The Tail at Scale

作者：Jeffrey Dean, Luiz André Barroso Google Inc 2013-2
原文：http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext
译者：phylips@bmy 2013-02-23
译文：http://duanple.blog.163.com/blog/static/709717672013351121744 5/

序

那些可以对用户动作进行快速响应(100ms 以内)的系统与那些响应慢的系统相比，可以提供更自然流畅的体验[3]。随着 Internet 连通性的提高以及 warehouse-scale 计算系统[2]的出现，使得 web 服务可以在访问存在数千台服务器上的 TB 级数据的同时还能保持流畅的响应；比如 Google 搜索系统会根据用户类型对查询结果进行交互式更新，会基于用户当前已经输入的前缀预测用户查询意图，在数十毫秒的时间内执行该查询并完成结果展示。新兴的增强现实(augmented-reality)设备(比如 Google Glass[7])为保证无缝的交互，对相关 web 服务的快速响应提出了更高要求。

对于服务提供者来说，随着系统规模和复杂度的上升或者是总体访问量的增加，要将交互式服务的延迟分布中的尾端也保持在很低的水平上会变得非常有挑战性。暂时性的高延迟(在中等规模的情况下影响并不大)在大规模场景下可能会严重影响服务整体性能。正如容错计算的目标是在一堆不可靠组件的基础上构建一个可靠的整体一样，大规模在线服务需要在一堆不可预测的组件的基础上创建一个响应可预测的整体；我们将这样的系统称为“lantency tail-tolerant”的，或者简称为“tail-tolerant”的。本文中，我们会列举一些大规模在线系统中发生高延迟的常见起因，同时会描述一些降低它们的严重程度或者是减轻对系统整体性能影响的技术。在很多情况下，“tail-tolerant”技术可以直接利用那些已经部署的用来实现容错的资源，以降低额外开销。我们会来探索一下这些技术是如何在不加重延迟长尾的情况下实现系统利用率的提高，进而避免冗余资源的浪费的。

要点提示

- 在大规模分布式系统中，偶发的系统性能卡壳(hiccup)可能会影响到所有请求的某个重要环节
- 对于大规模分布式系统来说，要完全消除产生延迟抖动的所有来源是不切实际的，尤其是在一个共享的环境中。
- 通过采用类似容错计算所采用的技术，“tail-tolerant”技术在不可预测的组件之上构建了一个可预测的整体

为什么会存在抖动(Variability)?

响应时间的抖动会导致服务的各组件产生高的长尾延迟，而抖动的产生有很多原

因，包括：
资源共享(*Shared resources*)：机器可能是由竞争相同共享资源(比如是 CPU core, 处理器缓存，内存带宽和网络带宽)的多个应用共享，同时相同应用的不同请求也可能发生资源竞争；
守护进程(Daemons)：通常后台守护进程可能只会使用有限的资源，但是在被调度起来的时候可能会产生数毫秒的卡壳(hiccups)；
全局性的资源共享(*Global resource sharing*)：运行在多个不同机器上的应用程序可能会竞争全局性的资源(比如网络交换机和共享的文件系统)；
维护性动作(*Maintenance activities*)：后台活动(比如分布式文件系统中的数据重构，像 Bigtable[4]这样的存储系统中的周期性日志 compaction，支持垃圾回收机制的语言的周期性垃圾回收)可能会导致延迟产生周期性的高峰；同时
队列机制(*Queueing*)：存在于中间服务器和网络交换机的多层队列机制会放大这种抖动。

硬件的发展也可能会成为加重抖动的原因，比如如下几个：
电力控制(*Power limits*)：现代 CPU 设计的可以临时超过其平均功率限制运行，但是如果这种情况持续很长时间就会通过节流来降低产生的热量影响；
垃圾回收(*Garbage collection*)：固态存储设备提供了快速的随机访问，但是大量数据块的周期性垃圾回收需求，即使是在普通的写入频度下仍可能会导致读延迟增加 100 倍[5]；同时
能源管理(*Energy management*)：很多支持节电模式的设备在节省了大量能源的同时也增加了从非活动模式转换到活动模式带来的额外延迟。

