Dapper，大规模分布式系统的跟踪系统

概述

当代的互联网的服务，通常都是用复杂的、大规模分布式集群来实现的。互联网应用构建在不同的软件模块集上，这些软件模块，有可能是由不同的团队开发、可能使用不同的编程语言来实现、有可能布在了几千台服务器，横跨多个不同的数据中心。因此，就需要一些可以帮助理解系统行为、用于分析性能问题的工具。

Dapper--Google 生产环境下的分布式跟踪系统，应运而生。那么我们就来介绍一个大规模集群的跟踪系统，它是如何满足一个低损耗、应用透明的、大范围部署这三个需求的。当然 Dapper 设计之初，参考了一些其他分布式系统的理念，尤其是 Magpie 和 X-Trace，但是我们之所以能成功应用在生产环境上，还需要一些画龙点睛之笔，例如采样率的使用以及把代码植入限制在一小部分公共库的改造上。

自从 Dapper 发展成为一流的监控系统之后，给其他应用的开发者和运维团队帮了大忙，所以我们今天才发表这篇论文，来汇报一下这两年来，Dapper 是怎么构建和部署的。Dapper 最初只是作为一个自给自足的监控工具起步的，但最终进化成一个监控平台，这个监控平台促生出多种多样的监控工具，有些甚至已经不是由 Dapper 团队开发的了。下面我们会介绍一些使用 Dapper 搭建的分析工具，分享一下这些工具在 google 内部使用的统计数据，展现一些使用场景，最后会讨论一下我们迄今为止从 Dapper 收获了些什么。

1. 介绍

我们开发 Dapper 是为了收集更多的复杂分布式系统的行为信息，然后呈现给 Google 的开发者们。这样的分布式系统有一个特殊的好处，因为那些大规模的低端服务器，作为互联网服务的载体，是一个特殊的经济划算的平台。想要在这个上下文中理解分布式系统的行为，就需要监控那些横跨了不同的应用、不同的服务器之间的关联动作。

下面举一个跟搜索相关的例子，这个例子阐述了 Dapper 可以应对哪些挑战。比如一个前段服务可能对上百台查询服务器发起了一个 Web 查询，每一个查询都有自己的 Index。这个查询可能会被发送到多个的子系统，这些子系统分别用来处理广告、进行拼写检查或是查找一些像图片、视频或新闻这样的特殊结果。根据每个子系统的查询结果进行筛选，得到最终结果，最后汇总到页面上。我们把这种搜索模型称为“全局搜索”（universal search）。总的来说，这一次全局搜索有可能调用上千台服务器，涉及各种服务。而且，用户对搜索的耗时是很敏感的，而任何一个子系统的低效都导致导致最终的搜索耗时。如果一个工程师只能知道这个查询耗时不正常，但是他无从知晓这个问题到底是由哪个服务调用造成的，或者为什么这个调用性能差强人意。首先，这个工程师可能无法准确的定位到这次全局搜索是调用了

哪些服务，因为新的服务、乃至服务上的某个片段，都有可能在任何时间上过线或修改过，有可能是面向用户功能，也有可能是一些例如针对性能或安全认证方面的功能改进。其次，你不能苛求这个工程师对所有参与这次全局搜索的服务都了如指掌，每一个服务都有可能是由不同的团队开发或维护的。再次，这些暴露出来的服务或服务器有可能同时还被其他客户端使用着，所以这次全局搜索的性能问题甚至有可能是由其他应用造成的。举个例子，一个后台服务可能要应付各种各样的请求类型，而一个使用效率很高的存储系统，比如 Bigtable，有可能正被反复读写着，因为上面跑着各种各样的应用。

上面这个案例中我们可以看到，对 Dapper 我们只有两点要求：无所不在的部署，持续的监控。无所不在的重要性不言而喻，因为在使用跟踪系统的进行监控时，即便只有一小部分没被监控到，那么人们对这个系统是不是值得信任都会产生巨大的质疑。另外，监控应该是 7x24 小时的，毕竟，系统异常或是那些重要的系统行为有可能出现过一次，就很难甚至不太可能重现。那么，根据这两个明确的需求，我们可以直接推出三个具体的设计目标：

1.低消耗：跟踪系统对在线服务的影响应该做到足够小。在一些高度优化过的服务，即使一点点损耗也会很容易察觉到，而且有可能迫使在线服务的部署团队不得不将跟踪系统关停。

2.应用级的透明：对于应用的程序员来说，是不需要知道有跟踪系统这回事的。如果一个跟踪系统想生效，就必须需要依赖应用的开发者主动配合，那么这个跟踪系统也太脆弱了，往往由于跟踪系统在应用中植入代码的 bug 或疏忽导致应用出问题，这样才是无法满足对跟踪系统“无所不在的部署”这个需求。面对当下想 Google 这样的快节奏的开发环境来说，尤其重要。

3.延展性：Google 至少在未来几年的服务和集群的规模，监控系统都应该能完全把控住。

一个额外的设计目标是为跟踪数据产生之后，进行分析的速度要快，理想情况是数据存入跟踪仓库后一分钟内就能统计出来。尽管跟踪系统对一小时前的旧数据进行统计也是相当有价值的，但如果跟踪系统能提供足够快的信息反馈，就可以对生产环境下的异常状况做出快速反应。

做到真正的应用级别的透明，这应该是当下面临的最挑战性的设计目标，我们把核心跟踪代码做的很轻巧，然后把它植入到那些无所不在的公共组件种，比如线程调用、控制流以及 RPC 库。使用自适应的采样率可以使跟踪系统变得可伸缩，并降低性能损耗，这些内容将在第 4.4 节中提及。结果展示的相关系统也需要包含一些用来收集跟踪数据的代码，用来图形化的工具，以及用来分析大规模跟踪数据的库和 API。虽然单独使用 Dapper 有时就足够让开发人员查明异常的来源，但是 Dapper 的初衷不是要取代所有其他监控的工具。我们发现，Dapper 的数据往往侧重性能方面的调查，所以其他监控工具也有他们各自的用处。

1.1 文献的总结

分布式系统跟踪工具的设计空间已经被一些优秀文章探索过了，其中的 Pinpoint[9]、Magpie[3]和 X-Trace[12]和 Dapper 最为相近。这些系统在其发展过程的早期倾向于写入研究报告中，即便他们还没来得及清楚地评估系统当中一些设计的重要性。相比之下，由于 Dapper 已经在大规模生产环境中摸爬滚打了多年，经过这么多生产环境的验证之后，我们认为这篇论文最适合重点阐述在部署 Dapper 的过程中我们有那些收获，我们的设计思想是如何决定的，以及以什么样的方式实现它才会最有用。Dappe 作为一个平台，承载基于 Dapper 开发的性能分析工具，以及 Dapper 自身的监测工具，它的价值在于我们可以在回顾评估中找出一些意想不到的结果。

虽然 Dapper 在许多高阶的设计思想上吸取了 Pinpoint 和 Magpie 的研究成果，但在分布式跟踪这个领域中，Dapper 的实现包含了许多新的贡献。例如，我们想实现低损耗的话，特别是在高度优化的而且趋于极端延迟敏感的 Web 服务中，采样率是很必要的。或许更令人惊讶的是，我们发现即便是 1/1000 的采样率，对于跟踪数据的通用使用层面上，也可以提供足够多的信息。

我们的系统的另一个重要的特征，就是我们能实现的应用级的透明。我们的组件对应用的侵入被先限制在足够低的水平上，即使想 Google 网页搜索这么大规模的分布式系统，也可以直接进行跟踪而无需加入额外的标注(Annotation)。虽然由于我们的部署系统有幸是一定程度的同质化的，所以更容易做到对应用层的透明这点，但是我们证明了这是实现这种程度的透明性的充分条件。

