> *最近在研究分布式链路跟踪系统，Google Dapper 当然是必读的论文了，目前网上能搜到一些中文翻译版，然而读下来个人感觉略生硬；这里试着在前人的肩膀上重新翻译一遍这个论文，权当是个人的学习笔记，如果同时能给其他人带来好处那就更好了。*
>
> *同时把译文放到了github，如您发现翻译错误或者不通顺之处，恳请提交github PR: https://github.com/AlphaWang/alpha-dapper-translation*

> - *原文：https://ai.google/research/pubs/pub36356*
> - *译文：http://alphawang.com/blog/google-dapper-translation*

Dapper, a Large-Scale Distributed Systems Tracing Infrastructure

摘要

现代互联网服务通常都是复杂的大规模分布式系统。这些系统由多个软件模块构成，这些软件模块可能由不同的团队开发、可能使用不同的编程语言实现、可能布在横跨多个数据中心的几千台服务器上。这种环境下就急需能帮助理解系统行为、能用于分析性能问题的工具。

本文将介绍 Dapper 这个在 Google 生产环境下的分布式系统跟踪服务的设计，并阐述它是如何满足在一个超大规模系统上达到低损耗（low overhead）、应用级透明（application-level transparency）、大范围部署（ubiquitous deployment）这三个需求的。Dapper 与其他一些跟踪系统的概念类似，尤其是 Magpie[3] 和X-Trace[12]，但是我们进行了一些特定的设计，使得 Dapper 能成功应用在我们的环境上，例如我们使用了采样并将性能测量（instrumentation）限制在很小一部分公用库里。

本文的主要目的是汇报两年多以来我们构建、部署并应用 Dapper 的经历，这两年多里 Dapper 对开发和运维团队非常有用，取得了显著的成功。最初 Dapper 只是一个自包含（self-contained）的跟踪工具，后来演化成了一个监控平台并促生出许多不同的工具，有些工具甚至 Dapper 的设计者都未曾预期到。我们将介绍一些基于 Dapper 构造的分析工具，分享这些工具在 Google 内部使用的统计数据，展示一些使用场景的例子，并讨论我们学习到的经验教训。

1 介绍

Dapper 的目的是为了将复杂分布式系统的更多行为信息提供给 Google 开发者。这种分布式系统利用大规模的小服务器，通常对于互联网服务是一个非常经济的平台，所以很受关注。要理解在这种上下文中要的系统行为的话，就需要观察横跨不同程序和不同机器的关联行为。

下面基于一个 web 搜索的例子来说明这种系统需要应对哪些挑战。前端服务器将一个 web 查询分发给上百台搜索服务器，每个搜索服务器在自己的 index 中完成搜索。同时这个 web 查询可能还会被发送给多个其他子系统，进行广告处理、拼写检查、查找相关的图片/视频/新闻等。所有这些服务的结果会被有选择地合并成结果页面；我们把这种模型称之为全局搜索 (universal search)。处理一次全局搜索查询，总计需要上千台机器，涉及多种服务。而且 web 搜索的用户对延时很敏感，而任何一个子系统的性能差了都可能导致延时。工程师如果只看总体耗时的话，他能知道出问题了，但是他猜不到是哪个系统出问题、为什么出问题。**首先，工程师可能无法准确知道到底调用了哪些服务**；每周我们都会添加新的服务，用于实现用户需求、提升性能或安全性。**其次，工程师不可能对每个服务的内部都了如指掌**；每个服务都是由不同的团队开发维护的。**第三，服务和服务器可能被许多不同的客户端调用**，所以性能问题有可能是其他应用造成的。举例来说，前端服务器可能要处理多个不同的请求类型，或者类似 Bigtable 这种存储系统在被多个应用共享时效率最高。

上面描述的场景就对 Dapper 提出了两条最基本的要求：大范围部署 (uniquitous deployment)、持续监控 (continuous monitoring)。即便只有很小一部分系统没有被监控到，跟踪系统的作用也会大打折扣，所以大范围部署非常重要。另外，应该始终开启监控，因为通常来说异常系统行为很难重现，甚至根本无法重现。这两条基本要求提出了三个具体的设计目标：

- **低消耗 (Low overhead)**：跟踪系统对在线服务的性能影响应该做到可忽略不计。对于一些高度优化过的服务，监控系统的一点小消耗都会很显眼，都可能迫使部署团队不得不关停跟踪系统。
- **应用级透明 (Application-level transparency)**：程序员应该不需要感知到跟踪系统。如果跟踪系统要求应用开发者的配合才能生效，那么这个跟踪系统就太脆弱了，经常会由于应用侵入代码的 bug 或者疏忽导致无法正常工作，这就违反了"大范围部署"的要求。这在我们这种快速开发的环境下尤为重要。
- **可扩展性 (Scalability)**：需要能处理 Google 在未来几年的服务和集群规模。

另外一个设计目标是生成跟踪数据后要很快可用于分析：最好是在一分钟内。尽管一个能处理几小时前数据的跟踪分析系统已经很有用了，但是能分析最新数据的话会让我们能对生产环境的异常情况作出快速反应。

我们通过把 Dapper 跟踪植入的核心代码限制在线程调用、控制流以及 RPC 等库代码中，实现了真正的应用透明这个最具挑战性的目标。使用自适应的采样（见4.4节），我们做到了可扩展性、降低性能损耗。最终的系统还包括收集跟踪数据的代码、可视化数据的工具、用于分析大规模跟踪数据的库和 API。尽管开发人员有时通过 Dapper 就足以找出性能问题的根源，但 Dapper 并不会替代所有其他的工具。我们发现 Dapper 的数据往往侧重于性能排查，所以其他工具也有自己的用处。

1.1 贡献总结

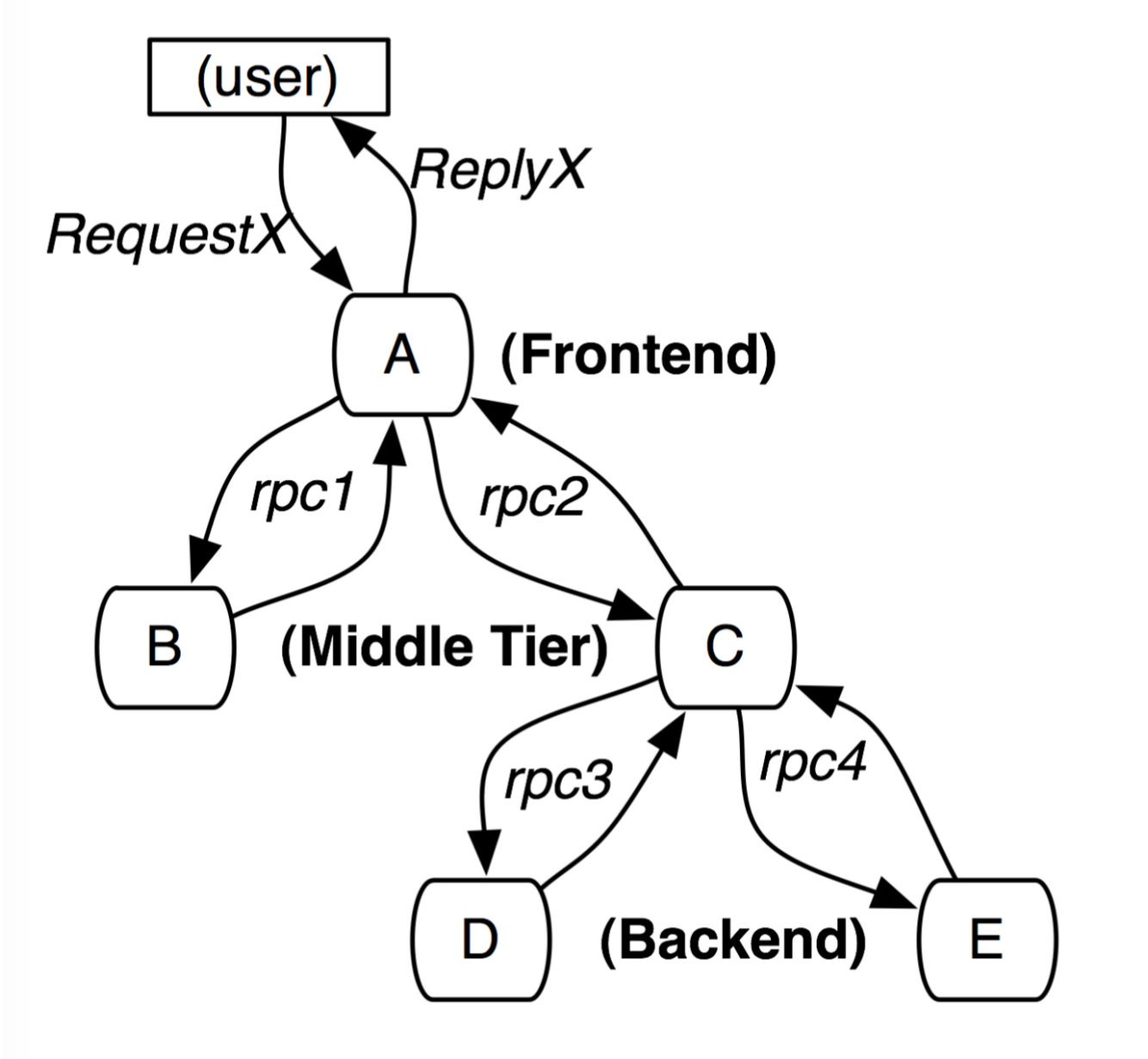
之前已有一些优秀的文章探讨了分布式系统跟踪工具的设计空间，其中 Pinpoint[9]、Magpie[3] 和 X-Trace[12] 与 Dapper 最为相关。这些系统倾向于在开发过程中的早期就写成研究报告，而此时还没有机会明确地评估重要的设计选型。Dapper 已经在生产环境中被大型系统应用好几年了，我们认为本文最适合的侧重点是讨论我们在 Dapper 开发过程中有哪些收获、我们的设计决策是如何制定的、它在哪些方面最有用。Dapper 作为一个开发性能分析工具的平台以及作为一个监控工具，其价值是我们可以在回顾评估中找到一些意想不到的产出。