组件级抖动在规模增长下的放大

在大规模在线服务中，降低延迟的一种通用技术就是将操作分为多个子操作在多台机器上并行执行，每个子操作操作大规模数据集中的一部分。并行化是通过将请求被从根节点发射到大量的叶子节点服务器执行来完成的，同时会通过请求分发树对响应结果进行归并。为保证服务尽快响应，所有子操作必须要在一个严格的截止期限内完成。

独立组件的延迟抖动，从整个服务的角度上看会被放大；比如，考虑这样一个系统，单个服务器大部分情况下都可以在 10ms 内完成响应，但是也只能保证 99%的延迟在 1 秒内。如果用户请求只是在一个这样的服务上进行处理，100 个请求里可能有 1 个会很慢(1 秒)。下图就展示了在上述假设下，服务级的延迟在规模增大情况下，发生延迟异常的概率变化情况。如果一个用户请求必须要收集来自 100 个这样的并行服务器响应的话，那么 63%的用户请求将会花费超过 1 秒的时间(图中标记为 X 的点)。即使是在单个服务器级别上 10000 个请求中只有一个超过一秒延迟的情况下，对于一个具有 2000 个这样的服务器的服务来说，仍然将会有大概五分之一的请求延迟会超过 1 秒(图中标记为 O 的点)。

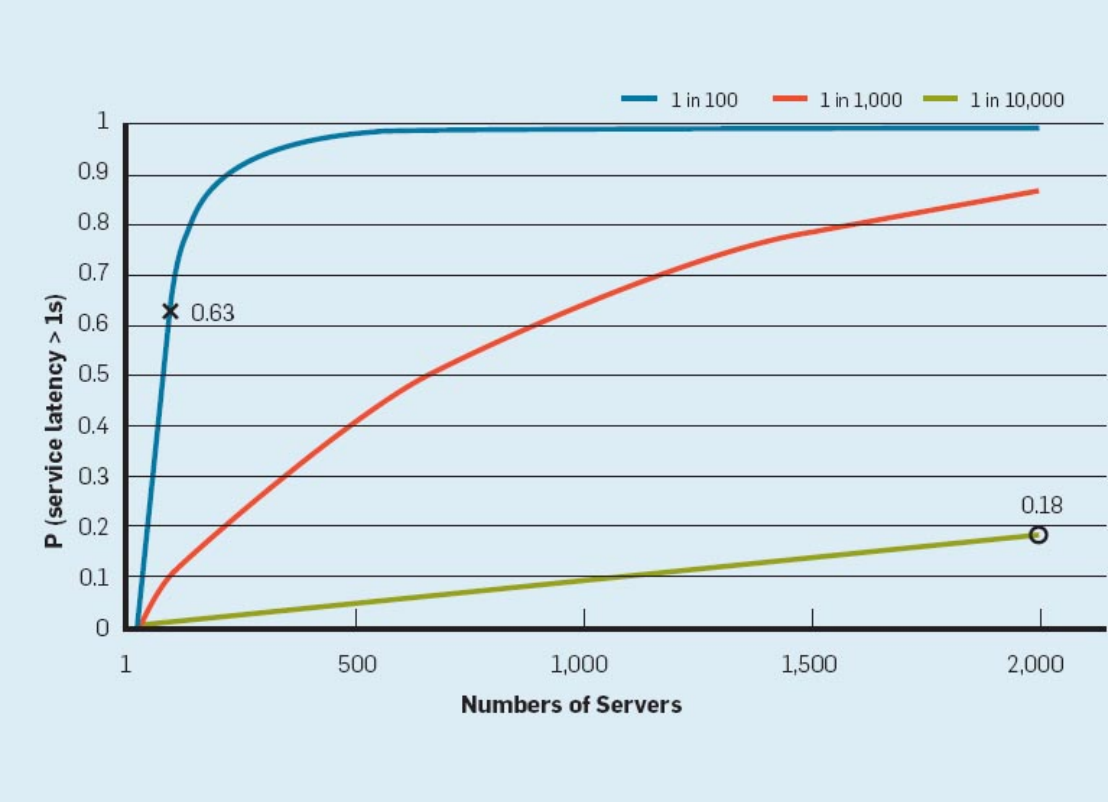

Figure. Probability of one-second service-level response time as the system scales and frequency of server-level high-latency outliers varies.