2. Dapper 的分布式跟踪

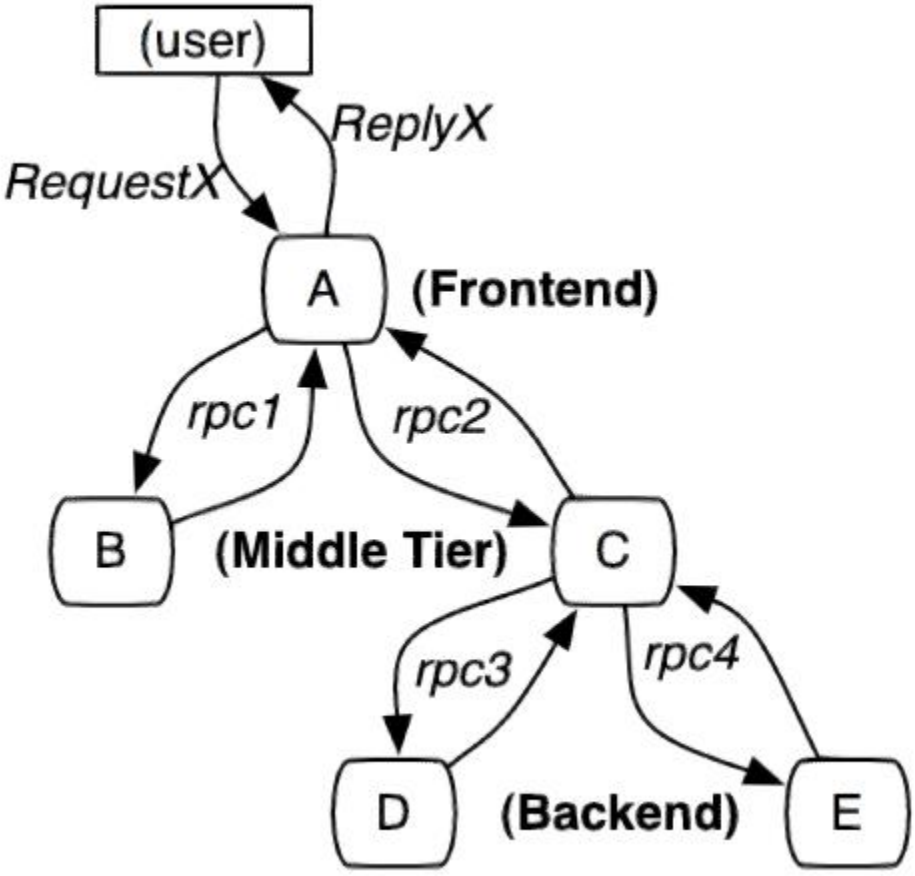

图 1：这个路径由用户的 X 请求发起，穿过一个简单的服务系统。用字母标识的节点代表分布式系统中的不同处理过程。

分布式服务的跟踪系统需要记录在一次特定的请求后系统中完成的所有工作的信息。举个例子，图 1 展现的是一个和 5 台服务器相关的一个服务，包括：前端（A），两个中间层（B 和 C），以及两个后端（D 和 E）。当一个用户（这个用例的发起人）发起一个请求时，首先到达前端，然后发送两个 RPC 到服务器 B 和 C。B 会马上做出反应，但是 C 需要和后端的 D 和 E 交互之后再返还给 A，由 A 来响应最初的请求。对于这样一个请求，简单实用的分布式跟踪的实现，就是为服务器上每一次你发送和接收动作来收集跟踪标识符(message identifiers)和时间戳(timestamped events)。

为了将所有记录条目与一个给定的发起者（例如，图 1 中的 RequestX）关联上并记录所有信息，现在有两种解决方案，黑盒(black-box)和基于标注(annotation-based)的监控方案。黑盒方案[1，15，2]假定需要跟踪的除了上述信息之外没有额外的信息，这样使用统计回归技术来推断两者之间的关系。基于标注的方案[3，12，9，16]依赖于应用程序或中间件明确地标记一个全局 ID，从而连接每一条记录和发起者的请求。虽然黑盒方案比标注方案更轻便，他们需要更多的数据，以获得足够的精度，因为他们依赖于统计推论。基于标注的方案最主要的缺点是，很明显，需要代码植入。在我们的生产环境中，因为所有的应用程序都使用相同的线程

模型，控制流和 RPC 系统，我们发现，可以把代码植入限制在一个很小的通用组件库中，从而实现了监测系统的应用对开发人员是有效地透明。

我们倾向于认为，Dapper 的跟踪架构像是内嵌在 RPC 调用的树形结构。然而，我们的核心数据模型不只局限于我们的特定的 RPC 框架，我们还能跟踪其他行为，例如 Gmail 的 SMTP 会话，外界的 HTTP 请求，和外部对 SQL 服务器的查询等。从形式上看，我们的 Dapper 跟踪模型使用的树形结构，Span 以及 Annotation。

2.1 跟踪树和 span

在 Dapper 跟踪树结构中，树节点是整个架构的基本单元，而每一个节点又是对 span 的引用。节点之间的连线表示的 span 和它的父 span 直接的关系。虽然 span 在日志文件中只是简单的代表 span 的开始和结束时间，他们在整个树形结构中却是相对独立的，任何 RPC 相关的时间数据、零个或多个特定应用程序的 Annotation 的相关内容会在 2.3 节中讨论。

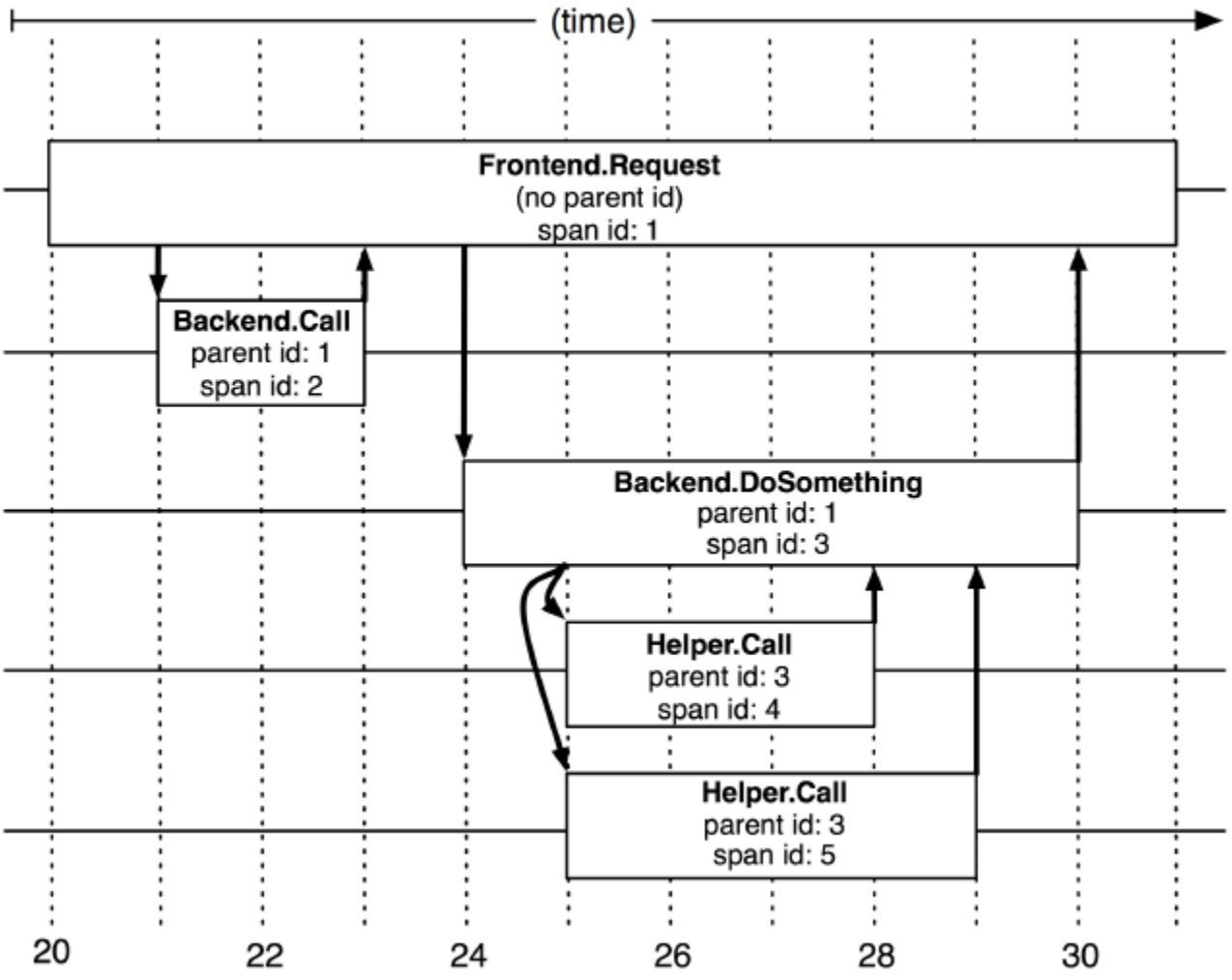

图 2：5 个 span 在 Dapper 跟踪树种短暂的关联关系

在图 2 中说明了 span 在一个大的跟踪过程中是什么样的。Dapper 记录了 span 名称，以及每个 span 的 ID 和父 ID，以重建在一次追踪过程中不同 span 之间的关系。如果一个 span 没有父 ID 被称为 root span。所有 span 都挂在一个特定的跟踪上，也共用一个跟踪 id（在图中未示出）。所有这些 ID 用全局唯一的 64 位整数标示。在一个典型的 Dapper 跟踪中，我们希望为每一个 RPC 对应到一个单一的 span 上，而且每一个额外的组件层都对应一个跟踪树型结构的层级。

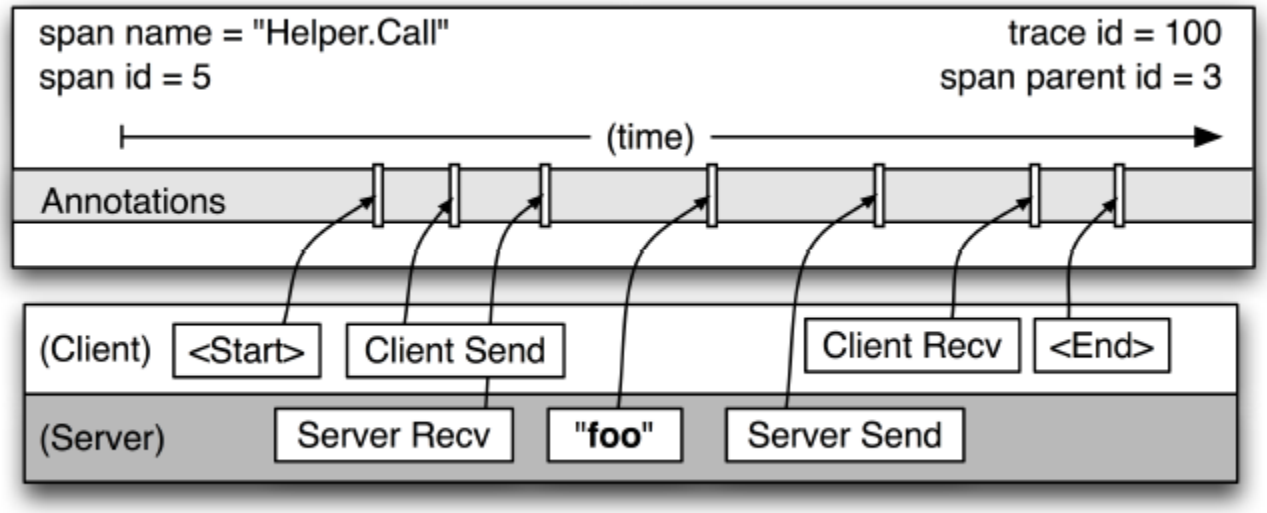

图 3：在图 2 中所示的一个单独的 span 的细节图

图 3 给出了一个更详细的典型的 Dapper 跟踪 span 的记录点的视图。在图 2 中这种某个 span 表述了两个“Helper.Call”的 RPC(分别为 server 端和 client 端)。span 的开始时间和结束时间，以及任何 RPC 的时间信息都通过 Dapper 在 RPC 组件库的植入记录下来。如果应用程序开发者选择在跟踪中增加他们自己的注释（如图中“foo”的注释）(业务数据)，这些信息也会和其他 span 信息一样记录下来。

