供了一个简单的 RESTful Web API, 以用来分别检索它们的 Pinglist 文件。Pingmesh Agent 需要定期向 Controller 请求 Pinglist 文件, Pingmesh Controller 不会将任何数据推送到 Pingmesh Agent 中。通过这样做, Pingmesh Controller 变得无状态且易于扩展。

作为整个 Pingmesh 系统的核心, Pingmesh Controller 需要为数万个 Pingmesh Agent 提供服务。因此, Pingmesh Controller 需要具有容错性和可扩展性。我们使用软件负载均衡 (SLB, Software Load Balancer)[8] 为 Pingmesh Controller 提供容错性和扩展性。有关 SLB 如何工作的详细信息, 请参见 [9, 14]。在单个 VIP(virtual IP address, 虚拟 IP 地址) 背后, Pingmesh Controller 有一个服务器集群。SLB 将来自 Pingmesh Agent 的请求分发至 Pingmesh Controller 服务器。每个 Pingmesh Controller 服务器运行着相同的代码块, 为所有服务器产生相同的 pinglist 文件集, 并可以为所有来自 Pingmesh Agent 的请求提供服务。然后, Pingmesh Controller 可以通过在同一 VIP 后添加服务器来轻松扩展。一旦 Pingmesh Controller 服务器停止运转, SLB 会自动将它从轮转中移除。我们在两个不同的数据中心集群中设置了两个不同 Pingmesh Controller, 使控制器在地理上更具容错能力。

3.4 Pingmesh Agent

3.4.1 Pingmesh Agent 设计

在所有的服务器上都会运行一个 Pingmesh Agent。它的任务很简单: 从 Pingmesh Controller 下载 pinglist 文件, 并对其中的服务器发起 ping, 然后把 ping 的结果集上传至 DSA。

基于 Pingmesh 能够区分用户感知延迟增加是否是由于网络的需求, Pingmesh 应该使用和应用程序相同类型的数据包。因为在我们的数据中心几乎所有的应用程序都使用 TCP 和 HTTP, 所以 Pingmesh 使用 TCP 和 HTTP 而非 ICMP 和 UDP 进行探测。

因为我们需要区分" 网络" 问题是由于网络还是应用程序本身, Pingmesh Agent 不会使用应用程序所使用的任何网络库。相反, 我们开发了自己的轻量级网络库, 专门用于网络延迟测量。Pingmesh Agent 可以配置发送和响应除 TCP SYN/SYN-ACK 数据包之外的不同长度的探测数据包。因此, Pingmesh Agent 需要同时充当客户端和服务器。客户端部分用于发起 ping, 服务器部分用于响应 ping。

每次探测需要建立一个新的连接和使用一个新的 TCP 源端口。这是为了尽可能地探索网络的多路径特性, 更重要的是, 减少 Pingmesh 创建的并发 TCP 连接的数量。

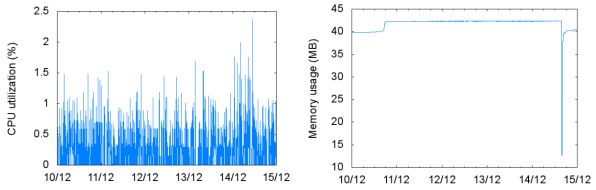
3.4.2 Pingmesh Agent 实现

虽然任务很简单, 但 Pingmesh Agent 仍是实现上最具挑战性的部分之一。他必须符合以下安全和性能要求。

首先, Pingmesh Agent 必须是失败封闭且不会产生现场事故。由于 Pingmesh Agent 运行在每台服务器上, 因此如果它发生故障, 可能会导致所有服务器都关闭 (例如: 占用大量 CPU 和内存资源, 产生大量探测流量等)。为了避免发生不良事故, Pingmesh Agent 已经实现了一些安全功能:

- Pingmesh Agent 的 CPU 和最大内存使用量受操作系统的限制。一旦最大内存使用量超过上限, Pingmesh Agent 将被终止。
- 任何两台服务器之间的最小探测间隔限制为 10 秒, 探测有效负载长度限制为 64KB。这些限制硬编码在源代码中的。通过这样做, 我们对 Pingmesh 可以带入网络的最坏情况流量进行了硬性限制。
- 如果 Pingmesh Agent 连续 3 次无法连接到它的控制器或者控制器已经启动但是没有可用的 pinglist 文件, 则 Pingmesh Agent 将删除其所有现有的 ping 对等项并停止其所有 ping 活动。(但是它仍会对 ping 作出响应)。借助此功能, 我们可以通过简单地移除控制器中所有的 pinglist 文件来让 Pingmesh Agent 停止运行。

- 如果服务器无法上传延迟数据, 它将尝试重传几次。之后它将停止尝试并丢弃内存中的数据。这是保证 Pingmesh Agent 使用有限的内存资源。Pingmesh Agent 还将延迟数据作为日志文件写入本地磁盘。日志文件的大小限制为可配置的大小。



(a) CPU usage (b) Memory usage

图 3. Pingmesh Agent 的 CPU 和内存使用情况

其次, 在设计上一个 Pingmesh Agent 需要对数万台服务器发起 ping 测试。但是作为共享服务, Pingmesh Agent 应该最小化它的资源 (CPU, 内存和硬盘空间) 使用。他应该使用接近 0 的 CPU 时间和尽可能小的内存占用, 以便最大限度地减少对客户应用程序的干扰。

为了实现性能目标和提高 Pingmesh 的延迟测量精确度, 我们使用 C++ 而不是 Java 或 C# 来编写 Pingmesh Agent。这是为了避免通用语言运行平台 (the common language runtime, CLR) 和 Java 虚拟机的开销。我们为 Pingmesh 专门开发了一套网络库。这个库的目标仅用于测量网络延迟, 它被设计成轻量级的并用于处理大量并发 TCP 连接。该库直接基于 Winsock API, 它使用 Windows IO Completion Port 编程模型进行高效的异步网络 IO 处理。这个库同时充当客户端和服务器, 并将探测处理负载均匀地分配给所有 CPU 核心。