表 1 列出了一个逻辑上类似于上面设想的场景的 Google 真实服务的测量数据；根服务器会通过很多中间服务器将请求分发到大量的叶子节点服务器上。表格展示了大规模的扇出(fan-out)对延迟分布的影响。在 root 节点的测量结果表明，如果只需要单个随机请求完成，所有延迟的第 99 个百分位数{! percentile(百分位数)，统计学术语，将一组数据按大小排序，并计算相应的累计百分位，则某一百分位所对应数据的值就称为这一百分位的百分位数。 即用 99 个数值或 99 个点，将按大小顺序排列的观测值划分为 100 个等分，则这 99 个数值或 99 个点就称为百分位数，分别代表第 1 个，第 2 个，…，第 99 个百分位数}为 10ms。如果要等所有请求都完成，第 99 个百分位数则变成了 140ms。而如果要等 95%的请求完成这个值是 70ms，这意味着为了等最慢的那 5%的请求完成要多花 70ms。那些专注于这些落后者的技术会显著影响系统整体性能。{!如何理解下面这个表呢？我们可以看做是一个采样过程，比如对于第一行来说，是在一个节点上进行的采样实验，三个列代表了延迟的分布，第一列表示 50%的延迟是在 1ms 以内，第二列表示 95%的延迟是在 5ms 以内，第三列表示 99%的延迟是在 10ms 以内。第二行则是在多个计算机上进行的采样实验同时要求 95%的节点做出响应即可。第三行也是在多个计算机上进行的采样实验但是要求所有节点都做出响应。}

	50%ile latency	95%ile latency	99%ile latency
One random leaf finishes	1ms	5ms	10ms
95% of all leaf requests finish	12ms	32ms	70ms
100% of all leaf requests finish	40ms	87ms	140ms

Table 1. Individual-leaf-request finishing times for a large fan-out service tree (measured from root node of the tree).

可以通过在所有层次和组件上预留(Overprovisioning)资源，进行细致的软件实时性设计以及提高可靠性，来减少抖动产生的根源。下面我们会讲述一些可以用来降低服务响应抖动的通用方法。

降低组件抖动

可以通过采用如下一些工程实现策略保证交互请求被及时处理来降低交互中的延迟抖动：

根据服务类别进行差异化及更高层次的队列机制(Differentiating service classes and higher-level queuing)：根据服务类别进行差异化就可以在那些非交互式请求之前优先调度那些用户正在等待的请求。保持底层队列的短小可以让上层策略更快产生效果；比如，Google 集群级文件系统的存储服务器会让操作系统的磁盘队列长度始终保持在非常低的水平，同时维护了一个自己的优先级队列来缓存磁盘请求。该队列允许服务器跳过那些更早到来的非延迟敏感的批处理操作，优先往下传送那些高优先级的交互式请求。

降低线头阻塞(Reducing head-of-line blocking)：{!线头阻塞（Head-of-line blocking, HOL）是一种出现在缓存式通信网络交换中的一种现象。交换通常由缓存式输入端口、一个交换架构以及缓存式输出端口组成。当在相同的输入端口上到达的包被指向不同的输出端口的时候就会出现线头阻塞。由于输入缓存以及交换设计的 FIFO 特性，交换架构在每一个周期中只能交换缓存头部的包。如果某一缓存头部的包由于拥塞而不能交换到一个输出端口，那么该缓存中余下的包也会被线头包所阻塞，即使这些包的目的端口并没有拥塞。这里位于缓存头部的包就类似于下文中的 long-running request}上层服务可以处理具有各种不同内在开销的请求。如果将需要长时间运行的请求(long-running request)切分成一系列的小请求，就可以让它与其他短时间即可运行完的请求(short-running request)交叉执行，有时对于系统来说是非常有用的；比如，Google 的网页搜索系统就会使用类似的分时机制来防止少数开销极高的查询请求影响到其他请求。