记住，任何一个 span 可以包含来自不同的主机信息，这些也要记录下来。事实上，每一个 RPC span 可以包含客户端和服务器两个过程的注释，使得链接两个主机的 span 会成为模型中所说的 span。由于客户端和服务器上的时间戳来自不同的主机，我们必须考虑到时间偏差。在我们的分析工具，我们利用了这个事实：RPC 客户端发送一个请求之后，服务器端才能接收到，对于响应也是一样的（服务器先响应，然后客户端才能接收到这个响应）。这样一来，服务器端的 RPC 就有一个时间戳的一个上限和下限。

2.2 植入点

Dapper 可以以对应用开发者近乎零浸入的成本对分布式控制路径进行跟踪，几乎完全依赖于基于少量通用组件库的改造。如下：

- 当一个线程在处理跟踪控制路径的过程中，Dapper 把这次跟踪的上下文的在 ThreadLocal 中进行存储。追踪上下文是一个小而且容易复制的容器，其中承载了 Scan 的属性比如跟踪 ID 和 span ID。
- 当计算过程是延迟调用的或是异步的，大多数 Google 开发者通过线程池或其他执行器，使用一个通用的控制流库来回调。Dapper 确保所有这样的回调可以存储这次跟踪的上下文，而当回调函数被触发时，这次跟踪的上下文会与适当的线程关联上。在这种方式下，Dapper 可以使用 trace ID 和 span ID 来辅助构建异步调用的路径。
- 几乎所有的 Google 的进程间通信是建立在一个用 C++和 Java 开发的 RPC 框架上。我们把跟踪植入该框架来定义 RPC 中所有的 span。span 的 ID 和跟踪的 ID 会从客户端发送到服务端。像那样的基于 RPC 的系统被广泛使用在 Google 中，这是一个重要的植入点。当那些非 RPC 通信框架发展成熟并找到了自己的用户群之后，我们会计划对 RPC 通信框架进行植入。

Dapper 的跟踪数据是独立于语言的，很多在生产环境中的跟踪结合了用 C++和 Java 写的进程的数据。在 3.2 节中，我们讨论应用程序的透明度时我们会把这些理论的是如何实践的进行讨论。

2.3 Annotation

```
// C++:
const string& request = ...;
if (HitCache())
  TRACEPRINTF("cache hit for %s", request.c_str());
else
  TRACEPRINTF("cache miss for %s", request.c_str());

// Java:
Tracer t = Tracer.getCurrentTracer();
String request = ...;
if (hitCache())
  t.record("cache hit for " + request);
else
  t.record("cache miss for " + request);
```

上述植入点足够推导出复杂的分布式系统的跟踪细节，使得 Dapper 的核心功能在不改动 Google 应用的情况下可用。然而，Dapper 还允许应用程序开发人员在 Dapper 跟踪的过程中添加额外的信息，以监控更高级别的系统行为，或帮助调试问题。我们允许用户通过一个简单的 API 定义带时间戳的 Annotation，核心的示例代码入图 4 所示。这些 Annotation 可以添加任意内容。为了保护 Dapper 的用户意外的过分热衷于日志的记录，每一个跟踪 span 有一个可配置的总 Annotation

量的上限。但是，应用程序级的 Annotation 是不能替代用于表示 span 结构的信息和记录着 RPC 相关的信息。

除了简单的文本 Annotation，Dapper 也支持的 key-value 映射的 Annotation，提供给开发人员更强的跟踪能力，如持续的计数器，二进制消息记录和在一个进程上跑着的任意的用户数据。键值对的 Annotation 方式用来在分布式追踪的上下文中定义某个特定应用程序的相关类型。

2.4 采样率

低损耗的是 Dapper 的一个关键的设计目标，因为如果这个工具价值未被证实但又对性能有影响的话，你可以理解服务运营人员为什么不愿意部署它。况且，我们想让开发人员使用 Annotation 的 API，而不用担心额外的开销。我们还发现，某些类型的 Web 服务对植入带来的性能损耗确实非常敏感。因此，除了把 Dapper 的收集工作对基本组件的性能损耗限制的尽可能小之外，我们还有进一步控制损耗的办法，那就是遇到大量请求时只记录其中的一小部分。我们将在 4.4 节中讨论跟踪的采样率方案的更多细节。

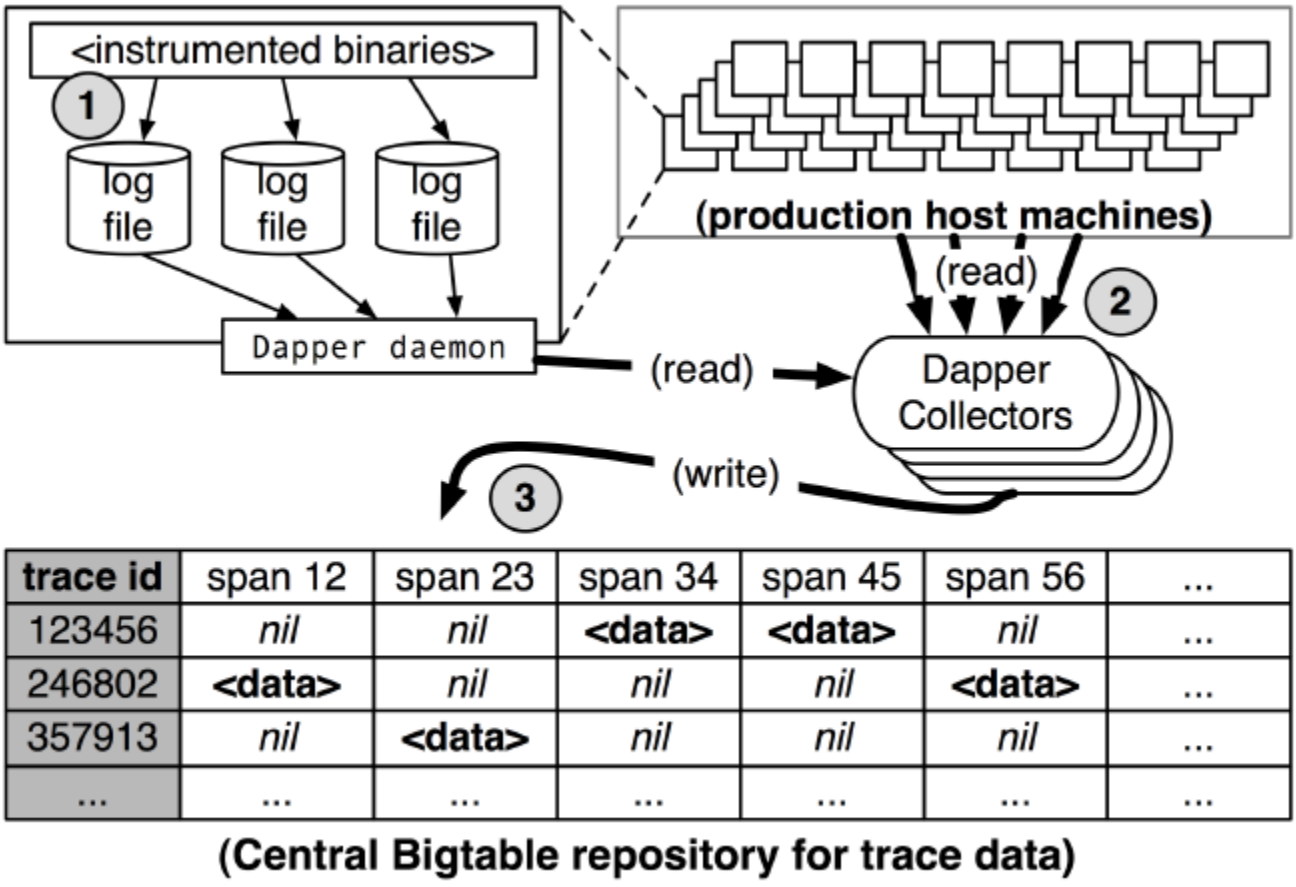

图 5：Dapper 收集管道的总览

2.5 跟踪的收集

Dapper 的跟踪记录和收集管道的过程分为三个阶段（参见图 5）。首先，span 数据写入（1）本地日志文件中。然后 Dapper 的守护进程和收集组件把这些数据从生产环境的主机中拉出来（2），最终写到（3）Dapper 的 Bigtable 仓库中。一次跟踪被设计成 Bigtable 中的一行，每一列相当于一个 span。Bigtable 的支持稀疏表格布局正适合这种情况，因为每一次跟踪可以有任意多个 span。跟踪数据收集（即从应用中的二进制数据传输到中央仓库所花费的时间）的延迟中位数少于 15 秒。第 98 百分位的延迟(The 98th percentile latency)往往随着时间的推移呈现双峰型;大约 75%的时间，第 98 百分位的延迟时间小于 2 分钟，但是另外大约 25%的时间，它可以增涨到几个小时。