我们已经进行了大量的测量来了解和优化 Pingmesh Agent 的性能。图 3展示了在一个典型的服务器上 Pingmesh Agent 的 CPU 和内存使用情况。在测试过程中, 这个 Pingmesh Agent 正在主动探测大约 2500 台服务器。这台服务器有 128GB 内存和两个 Intel Xeon E5-2450 8 核处理器。平均内存占用量小于 45MB, 平均 CPU 使用率为 0.26%。

我们注意到 Pingmesh Agent 产生的探测流量很小, 通常为几十 kb/s。作为比较, 我们的数据中心网络为数据中心的任意两台服务器之间提供几个 Gb/s 的吞吐量。

3.5 数据存储和分析

对于 Pingmesh 数据存储和分析, 我们使用完善的现有系统, Cosmos/SCOPE 和 Autopilot 的 PerfCounter Aggregator(PA) 服务, 而不是重新发明轮子。

Pingmesh Agent 将聚合后的记录周期性地上传至 Cosmos。和 Pingmesh Controller 类似, Cosmos 的前端采用负载均衡器和 VIP 进行扩展。同时, Pingmesh Agent 对延迟数据执行本地计算, 并生成一组性能计数器, 包括丢包率, 第 50 和第 99 百分位数的网络延迟等。所有这些的性能计数器都由 Autopilot 的 PA 服务收集, 汇总并存储。

一旦结果在 Cosmos 中, 我们就会运行一组 SCOPE 任务进行数据处理。我们在不同的时间范围内有 10 分钟, 1 小时, 1 天的任务。10 分钟的任务是近乎实时的任务。对于 10 分钟的任务, 从生成延迟时间数据到消耗完数据 (例如: 发出报警, 生成仪表板图表) 的时间间隔约为 20 分钟。1 小时和 1 天的管道用于非实时任务, 包括网络 SLA 跟踪, 网络黑洞检测, 丢包检测等等。我们所有的任务都由 Jonb Manager 自动定期地提交给 SCOPE, 无需用户干预。SCOPE 任务的结果存储在 SQL 数据中, 从中生成可视化, 报告和警报。

实际上, 对于系统层次的 SLA 追踪, 我们发现这 20 分钟的延迟工作良好。为了进一步降低响应时间, 我们并行使用 Autopilot PA 管道来收集和汇总一组 Pingmesh 计数器。Autopilot PA 管道是一种分布式设计, 每个数据中心都有自己的管道。PA 计数器收集延迟为 5 分钟, 比 Cosmos/SCOPE 管道更快。PA 管道比 Cosmos/SCOPE 更快, 而 Cosmos/SCOPE 在数据处理上比 PA 更具表现力。通过综合使用它们, 我们为 Pingmesh 提供了比其中任何一个更高的可用性。

我们把 Pingmesh 作为一个始终运行的服务和周期性运行的脚本集区分开。Pingmesh 的所有组件都有 watchdog 来观察它们是否运行正常, 例如: 是否正确生成 pinglist 文件, CPU 和内存使用是否在限制之内, pingmesh 的数据是否被报告和存储, DSA 是否及时报告网络 SLAs 等。此外, Pingmesh 旨在以轻量级和安全的方式探测成千上万的对等服务器。

所有的 Pingmesh Agent 每天上传 24TB 的延迟测量数据到 Cosmos 中。这超过了 2Gb/s 的上传速率。虽然这些数字看起来很大, 但它们只是我们的网络和 Cosmos 所能提供的总量中可忽略不计的一

小部分。

4. 延迟数据分析

在本节中, 我们将介绍 Pingmesh 如何帮助我们更好地了解网络延迟和数据包丢失, 定义和追踪网络 SLA 以及判断一个现场事件是否是因为网络问题。我们在本节中描述的所有数据中心都具有与我们在图 1中介绍的类似的网络架构, 尽管它们的大小可能不同, 并且可能在不同的时间构建。

4.1 网络延迟

图 4展示了两个代表性数据中心 (美国西部的 DC1 和美国中部的 DC2) 的 Intra-DC 延迟分布。DC1 用于分布式存储和 MapReduce 计算, DC2 用于交互式搜索服务。DC1 中的服务器是吞吐量密集型的, 平均服务器 CPU 利用率高达 90%。DC1 中的服务器大量使用网络, 且一直发送和接收平均几百 Mb/s 的数据。DC2 对延迟敏感, 服务器具有高扇入和高扇出, 因为服务器需要与其它大量的服务器交互来服务一个搜索查询。DC2 的服务器平均 CPU 利用率适中且平均网络吞吐量较低, 但是流量具有突发性。

图 4中 CDF 是从一个正常工作日的延迟数据中计算出的, 这天没有到检测网络故障也没有由于网络故障导致的现场事件。我们分别增加和不增加 TCP 有效负载 (payload) 来跟踪 intra-pod 和 inter-pod 的延迟分布。如果没有特别提及, 我们在本文中使用的延迟时间是基于无负载的 inter-pod TCP SYN/SYN-ACK 报文的 RTT 时间。图 4(a) 展示了整个 inter-pod 的延迟分布, 图 4(b 展示了 inter-pod 在高百分位下的延迟分布。我们曾经预计 DC1 中的延迟应该远大于 DC2 中的延迟, 因为 DC1 中的服务器和网络负载都很高。但事实证明, 在第 90 或更低百分位数的延迟并非如此。