虽然 Dapper 的许多高层理念和 Pinpoint、Magpie 等其他系统是共通的，但是我们的实现包含了一系列新的贡献。举个例子，我们发现要想降低消耗的话采样就必不可少，尤其是在高度优化后的对延迟非常敏感的 web 服务中。或许最令人惊讶的是，我们发现即便只使用 1/1000 的采样率，已经能为跟踪数据的通用用例提供足够多的信息了。

Dapper 的另一个重要特征是我们实现的应用透明程度非常高。我们将性能测量限制在足够底层，所以即便是像 Google web 搜索这样的大型分布式系统也能进行跟踪，而无需额外的注解。虽然由于我们的部署环境具有一定的同质性，所以更容易实现应用透明这个目标，但是我们的结果也论证了实现透明性的充分条件。

2 Dapper 的分布式跟踪

分布式服务的跟踪系统需要记录在一次请求后系统完成的所有工作的信息。举个例子，图-1展示了拥有 5 台服务器的服务：一个前端服务器 A，两个中间层 B 和 C，两个后端服务器 D 和 E。当用户发起请求到前端服务器 A 之后，会发送两个 RPC 调用到 B 和 C。B 马上会返回结果，但是 C 还需要继续调用后端服务器 D 和 E，然后返回结果给 A，A 再响应最初的请求。对这个请求来说，一个简单的分布式跟踪系统需要记录每台机器上的每次信息发送和接收的信息标识符和时间戳。



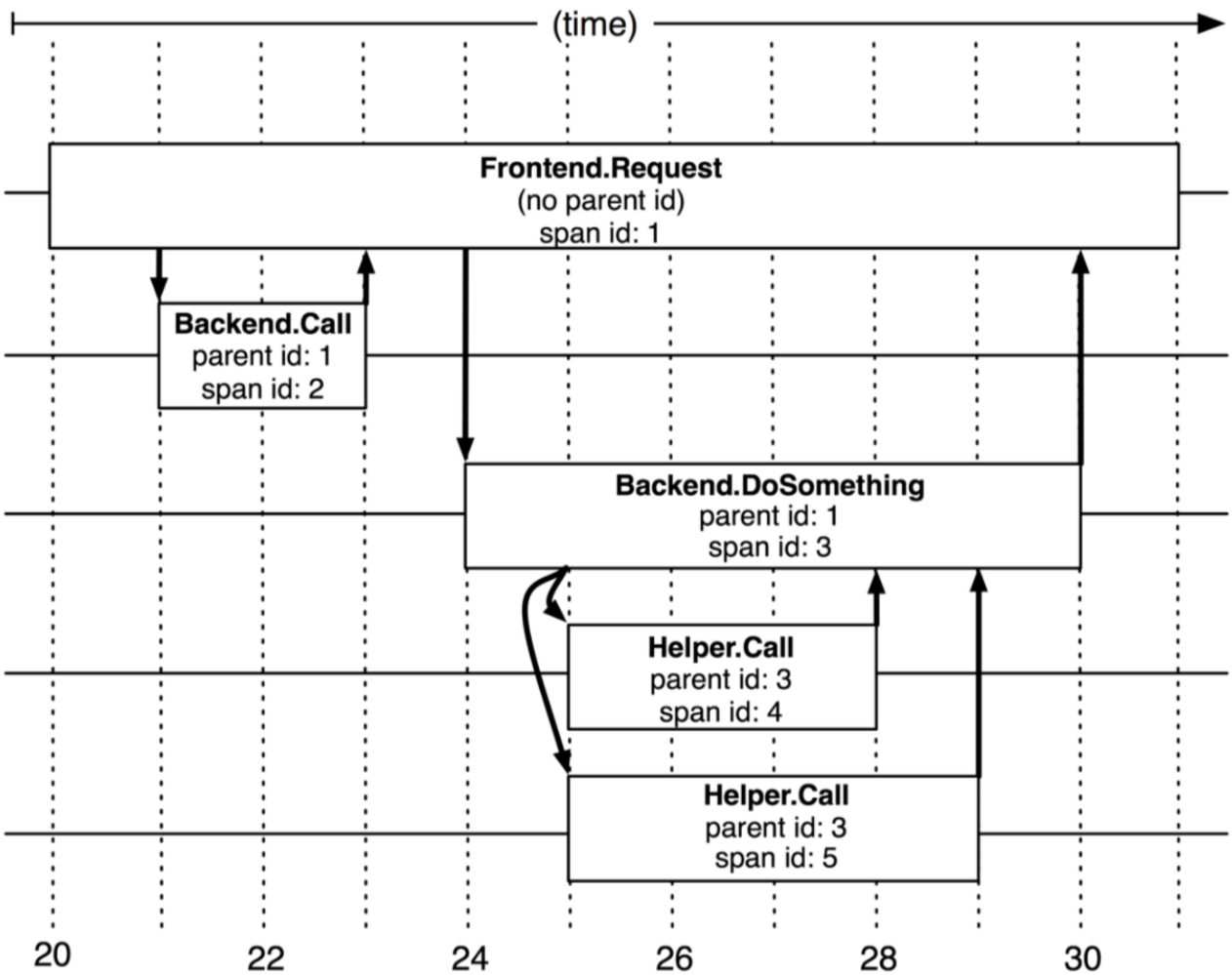
(*图-1. 由用户请求X 发起的穿过一个简单服务系统的请求路径。字母标识的节点表示分布式系统中的处理过程*)

为了能将信息聚合到一起以便人们能将所有记录信息关联到一个初始请求（如图1中的请求 X），我们提出了两种解决方案：黑盒监控模式 和 基于标注的监控模式。黑盒模式[1, 15, 2] 假定除了上面描述的信息记录之外无需任何额外的信息，而使用统计回归技术来推断关联关系。基于标注的模式[3, 12, 9, 16] 则要求应用程序或中间件显式地将每个记录关联到一个全局 ID，从而将这些信息记录关联回初始请求。黑盒模式比基于标注的模式更加轻便，但是它依赖统计推断，所以需要更多的数据以便获取足够的准确性。很明显，基于标注的模式关键缺点是需要有代码侵入。在我们的环境中，由于所有应用系统都使用相同的线程模型、控制流和 RPC 系统，所以我们可以将性能测量限制在小规模的公用库中，以此实现对开发人员有效透明的监控系统。