Dapper 还提供了一个 API 来简化访问我们仓库中的跟踪数据。 Google 的开发人员用这个 API，以构建通用和特定应用程序的分析工具。第 5.1 节包含更多如何使用它的信息。

2.5.1 带外数据跟踪收集

tip1:带外数据:传输层协议使用带外数据(out-of-band，OOB)来发送一些重要的数据,如果通信一方有重要的数据需要通知对方时,协议能够将这些数据快速地发送到对方。为了发送这些数据，协议一般不使用与普通数据相同的通道,而是使用另外的通道。

tip2:这里指的 in-band 策略是把跟踪数据随着调用链进行传送，out-of-band 是通过其他的链路进行跟踪数据的收集，Dapper 的写日志然后进行日志采集的方式就属于 out-of-band 策略

Dapper 系统请求树树自身进行跟踪记录和收集带外数据。这样做是为两个不相关的原因。首先，带内收集方案--这里跟踪数据会以 RPC 响应头的形式被返回--会影响应用程序网络动态。在 Google 里的许多规模较大的系统中，一次跟踪成千上万的 span 并不少见。然而，RPC 回应大小--甚至是接近大型分布式的跟踪的根节点的这种情况下-- 仍然是比较小的：通常小于 10K。在这种情况下，带内 Dapper 的跟踪数据会让应用程序数据和倾向于使用后续分析结果的数据量相形见绌。其次，带内收集方案假定所有的 RPC 是完美嵌套的。我们发现，在所有的后端的系统返回的最终结果之前，有许多中间件会把结果返回给他们的调用者。带内收集系统是无法解释这种非嵌套的分布式执行模式的。

2.6 安全和隐私考虑

记录一定量的 RPC 有效负载信息将丰富 Dapper 的跟踪能力，因为分析工具能够在有效载荷数据（方法传递的参数）中找到相关的样例，这些样例可以解释被监控系统的为何表现异常。然而，有些情况下，有效载荷数据可能包含的一些不应该透露给未经授权用户(包括正在 debug 的工程师)的内部信息。

由于安全和隐私问题是不可忽略的，dapper 中的虽然存储 RPC 方法的名称，但在这个时候不记录任何有效载荷数据。相反，应用程序级别的 Annotation 提供了一个方便的可选机制：应用程序开发人员可以在 span 中选择关联那些为以后分析提供价值的数据。

Dapper 还提供了一些安全上的便利，是它的设计者事先没有预料到的。通过跟踪公开的安全协议参数，Dapper 可以通过相应级别的认证或加密，来监视应用程序是否满足安全策略。例如。Dapper 还可以提供信息，以基于策略的的隔离系统按预期执行，例如支撑敏感数据的应用程序不与未经授权的系统组件进行了交互。这样的测算提供了比源码审核更强大的保障。

3. Dapper 部署状况

Dapper 作为我们生产环境下的跟踪系统已经超过两年。在本节中，我们会汇报系统状态，把重点放在 Dapper 如何满足了我们的目标——无处不在的部署和应用级的透明。

3.1 Dapper 运行库

也许 Dapper 代码中中最关键的部分，就是对基础 RPC、线程控制和流程控制的组件库的植入，其中包括 span 的创建，采样率的设置，以及把日志写入本地磁盘。除了做到轻量级，植入的代码更需要稳定和健壮，因为它与海量的应用对接，维护和 bug 修复变得困难。植入的核心代码是由未超过 1000 行的 C++和不超过 800 行 Java 代码组成。为了支持键值对的 Annotation 还添加了额外的 500 行代码。

3.2 生产环境下的涵盖面

Dapper 的渗透可以总结为两个方面：一方面是可以创建 Dapper 跟踪的过程(与 Dapper 植入的组件库相关)，和生产环境下的服务器上在运行 Dapper 跟踪收集守护进程。Dapper 的守护进程的分布相当于我们服务器的简单的拓扑图，它存在于 Google 几乎所有的服务器上。这很难确定精确的 Dapper-ready 进程部分，因为过程即便不产生跟踪信息 Dapper 也是无从知晓的。尽管如此，考虑到无处不在 Dapper 组件的植入库，我们估计几乎每一个 Google 的生产进程都是支持跟踪的。

在某些情况下 Dapper 的是不能正确的跟踪控制路径的。这些通常源于使用非标准的控制流，或是 Dapper 的错误的把路径关联归到不相关的事件上。Dapper 提供了一个简单的库来帮助开发者手动控制跟踪传播作为一种变通方法。目前有 40 个 C++应用程序和 33 个 Java 应用程序需要一些手动控制的追踪传播，不过这只是上千个的跟踪中的一小部分。也有非常小的一部分程序使用的非组件性质的通信库（比如原生的 TCP Socket 或 SOAP RPC），因此不能直接支持 Dapper 的跟踪。但是这些应用可以单独接入到 Dapper 中，如果需要的话。

考虑到生产环境的安全，Dapper 的跟踪也可以关闭。事实上，它在部署的早起就是默认关闭的，直到我们对 Dapper 的稳定性和低损耗有了足够的信心之后才把它开启。Dapper 的团队偶尔会执行审查寻找跟踪配置的变化，来看看那些服务关闭了 Dapper 的跟踪。但这种情况不多见，而且通常是源于对监控对性能消耗的担忧。经过了对实际性能消耗的进一步调查和测量，所有这些关闭 Dapper 跟踪都已经恢复开启了，不过这些已经不重要了。

3.3 跟踪 Annotation 的使用

程序员倾向于使用特定应用程序的 Annotation，无论是作为一种分布式调试日志文件，还是通过一些应用程序特定的功能对跟踪进行分类。例如，所有的 Bigtable 的请求会把被访问的表名也记录到 Annotation 中。目前，70%的 Dapper span 和 90%的所有 Dapper 跟踪都至少有一个特殊应用的 Annotation。

41 个 Java 应用和 68 个 C++应用中都添加自定义的 Annotation 为了更好地理解应用程序中的 span 在他们的服务中的行为。值得注意的是，迄今为止我们的 Java 开发者比 C++开发者更多的在每一个跟踪 span 上采用 Annotation 的 API。这可能是因为我们的 Java 应用的作用域往往是更接近最终用户(C++偏底层);这些类型的应用程序经常处理更广泛的请求组合，因此具有比较复杂的控制路径。

4. 处理跟踪损耗

跟踪系统的成本由两部分组成：1.正在被监控的系统在生成追踪和收集追踪数据的消耗导致系统性能下降，2。需要使用一部分资源来存储和分析跟踪数据。虽然你可以说一个有价值的组件植入跟踪带来一部分性能损耗是值得的，我们相信如果基本损耗能达到可以忽略的程度，那么对跟踪系统最初的推广会有极大的帮助。

在本节中，我们会展现一下三个方面：Dapper 组件操作的消耗，跟踪收集的消耗，以及 Dapper 对生产环境负载的影响。我们还介绍了 Dapper 可调节的采样率机制如何帮我们处理低损耗和跟踪代表性之间的平衡和取舍。

4.1 生成跟踪的损耗

生成跟踪的开销是 Dapper 性能影响中最关键的部分，因为收集和分析可以更容易在紧急情况下被关闭。Dapper 运行库中最重要的跟踪生成消耗在于创建和销毁 span 和 annotation，并记录到本地磁盘供后续的收集。根 span 的创建和销毁需要损耗平均 204 纳秒的时间，而同样的操作在其他 span 上需要消耗 176 纳秒。时间上的差别主要在于需要在跟 span 上给这次跟踪分配一个全局唯一的 ID。

如果一个 span 没有被采样的话，那么这个额外的 span 下创建 annotation 的成本几乎可以忽略不计，他由在 Dapper 运行期对 ThreadLocal 查找操作构成，这平均只消耗 9 纳秒。如果这个 span 被计入采样的话，会用一个用字符串进行标注--在

图 4 中有展现--平均需要消耗 40 纳秒。这些数据都是在 2.2GHz 的 x86 服务器上采集的。

在 Dapper 运行期写入到本地磁盘是最昂贵的操作，但是他们的可见损耗大大减少，因为写入日志文件和操作相对于被跟踪的应用系统来说都是异步的。不过，日志写入的操作如果在大流量的情况，尤其是每一个请求都被跟踪的情况下就会变得可以察觉到。我们记录了在 4.3 节展示了一次 Web 搜索的负载下的性能消耗。

4.2 跟踪收集的消耗

读出跟踪数据也会对正在被监控的负载产生干扰。表 1 展示的是最坏情况下，Dapper 收集日志的守护进程在高于实际情况的负载基准下进行测试时的 cpu 使用率。在生产环境下，跟踪数据处理中，这个守护进程从来没有超过 0.3%的单核 cpu 使用率，而且只有很少量的内存使用（以及堆碎片的噪音）。我们还限制了 Dapper 守护进程为内核 scheduler 最低的优先级，以防在一台高负载的服务器上发生 cpu 竞争。

Dapper 也是一个带宽资源的轻量级的消费者，每一个 span 在我们的仓库中传输只占用了平均 426 的 byte。作为网络行为中的极小部分，Dapper 的数据收集在 Google 的生产环境中的只占用了 0.01%的网络资源。