管理后台活动和对破坏进行同步(Managing background activities and

synchronized disruption)：后台任务会产生显著的 CPU，磁盘或网络负载；像基于日志的(log-oriented)存储系统中的 log compaction 以及支持自动垃圾回收的语言的垃圾回收活动。可以通过综合使用节流(throttling)，将重量级操作切分为小操作以及在总体负载较低时触发这些后台活动降低交互请求的延迟。对于具有大规模扇出(fan-out)的服务，在多个机器上进行后台活动的同步有时对于系统来说是非常有用的。这种同步会导致每台机器上同时出现短暂性的活动爆发，这样就只会影响到那些出现在这个短暂的时间区间内的请求。相比之下，如果没有同步，总有一些机器在进行后台活动，从而导致所有的请求会产生延迟长尾。

目前为止的讨论中一直未涉及到缓存这个话题。尽管对于很多系统来说，高效的缓存层是非常有用的甚至是必需的，但是它并没有直接解决长尾延迟问题，除非可以保证将应用程序的整个工作集完全放入缓存。

Living with Latency Variability

上一节提到的那些细致的工程技术对于构建高性能交互式服务系统是非常必要的，但是现代 web 服务的规模和复杂性使得消除所有延迟抖动是不可能的。即使是能在孤立系统中完美实现，但是那些共享计算资源的系统的性能波动已经超出了应用开发者所能控制的范围。因此，在 Google 都是通过开发“tail-tolerant”技术来掩盖或绕过延迟问题，而不是试图完全消除它。我们将这些技术分为两大类：第一类是专注于请求内部的立即响应技术，操作对象的时间尺度为数十毫秒；第二类为跨请求的 long-term adaptations，所工作的时间尺度从数十秒到数分钟，是为了掩盖持续时间更长的现象的影响。

Within Request Short-Term Adaptations

很多 web 服务器都会为单个数据项部署多个副本来提供额外的吞吐容量以及保证故障发生时的可用性。在大多数请求操作针对的都是只读，松散一致的数据集合的情况下，这种方法非常有效；一个例子是拼写校正服务，对于该服务来说它的模型每天更新一次，而每秒要处理成千上万的校正请求。类似的，分布式文件系统可能有一个给定数据块的多个副本，每个副本都可以被用来服务读请求。这里的技术展示了如何采用副本来减少单个请求的延迟抖动。

对冲请求(Hedged requests)：一种用来抑制延迟抖动的简单方法就是将相同请求发送给多个副本，然后使用最先返回的那个作为结果。我们将这些请求称为“对冲请求”，因为客户端会首先发送一个请求到被认为是最合适的那个副本，随后在短暂的延迟后会再发送一个请求。一旦接收到响应结果客户端会忽略剩余未完成的请求。虽然该技术的这种简单实现版本会引入不可接受的额外负载，但是它的一些变种实现却可以在对负载略有影响的情况下得到很好的延迟降低效果。

比如可以将第二个请求延迟到发出的第一个请求在超过该类请求延迟的第 95 个

百分位数代表的延迟还未返回后再发送。这样就在大大降低了延迟的同时，把额外负载限制在大概 5%上。这项技术之所以可行是因为延迟通常不是请求本身固有的而是由其他形式的干扰造成的。比如，Google 的某个基准测试程序会从跨越 100 个不同机器的 Bigtable 系统中读取 1000 个 key 所对应的 value 值。通过在 10ms 的延迟后发送一个对冲请求，将检索 1000 个 key 的 99.9 th 百分位延迟从 1800ms 降到了 74ms，同时仅多发了 2%的请求。还可以通过让后发请求优先级低于主请求来进一步降低对冲请求的开销。

Tied requests：对冲请求技术存在一个脆弱窗口，在该窗口内多个服务器可能会不必要地对同一个请求进行处理。这些额外的工作量可以通过在发送对冲请求之前等待所有延迟的第 95 个的百分位数代表的时间进行限制，但是这就将收益限制在了一少部分请求之上。如果要在合理的资源消耗下更积极的使用对冲请求，则需要能对请求进行更快速的取消。