我们倾向于认为 Dapper 的跟踪是一个嵌入式的 RPC 树。然而，我们的核心数据模型并不局限于特定的 RPC 框架；我们也能跟踪例如 Gmail SMTP 会话、来自外界的 HTTP 请求、对 SQL 服务器的查询等行为。正式一点说，Dapper 跟踪模型使用了树、`span`和 标注。

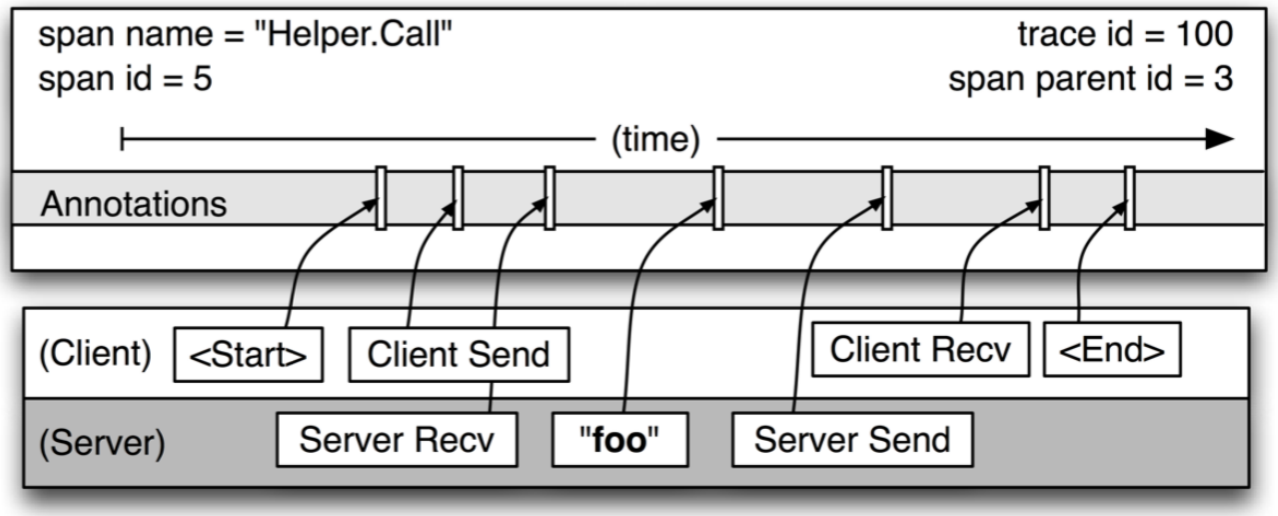
2.1 跟踪树与span

在 Dapper 跟踪树中，树节点是基本单元，我们称之为 `span`。节点之间的连线表示 span 与其父`span` 之间的关系。虽然节点在整个跟踪树中的位置是独立的，但 span 也是一个简单的时间戳日志，其中编码了这个 span 的开始时间、结束时间、RPC 时间数据、以及0或多个应用程序相关的标注，我们将在 2.3 节讨论这些内容。



(*图-2. Dapper 跟踪树中5个 span 的因果和实时关系*)

图2 阐释了 span 是如何构造成更大的跟踪结构的。Dapper 为每个 span 记录了一个可读的`span name`、`span id`和`parent id`，这样就能重建出一次分布式跟踪过程中不同 span 之间的关系。没有 parent id 的 span被称为 根`span`。一次特定跟踪的所有相关 span 会共享同一个通用的`trace id`（trace id在图中没有绘出）。所有这些 ID 可能是唯一的 64 位整数。在一个典型的 Dapper 跟踪中，我们希望每个 RPC 对应一个 span，每一个组件层对应跟踪树上的一个层级。



(*图-3. span 的详细视图*)

图3 给出了 Dapper 跟踪 span 中记录的事件的更详细视图。这个 span 标示图 2 中更长的那次`Helper.Call` RPC 调用。Dapper 的 RPC 库记录下了 span 的开始时间和结束时间、RPC 的计时信息。如果应用程序负责人选择用他们自己的标注来注释这次跟踪（例如图中的`foo`），那么这些信息也会跟随 span 的其他信息一起记录下来。

要着重强调的是，一个 span 中的信息可能来自多个不同的主机；实际上，每个 RPC span 都包含 client和 server 端的标注，这使得二主机`span (two host span)`是最常见的情况。由于 client 和 server 的时间戳来自不同的主机，所以我们需要注意时钟偏差。在我们的分析工具中，我们利用了如下事实：RPC client 发送请求总是会先于 server 接受到请求，对于 server 响应也是如此。这样一来，RPC server 端的 span 时间戳就有了下限和上限。

2.2 性能测量点 Instrumention points

通过对部分通用库进行性能测量，Dapper 能够做到在对应用程序开发者零干扰的情况下进行分布式路径跟踪：

- 当一个线程处理被跟踪的控制路径时，Dapper 会把一个**跟踪上下文**（`trace context`）存储到 ThreadLocal 中。跟踪上下文是一个小而容易复制的容器，里面包含了 trace id 和 span id 等 span属性。
- 当计算过程是延迟调用或异步执行时，多数 Google 开发者会使用一个通用的控制流程库来构造回调函数，并用线程池或其他 executor 来执行回调。Dapper 确保所有的回调都会存储其创建者的跟踪上下文，而当执行回调时这个跟踪上下文会关联到合适的线程上。通过这种方式，Dapper 用于重建跟踪的 ID 也能透明地用于异步控制流程。
- Google 进程间的通讯几乎都是建立在一个用 C++ 和 Java 开发的 RPC 框架上。我们在这个框架上进行性能测量，定义了所有 RPC 调用相关的 span。被跟踪的 RPC 调用的 span id 和 trace id

会从客户端传送到服务端。对于这种在Google内广泛使用的基于RPC的系统来说，这是一个非常必要的性能测量点。我们计划当非 RPC 通讯框架发展成熟并找到其用户群后，再对非 RPC 通信框架进行性能测量。

Dapper 的跟踪数据是语言无关的，生产环境中的许多跟踪结合了 C++ 和 Java 进程中的数据。在 3.2 节我们将讨论我们在实践中达到了何种程度的应用程序透明。

2.3 标注 Annotation

上述性能测量点足够推导出复杂分布式系统的跟踪细节，这使得 Dapper 的核心功能也适用于那些不可修改的 Google 应用程序。然而，Dapper 也允许应用程序开发者添加额外的信息，以丰富 Dapper 的跟踪数据，从而帮助监控更高级别的系统行为，或者帮助调试问题。我们允许用户通过一个简单的 API 来定义带时间戳的标注，其核心代码如图4 所示。这些标注支持任意内容。为了保护 Dapper 用户不至于意外加入太多日志，每个跟踪 span 都可配置一个标注量的上限。应用程序级别的标注是不能替代结构化的 span 信息以及 RPC 信息的。

```
// 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);
```