Process Count (per host)	Data Rate (per process)	Daemon CPU Us (single CPU co
25	10K/sec	0.12
10	200K/sec	0.26
50	2K/sec	0.13

表 1：Dapper 守护进程在负载测试时的 CPU 资源使用率

4.3 在生产环境下对负载的影响

每个请求都会利用到大量的服务器的高吞吐量的线上服务，这是对有效跟踪最主要的需求之一；这种情况需要生成大量的跟踪数据，并且他们对性能的影响是最敏感的。在表 2 中我们用集群下的网络搜索服务作为例子，我们通过调整采样率，来衡量 Dapper 在延迟和吞吐量方面对性能的影响。

Sampling frequency	Avg. Latency (% change)	Avg. Throughput (% change)
1/1	16.3%	−1.48%
1/2	9.40%	−0.73%
1/4	6.38%	−0.30%
1/8	4.12%	−0.23%
1/16	2.12%	−0.08%
1/1024	−0.20%	−0.06%

表 2：网络搜索集群中，对不同采样率对网络延迟和吞吐的影响。延迟和吞吐的实验误差分别是 2.5%和 0.15%。

我们看到，虽然对吞吐量的影响不是很明显，但为了避免明显的延迟，跟踪的采样还是必要的。然而，延迟和吞吐量的带来的损失在把采样率调整到小于 1/16 之后就全部在实验误差范围内。在实践中，我们发现即便采样率调整到 1/1024 仍然是有足够量的跟踪数据的用来跟踪大量的服务。保持 Dapper 的性能损耗基线在一个非常低的水平是很重要的，因为它为那些应用提供了一个宽松的环境使用完整的 Annotation API 而无惧性能损失。使用较低的采样率还有额外的好处，可以让持久化到硬盘中的跟踪数据在垃圾回收机制处理之前保留更长的时间，这样为 Dapper 的收集组件给了更多的灵活性。

4.4 可变采样

任何给定进程的 Dapper 的消耗和每个进程单位时间的跟踪的采样率成正比。Dapper 的第一个生产版本在 Google 内部的所有进程上使用统一的采样率，为 1/1024。这个简单的方案是对我们的高吞吐量的线上服务来说是非常有用，因为那些感兴趣的事件(在大吞吐量的情况下)仍然很有可能经常出现，并且通常足以被捕捉到。

然而，在较低的采样率和较低的传输负载下可能会导致错过重要事件，而想用较高的采样率就需要能接受的性能损耗。对于这样的系统的解决方案就是覆盖默认的采样率，这需要手动干预的，这种情况是我们试图避免在 dapper 中出现的。

我们在部署可变采样的过程中，参数化配置采样率时，不是使用一个统一的采样方案，而是使用一个采样期望率来标识单位时间内采样的追踪。这样一来，低流量低

负载自动提高采样率，而在高流量高负载的情况下会降低采样率，使损耗一直保持在控制之下。实际使用的采样率会随着跟踪本身记录下来，这有利于从 Dapper 的跟踪数据中准确的分析。

4.5 应对积极采样(Coping with aggressive sampling)

新的 Dapper 用户往往觉得低采样率--在高吞吐量的服务下经常低至 0.01％--将会不利于他们的分析。我们在 Google 的经验使我们相信，对于高吞吐量服务，积极采样(aggressive sampling)并不妨碍最重要的分析。如果一个显着的操作在系统中出现一次，他就会出现上千次。低吞吐量的服务--也许是每秒请求几十次，而不是几十万--可以负担得起跟踪每一个请求，这是促使我们下决心使用自适应采样率的原因。

4.6 在收集过程中额外的采样

上述采样机制被设计为尽量减少与 Dapper 运行库协作的应用程序中明显的性能损耗。Dapper 的团队还需要控制写入中央资料库的数据的总规模，因此为达到这个目的，我们结合了二级采样。

目前我们的生产集群每天产生超过 1TB 的采样跟踪数据。Dapper 的用户希望生产环境下的进程的跟踪数据从被记录之后能保存至少两周的时间。逐渐增长的追踪数据的密度必须和 Dapper 中央仓库所消耗的服务器及硬盘存储进行权衡。对请求的高采样率还使得 Dapper 收集器接近写入吞吐量的上限。

为了维持物质资源的需求和渐增的 Bigtable 的吞吐之间的灵活性，我们在收集系统自身上增加了额外的采样率的支持。我们充分利用所有 span 都来自一个特定的跟踪并分享同一个跟踪 ID 这个事实，虽然这些 span 有可能横跨了数千个主机。对于在收集系统中的每一个 span，我们用 hash 算法把跟踪 ID 转成一个标量 Z，这里 0<=Z<=1。如果 Z 比我们收集系统中的系数低的话，我们就保留这个 span 信息，并写入到 Bigtable 中。反之，我们就抛弃他。通过在采样决策中的跟踪 ID，我们要么保存、要么抛弃整个跟踪，而不是单独处理跟踪内的 span。我们发现，有了这个额外的配置参数使管理我们的收集管道变得简单多了，因为我们可以很容易地在配置文件中调整我们的全局写入率这个参数。

如果整个跟踪过程和收集系统只使用一个采样率参数确实会简单一些，但是这就不能应对快速调整在所有部署的节点上的运行期采样率配置的这个要求。我们选择了运行期采样率，这样就可以优雅的去掉我们无法写入到仓库中的多余数据，我们还可以通过调节收集系统中的二级采样率系数来调整这个运行期采样率。Dapper 的管道维护变得更容易，因为我们就可以通过修改我们的二级采样率的配置，直接增加或减少我们的全局覆盖率和写入速度。

5. 通用的 Dapper 工具

几年前，当 Dapper 还只是个原型的时候，它只能在 Dapper 开发者耐心的支持下使用。从那时起，我们逐渐迭代的建立了收集组件，编程接口，和基于 Web 的交互式用户界面，帮助 Dapper 的用户独立解决自己的问题。在本节中，我们会总结一下哪些的方法有用，哪些用处不大，我们还提供关于这些通用的分析工具的基本的使用信息。

5.1 Dapper Depot API

Dapper 的“Depot API”或称作 DAPI，提供在 Dapper 的区域仓库中对分布式跟踪数据一个直接访问。DAPI 和 Dapper 跟踪仓库被设计成串联的，而且 DAPI 意味着对 Dapper 仓库中的元数据暴露一个干净和直观的的接口。我们使用了以下推荐的三种方式去暴露这样的接口：

- 通过跟踪 ID 来访问：DAPI 可以通过他的全局唯一的跟踪 ID 读取任何一次跟踪信息。
- 批量访问：DAPI 可以利用的 MapReduce 提供对上亿条 Dapper 跟踪数据的并行读取。用户重写一个虚拟函数，它接受一个 Dapper 的跟踪信息作为其唯一的参数，该框架将在用户指定的时间窗口中调用每一次收集到的跟踪信息。
- 索引访问：Dapper 的仓库支持一个符合我们通用调用模板的唯一索引。该索引根据通用请求跟踪特性(commonly-requested trace features)进行绘制来识别 Dapper 的跟踪信息。因为跟踪 ID 是根据伪随机的规则创建的，这是最好的办法去访问跟某个服务或主机相关的跟踪数据。

所有这三种访问模式把用户指向不同的 Dapper 跟踪记录。正如第 2.1 节所述的，Dapper 的由 span 组成的跟踪数据是用树形结构建模的，因此，跟踪数据的数据结构，也是一个简单的由 span 组成遍历树。Span 就相当于 RPC 调用，在这种情况下，RPC 的时间信息是可用的。带时间戳的特殊的应用标注也是可以通过这个 span 结构来访问的。

选择一个合适的自定义索引是 DAPI 设计中最具挑战性的部分。压缩存储要求在跟踪数据种建立一个索引的情况只比实际数据小 26%，所以消耗是巨大的。最初，我们部署了两个索引：第一个是主机索引，另一个是服务名的索引。然而，我们并没有找到主机索引和存储成本之间的利害关系。当用户对每一台主机感兴趣的时候，他们也会对特定的服务感兴趣，所以我们最终选择把两者相结合，成为一个组合索引，它允许以服务名称，主机，和时间戳的顺序进行有效的查找。

5.1.1 DAPI 在 Google 内部的使用

DAPI 在谷歌的使用有三类：使利用 DAPI 的持续的线上 Web 应用，维护良好的可以在控制台上调用的基于 DAPI 的工具，可以被写入，运行、不过大部分已经被忘记了的一次性分析工具。我们知道的有 3 个持久性的基于 DAPI 的应用程序，8 个额外的按需定制的基于 DAPI 分析工具，以及使用 DAPI 框架构建的约 15~20 一次性的分析工具。在这之后的工具就这是很难说明了，因为开发者可以构建、运行和丢弃这些项目，而不需要 Dapper 团队的技术支持。