但是在高百分位数上, DC1 确实有更高的延迟, 正如图 4(b) 中所示。在 P99.9, DC1 和 DC2 的 inter-pod 延迟分别是 23.35ms 和 11.07ms。在 P99.99, DC1 和 DC2 的 inter-pod 延迟分别为 1397.63.ms 和 105.84ms。我们的测量结果表明, 即使服务器和网络在宏观时间尺度上都是轻加载的, 也很难在三个或四个 9 提供低延迟 (例如: 亚毫秒级)。这是因为服务器的操作系统不是实时操作系统, 而且网络中的流量是突发的。即使平均网络利用率低至中等, 我们也会看到在 intra-pod 的通信 (4.2 节) 中有 5 到 10 个数据包丢失。

数据中心	丢包率	
	Intra-pod	Inter-pod
DC1(美国西部)	1.31×10^{-5}	7.55×10^{-5}
DC2(美国中部)	2.10×10^{-5}	7.63×10^{-5}
DC3(美国东部)	9.58×10^{-5}	4.00×10^{-5}
DC4(欧洲)	1.52×10^{-5}	5.32×10^{-5}
DC5(亚洲)	9.82×10^{-5}	1.54×10^{-5}

表 1. Intra-pod 和 inter-pod 的丢包率。

图 4(c) 比较了 intra-pod 和 inter-pod 的延迟分布, 图 4(d) 研究了有无负载情况下的 inter-pod 延迟, 所有这些都在 DC1 中完成。对于使用有负载的延迟测量, 在 TCP 连接建立以后, 我们让客户端发送一个消息 (通常一个数据包内包含 800-1200 字节)。一旦客户端收到来自服务器的回送消息吼, 立即测量有效负载延迟。

正如图 4(c) 所示, 和预期一样 intra-pod 的延迟总是小于 inter-pod 的延迟。在 DC1 中, 第 50(P50) 和第 99(P99) 百分位数的 intra-pod 和 inter-pod 的延迟分别是 (216us, 1.26ms) 和 (268us, 1.34ms)。在 P50 和 P99 的差值分别是 52us 和 80us。这些数字表明, 由于排队延迟, 网络确实引入了数十微秒的延迟。但是排队延迟较小。因此我们可以推断出网络提供了足够的网络容量。

图 4(d) 展示了有无负载情况的延迟。有负载时, P50 和 P99 的网络延迟分别从 286us 增加到 326us, 从 1.34ms 增加到 2.43ms。这个增加主要是因为传输延迟增加以及接受服务器回送消息的用户空间处理开销。在大多情况下, 有无负载的延迟分布是近似的。我们引入了有效载荷, 因为它可以帮助检测与数据包长度相关的数据包丢失 (例如: 光纤 FCS 错误和误码率相关的切换 SerDes 错误)。

基于 Pingmesh 数据, 我们不仅可以计算出数据中心的延迟分布, 还可以计算所有应用程序和服务的延迟 CDF。从这些结果中, 我们能够一直跟踪它们所有的网络延迟。

4.2 丢包率

Pingmesh 不直接测量丢包率。然而, 我们可以通过 TCP 连接建立阶段推断出丢包率。当第一个 SYN 数据包丢失时, TCP 发送器将会经过一个初始的超时时间后重新发送这个数据包。对于其余的连续重传, TCP 每次都会使超时时间值加倍。在我们的数据中心, 初始超时时间值为 3s, 发送器会重

传两次 SYN 数据包。因此, 如果测量的 TCP 连接 RTT 时间大约是 3s, 丢失了一个数据包; 如果测量的 TCP 连接 RTT 时间大约是 9s, 丢失了两个数据包。我们使用下列启发式来估测丢包率:

$$\frac{probes_with\_3s\_rtt + probes\_with\_9s\_rtt}{total\_successful\_probes} \quad (1)$$

注意这里我们仅使用成功的 TCP 探测数而不是总探测数作为分母。这是因为对于失败的探测, 我们无法区分出是由于数据包丢失还是接收的服务器故障。在分子中, 对于 9 秒 RRT 时间的连接, 我们只计算一个数据包丢失而不是两个数据包丢失。这是因为 TCP 连接中连续的数据包丢失不是相互独立的: 如果第一个 SYN 数据包丢失, 则第二次丢失 SYN 数据包的概率要高得多。我们已经通过计算 NIC 和 ToR 丢失的数据包数量验证了启发式对于单 ToR 网络的精确度。

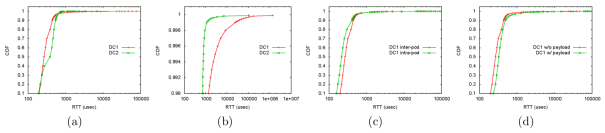


图 4. (a) 两个数据中心的 Inter-pod 延迟. (b) 高百分位数的 Inter-pod 延迟. (c)Intra-pod 和 Inter-pod 的延迟比较. (d) 有无负载的延迟比较.

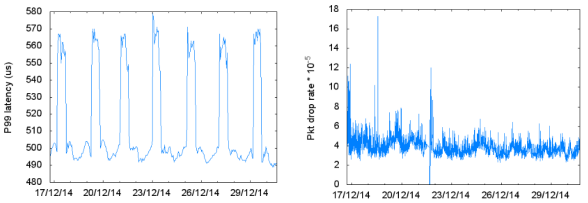
在我们的网络中, SYN 数据包的处理方式和其他数据包相同。因此 SYN 数据包的丢包率可以代表正常情况下其他数据包的丢包率。然而, 当丢包率和数据包大小相关时 (例如, 由于 FCS 错误), 该假设可能不成立。我们确实看到较大尺寸的数据包在 FCS 错误相关事件中可能会遇到更高的丢包率。在下文中, 我们展示的都是网络处于正常状态时的结果。

我们的网络不会区分不同 IP 协议 (例如: TCP vs. UDP) 的数据包。因此, 我们的数据包丢失计算也适用于非 TCP 流量。

表1显示了五个数据中心的丢包率。我们同时展示了 intra-pod 和 inter-pod 的丢包率。对于 intra-pod 丢包, 通常丢失在 ToR 交换机, NIC 和终端主机的网络栈。除了 ToR, NIC 和终端主机网络栈外, inter-pod 的丢包可能来自于 Leaf 和 Spine 交换机以及相关的链路。