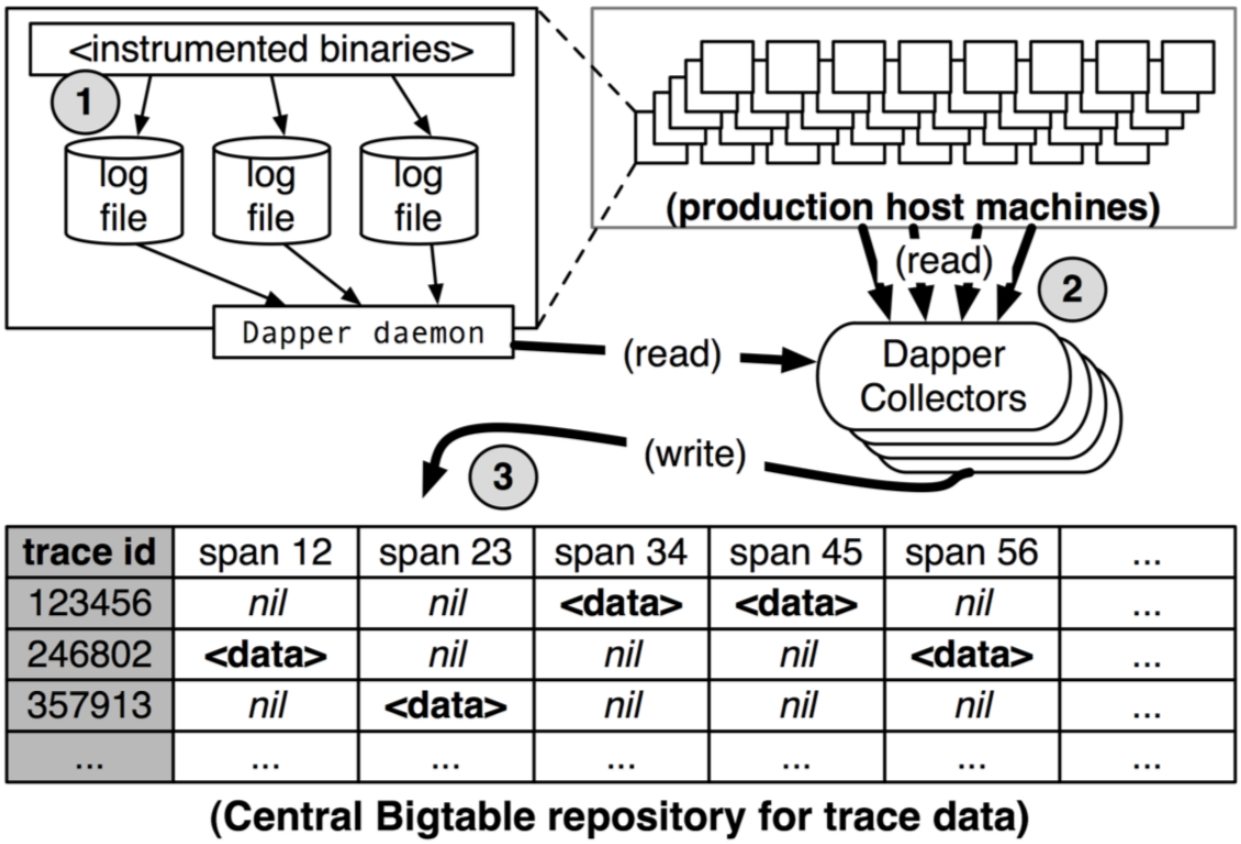
(图-4. Dapper 标注 API 在 C++ 和 Java 中的通用使用模式)

除了简单的文本标注，Dapper 也支持 key-value map 的标注，给开发者提供更强的跟踪能力，例如维护计数器、记录二进制消息、传输任意用户自定义的数据。这些 key-value 标注可用于在分布式跟踪上下文中定义应用程序相关的对等类（equivalence classes）。

2.4 采样 Sampling

Dapper 的一个关键设计目标是低损耗，因为如果一个新工具的价值还未证实，而对性能有影响的话，服务运维人员是不会愿意去部署这个工具的。而且，我们还想要允许开发人员使用标注 API，而无需担心额外的损耗。我们同时也发现 web 服务确实对性能测量的损耗很敏感。所以，除了把 Dapper 的基本性能测量损耗限制得尽可能小，我们还通过仅记录一部分跟踪信息，来进一步降低损耗。我们将在 4.4 节详细讨论这种跟踪采样模式。

2.5 跟踪收集



(*图-5. Dapper 收集管道概览*)

Dapper 的跟踪记录和收集管道分为三个阶段（如图5）。首先，把 span 数据写入(1) 到本地日志文件。然后 Dapper 守护进程从所有生产主机中将他们拉取出来(2)，最终写入(3) 到 Dapper 的 Bigtable 仓库中。Bigtable 中的行表示一次跟踪，列表示一个 span。Bigtable 对稀疏表格布局的支持正适合这种情况，因为每个跟踪都可能有任意多个 span。跟踪数据收集即将应用程序二进制数据传输到中央仓库，其延迟中位数小于 15 秒。98 分位延迟呈现双峰形；大约 75% 时间里，98 分位延迟小于 2 分钟，但是在另外 25% 时间里可能会涨到几小时。

Dapper 还提供了一个 API 来简化对仓库中跟踪数据的访问。Google 开发者利用这个API来构造通用的或者特定应用程序的分析工具。5.1 节将介绍这个 API 的使用。

2.5.1 带外（out-of-band）跟踪收集

Dapper 系统在请求树 带外(`out-of-band`) 进行日志跟踪与收集。这样做有两个原因：首先，带内收集模式（in-band collection scheme）通过 RPC 响应头回传跟踪数据，这会影响应用的网络动态。Google 的许多大型系统里，一次跟踪有几千个 span 的情况并不少见。而即便是在大型分布式跟踪的根节点附近，RPC 响应仍然是相当小的：通常小于 10K。在这种情况下，带内跟踪数据会影响应用数据，并且使后续的分析结果产生偏差。其次，带内收集模式假定所有 RPC 调用时完美嵌套的。而我们发现许多中间件系统会在其后端服务返回最终结果前，返回一个结果给其调用者。带内收集系统不能适用于这种非嵌套的分布式执行模式。

2.6 安全和隐私考虑

记录 RPC payload 信息会丰富 Dapper 的跟踪能力，因为分析工具可能能从 payload 数据中找到导致性能异常的模式。然而在某些情况下，payload 数据可能会包含一些信息，这些信息不应该暴露给非授权内部用户，包括正在调试性能的工程师。

由于安全和隐私是不可忽略的问题，所以 Dapper 存储了 RPC 方法名，但不会存储任何 payload 数据。相反，应用级别的标注则提供了一个方便的可选机制：应用开发人员可以选择将那些对以后分析有用的任何数据关联到一个 span 上。

Dapper 还提供了一些设计者没料到的安全性好处。例如 Dapper 通过跟踪公开的安全协议参数，用来监控应用是否满足认证或加密的安全策略。Dapper 还可以提供信息以确保系统是否执行了预期的基于策略的隔离，例如承载敏感数据的应用不与未授权的系统组件交互。这种方法可比代码审核强多了。

3 Dapper 的部署状况

我们把 Dapper 作为生产环境跟踪系统超过两年了。本节我们将汇报 Dapper 系统的状态，着重讲解 Dapper 如何很好地满足大范围部署、应用级透明等目标的。

3.1 Dapper 运行时库

Dapper 代码中最关键的部分也许就是对基础 RPC、线程、控制流库的性能测量了，包含创建 span、采样以及记录到本地磁盘。我们的代码不仅需要轻量，还需要稳定、健壮，因为它与海量应用连接，维护和 bug 修复是很困难的。我们的C++ 性能测量的核心代码少于 1000 行，而 Java 代码则少于 800 行。key-value 标注的代码实现额外有 500 行代码。

3.2 生产环境覆盖率

Dapper 的渗透率可以通过两方面来衡量：其一是可以产生 Dapper 跟踪的生产环境进程比率（即与 Dapper 性能测量运行时库连接的那些），其二是运行 Dapper 跟踪收集守护进程的生产环境机器比率。Dapper 守护进程是我们基本机器镜像的一部分，所以实际上它在 Google 的每台服务器上都有。很难确定 Dapper-ready 进程精确比率，因为那些不产生跟踪信息的进程是对 Dapper 不可见的。尽管如此，因为 Dapper 性能测量库几乎无处不在，我们估么着几乎每一个 Google 生产环境进程都支持跟踪。

在有些情况下 Dapper 不能正确地跟踪控制流程。这通常是由于使用了非标准的控制流程，或是由于 Dapper 错误地将因果关系归到无关的事件上。Dapper 提供了一个简单的库作为一种变通方法，可以帮助开发者手动控制跟踪的传播。目前有 40 个 C++ 应用和 33 个 Java 应用需要手工的跟踪传播，这对总计几千个应用来说只是很小的一部分。还有很小一部分程序使用的是没有性能测量的通讯库（例如通过原生 TCP Socket 或者 SOAP RPC），所以是不支持 Dapper 跟踪的。但如果真的需要的话，这些应用也可以做到支持 Dapper。