5.2 Dapper 的用户接口

绝大多数用户使用发生在基于 web 的用户交互接口。篇幅有限，我们不能列出每一个特点，而只能把典型的用户工作流在图 6 中展示。

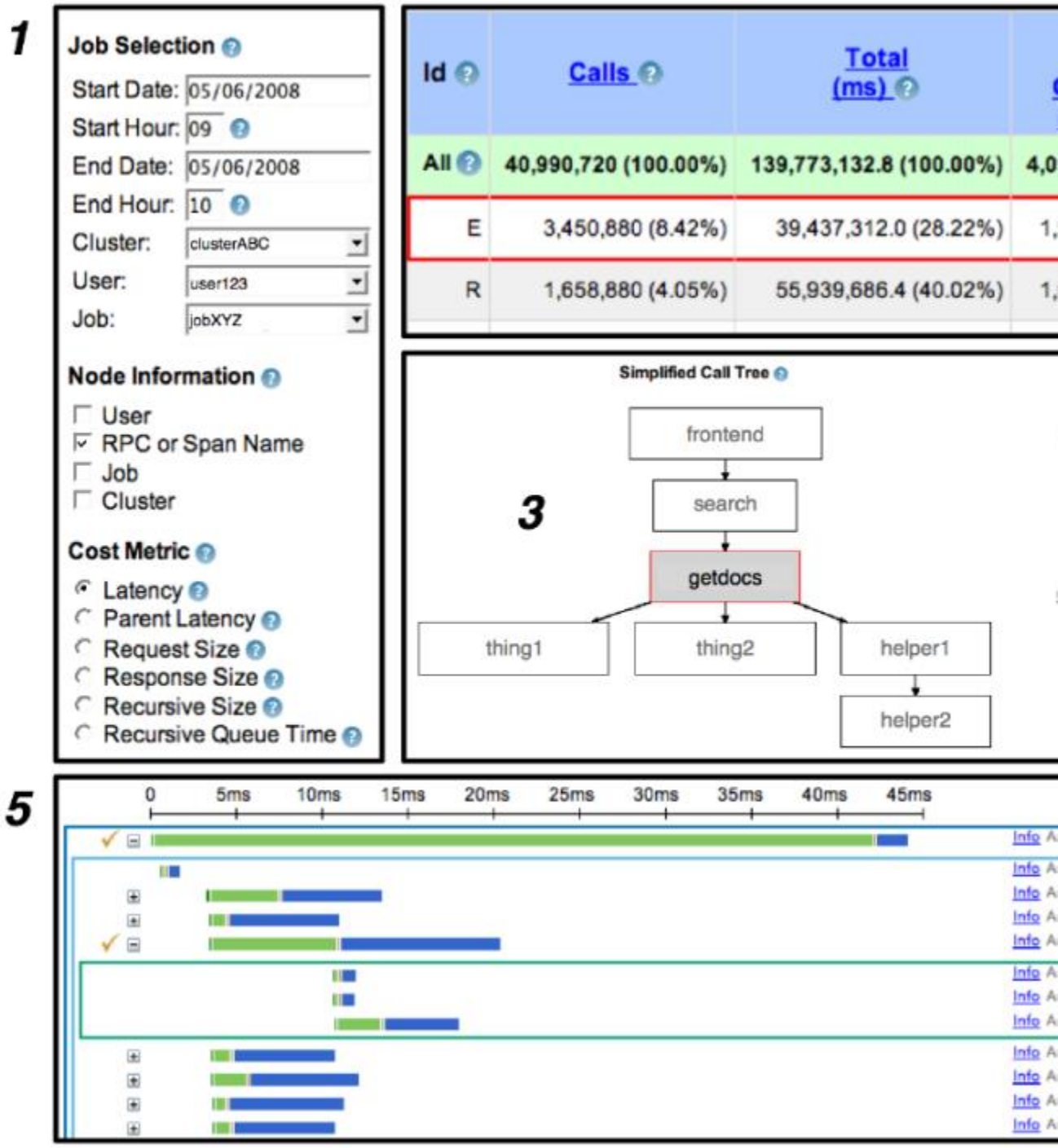

图 6

1. 用户描述的他们关心的服务和时间，和其他任何他们可以用来区分跟踪模板的信息（比如，span 的名称）。他们还可以指定与他们的搜索最相关的成本度量(cost metric)(比如，服务响应时间)。
2. 一个关于性能概要的大表格，对应确定的服务关联的所有分布式处理图表。用户可以把这些执行图标排序成他们想要的，并选择一种直方图去展现出更多的细节。
3. 一旦某个单一的分布式执行部分被选中后，用户能看到关于执行部分的的图形化描述。被选中的服务被高亮展示在该图的中心。
4. 在生成与步骤 1 中选中的成本度量(cost metric)维度相关的统计信息之后，Dapper 的用户界面会提供了一个简单的直方图。在这个例子中，我们可以看到一个大致的所选中部分的分布式响应时间分布图。用户还会看到一个关于具体的跟踪信息的列表，展现跟踪信息在直方图中被划分为的不同区域。在这个例子中，用户点击列表种第二个跟踪信息实例时，会在下方看到这个跟踪信息的详细视图(步骤 5)。
5. 绝大多数 Dapper 的使用者最终的会检查某个跟踪的情况，希望能收集一些信息去了解系统行为的根源所在。我们没有足够的空间来做跟踪视图的审查，但我们使用由一个全局时间轴（在上方可以看到），并能够展开和折叠树形结构的交互方式，这也很有特点。分布式跟踪树的连续层用内嵌的不同颜色的矩形表示。每一个 RPC 的 span 被从时间上分解为一个服务器进程中的消耗（绿色部分）和在网络上的消耗（蓝色部分）。用户 Annotation 没有显示在这个截图中，但他们可以选择性的以 span 的形式包含在全局时间轴上。

为了让用户查询实时数据，Dapper 的用户界面能够直接与 Dapper 每一台生产环境下的服务器上的守护进程进行交互。在该模式下，不可能指望能看到上面所说的系统级的图表展示，但仍然可以很容易基于性能和网络特性选取一个特定的跟踪。在这种模式下，可在几秒钟内查到实时的数据。

根据我们的记录，大约有 200 个不同的 Google 工程师在一天内使用的 Dapper 的 UI;在一周的过程中，大约有 750-1000 不同的用户。这些用户数，在新功能的内部通告上，是按月连续的。通常用户会发送特定跟踪的连接，这将不可避免地在查询跟踪情况时中产生很多一次性的，持续时间较短的交互。

6. 经验

Dapper 在 Google 被广泛应用，一部分直接通过 Dapper 的用户界面，另一部分间接地通过对 Dapper API 的二次开发或者建立在基于 api 的应用上。在本节中，我们并不打算罗列出每一种已知的 Dapper 使用方式，而是试图覆盖 Dapper 使用方式的“基本向量”，并努力来说明什么样的应用是最成功的。

6.1 在开发中使用 Dapper

Google AdWords 系统是围绕一个大型的关键词定位准则和相关文字广告的数据库搭建的。当新的关键字或广告被插入或修改时，它们必须通过服务策略术语的检查（如检查不恰当的语言，这个过程如果使用自动复查系统来做的话会更加有效）。

当轮到从头重新设计一个广告审查服务时，这个团队迭代的从第一个系统原型开始使用 Dapper，并且，最终用 Dapper 一直维护着他们的系统。Dapper 帮助他们从以下几个方面改进了他们的服务：

- 性能：开发人员针对请求延迟的目标进行跟踪，并对容易优化的地方进行定位。Dapper 也被用来确定在关键路径上不必要的串行请求--通常来源于不是开发者自己开发的子系统--并促使团队持续修复他们。
- 正确性：广告审查服务围绕大型数据库系统搭建。系统同时具有只读副本策略（数据访问廉价）和读写的主策略（访问代价高）。Dapper 被用来在很多种情况中确定，哪些查询是无需通过主策略访问而可以采用副本策略访问。Dapper 现在可以负责监控哪些主策略被直接访问，并对重要的系统常量进行保障。
- 理解性：广告审查查询跨越了各种类型的系统，包括 BigTable—之前提到的那个数据库，多维索引服务，以及其他各种 C++和 Java 后端服务。Dapper 的跟踪用来评估总查询成本，促进重新对业务的设计，用以在他们的系统依赖上减少负载。
- 测试：新的代码版本会经过一个使用 Dapper 进行跟踪的 QA 过程，用来验证正确的系统行为和性能。在跑测试的过程中能发现很多问题，这些问题来自广告审查系统自身的代码或是他的依赖包。

广告审查团队广泛使用了 Dapper Annotation API。Guice[13]开源的 AOP 框架用来在重要的软件组件上标注“@Traced”。这些跟踪信息可以进一步被标注，包含：重要子路径的输入输出大小、基础信息、其他调试信息，所有这些信息将会额外发送到日志文件中。

同时，我们也发现了一些广告审查小组在使用方面的不足。比如：他们想根据他们所有跟踪的 Annotation 信息，在一个交互时间段内进行搜索，然而这就必须跑一个自定义的 MapReduce 或进行每一个跟踪的手动检查。另外，在 Google 还有一些其他的系统在也从通用调试日志中收集和集中信息，把那些系统的海量数据和 Dapper 仓库整合也是有价值的。

总的来说，即便如此，广告审查团队仍然对 Dapper 的作用进行了以下评估，通过使用 Dapper 的跟踪平台的数据分析，他们的服务延迟性已经优化了两个数量级。

6.1.1 与异常监控的集成

Google 维护了一个从运行进程中不断收集并集中异常信息报告的服务。如果这些异常发生在 Dapper 跟踪采样的上下文中，那么相应的跟踪 ID 和 span 的 ID 也会作为元数据记录在异常报告中。异常监测服务的前端会提供一个链接，从特定的异常信息的报告直接导向到他们各自的分布式跟踪。广告审查团队使用这个功能可以了解 bug 发生的更大范围的上下文。通过暴露基于简单的唯一 ID 构建的接口，Dapper 平台被集成到其他事件监测系统会相对容易。