从表1中可以得出几个观察结果。首先, 丢包率的取值处于 $10^{-5} \sim 10^{-4}$ 范围内。我们每天跟踪我们所有数据中心的丢包率, 并且我们发现除非发生了网络故障, 丢包率始终处于这个范围中。其次, inter-pod 的丢包率通常比 intra-pod 的丢包率要高好几倍。这表明大多的数据包丢失发生在网络中而不是在主机上。第三, intra-pod 的丢包率大约是 10^{-5}, 比我们预期的要大。

实验表明, 有许多原因可能会导致数据包丢失发生, 例如: 交换机缓存溢出, NIC 接收缓存溢出, 光纤 FCS 错误, 切换 ASIC 故障等。尽管我们的测量结果表明, 丢包率在正常情况下大约是 $10^{-5} \sim 10^{-4}$, 我们仍处于了解它为何保持在这个范围内的早期阶段。



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

图 5. 某服务第 99 百分位数的网络延迟和丢包率.

许多数据中心应用, 例如, 搜索引擎可能会同时使用数百甚至数千个 TCP 连接。对于这些应用, 由于使用大量连接, 因此高延迟尾 (high latency tail) 成为常态。应用程序已经引入几种应用层技巧来处理数据包丢失。

我们通过每个服务器的延迟数据可以计算和跟踪服务器, pod, podset 和数据中心层次的网络 SLA。同样, 我们可以计算和跟踪各个服务的网络 SLA。

4.3 这是一个网络问题吗?

在大型的分布式数据中心中, 许多部分可能会出错。当一个现场事故发生时, 识别出是哪个部分造成的问题是不容易的。有时候所有的组件看起来都很正常, 但整个系统是坏的。如果网络无法证明它是正常的, 那么这个问题就会被称为" 网络问题": 我没有对我的服务做任何错误的事情, 这一定是网络的错误。

然后网络团队开始调查。通常如同下面所描述的过程。on-call 网络工程师询问发生故障的服务的详细症状和源-目的服务器对; 然后, 他登录到源服务器和/或目的服务器并运行各种网络工具来重现该问题; 他也可能会看到异常可能路径上的交换机计数器; 如果他无法重现, 他可能会询问更多的源-目的服务器对。这个过程可能需要重复迭代很多次。

上述方法对我们来说效果不佳, 因为它是一个手工过程而且不能扩展。如果问题不是由网络造成的, 那么服务所有者就会浪费时间与错误的团队合作。如果问题确实是由于网络问题, 则手工过程会导致长时间的检测 (TTD, time-to-detect), 缓解时间 (TTM, time-to-mitigate) 和解决时间 (TTR, time-to-resolve)。

Pingmesh 改变了这种情况。因为 Pingmesh 收集的延迟数据来自于所有的服务器, 我们总是可以提取出 Pingmesh 数据来判断特定的服务是否存在网络问题。如果 Pingmesh 的数据和应用程序感知的问题无关, 那么它不是网络问题。如果 Pingmesh 数据显示它确实是网络问题, 我们可以进一步从 Pingmesh 获得详细数据用于进一步调查, 例如, 问题的规模 (例如: 受影响的服务器和应用程序的数量), 源-目标服务器 IP 地址和 TCP 端口号。

我们将网络 SLA 定义为一组指标, 包括第 50 和第 99 百分位数的丢包率和网络延迟。然后通过使用 Pingmesh 的数据可以在不同的范围内追踪网络 SLA, 包括每个服务器, 每个 pod/podst, 每个服务, 每个数据中心。在实践中, 我们发现两个网络 SLA 指标: 第 99 百分位数的丢包率和网络延迟有助于判断一个问题是否是由于网络造成的。图 5展示了一个正常周内某服务的这两个指标的值。丢包率大约是 4×10^{-5}, 数据中心中第 99 百分位数的网络延迟是 500 ~ 560us。(这个延迟显示了一种周期模式。这是因为该服务定期执行高吞吐量数据同步, 增加了第 99 百分位数的延迟。) 如果这两个指标发生重大变化, 那么这就是网络问题。

5. 静默丢包检测

在本节中, 我们介绍如何用 Pingmesh 检测交换机静默数据包丢弃。当静默丢包发生时, 交换机由于各种原因不会显示数据包丢失的信息并且交换机看起来是正常的。但是应用程序受到了延迟增加和数据包丢失的影响。因此如何快速识别正在发生的现场事件是否是由于交换机静默丢包引起的变得至关重要。

在过去, 我们已经能够识别两种类别的交换机静默丢包: 数据包黑洞和静默随机丢包。接下来, 我们介绍如何使用 Pingmesh 来检测它们。

5.1 数据包黑洞

数据包黑洞是一种特殊的交换机丢包类型。对于正在发生数据包黑洞的交换机, 交换机会确定性地 (例如: 100%) 丢弃某些满足特定" 模式" 的数据包。我们已经识别出两种类型数据包黑洞。第一类黑洞, 包含特定源-目标 IP 地址对的数据包会被丢弃。症状如下: 服务器 A 无法访问服务器 B, 但是它可以正常访问服务器 C 和 D。所有 A-D 的服务器都是正常的。

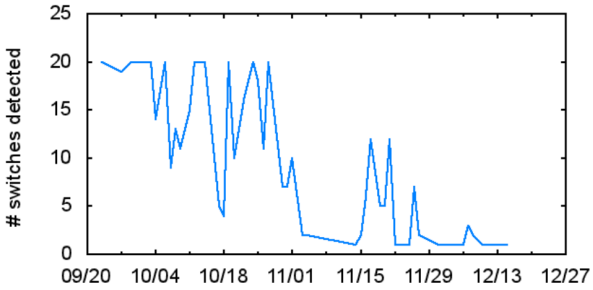


图 6. 检测到网络黑洞的交换机的数量

第二类黑洞, 包含特定源-目标地址和端口号的数据包会被丢弃。注意对于这种类型的黑洞, 具有相同源-目标地址对但不同源目标端口号的数据包的处理方式不同。例如, 服务器 A 使用源端口 X 可以访问服务器 B 的端口 Y, 但是源端口 Z 不行。