抖动的一个常见来源是在请求被执行前在服务端的排队延迟。对于很多服务来说，一旦请求被调度起来开始执行，它的完成时间的变化实际很小。Mitzenmacher[10]曾说过，与一个均匀随机的方案相比，让请求在入队时可以基于队列长度在两个服务器间进行选择，负载平衡性能将会有指数级的提升。我们提倡的不是选择某个服务器而是将某个请求同时在多个服务器上入队，同时允许这些服务器相互之间进行通信更新关于这些拷贝的状态。我们将这些会进行跨服务器状态更新的请求称为“tied requests”。“tied requests”最简单的形式就是客户端将请求发送给两个不同的服务器，每个都绑定了另一个服务器上的请求的标识符。当一个请求开始执行时，会向另一个服务器发送取消消息。另一个请求如果还正在服务器上排队，就可以立即终止或者是将优先级大大调低。

此处有一个平均网络延迟大小的窗口时间，在这个窗口内，两个服务器可能都已经开始处理请求，但是它们的取消消息都还在路上。会发生这种情况的一个常见场景是，两个服务器的队列都为空。因此，对于客户端来说在发送第一个请求与第二个请求之间引入一个平均网络消息延迟大小(在现代数据中心里这个值通常都是小于 1ms)的等待时间会比较好。Google 将这种技术在分布式文件系统中进行了实现，有效地降低了延迟的中位数和长尾值。表 2 列出了 Bigtable 中读请求的响应时间，读取的数据没有被缓存都必须要从文件系统中读出，每个文件块有分布在不同机器上的三个副本。表中包含了有和没有“tied requests”在两种场景下的读请求延迟：第一个场景来自于一个只有基准测试程序运行的独立集群，在这种情况下，延迟抖动主要来自于内部干扰及常规的集群管理活动。此时，发送“tied requests”将延迟的中位数降低了 16%，而且越往后看效果越明显，在 99.9th-percentile 时快达到了 40%。第二个场景类似于第二个场景，只是与此同时还有另一个排序 job 正在集群上运行，并与其竞争它们所共享的文件系统的磁盘资源。尽管由于资源使用率偏高导致总体延迟也要偏高，但是延迟的降低上也达到了之前的类似效果。而采用了“tied requests”同时具有排序 job 的延迟测试结果，与没有采用“tied requests”同时大部分空闲的集群上的测试结果非常接近{!即下图中的第 2 列数据与第 6 列数据}。“tied requests”可以将负载在单个集群上进行合并，从而显著地降低计算开销。在表 2 的两种场景下，“tied requests”带来的磁盘负载开销小于 1%，这表明取消策略在降低冗余的读操作上非常有效。

	Mostly idle cluster			With concurrent terasort		
	No hedge	Tied request after 1ms		No hedge	Tied request after 1ms	
50%ile	19ms	16ms	(–16%)	24ms	19ms	(–21%)
90%ile	38ms	29ms	(–24%)	56ms	38ms	(–32%)
99%ile	67ms	42ms	(–37%)	108ms	67ms	(–38%)
99.9%ile	98ms	61ms	(–38%)	159ms	108ms	(–32%)

Table 2. Read latencies observed in a BigTable service benchmark.

针对 tied-request 和对冲请求的一种改进是：首先对远程队列进行探测，然后将请求提交到负载最轻的服务器上[10]。这是有益的，但是效果不如将任务同时提交给两个队列，主要原因有三个：在探测和发送请求之间，负载可能发生变化；由于底层系统和硬件的不同，请求服务时间很难估计；当所有客户端都同时选择向那个负载最低的服务器发送请求时，它会成为新的热点。“Distributed Shortest-Positioning Time First system”[9]采用了另一种变种实现：请求首先被发送到一个服务器，然后只有在缓存命中失败时才会将它转发给另一个服务器，同时采用了跨机器的取消机制。