为了生产环境的安全性，Dapper 跟踪是可以被关闭的。实际上在早期它默认是关闭的，直到我们对 Dapper 的稳定性和低损耗有信心之后，我们才把它开启了。Dapper 团队偶尔会进行审计检查配置文件的变化，找到那些关闭了跟踪配置的服务。这种变化很少见，并且通常是因为担心监控的消耗。经过对实际消耗的进一步调查和衡量，发现其消耗已经很小了，所以现在这些改动都已经被回退回去了。

3.3 跟踪标注的使用

程序员们喜欢用应用程序特定的标注来作为一种分布式调试日志文件，或者通过应用程序的特定功能来对跟踪进行分类。例如所有 Bigtable 的请求都标注了访问的表名。目前 Dapper 中 70% 的 span 和 90% 的 trace 都至少有一个应用指定的标注。

我们有 41 个 Java 应用和 68 个 C++ 应用添加了自定义的标注以便更好地理解 span 内的行为。值得注意的是 Java 开发者在每个 span 上加的标注比 C++ 开发者更多，这也许是因为 Java 的负载更接近最终用户；这类应用经常处理更广的请求，所以控制路径也相对更复杂。

4 管理跟踪损耗

跟踪系统的成本是由于生成追踪和收集数据造成的系统性能下降，以及用来存储和分析跟踪数据的资源量。尽管你可以说一个有价值的跟踪系统即便造成一点性能损耗也是值得的，但是我们相信如果基线损耗达到可以忽略的程度，那么一定会对跟踪系统的最初推广大有裨益。

本节我们将展示 Dapper 性能测量操作的消耗、跟踪收集的消耗、以及 Dapper 对生产环境负载的影响。同时还会介绍 Dapper 的适应性采样机制是如何帮助我们平衡低损耗的需求与代表性跟踪的需求。

4.1 跟踪生成的损耗

跟踪生成的损耗是 Dapper 性能影响中最重要的部分，因为收集和分析可以在紧急情况下关闭掉。Dapper 运行库生成跟踪的消耗最重要的原因是创建销毁 span 和标注、以及记录到本地磁盘以便后续的收集。非根 span 的创建和销毁平均需要 176 纳秒，而根 span 则需要 204 纳秒。这个差别是因为要对根 span 分配全局唯一 trace id 的时间。

如果一个 span 没有被采样的话，那么额外标注的成本则几乎可以忽略不计，只需 Dapper 运行时在 ThreadLocal 中查找，平均花费 9 纳秒。而如果这个 span 被采样的话，对 span 进行字符串标注（见图4）则平均需要花费 40 纳秒。这些数据都是基于 2.2GHz x86 服务器测量得出的。

Dapper 运行时库最昂贵的操作就是写入本地磁盘了，不过这个损耗可以大大减少，因为每个磁盘都会合并对多个日志文件写入操作，并且相对于被跟踪的应用系统来说是异步执行的。尽管如此，日志写入对高吞吐量系统仍然可能有可见的性能影响，尤其是当所有请求都都被跟踪时。在 4.3 节我们对一次 Web 搜索过程中生成跟踪的损耗进行了量化。

4.2 跟踪收集的损耗

读出本地的跟踪数据也会对正在监控的系统产生影响。表1 展示了在高于实际负载的测试情况下，Dapper 守护进程 CPU 使用率的最坏情况。

Process Count (per host)	Data Rate (per process)	Daemon CPU Usage (single CPU core)
25	10K/sec	0.125%
10	200K/sec	0.267%
50	2K/sec	0.130%

(表-1. Dapper 守护进程在负载测试中CPU 资源使用率)

Dapper 守护进程在跟踪收集过程中，对生产环境单核 CPU 的占用率从未超过 0.3%，并且内存占用也很小。同时我们把 Dapper 守护进程在内核 scheduler 中的优先级限制到尽可能最低，以防在高负载机器上出现 CPU 竞争。

Dapper 对网络资源的消耗也轻量，我们仓库中每个 span 平均只有 426 byte。Dapper 跟踪数据在 Google 生产环境中占用的网络流量小于 0.01%。

4.3 对生产环境负载的影响

高吞吐量的在线服务处理每个请求都会用到大量的机器，这种在线服务最有需求进行高效的跟踪；他们会生成大量的跟踪数据，同时也对性能影响是最敏感的。在表2 中我们用 web 搜索集群作为例子，通过调整采样率，来测量 Dapper 对平均延迟和吞吐量的性能影响。

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. Dapper 采样频率对Web 搜索集群延迟及吞吐量的影响。延迟及吞吐量的实验误差分别为2.5% 和0.15%)

可以看到，虽然对吞吐量的影响不是很明显，但为了避免明显的延迟，跟踪采样还是非常有必要的。然而，当采样率小于 1/16 时的延迟和吞吐量损失是在实验误差范围内的。实践中，我们发现对于高容量服务来说，即便把采样率设为 1/1024 这样低，仍然能够得到足够多的跟踪数据。保持 Dapper 的性能损耗基线极其低是很重要的，因为这就能为应用程度提供一个宽松的环境来使用完整的标注 API，而无需担心性能损失。使用低采样率还有一个额外的好处，可以让主机磁盘中的数据在被垃圾回收之前能持久化更长的时间，这就让收集组件有了更多的灵活性。

4.4 适应性采样

Dapper 对于任何给定进程的损耗是与单位时间内进程跟踪的数目成正比的。Dapper 的第一个生产版本在 Google 的几乎所有进程上使用同一个采样率，每 1024 个候选中平均采样一个。这个简单的方案对高吞吐量在线服务是有效的，因为大多数我们感兴趣的事件仍然会经常出现并被捕捉到。

然而，低流量的服务在这种低采样率下就可能会错失重要的事件，而更高采样率带来的性能损耗是可接受的。针对这种系统的解决方案是覆盖默认采样率，而这就需要手工干预，我们不想在 Dapper 中出现这种手工干预。

我们正在部署一种适应性的采样机制，不使用统一的采样率，而使用单位时间内的期望采样率。这样，低流量负载会自动提高采样率，而高流量负载则会自动降低采样率，从而掌控损耗。实际采样率会和跟踪数据一起记录下来；这有利于在基于 Dapper 数据的分析工具中精准使用采样率。

4.5 应对激进采样

Dapper 新用户往往觉得低采样率（高流量服务中通常会低于 0.01%）会干扰他们的分析。我们在 Google 中应用的经验让我们相信，对于高吞吐量服务来说，激进采样并不会妨碍最重要的那些分析。如果一个重要的执行模式在这种系统中出现过一次，那么就会出现上千次。每秒请求几十次而不是上万次的那些低流量服务则可以承受跟踪每一个请求；这驱动着我们往适应性采样方向前进。

4.6 收集过程中的额外采样

上述采样机制用来尽量减少与 Dapper 运行时库协作的应用程序中的性能损耗。Dapper 团队还需要控制写入中央仓库的数据量，为此我们引入了第二轮采样。

目前我们生产集群每天产生超过 1 TB 的采样跟踪数据。Dapper 用户希望跟踪数据从生产进程中记录下来后最少保留两周时间。逐渐增长的跟踪数据带来了好处，同时 Dapper 仓库的机器和磁盘存储成本也在增加，我们需要作出权衡。对请求的高采样率还会使得 Dapper 收集器接近 Dapper Bigtable 仓库的写入吞吐量极限。

为了维持物资资源的需求和 Bigtable 的累积写入吞吐量之间的灵活性，我们在收集系统自身上增加了额外的采样。一个特定 trace 中的所有 span 都共享同一个trace id，即便这些span可能横跨数千个不同的主机。**对于在收集系统中的每个 span，我们将其 trace id 哈希成一个标量 z （0<=z<=1）。如果 z 小于我们的收集采样系数，我们就保留这个 span 并将它写入 Bigtable；否则就丢弃。**在采样决策中通过依靠 trace id，我们要么采样整个 trace，要么抛弃整个 trace，而不会对 trace 中的某些 span进行处理。我们发现这种额外配置参数让我们对收集管道的管理变得简单得多，因为可以很容易地调整全局写入率，仅仅修改配置文件中的一个参数即可。