第一类黑洞通常由交换机 ASIC 中的 TCAM 缺陷 (例如: 奇偶校验错误) 引起。TCAM 表中的某些 TCAM 条目可能会损坏, 这些损坏会导致某些具有特定源目标地址模式的数据包被丢弃 (由于只有目标地址用于 IP 路由中下一跳查找, 因此想知道为什么源 IP 地址也有作用。我们的猜想是一个 TCAM 条目不仅包含目标地址还包含源地址和其他元数据)。

我们对第二类黑洞的根本原因了解很少。我们怀疑这可能是因为与 ECMP 相关的错误, 它使用源目标 IP 地址和端口号来决定下一个转发跳。

基于我们的实验, 这两类黑洞可以通过重启交换机修复。因此问题变成如何检测存在黑洞的交换机。

我们设计了一种基于 Pingmesh 数据的 ToR 交换机黑洞检测算法。这个算法的设想是如果 ToR 交换机下的许多服务器遇到黑洞症状, 那么我们将 ToR 交换机标记为一个黑洞候选并给它分配一个分数, 即服务器与黑洞症状的比率。然后我们选择黑洞分数大于阀值的交换机作为候选者。在一个 podset 中, 如果只有部分 ToR 遇到黑洞症状, 则这些 ToR 是黑洞数据包。然后我们调用一个网络修复服务来重启这些 ToR。如果 podset 中所有的 ToR 都遇到

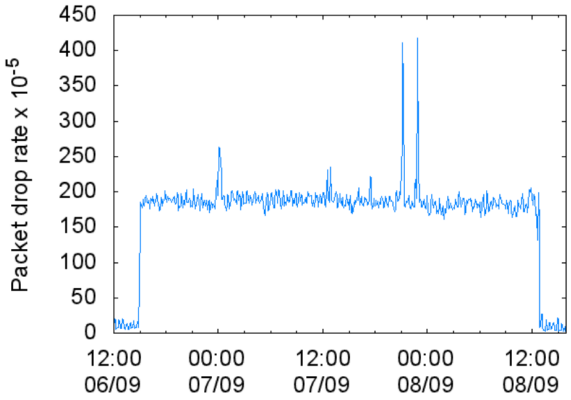


图 7. 在发生故障时, Pingmesh 检测到的 Spine 交换机的静默随机丢包.

黑洞症状, 则问题可能出现在 Leaf 或 Spine 层。通知网络工程师做进一步的调查。

图 6展示了使用算法检测到具有黑洞的交换机的数量。正如我们从图中看到的, 一旦算法开始运行, 存在数据包黑洞的交换机的数量马上就会降下来。在我们的算法中, 我们限制算法每天最多重启 20 个交换机。这是为了限制交换机重启的数量。我们可以看到, 经过一段时间后, 检测到的交换机数量下降到每天只有几个。

我们想要注意的是, Pingmesh Agent 每个探测的 TCP 源端口是不同的。通过大量的源/目标 IP 地址对, Pingmesh 扫描了整个源/目标地址和端口空间的很大一部分。在我们的 Pingmesh 黑洞检测上线后, 我们的客户不再抱怨数据黑洞了。

5.2 静默随机丢包

交换机位于网络中的层次越高, 当它开始发生丢包时产生的影响越严重。当 Spine 交换机发生静默丢包时, 将会影响到数万台服务器和许多服务, 并触发严重程度较高的现场事件。

这里我们介绍 Pingmesh 如何用于定位 Spine 交换机上的静默随机丢包。在一次事件中, 数据中心的所有用户在第 99 百分位数遇到网络延迟增加。使用 Pingmesh, 我们可以确定数据中心中的数据包丢失数量显著增加, 并且丢失是不确定性的。图 7展示了丢包率的变化。在正常情况下, 延迟的比率应该大约是 $10^{-4} \sim 10^{-5}$。但是它突然增加到大约 2×10^{-3}。

使用 Pingmesh, 我们不久就认识到只有一个数据中心受到影响, 而其他数据中心状态良好。ToR 和 Leaf 层的丢包不会导致所有用户的延迟增加, 因为它们下面的服务器数量要少得多。图 8中的延迟增加模式指出问题发生在 Spine 交换机层。

但是在这些交换机上我们无法找到任何数据包丢失提示 (FCS 错误, 输入/输出数据包丢弃, 系统日志错误等等)。然后我们怀疑这可能是一个静默丢包问题。下一个步骤是定位发生丢包的交换机。

再次, 通过使用 Pingmesh, 我们可以找到几个遇到大约 1% ~ 2% 随机数据包丢失的源和目标对。然后我们对这些源/目标对进行 traceroute(TCP) 检测, 并且最终精确定位到一个 Spine 交换机上。在我们将这个交换机于服务实时流量隔离开后, 静默随机丢包问题就解决了。交换机供应商的事后分析显示, 数据包丢失是由于该交换机结构模块的位翻转造成的。

上面的案例是我们遇到的第一个静默丢包案例, 我们花了很长时间才解决。之后我们遇到了更多的案例, 并改进了 Pingmesh 数据分析和其它工具, 用以实现更好的自动随机静默丢包检测。我们的实验告诉偶们随机静默丢包可能是由于不同的原因, 例如, 交换机结构 CRC 校验和错误, 交换机 ASIC 缺陷, 线路卡 (linecard) 不太好等等。这些类型的交换机静默丢包无法通过重启交换机来修复, 我们必须通过 RMA(return merchandise authorization) 故障交换机或组件。