6.2 解决延迟的长尾效应

考虑到移动部件的数量、代码库的规模、部署的范围，调试一个像全文搜索那样服务（第 1 节里提到过）是非常具有挑战性的。在这节，我们描述了我们在减轻全文

搜索的延迟分布的长尾效应上做的各种努力。Dapper 能够验证端到端的延迟的假设，更具体地说，Dapper 能够验证对于搜索请求的关键路径。当一个系统不仅涉及数个子系统，而是几十个开发团队的涉及到的系统的情况下，端到端性能较差的根本原因到底在哪，这个问题即使是我们最好的和最有经验的工程师也无法正确回答。在这种情况下，Dapper 可以提供急需的数据，而且可以对许多重要的性能问题得出结论。

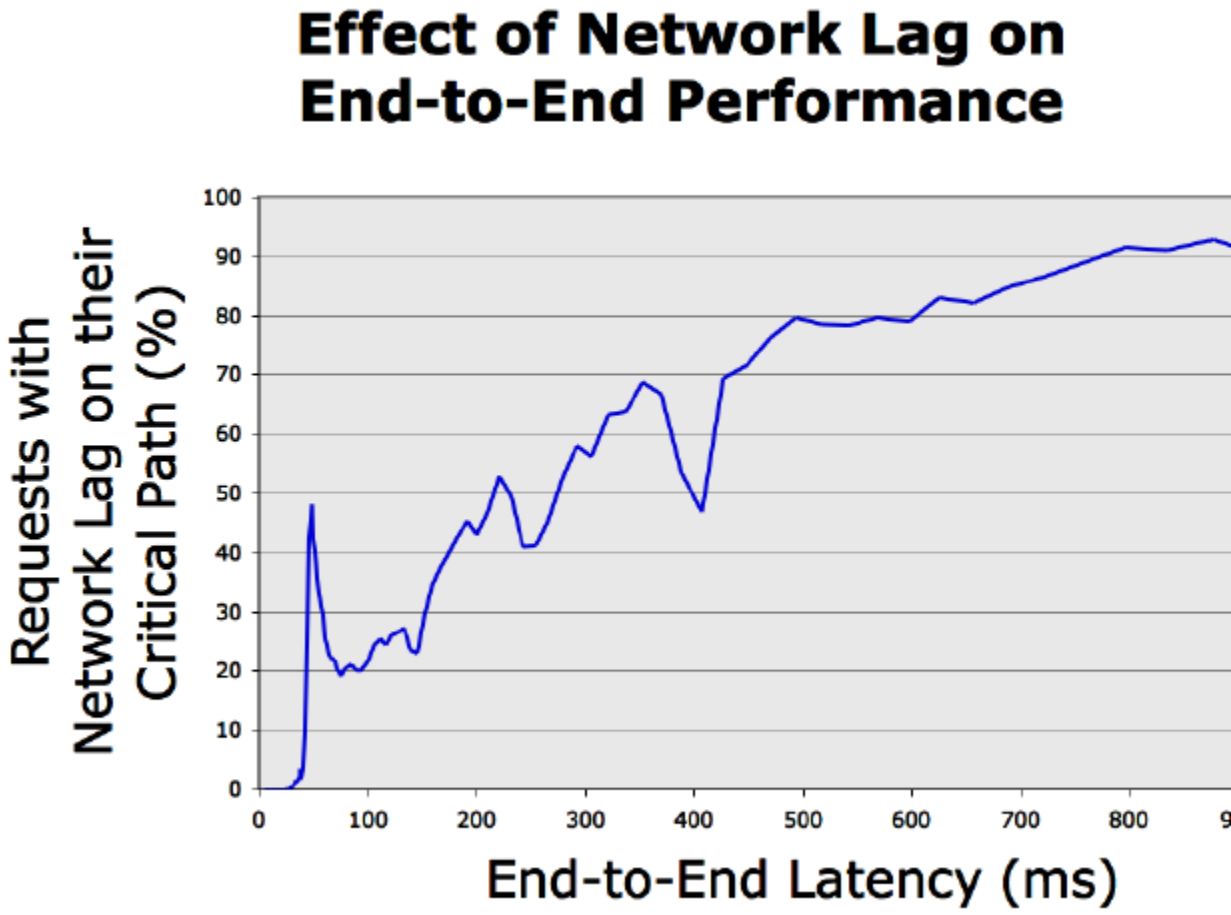

图 7：全局搜索的跟踪片段，在不常遇到高网络延迟的情况下，在沿着关键路径的端到端的请求延迟，如图所示。

在调试延迟长尾效应的过程中，工程师可以建立一个小型库，这个小型库可以根据 DAPI 跟踪对象来推断关键路径的层级结构。这些关键路径的结构可以被用来诊断问题，并且为全文搜索提供可优先处理的预期的性能改进。Dapper 的这项工作导致了下列发现：

- 在关键路径上的短暂的网络性能退化不影响系统的吞吐量，但它可能会对延迟异常值产生极大的影响。在图 7 中可以看出，大部分的全局搜索的缓慢的跟踪都来源于关键路径的网络性能退化。

- 许多问题和代价很高的查询模式来源于一些意想不到的服务之间的交互。一旦发现，往往容易纠正它们，但是 Dapper 出现之前想找出这些问题是相当困难的。
- 通用的查询从 Dapper 之外的安全日志仓库中收取，并使用 Dapper 唯一的跟踪 ID，与 Dapper 的仓库做关联。然后，该映射用来建立关于在全局搜索中的每一个独立子系统都很慢的实例查询的列表。

6.3 推断服务依赖

在任何给定的时间内，Google 内部的一个典型的计算集群是一个汇集了成千上万个逻辑“任务”的主机，一套的处理器在执行一个通用的方法。Google 维护着许多这样的集群，当然，事实上，我们发现在一个集群上计算着的这些任务通常依赖于其他的集群上的任务。由于任务们之间的依赖是动态改变的，所以不可能仅从配置信息上推断出所有这些服务之间的依赖关系。不过，除了其他方面的原因之外，在公司内部的各个流程需要准确的服务依赖关系信息，以确定瓶颈所在，以及计划服务的迁移。Google 的可称为“Service Dependencies”的项目是通过使用跟踪 Annotation 和 DAPI MapReduce 接口来实现自动化确定服务依赖归属的。

Dapper 核心组件与 Dapper 跟踪 Annotation 一并使用的情况下，“Service Dependencies”项目能够推算出任务各自之间的依赖，以及任务和其他软件组件之间的依赖。比如，所有的 BigTable 的操作会加上与受影响的表名称相关的标记。运用 Dapper 的平台，Service Dependencies 团队就可以自动的推算出依赖于命名的不同资源的服务粒度。

6.4 不同服务的网络使用率

Google 投入了大量的人力和物力资源在他的网络结构上。从前网络管理员可能只关注独立的硬件信息、常用工具及以及搭建出的各种全局网络鸟瞰图的 dashboard 上的信息。网络管理员确实可以一览整个网络的健康状况，但是，当遇到问题时，他们很少有能够准确查找网络负载的工具，用来定位应用程序级别的罪魁祸首。

虽然 Dapper 不是设计用来做链路级的监控的，但是我们发现，它是非常适合去做集群之间网络活动性的应用级任务的分析。Google 能够利用 Dapper 这个平台，建立一个不断更新的控制台，来显示集群之间最活跃的网络流量的应用级的热点。此外，使用 Dapper 我们能够为昂贵的网络请求提供指出的构成原因的跟踪，而不是面对不同服务器之间的信息孤岛而无所适从。建立一个基于 Dapper API 的 dashboard 总共没花超过 2 周的时间。

6.5 分层和共享存储系统

在 Google 的许多存储系统是由多重独立复杂层级的分布式基础设备组成的。例如，Google 的 App Engine[5]就是搭建在一个可扩展的实体存储系统上的。该实体存储系统在基于 BigTable 上公开某些 RDBMS 功能。 BigTable 的同时使用 Chubby[7]（分布式锁系统）及 GFS。再者，像 BigTable 这样的系统简化了部署，并更好的利用了计算资源。

在这种分层的系统，并不总是很容易确定最终用户资源的消费模式。例如，来自于一个给定的 BigTable 单元格的 GFS 大信息量主要来自于一个用户或是由多个用户产生，但是在 GFS 层面，这两种明显的使用场景是很难界定。而且，如果缺乏一个像 Dapper 一样的工具的情况下，对共享服务的竞争可能会同样难于调试。

第 5.2 节中所示的 Dapper 的用户界面可以聚合那些调用任意公共服务的多个客户端的跟踪的性能信息。这就很容易让提供这些服务的源从多个维度给他们的用户排名。（例如，入站的网络负载，出站的网络负载，或服务请求的总时间）

6.6 Dapper 的救火能力(Firefighting)

对于一些“救火”任务，Dapper 可以处理其中的一部分。“救火”任务在这里是指一些有风险很高的在分布式系统上的操作。通常情况下，Dapper 用户当正在进行“救火”任务时需要使用新的数据，并且没有时间写新的 DAPI 代码或等待周期性的报告运行。

对于那些高延迟，不，可能更糟糕的那些在正常负载下都会响应超时的服务，Dapper 用户界面通常会把这些延迟瓶颈的位置隔离出来。通过与 Dapper 守护进程的直接通信，那些特定的高延迟的跟踪数据轻易的收集到。当出现灾难性故障时，通常是没有必要去看统计数据以确定根本原因，只查看示例跟踪就足够了(因为前文提到过从 Dapper 守护进程中几乎可以立即获得跟踪数据)。