值得指出的是这种技术不仅适用于存在副本的情况，也可以将它应用在复杂的编码模式(比如 Reed-Solomon)下，在这种情况下首先将主请求发送给包含了所需数据块的机器，在简短的等待后如果没有收到响应，那么将会产生一组请求发送给足以将数据重构出来的一组机器，实际上组成了一个 tied-request 集合。

另外还需要注意的是，此处描述的这类技术只有在引发抖动的病症不会同时影响到多个请求副本时才会是有效的。同时我们认为这种非关联病症在大规模系统中是更普遍的。

Cross-Request Long-Term Adaptations

现在，我们将目光转向那些用于降低由粗粒度的现象(比如服务时间变化和负载不平衡)引发的延迟抖动的技术。尽管很多系统都尽量将数据划分得每个分区都具有相同开销，但是简单地为每个机器分配单个分区的静态分配方式远远不够，原因有二：首先，随着时间的推移，由于各种各样的原因(比如前面提到的资源共享，CPU 的过热降频保护机制)底层机器的性能表现既不均匀也不是不变的；其次，分区内的异常数据项也可能导致由数据诱发的负载不均衡(比如某个数据项突然成为访问热点，那么它所在分区的负载会急剧上升)。

Micro-partitions：为了应对不平衡的情况， Google 的很多系统都会产生数量上远大于机器数的分区(partition)，然后将这些分区在特定机器集合上进行动态的分配和负载平衡。这样负载平衡所需要做的就是将一个分区从一台机器搬到另一台上去。假设平均每台机器 20 个分区，跟机器分区一对一分配的方式相比，粒度更细移动单个分区所需的时间也更少。Bigtable 分布式存储系统就是将数据以 tablet 为单位进行存储的，同时每台机器大概管理了 20 到 1000 个 tablet。此外 micro-partitioning 还提高了故障恢复速度，因为当某台机器出现故障时，可以由其他多个机器接收它上面的工作单元。这种方法非常类似于 Stoica[12]中描述的虚服务器概念以及 DeWitt 等人提出的 virtual-processor-partitioning 技术[6]。

Selective replication：针对 micro-partitioning 模式的一个改进是检测甚至是预测出那些可能导致负载不平衡的数据项，并为它们创建额外的副本。负载平衡系统之后就可以通过这些额外的副本将那些热点 micro-partition 的负载分摊到多台机器上而不用实际移动它们。Google 的主搜索系统就采用了这个策略，在多个 micro-partition 上为那些最流行和最重要的文档创建副本。在 Google 网页搜索系统的长期演化中，已经建立起了偏向特定文档语言的 micro-partitions，并能够根据典型的一天内的查询变化模式对这些 micro-partitions 的副本进行动态调整。此外查询模式也可能发生突变，比如亚洲数据中心的宕机将会导致大量亚洲语系的相关查询被指向位于北美的设施，从实质上改变负载行为。

Latency-induced probation：通过观测系统中各个机器的响应延迟分布，中间服务器有时能够检测出某些情况，在这些情况下将那些特别慢的机器排除或者是将它判为缓刑{!probation：缓刑，即不是立即将它判死刑排除掉，比如可以临时放入黑名单暂时不用}，可以提高系统性能。运行缓慢通常是因为暂时性的现象，像不相关的网络流量的干扰或者是由于机器上另一个作业导致的 CPU 使用飙升，而且系统负载越高这种现象导致的缓慢就越明显。但是系统还会持续的向被排除的服务器发送影子(shadow)请求，并收集延迟统计信息，以在问题消失时让它们重新加入以提供服务。这种情况有点奇特，因为在系统处于高负载期间去除了一部分服务容量但是延迟却改进了。