与由于网络拥塞和链路 FCS 错误导致的丢包相比, 数据包黑洞和静默随机丢包是新的问题, 我们对此也不太了解。由于 Pingmesh 的全局覆盖和始终在线的特性, 我们能够确认现实中交换机确实存在静默丢包问题, 并对不同的静默丢包类型进行分类以及进一步定位静默丢包发生的位置。

6. 经验教训

Pingmesh 旨在提供可扩展性。我们知道并非所有网络都符合我们的规模。我们相信我们从 Pingmesh 学到的经验教训对无论大小规模的网络都是有益的。我们吸取的教训之一是, 全覆盖且可信的延迟数据是非常有价值的。如果数据本身是不可信的, 那么构建在其上的结果也是不可信的。我们的实验告诉我们并非所有 SNMP 数据都值得信赖。即使 SNMP 告诉我们一切正常, 交换机也可能会发生丢包。我们信任 Pingmesh 的数据是因为我们编写了代码, 并对它进行了测试和运行。当有 bug 时, 我们会立即修复。经过几轮迭代后, 我们知道我们可以信任这些数据。由于其延迟数据的全覆盖和可靠性, Pingmesh 可以执行准确的黑洞和静默丢包检测。作

为比较, 简单地使用交换机 SNMP 和 syslog 数据是无效的, 因为它们不能告知我们存在数据包黑洞和静默丢包问题。

在下文中, 我们将介绍我们从构建和运行 Pingmesh 中学到的其它几个经验教训, 我们相信它们也可以应用于不同规模的网络。

6.1 Pingmesh 是一个 always-on 服务

从项目一开始, 我们相信 Pingmesh 需要覆盖所有的服务器并一直保持在线状态。但不是所有人都同意。有人认为应该按需收集需求; 我们应该只选择一些服务器参与到延迟测量中, 以减小开销。我们不同意上述两个观点。

从本质上讲, 第一个争论在于 always-on 和 on-demand。有人可能会说, 如果用不到 always-on 的延迟数据, 是对资源的浪费, 因此我们应该只在需要时收集延迟数据。这个观点存在两个问题。首先, 我们无法预测需要何时的延迟数据, 因为我们不知道什么时候会发生现场事件。当发生现场事件时, 手头有网络延迟数据而不是在那时收集它们是一个更好的选择。其次, 当不好的事情发生时, 我们通常不知道是哪个设备造成的故障, 因此我们甚至没有源/目标对来发起延迟测试。

仅使用少量选定的服务器进行延迟测量会限制 Pingmesh 数据的覆盖范围, 并且应该选择哪些服务器也是一个难题。正如我们在本文中所演示的那样, 让所有的服务器都参与以为我们提供最大可能的覆盖, 并平衡所有服务器之间的探测活动。正如我们在本文中所演示的那样, Pingmesh 所使用的 CPU, 内存和带宽开销都是在可承受范围内的。

always-on 的延迟数据带给我们在一开始没有认识到的好处。在经历了一些由数据包黑洞和交换机静默丢包导致的现场事件后, 我们发现因为 Pingmesh 数据的全覆盖和 always-on 的特性, 可以使用 Pingmesh 数据来自动检测这些类型的交换机事故。(第 5 节)

6.2 松散耦合组件有益于扩展

Pingmesh 获益于松散耦合系统设计。Pingmesh Controller 和 Pingmesh Agent 仅通过 pinglist 文件交互, pinglist 文件是由标准 Web API 提供的标准 XML 文档。Pingmesh Agent 以 CSV 文件和标准性能计数器形式提交延迟数据。

采用松散耦合设计, Pingmesh 可以分为三个阶段来构建。在第一阶段, 我们重点关注于 Pingmesh Agent。我们构建了一个简单的 Pingmesh Controller, 它使用简化的 pinglist 生成算法来静态生成 pinglist 文件。直接将没有经过自动分析的延迟数据提交到 Cosmos 中。这个阶段演示了 Pingmesh 的可行性。在这个阶段结束后, 延迟数据已经用于网络 SLA 计算。

在第二阶段, 我们构建了一个完整且成熟的 Pingmesh 控制器, 一旦更新了网络拓扑或调整了配置, 它就会自动更新 pinglists。通过在地理上分布的数据中心设置多个控制器, 新版本的 Pingmesh Controller 还具有更高的容量和更强的容错性。

在第三阶段, 我们集中精力进行数据分析和可视化。我们构建了一个数据处理通道用于分别每 10 分钟, 每 1 小时和每 1 天自动分析所收集的延迟数据。处理后的结果然后存储在数据库中用于可视化, 报告和报警服务。

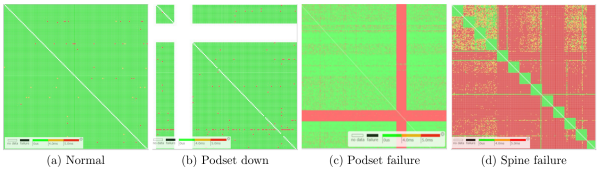


图 8. 可视化的网络延迟图像.