如果整个跟踪和收集系统都是用同一个采样参数则会更简单，但是那样就无法灵活地快速调整所有部署环境中的运行时采样配置。我们选择的运行时采样率产生的数据会稍微高于我们能写入仓库的数据，而我们可以通过调整收集系统中的二级采样参数对写入速度进行限流。因为我们可以通过对二级采样配置一下就能增加或减少全局覆盖率和写入速率，所以 Dapper 管道的维护工作变得更简单了。

5 通用 Dapper 工具

几年前当 Dapper 还是一个原型时，在开发者的耐心支持下才能把 Dapper 用起来。从那时起，我们逐渐建立了收集组件、编程接口、以及基于 web 的用户交互界面，帮助 Dapper 用户独立地解决自己的问题。本节将总结哪些方法有用，哪些没用，并提供这些通用的分析工具的基本使用信息。

5.1 Dapper Depot API

Dapper Deport API 又称 DAPI，通过它可以直接访问 Dapper 区域仓库中的分布式跟踪数据。DAPI 和 Dapper 跟踪仓库是串行设计的，DAPI 意在为 Dapper 仓库中的原始数据提供一个干净而直观的接口。我们的用例推荐如下三种方式来访问跟踪数据：

通过trace id访问（Access by trace id）：DAPI 可以根据全局唯一的 trace id 来加载任何一次跟踪。

批量访问（Bulk access）：DAPI 可通过 MapReduce 来并行访问数亿条 Dapper 跟踪数据。用户重写一个虚拟函数，它的唯一参数接受一个 Dapper 跟踪信息，然后框架将会对用户指定时间窗口内的每一条跟踪信息调用一次该函数。

索引访问（Indexed access）：Dapper 仓库支持一个唯一索引，可用于匹配我们通用的访问模式。该索引将通用请求的跟踪特性映射到特定的 Dapper 跟踪。因为 trace id 是伪随机创建的，所以这是快速访问某个特定服务或特定主机追踪信息的最佳方式。

所有这三种访问方式都将用户引导到特定的 Dapper 追踪记录。正如 2.1 节所述，Dapper 的跟踪信息是由 trace span 组成的树，所以 `Trace` 数据结构就是一个由不同 `Span` 结构组成的遍历树。Span 通常对应 RPC 调用，在这种情况下，RPC 的耗时信息是有的。通过 span 结构还可访问基于时间戳的引用标注信息。

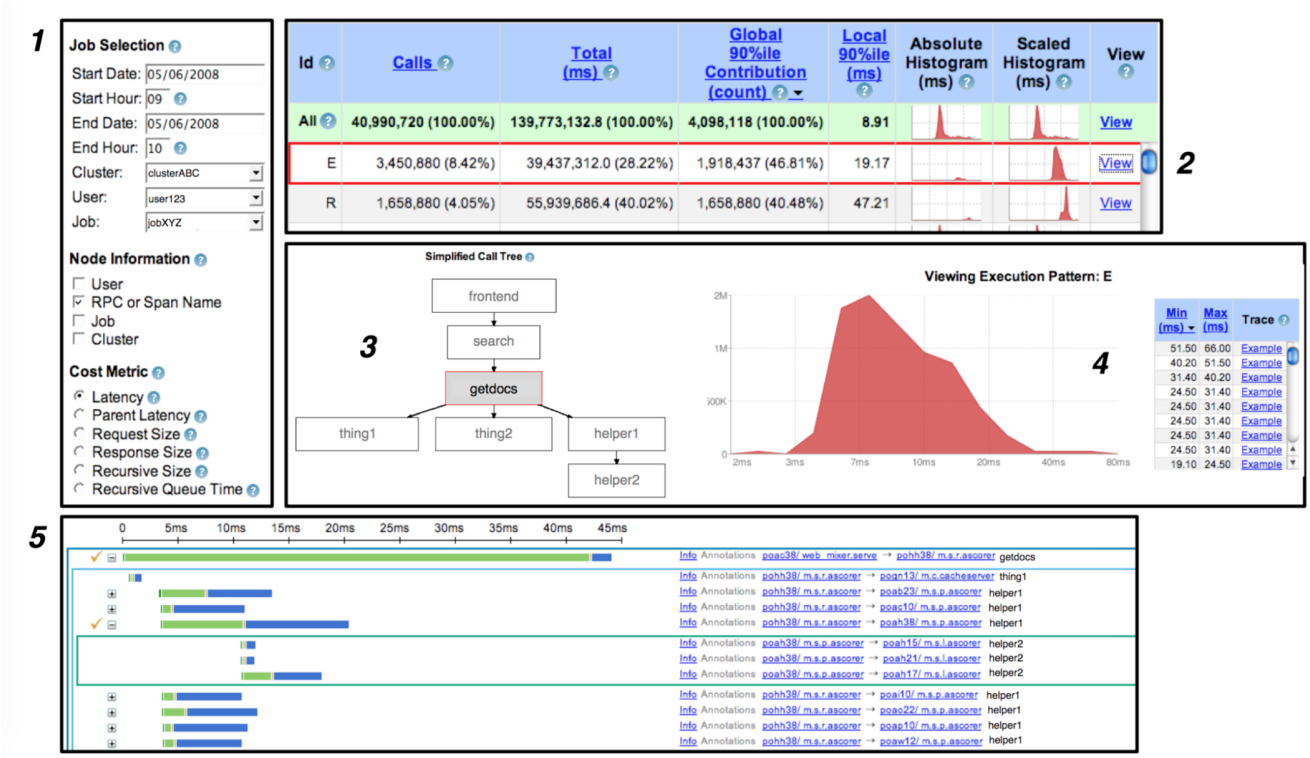
选择合适的用户索引是DAPI 设计中最具挑战性的部分。索引要求的压缩存储只比实际数据本身小 26%，所以成本是巨大的。最初我们部署了两个索引：一个是主机索引，另一个是服务名索引。然而我们发现相对于存储成本来说，用户对主机索引的兴趣尚不足够。当用户对某台机器的跟踪感兴趣的时候，他们也会对特定的服务感兴趣，所以我们最终将这两个索引合并成一个组合索引，允许按服务名、主机、时间戳高效地进行查找。

5.1.1 DAPI 在 Google 内部的使用

Dapper 在 Google 的使用有三类：使用 DAPI 的持久在线 web 应用，可在命令行启动的维护良好的基于 DAPI 的工具，以及编写、运行、然后即被遗忘的一次性分析工具。我们知道的有3 个基于DAPI的持久性应用、8个基于DAPI的分析工具、约15~20个一次性分析工具。在这之后就很难统计这些工具了，因为开发者可以构建、运行、然后丢弃，而不需要让 Dapper 团队知道。

5.2 Dapper 用户接口

绝大多数情况下，人们通过基于 web 的用户交互接口来使用 Dapper。篇幅所限我们不能展示每一个特性，不过图6 列出了一个典型的用户工作流。



(*图-6. 通用Dapper 用户接口中的一个典型用户工作流*)

1. 用户输入他们关心的服务名以及时间窗口，再加上任何需要来区分跟踪模式的信息（例如span名称）。同时指定与他们的搜索最相关的成本度量（例如服务响应时间）。
2. 然后就会出现一个性能概要的大表格，总结了与给定服务相关的所有分布式执行模式。用户可以根据他们的需要对执行模式进行排序，并选择其中一个查看更多细节。
3. 一旦选中一个分布式执行模式，用户则会看到关于这个执行模式的图形化描述。被选中的服务在图表中央被高亮显示。
4. 在创建与第 1 步选中的成本度量相关的统计信息后，Dapper 用户界面会展示一个简单的频率直方图。所以在这个例子中，我们能看到选中的执行模式相关的响应时间大概是对数正态分布的。用户还会看到一个特定跟踪样例的列表，这些样例分布在直方图的不同区间。本例中，用户点击第二个跟踪样例，在 Dapper 用户界面打开跟踪详细视图。
5. 绝大多数 Dapper 用户最终会检查特定的跟踪，希望收集系统行为根本原因的信息。我们没有足够的空间去做跟踪视图的审查，但我们有个全局时间线，并能交互地展开或折叠子树，这是我们的特点。分布式跟踪树的连续层用内嵌的不同颜色的矩形表示。每个 RPC span 分为服务进程处理时间（绿色）和网络消耗时间（蓝色）。用户标注没有显示在这个截图中，不过可以以 span 为基础将他们选择性地包含在全局时间线上。