Large Information Retrieval Systems

在大型信息检索(IR)系统中，速度不仅仅是一个性能指标；它也是一个关键的质量指标，因为返回一个够好的结果要优于很慢地返回一个最好的结果。两种技术都可以应用到这类系统，以及其他的那些本质上并不需要精确结果的系统中：
Good enough.在大型 IR 系统中，一旦所有叶子服务器中返回响应的占到足够的比例，为了获取更好的端到端延迟，就可以直接将这些稍有欠缺(good enough)的结果返回给用户。某个叶子服务器具有最好的查询结果的概率，1000 个查询里不到一个，此外还可以进一步地通过将那些最重要的文档备份到多个叶子服务器减少这个概率。由于如果要等待那些运行地及其缓慢的服务器可能会将整个服务的延迟拖到不可忍受的级别，Google 的 IR 系统已经调整地只要整个语料库有足够比例的部分被搜索到就可以提供足够好的返回结果，尽管需要小心确保结果足够好的情况比较罕见。通常，good enough 模式也会被用来跳过非必要的子系

统来提高系统响应；比如，对于网页搜索来说，如果广告系统和拼写校正系统无法及时响应，可以直接跳过它们。

A simple way to curb latency variability is to issue the same request to multiple replicas and use the results from whichever replica responds first.

Canary requests. 可能发生在高扇出系统中的另一个问题是一个特定的请求可能触发特定的代码路径，触发 crash 甚至导致数千台机器同时产生极大的延迟。为了避免这种情况，Google 的 IR 系统采用了一种称为“canary requests”的技术。不是一开始就将单个请求直接发送到数千个叶子服务器上，根服务器会首先将它发送到 2 到 3 台叶子服务器，只有当根服务器在合理的时间段内成功收到响应后再将请求发送给其他服务器。如果在请求被发出后服务器崩溃或者 hang 住了，那么系统会将该请求标识为具有潜在危险性，同时不再将请求发送给其他服务器来避免进一步的执行。“canary requests”提供了一种在面对难以预测的编程错误以及恶意的拒绝服务攻击时，衡量后端系统健壮性的方法。

“canary requests”阶段只增加了少量的系统总体延迟。因为与大规模扇出情况下需要等待所有服务器返回相比，系统只需要等待单个服务器返回的延迟抖动要小多了；比较表 1 中的第一行和最后一行也可以看出。尽管“canary requests”会对延迟产生轻微影响，但是由于它们提供的这种安全性保证，我们更倾向于在 Google 的所有大规模扇出搜索系统中采用这种请求方式。

Mutations

目前为止我们所讨论的技术都是最适用于那些不会对系统状态进行关键变更的操作，涵盖了一大类数据密集型服务。对于那些会改变系统状态的操作来说，要容忍它们产生的延迟抖动相对容易，原因如下：首先，在这些服务中对延迟要求比较高的修改操作的规模通常很小；第二，实际的更新操作可以从关键路径上剥离，在将响应返回后再执行；第三，很多服务都可以设计地可以容忍不一致的更新模型(延迟容忍度更高)；最后，对于那些需要一致性更新的服务来说，最通用的技术就是采用 quorum-based 算法(比如 Lamport 的 Paxos 算法[8])；由于这些算法在五个副本的情况下只要三个提交就 ok 了，天生就是 tail-tolerant 的。

硬件发展趋势及其影响

未来由于更激进的功率优化技术的应用以及在深亚微米级的制造挑战带来的设备异构性，硬件层面产生的抖动可能会变得更高。伴随着系统规模的不断增长，设备异构性将会使基于软件层面的抖动容忍技术变得日益重要。幸运的是，一些新兴的硬件趋势会增加延迟容忍技术的有效性。比如，数据中心内部网络更高的等分带宽以及用来降低单消息传输开销的优化技术(如远程 DMA)，将会降低“tied

requests”的开销，使得取消消息可以更容易被及时接收到从而避免冗余工作。更低的单消息开销，将会允许产生更多的细粒度请求，从而更好地进行多路复用以及避免线头阻塞带来的影响。