这三个阶段的任务在 2012 六月完成。在此之后, 我们添加了很多新的特性到 Pingmesh 中。

Inter-DC Pingmesh。Pingmesh 原本是用于 intra-DC 的。但是, 扩展它以覆盖 inter-DC 是很容易的。我们扩展了 Pingmesh Controller 的 pinglist 生成算法, 以便从每个数据中心选择一组服务器让它们执行 inter-DC ping(而这个工作已经完成了)。我们没有更改 Pingmesh Agent 的任何一行代码和配置。但是我们增加了一个新的 inter-DC 数据处理管道。

QoS 监控。在部署 Pingmesh 服务后, 我们的数据中心引入了网络 QoS 用于根据 DSCP(differentiated service code point, 差异化服务码点) 区分高优先级和低优先级的数据包。同样, 我们扩展了 Pingmesh Generator 为高优先级和低优先级类生成 pinglists。在这种情况下, 我们确实需要对 Pingmesh Agent 进行简单的配置以便让它监听为低优先级流量配置的其他 TCP 端口。VIP 监控。Pingmesh 最初设计用于测量物理网络的网络延迟。在我们的数据中心中, 广泛应用了负载均衡和 IP 地址虚拟化技术。地址虚拟化为用户提供了一个逻辑虚拟 IP 地址, 而 VIP 被映射到一组物理服务器。这

些服务器的物理 IP 地址被称为 DIP(目的 IP)。在我们的负载均衡系统中, 有一个维护 VIP 和 DIP 映射的控制任务, 还有一个通过数据包封装将发送给 VIP 的数据包转发给 DIP 的数据任务。在部署 Pingmesh 时, Pingmesh 自然需要监控 VIP 的可用性。这也是通过扩展 Pingmesh 生成算法来覆盖目标 VIP 来完成的, 而不涉及 Pingmesh 管道的其余部分。

静默丢包检测。正如我们在第 5 节中讨论的, 我们已经可以使用 Pingmesh 来进行静默丢包检测。因为已经有了延迟数据, 我们只需要思考出检测算法并在 DSA 管道中实现该算法, 而无需接触 Pingmesh 的其他组件。

服务网络指标。服务开发人员使用两个 Pingmesh 指标来设计和实现更好的服务。Pingmesh Agent 为每个服务器提供两个指标: 第 99 百分位数的延迟和丢包率。根据第 99 百分位数的延迟, 服务开发者可以在服务器层次更好地理解数据中心的网络延迟。一些服务已经将每个服务器的丢包率用于服务器选择的度量指标之一。

对于上述扩展, 只有 inter-DC Pingmesh 和 QoS 监控在预期设计中, 剩下的三个超出了我们的预期。感谢松散耦合设计, 所有这些功能都顺利完成了, 并且无需调整 Pingmesh 的体系架构。

6.3 可视化检测模式

我们在 Pingmesh 数据分析和可视化方面投入了大量资金。我们很高兴地发现数据是不言而喻的, 而可视化有助于帮助我们更好地理解和检测各种延迟图案。

图 8展示了一些典型的可视化延迟图案。在图中, 一个绿色、黄色或红色的小块或像素表示一对源/目标 pod 在第 99 百分位数的网络延迟。绿色表示延迟小于 4ms, 黄色表示延迟在 4 ∼ 5ms, 而红色表示延迟大于 5ms, 一个白色的色块表示没有可用的网络延迟数据。

图 8(a) 展示了一个几乎全绿的图案, 表示网络工作正常。虽然看起来很简单, 但是这种全绿图案是 Pingmesh 最广泛使用的功能之一。通过这个图案, 我们可以轻易地判断出全局网络的健康状态。

图 8(b) 显示一个白色十字图案。白色十字的宽度对应于一个 Podset, 包含大约 20 个 pods。这个图案显示的是 Podset 关闭的场景。Podset 关闭通常是因为整个 Podset 都断电了。

图 8(c) 显示一个红色十字图案。红色十字的宽度也对应于一个 Podset。红色十字表明来自该 Podset 的网络流量存在高延迟。这个图案表明该 Podset 中存在网络故障, 因为其他 Podset 的网络延迟是正常的。有多种可能的原因导致该 Podset 显示红色十字。如果 Leaf 和 ToR 交换机都是 L3 交换机, 则至少有一个 Leaf 交换机发生丢包。如果整个 Podset 都是 L2 域, 则可能是因为广播风暴, 例如, 由于一些交换机丢失了配置信息。

图 8(d) 显示了一个仅对角线上的方块呈绿色其余为红色的图案。这里每一个绿色方块都是一个 Podset。这表明每个 Podset 内部的网络延迟是正常的, 但是跨 Podset 的网络延迟都超出了网络 SLA。这说明在 Spine 交换机层存在网络故障。

可视化的成功超出了我们的预期。我们很多人都习惯于定期打开可视化门户网站来看看网络是否正常。这个可视化门户站点不仅被用于网络开发人员和工程师, 还被我们的用户用于了解是否存在一个网络问题。我们还观察到一个有趣的使用模式: 当可视化系统首次投入使用时, 我们的网络团队通常会使用它们向我们的客户" 证明" 网络状况良好。现在我们的客户通常使用这个可视化来查看这里是否发生了网络问题。我们很高兴看到这种使用模式的改变。

6.4 Pingmesh 的局限性

在运行 Pingmesh 期间, 我们发现了 Pingmesh 的两个局限性。首先, 尽管 Pingmesh 能够检测到故障网络设置所在的层, 但它无法确定具体的位置。在我们的网络中, Spine 层有数千台交换机。了解到 Spine 层发生了一些问题是好的但是还不够。我们需要尽快定位和隔离故障设备的方法。这是从一开始就已知的 Pingmesh 的限制。和在 5.2 节所描述的一样, 我们结合 Pingmesh 和 TCP traceroute 来定位这个问题。

第二个限制在于 Pingmesh 测量实时延迟。虽然 Pingmesh Agent 可以发送和接收高达 64KB 的探测消息, 但我们只使用 SYN/SYN-ACK 和单个数据包进行单个 RTT 测量。单个数据包 RTT 可以很好地检测网络可达性和数据包层延迟问题。但它不包括多次往返的情况。我们最近遇到了由于 TCP 参数调整引起的现场事故。我们 TCP 参数配置软件中的一个 Bug 误将 TCP 参数重写为其默认值。结果, 对于一些我们的服务, 初始拥塞窗口 (ICW) 从 16 减少到 4。对于长距离 TCP 会话, 如果会话需要多次往返, 则会话结束时间会增加几百毫秒。Pingmesh 没