对于想查询实时数据的用户，Dapper 用户界面支持直接与每台生产环境服务器上的守护进程通信。在这个模式下，不能像上图那样查看系统级别的图表，不过仍然很容易地基于耗时和网络特性选择一个跟踪。在这个模式下，可在几秒内实时地查到数据。

根据我们的日志，每个工作日大概有 200 个 Google 工程师使用 Dapper UI；每周大约有 750 到 1000 个独立用户访问。忽略掉发布新功能的因素，这个数据每个月都是一致的。用户通常会发送出特定跟踪的链接，这会不可避免地在跟踪查询中产生很多一次性的、短期的流量。

6 经验

Dapper 在 Google中被广泛使用，通过 Dapper 用户界面直接访问，或者通过编程 API 以及基于这些 API 构建的程序访问。本节我们不打算罗列出每一种已知的 Dapper 的使用方式，而会尝试讲解 Dapper 使用的"基本向量"，阐述何种应用是最成功的。

6.1 开发过程中使用 Dapper

Google AdWords 系统建立在关键词定位准则和相关文字广告的大型数据库之上。当新的关键词被插入或修改时，必须对他们进行校验，以遵循服务策略条款（例如检查不恰当的语言）；这个过程使用自动审查系统来做的话会更有效率。

当从头开始重新设计一个广告审查服务时，团队从第一个系统原型开始，直到最终的系统维护，都使用了 Dapper。他们的服务通过 Dapper 有了以下方面的提高：

性能（Performance）：开发人员跟踪请求延迟目标的进度，精确找到可优化的机会。Dapper 还被用来找出关键路径中的不必要请求序列（这种不必要请求通常源于不是开发者自己开发的子系统），然后促使相关团队修复这些问题。

正确性（Correctness）：广告审查服务是围绕大型数据库系统的。系统同时具有只读副本服务器（廉价访问），以及可读写的主服务器（昂贵访问）。他们通过 Dapper 找到了好些不必要地访问主服务器而不是访问副本服务器的查询。Dapper 现在可用于解释主服务器被直接访问的原因，确保重要系统的不变式。

理解性（Understanding）：广告审查查询跨越多种类型的系统，包括 Bigtable（即前文提到的数据库）、多维索引服务、以及许多其他 C++ 和 Java 后端服务。Dapper 跟踪用来评估总查询成本，促进对业务重新设计，使得系统依赖的负载最小。

测试（Testing）：新代码的发布会经过一个 Dapper 跟踪的 QA 过程，验证正确的系统行为和性能。这个过程中发现了很多问题，包括广告审查代码自身的问题，及其依赖包的问题。

广告审查团队广泛使用了 Dapper 标注 API。Guice[13] 开源的 AOP 框架用来在重要的软件组件上标注 `@Traced`。跟踪信息进一步标注的信息有重要子程序的输入输出大小、状态消息、以及其他调试信息；否则这些信息会被发到日志文件中。

Dapper 在广告审查团队的应用有一些不足的地方。例如，他们想在交互时间内搜索所有的跟踪标注，然而必须运行自定义的 MapReduce 或者手工检查每个跟踪。另外，Google 内还有其他的系统对通用目的的调试日志进行收集并进行集中化，把这些系统中的海量数据和 Dapper 仓库进行整合是有价值的。

即便如此，总的来说广告审查团队估计通过 Dapper 跟踪平台的数据分析，他们的延迟数据已经优化了两个数量级。

6.1.1 与异常监控的集成

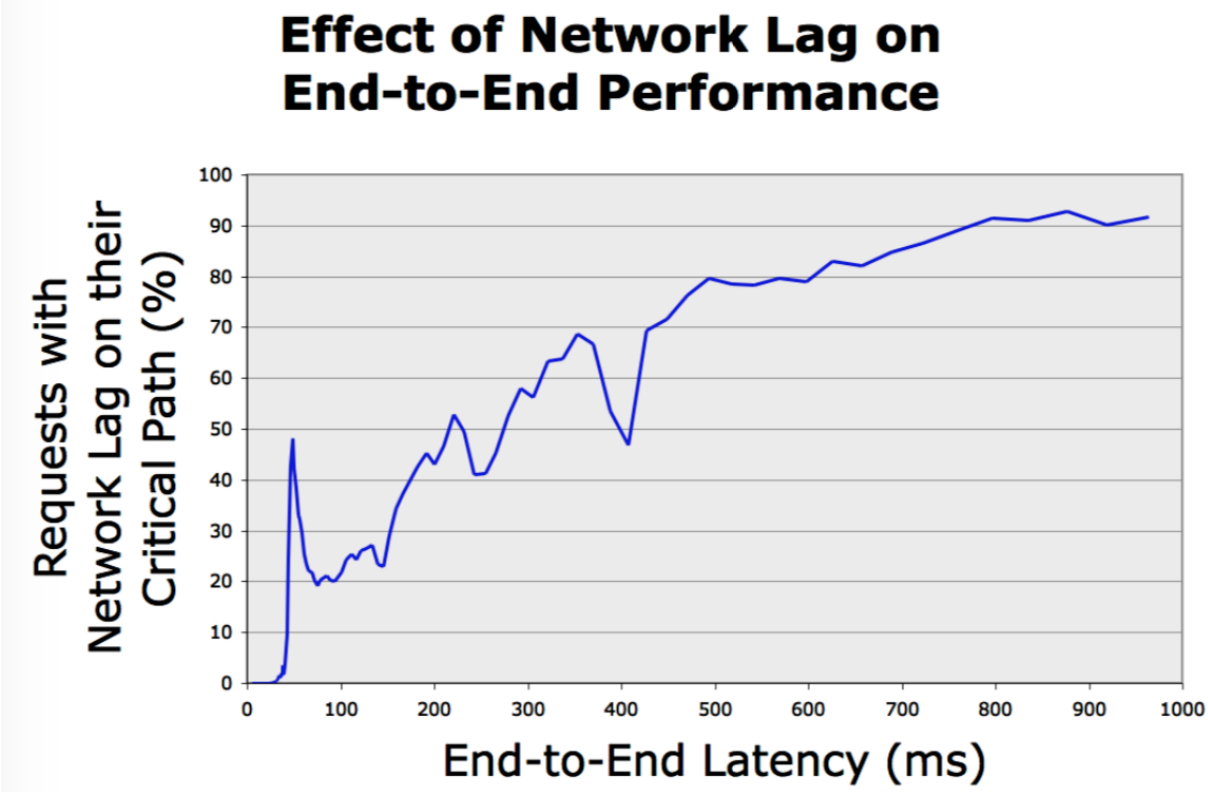
Google 维护了一个从运行进程中不断收集并集中异常报告的服务。如果这些异常发生在被采样的 Dapper 跟踪中，则异常报告中会包含相关的 trace id 和 span id。然后异常监控服务前端就会在特定异常报告里提供一个链接，指向相应分布式跟踪。广告审查团队利用这个特性，来了解异常监控服务发现的那些 bug 的更大范围的上下文。Dapper 平台通过导出基于简单唯一 ID 构建的接口，相对容易地集成到其他事件监控系统中。

6.2 解决长尾延迟

由于移动部件的数量、代码库及部署的规模，调试一个像全文搜索（universal search）那样的服务是非常有挑战性的。这里我们描述在减轻全文搜索延迟分布的长尾效应上做的努力。Dapper 能够验证端到端延迟的假设，更具体地说，它能够**验证全文搜索请求的关键路径**。当系统不仅涉及多个子系统，还涉及多个开发团队时，即便我们最好最有经验的工程师也经常猜错端到端性能差的根本原因。在这种情况下，Dapper 可以提供必需的事实，可以回答许多重要的性能问题。

一个工程师在调试长尾延迟的过程中建立了一个小型库，可以根据 DAPI `Trace`对象推断出层次性的关键路径。这些关键路径结构可用来诊断问题、为全文搜索可预期的性能改进调整优先级。Dapper 的这项工作引出了下列发现：