总结

伴随着下一代计算密集型、无缝交互的云服务的兴起，可以持续提供一致性响应的大规模计算系统开始受到越来越多的关注。随着系统规模扩大，简单地铲除产生性能异变的所有来源无法达到这样的响应能力。容错技术(Fault-tolerant)的出现是因为当超过一定的系统规模后，已经无法保证无故障的运行。类似的，“tail-tolerant”技术的出现也是因为对于大规模服务来说已经无法消除产生抖动的所有来源。虽然那些用来解决特定延迟抖动来源的方法也是有用的，但是最有力的“tail-tolerant”技术应是可以在不管根源的情况下就可以降低延迟。这些“tail-tolerant”技术允许设计者针对常见情况(common case)进行不断优化，同时能够应对非常见的情况。我们已经列出了一些“tail-tolerant”技术，它们已在 Google 的几个大规模软件系统中起到了非常好的效果。随着互联网服务需求的不断增长，warehouse-scale 系统的增多以及底层硬件性能抖动的增大，它们的重要性也会不断提升。

虽然一些最强大的“tail-tolerant”技术需要占用额外资源，但是它们的开销可以控制在合理范围内，通常只利用现有为容错所预留的资源就可以带来延迟方面的很大改进。此外，其中很多技术都可以直接封装到基础库和系统中，同时延迟方面的改进通常可以从大大简化应用层的设计。除了确保大规模情况下的低延迟外，这些技术还使得在不牺牲服务响应能力情况下获得更高的系统利用率成为可能。

致谢

We thank Ben Appleton, Zhifeng Chen, Greg Ganger, Sanjay Ghemawat, Ali Ghodsi, Rama Govindaraju, Lawrence Greenfield, Steve Gribble, Brian Gustafson, Nevin Heintze, Jeff Mogul, Andrew Moore, Rob Pike, Sean Quinlan, Gautham Thambidorai, Ion Stoica, Amin Vahdat, and T.N. Vijaykumar for their helpful feedback on earlier drafts and presentations of this work. Numerous people at Google have worked on systems that use these techniques.

参考文献

1. Barroso, L.A. and Höelzle, U. The case for energy proportional computing. *IEEE Computer 40*, 12 (Dec. 2007), 33–37.

2. Barroso, L.A. and Höelzle, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*. Synthesis Series on Computer Architecture, Morgan & Claypool Publishers, May 2009.
3. Card, S.K., Robertson, G.G., and Mackinlay, J.D. The information visualizer: An information workspace. In*Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems* (New Orleans, Apr. 28–May 2). ACM Press, New York, 1991, 181–188.
4. Chang F., Dean J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R.E. BigTable: A distributed storage system for structured data. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation* (Seattle, Nov.). USENIX Association, Berkeley CA, 2006, 205–218.
5. Charles, J., Jassi, P., Ananth, N.S., Sadat, A., and Fedorova, A. Evaluation of the Intel Core i7 Turbo Boost feature. In *Proceedings of the IEEE International Symposium on Workload Characterization* (Austin, TX, Oct. 4–6). IEEE Computer Society Press, 2009, 188–197.
6. DeWitt, D.J., Naughton, J.F., Schneider, D.A., and Seshadri, S. Practical skew handling in parallel joins. In*Proceedings of the 18th International Conference on Very Large Data Bases*, Li-Yan Yuan, Ed. (Vancouver, BC, Aug. 24–27). Morgan Kaufmann Publishers, Inc., San Francisco, 1992, 27–40.
7. Google, Inc. Project Glass; http://g.co/projectglass
8. Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.
9. Lumb, C.R. and Golding, R. D-SPTF: Decentralized request distribution in brick-based storage systems.*SIGOPS Operating System Review 38*, 5 (Oct. 2004), 37–47.
10. Mitzenmacher, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Computing 12*, 10 (Oct. 2001), 1094–1104.
11. Mudge, T. and Hölzle, U. Challenges and opportunities for extremely energy-efficient processors. *IEEE Micro 30*, 4 (July 2010), 20–24.
12. Stoica I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of SIGCOMM* (San Diego, Aug. 27–31). ACM Press, New York, 2001, 149–160.

作者介绍

Jeffrey Dean (jeff@google.com) is a Google Fellow in the Systems Infrastructure Group of Google Inc., Mountain View, CA.
Luiz André Barroso (luiz@google.com) is a Google Fellow and technical lead of core computing infrastructure at Google Inc., Mountain View, CA.