但是，如在 6.5 节中描述的共享的存储服务，要求当用户活动过程中突然中断时能尽可能快的汇总信息。对于事件发生之后，共享服务仍然可以利用汇总的的 Dapper 数据，但是，除非收集到的 Dapper 数据的批量分析能在问题出现 10 分钟之内完成，否则 Dapper 面对与共享存储服务相关的“救火”任务就很难按预想的那般顺利完成。

7. 其他收获

虽然迄今为止，我们在 Dapper 上的经验已经大致符合我们的预期，但是也出现了一些积极的方面是我们没有充分预料到的。首先，我们获得了超出预期的 Dapper 使用用例的数量，对此我们可谓欢心鼓舞。另外，在除了几个的在第 6 节使用经验中提到过的一些用例之外，还包括资源核算系统，对指定的通讯模式敏感的服务的检查工具，以及一种对 RPC 压缩策略的分析器，等等。我们认为这些意想不到的用例一定程度上是由于我们向开发者以一种简单的编程接口的方式开放了跟踪数据存储的缘故，这使得我们能够充分利用这个大的多的社区的创造力。除此之外，Dapper 对旧的负载的支持也比预期的要简单，只需要在程序中引入一个用新版本的重新编译过的公共组件库(包含常规的线程使用，控制流和 RPC 框架)即可。

Dapper 在 Google 内部的广泛使用还为我们在 Dapper 的局限性上提供了宝贵的反馈意见。下面我们将介绍一些我们已知的最重要的 Dapper 的不足：

- 合并的影响：我们的模型隐含的前提是不同的子系统在处理的都是来自同一个被跟踪的请求。在某些情况下，缓冲一部分请求，然后一次性操作一个请求集会更加有效。（比如，磁盘上的一次合并写入操作）。在这种情况下，一个被跟踪的请求可以看似是一个大型工作单元。此外，当有多个追踪请求被收集在一起，他们当中只有一个会用来生成那个唯一的跟踪 ID，用来给其他 span 使用，所以就无法跟踪下去了。我们正在考虑的解决方案，希望在可以识别这种情况的前提下，用尽可能少的记录来解决这个问题。
- 跟踪批处理负载：Dapper 的设计，主要是针对在线服务系统，最初的目标是了解一个用户请求产生的系统行为。然而，离线的密集型负载，例如符合 MapReduce[10]模型的情况，也可以受益于性能挖潜。在这种情况下，我们需要把跟踪 ID 与一些其他的有意义的工作单元做关联，诸如输入数据中的键值（或键值的范围），或是一个 MapReduce shard。
- 寻找根源：Dapper 可以有效地确定系统中的哪一部分致使系统整个速度变慢，但并不总是能够找出问题的根源。例如，一个请求很慢有可能不是因为它自己的行为，而是由于队列中其他排在它前面的(queued ahead of)请求还没处理完。程序可以使用应用级的 annotation 把队列的大小或过载情况写入跟踪系统。此外，如果这种情况屡见不鲜，那么在 ProfileMe[11]中提到的成对的采样技术可以解决这个问题。它由两个时间重叠的采样率组成，并观察它们在整个系统中的相对延迟。
- 记录内核级的信息：一些内核可见的事件的详细信息有时对确定问题根源是很有用的。我们有一些工具，能够跟踪或以其他方式描述内核的执行，但是，想用通用的或是不那么突兀的方式，是很难把这些信息到捆绑到用户级别的跟踪上下文中。我们正在研究一种妥协的解决方案，我们在用户层面上把一些内核级的活动参数做快照，然后绑定他们到一个活动的 span 上。

8. 相关产品

在分布式系统跟踪领域，有一套完整的体系，一部分系统主要关注定位到故障位置，其他的目标是针对性能进行优化。 Dapper 确实被用于发现系统问题，但它更通常用于探查性能不足，以及提高全面大规模的工作负载下的系统行为的理解。

与 Dapper 相关的黑盒监控系统，比如 Project5[1]，WAP5[15]和 Sherlock[2]，可以说不依赖运行库的情况下，黑盒监控系统能够实现更高的应用级透明。黑盒的缺点是一定程度上不够精确，并可能在统计推断关键路径时带来更大的系统损耗。

对于分布式系统监控来说，基于 Annotation 的中间件或应用自身是一个可能是更受欢迎的解决办法.拿 Pip[14]和 Webmon[16]系统举例，他们更依赖于应用级的 Annotation，而 X-Trace[12]，Pinpoint[9]和 Magpie[3]大多集中在对库和中间件的修改。Dapper 更接近后者。像 Pinpoint，X-Trace，和早期版本的 Magpie 一样，Dapper 采用了全局标识符把分布式系统中各部分相关的事件联系在一起。和这些系统类似，Dapper 尝试避免使用应用级 Annotation，而是把的植入隐藏在通用组件模块内。Magpie 放弃使用全局 ID，仍然试图正确的完成请求的正确传播，他通过采用应用系统各自写入的事件策略，最终也能精确描述不同事件之间关系。但是目前还不清楚 Magpie 在实际环境中实现透明性这些策略到底多么有效。 X-Trace 的核心 Annotation 比 Dapper 更有野心一些，因为 X-Trace 系统对于跟踪的收集，不仅在跟踪节点层面上，而且在节点内部不同的软件层也会进行跟踪。而我们

对于组件的低性能损耗的要求迫使我们不能采用 X-Trace 这样的模型，而是朝着把一个请求连接起来完整跟踪所能做到的最小代价而努力。而 Dapper 的跟踪仍然可以从可选的应用级 Annotation 中获益。

9. 总结

在本文中，我们介绍 Dapper 这个 Google 的生产环境下的分布式系统跟踪平台，并汇报了我们开发和使用它的相关经验。 Dapper 几乎在部署在所有的 Google 系统上，并可以在不需要应用级修改的情况下进行跟踪，而且没有明显的性能影响。Dapper 对于开发人员和运维团队带来的好处，可以从我们主要的跟踪用户界面的广泛使用上看出来，另外我们还列举了一些 Dapper 的使用用例来说明 Dapper 的作用，这些用例有些甚至都没有 Dapper 开发团队参与，而是被应用的开发者开发出来的。

据我们所知，这是第一篇汇报生产环境下分布式系统跟踪框架的论文。事实上，我们的主要贡献源于这个事实：论文中回顾的这个系统已经运行两年之久。我们发现，结合对开发人员提供简单 API 和对应用系统完全透明来增强跟踪的这个决定，是非常值得的。

我们相信，Dapper 比以前的基于 Annotation 的分布式跟踪达到更高的应用透明度，这一点已经通过只需要少量人工干预的工作量得以证明。虽然一定程度上得益于我们的系统的同质性，但它本身仍然是一个重大的挑战。最重要的是，我们的设计提出了一些实现应用级透明性的充分条件，对此我们希望能够对更错杂环境下的解决方案的开发有所帮助。

最后，通过开放 Dapper 跟踪仓库给内部开发者，我们促使更多的基于跟踪仓库的分析工具的产生，而仅仅由 Dapper 团队默默的在信息孤岛中埋头苦干的结果远达不到现在这么大的规模，这个决定促使了设计和实施的展开。

Acknowledgments

We thank Mahesh Palekar, Cliff Biffle, Thomas Kotzmann, Kevin Gibbs, Yonatan Zunger, Michael Kleber, and Toby Smith for their experimental data and feedback about Dapper experiences. We also thank Silvius Rus for his assistance with load testing. Most importantly, though, we thank the outstanding team of engineers who have continued to develop and improve Dapper over the years; in order of appearance, Sharon Perl, Dick Sites, Rob von Behren, Tony DeWitt, Don Pazel, Ofer Zajicek, Anthony Zana, Hyang-Ah Kim, Joshua MacDonald, Dan Sturman, Glenn Willen, Alex Kehlenbeck, Brian McBarron, Michael Kleber, Chris Povirk, Bradley White, Toby Smith, Todd Derr, Michael De Rosa, and Athicha Muthitacharoen. They have all done a tremendous amount of work to make Dapper a day-to-day reality at Google.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In Proceedings of the 19th ACM Symposium on Operating Systems Principles, December 2003.

[2] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies. In Proceedings of SIGCOMM, 2007.

[3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: online modelling and performance-aware systems. In Proceedings of USENIX HotOS IX, 2003.

[4] L. A. Barroso, J. Dean, and U. Holzle. Web Search for a Planet: The Google Cluster Architecture. IEEE Micro, 23(2):22–28, March/April 2003.

[5] T. O. G. Blog. Developers, start your engines. http://googleblog.blogspot.com/2008/04/developers-start-your-engines.html,2007.

[6] T. O. G. Blog. Universal search: The best answer is still the best answer. http://googleblog.blogspot.com/2007/05/universal-search-best-answer-is-still.html, 2007.

[7] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, pages 335 – 350, 2006.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), November 2006.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In Proceedings of ACM International Conference on Dependable Systems and Networks, 2002.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04), pages 137 – 150, December 2004.

[11] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In Proceedings of the IEEE/ACM International Symposium on Microarchitecture, 1997.

[12] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In Proceedings of USENIX NSDI, 2007.

[13] B. Lee and K. Bourrillion. The Guice Project Home Page. http://code.google.com/p/google-guice/, 2007.

[14] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In Proceedings of USENIX NSDI, 2006.

[15] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: Black Box Performance Debugging for Wide-Area Systems. In Proceedings of the 15th International World Wide Web Conference, 2006.

[16] P. K. G. T. Gschwind, K. Eshghi and K. Wurster. WebMon: A Performance Profiler for Web Transactions. In E-Commerce Workshop, 2002.