有捕捉到这个问题, 因为它只测量单次数据包 RTT。

7. 相关工作

我们运行在世界上最大的数据中心网络之一的实验表明, 包括应用程序, 操作系统内核, NIC, 交换机 ASIC 和固件, 以及光纤在内的所有组件都可能导致通信故障。有关各种可能导致网络隔离的故障, 请参见 [15]。

[16] 和 [17] 通过收集网络跟踪数据, 来研究不同类型数据中心的流量和流量特性。Pingemsh 专注于网络延迟, 是对这些工作的补充。

Pingemsh 和 [18] 都旨在检测网络中的数据包丢失。两者都采用主动探测数据包的方法, 并能够覆盖整个网络。但是它们的方式不同。[18] 采用 RSVP-TE 基源路由来精确定位探测数据包的路由路径。因此它需要先构建路由路径和映射。这也意味着不同于非探测分组通过网络的方式, 探测分组需要经过 LSP(标记交换路径, label switched paths) 来通过网络。其次, RSVP-TE 基于 MPLS, 尽管 MPLS 广泛地应用于 WAN 流量工程, 但是并未在数据中心内使用。Pingmesh 可以同时用于 intra-DC 和 inter-DC 网络。使用源路由确实存在一个优势: [18] 可以直接精确定位发生丢包的交换机或者链路。我们在 5.2 节中展示了 Pingmesh 可以结合 traceroute 来定位故障设备。

Cisco IPSLA[19] 也使用活动数据包进行网络性能监控。IPSLA 在 Cisco 交换机上配置运行, 能够发送 ICMP, IP, UDP, TCP 和 HTTP 数据包。IPSLA 收集网络延迟, 抖动, 丢包, 服务器响应时间甚至噪声质量分数。这些结果存储在交换机本地, 可以通过 SNMP 或 CLI(命令行界面) 检索。Pingmesh 和 IPSLA 有几方面的不同。首先, Pingmesh 使用服务器而不是交换机进行数据收集。通过这样, Pingmesh 不依赖于任何网络设备而 IPSLA 只能在 Cisco 设备上运行。其次, Pingmesh 旨在测量和分析延迟数据。为了实现这个目标, Pingmesh 不仅有 Pingmesh Agent 用于收集数据, 还有一个中央控制面板和数据存储和分析管道。IPSLA 没有没有控制面板和数据存储和分析管道。

NetSight[20] 通过在交换机上引入明信片 (postcard) 过滤器来跟踪数据包历史, 以生成称为明信片 (postcard) 的数据包捕获事件。NetSight 可以构建多种网络定位服务, 例如, nprof, netshark, netwatch, ndb。与 NetSight 相比, Pingmesh 是基于服务器的, 因此它不需要在交换机中引入额外的规则。此外, Pingmesh 能够检测交换机的静默丢包。目前尚不清楚如何为 NetSight 编写静默丢包规则, 因为事先不知道会丢弃哪种类型的数据包。

ATPG[21] 确定了覆盖所有网络链路和转发规则的最小探测包集合。Pingmesh 没有尝试最小化探测的数量。只要开销在可承受范围内, 我们更希望让 Pingmesh 一直运行。此外, 由于无法预先确定黑洞的规则, 因此尚不清楚 ATPG 如何处理数据包黑洞, 。

Pingmesh 专注于物理网络, 它通过在服务器中安装 Pingmesh Agent 来使用主动探测。但是, 对于第三方 VM(虚拟机, Virtual Machine) 和虚拟网络, 安装 Pingmesh Agent 可能不可行。在这种情况下, 可以使用 VND[22] 探索被动流量集合。

8. 总结

我们介绍了用于数据中心网络延迟测量和分析的 Pingmesh 的设计和实现。Pingmesh 始终处于运行状态, 通过所有的服务器收集网络延迟数据并为所有的服务器提供分析结果。Pingmesh 已经在微软的数据中心运行超过 4 年了。它可以帮助我们判断服务问题是否是由网络问题引起的, 并在宏观和微观层次定义和追踪网络 SLA, 它已成为网络故障排除不可缺少的一部分。

基于松散耦合设计, Pingmesh 变得易于扩展。在 Pingmesh 的架构保持不变的情况下, 添加了许多相同的功能。通过研究 Pingmesh 延迟数据以及从可视化和数据挖掘图像中学习, 我们能够不断提高网络质量, 例如: 自动修复数据包黑洞和检测交换机静默随机丢包。

9. 致谢

感谢 Lijiang Fang, Albert Greenberg, Wilson Lee, Randy Kern, Kelvin Yiu, Dongmei Zhang, Yongguang Zhang, Feng Zhao 以及微软亚洲研究院无线和网络部门的成员在这个项目的各个阶段提供支持。也感谢我们的领导 Sujata Banerjee 和匿名 SIGCOMM 评论员提供的宝贵而详细的反馈和评论。

参考文献

[1] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE micro*, (2):22–28, 2003.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. 2003.

[3] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[6] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

[7] Alexey Andreyev. Introducing data center fabric, the next-generation facebook data center network. *Facebook, Nov*, 14:13, 2014.

[8] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 207–218. ACM, 2013.

[9] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[10] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. Bullet trains: a study of nic burst behavior at microsecond timescales. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 133–138. ACM, 2013.

[11] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[12] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[13] Cade Metz. Return of the borg: How twitter rebuilt google' s secret weapon, 2013.

[14] Hadoop. `http://hadoop.apache.org/`.

[15] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *Commun. ACM*, 2014.

[16] 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*, pages 202–208. ACM, 2009.

[17] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[18] Nicolas Guilbaud and Ross Cartlidge. Google localizing packet loss in a large complex network. *Feb*, 5:1–43, 2013.

[19] 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`.

[20] 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 *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 71–85, 2014.

[21] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.

[22] Wenfei Wu, Guohui Wang, Aditya Akella, and Anees Shaikh. Virtual network diagnosis as a service. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 9. ACM, 2013.