- 关键路径上短暂的网络性能退化不会影响系统吞吐量，但能对延迟异常值产生巨大影响。在图7中，大多数全文搜索的慢跟踪都在关键路径上有网络退化。



(*图-7. 关键路径上遇到非正常网络延迟的全文搜索跟踪，与端到端请求延迟的关系*)

- 许多有问题的昂贵查询模式都源自服务间不经意的交互。一旦发现，他们往往很容易纠正；但是在没有 Dapper 时如何发现他们是很困难的。
- 通用查询是从 Dapper 之外的安全日志仓库中获取，并且使用 Dapper 的唯一 trace id，与 Dapper 仓库做关联。这种映射随后被用于构建全文搜索每个独立子系统中的慢查询列表。

6.3 推断服务依赖

在任意指定时刻，Google 的典型计算集群是成千上万个逻辑"任务"组成；一系列进程执行通用函数。Google 维护着许多这种集群，当然我们发现一个计算集群中的任务往往依赖其他集群中的任务。由于任务间的依赖是动态改变的，所以不可能仅仅从配置信息中推断出所有的服务间依赖。尽管如此，公司内部的许多进程要求知道准确的服务依赖信息，以便找出瓶颈，计划服务的迁移。Google 的"服务依赖"项目通过使用跟踪标注以及 DAPI MapReduce 接口，自动探测服务间的依赖。

使用 Dapper 核心性能检测以及 Dapper 的跟踪标注，服务依赖项目能够推断出任务之间的依赖关系，还能推断出这些任务所依赖的程序组件。例如，所有 Bigtable 的操作被标记上受影响的表名。通过 Dapper 平台，服务依赖团队就可以自动推断出多种服务粒度的依赖关系。

6.4 不同服务的网络使用率

Google 在网络结构上投入了大量的人力物力。毫无疑问，网络运维人员要关注单个硬件的监控信息、自定义工具和 dashboard，来查看全局网络使用情况的鸟瞰图。网络运维人员可以一览整个网络的健康状况，但是当出现问题时，他们却缺少工具找到网络负载问题在应用级别的罪魁祸首。

虽然 Dapper 并不是设计用来做链路级的监控，但我们发现它非常适合集群之间网络活动应用级别分析的任务。Google 利用 Dapper 平台得以建立不断更新的终端，来显示集群间网络流量中最活跃的那些应用级别端点。此外，通过 Dapper 我们可以找出引起昂贵网络请求的跟踪，而不是面对孤立的机器。在 Dapper API 之上建立 dashboard 花费的时间没超过两周。

6.5 分层及共享的存储系统

Google 的许多存储系统都由多个独立的复杂层次的分布式基础设施组成。例如，Google App Engine[5] 就是建立在一个可扩展实体存储系统之上。这个实体存储系统基于底层的 BigTable 暴露出一些 RDBMS 功能。Bigtable 则同时使用 Chubby[7]（一个分布式锁系统）及 GFS。此外，像 BigTable这类系统会作为共享服务来管理，以简化部署并更好地利用计算资源。

在这种分层系统中，并不总是很容易发现终端用户的资源消费模式。例如，给定 BigTable 单元对 GFS 的大量请求可能来自一个用户或者许多用户，而在 GFS 层面这两种不同的使用模式的区别是模糊的。而且，如果缺乏像 Dapper 这种工具的话，对这种共享服务的竞争同样是难以调试的。

5.2节展示的 Dapper 用户界面可以分组聚合共享服务横跨多个客户端的跟踪性能信息。这就使得共享服务的负责人可以容易地根据多个指标对其用户进行排名（例如根据inbound网络负载、outbound网络负载、或者服务请求的总时间）。

6.6 用 Dapper 来救火

Dapper 对于某些救火任务是有用的。这里的"救火"指的是对处于危险中的分布式系统进行的操作。典型情况下，Dapper 用户在进行救火时需要访问新鲜数据，并且没有时间写新的 DAPI 代码，也没时间等待周期性的报告运行。

对于那些正在经历高延迟的服务，或者更糟的在正常负载下都会超时的服务，Dapper 用户界面通常能把这些延迟的瓶颈隔离出来。通过与 Dapper 守护进程直接通信，可以容易地收集特定高延迟跟踪的新鲜数据。在灾难性故障时，通常没必要分析统计数据来确定根本原因，而查看示例跟踪就足够了。

然而，6.5 节描述的那种共享存储服务则要求当用户活动突然激增时能快速聚合信息。对于事后检验，共享服务仍然可以利用 Dapper 的聚合数据，但是除非可以在十分钟之内完成对 Dapper 数据的批量分析，否则 Dapper 对共享存储服务的救火就不会那么有用了。

7 其他经验教训

虽然我们在 Dapper 上的经验已经基本满足我们的预期，但是也有一些积极的方面是我们没有充分预料到的。我们对非计划中的用例数目感到高兴。除了在第6节描述的一些经验外，还包括资源核算系统，用来检查敏感服务是否遵从指定的通讯模式的工具，RPC 压缩策略的分析工具，等等。这些非计划中的用例一定程度上归功于我们通过一个简单的编程接口开放了跟踪数据存储，这就允许我们利用上这个大得多的社区的创造力。Dapper 对旧系统的支持也比预期更简单，只需要基于新版本的库重新编译即可，这个库提供通用线程、控制流和 RPC 框架。

Dapper 在 Google 内部的广泛使用还为我们提供了关于其局限性的宝贵反馈。下面我们将介绍一些我们已知的最重要的一些不足之处。

合并的影响（Coalescing effects）：Dapper 模型隐式地设想不同子系统一次只会处理一个跟踪请求。在某些情况下，在对一组请求执行操作之前缓冲一些请求会更有效率（例如对磁盘写入进行合并）。在这些情况下，一个跟踪请求可以看做是一个大型工作单元(a traced request can be blamed for a deceptively large unit of work)。此外，如果多个跟踪请求被批量执行，那么只会有一个请求被span使用，这是因为我们我们对每个跟踪只会有一个唯一 trace id（if multiple traced requests are batched together, only one of them will appear responsible for the span due to our reliance on a single unique trace id for each trace）。我们正在考虑解决方案以识别这种情况，并记录最少的信息来区别这些请求。

跟踪批处理系统（Tracing batch workloads）：Dapper 的设计是针对在线服务系统，最初的目标是了解 Google 的用户请求引起的系统行为。然而，离线的数据密集型系统也可以从对性能的洞悉中获益，例如适合 MapReduce 模型的系统。在这种情况下，我们需要把 trace id 关联到一些其他的有意义的工作单元，例如输入数据的 key（或key范围），或是一个 MapReduce shard。

寻找根本原因（Finding a root cause）：Dapper 可以有效地确定系统中的哪个部分正在经历速度变慢，但并不总是足够找出问题的根本原因。举个例子，一个请求变慢可能并不是因为他自己的行为，而是因为其他请求还排在他前面。程序可以利用应用级别的标注把队列大小和过载情况转播到跟踪系统。同时，如果这种情况很常见，那么在ProfileMe[11] 中提出的成对采样技术就很有用了。它对两个时间重叠的请求进行采样、并观察它们在系统中的相对延迟。

记录内核级别的信息（Logging kernel-level information）：内核可见事件的详细信息有时对确定问题根本原因很有用。我们有一些工具能够跟踪或者描述内核的执行，但是要想将这些信息绑定到用户级别的跟踪上下文上，用通用或是不那么突兀的方式是很难的。我们正在研究一种可能的妥协方案，对用户层面上的一些内核级别活动参数做快照，将其关联到一个活动 span 上。

8 相关工作

在分布式系统跟踪领域，有一套完整的体系，一些系统主要关注定位到故障位置，另一些系统关注性能优化。Dapper 曾被用于故障发现，但它在发现性能问题、提升对大型复杂系统行为的理解方面更有用。

Dapper 与黑盒监控系统有关，就像 Project5[1]、WAP5[15] 和 Sherlock[2]，黑盒监控系统不依赖于运行时库的性能测量，能够实现更高度的应用级透明。黑盒的缺点是有些不精确，并在统计推断因果路径过程中可能损耗更大。

对分布式系统的监控来说，显式的基于标注的中间件或应用本身的性能测量或许是更受欢迎的方式。Pip[14] 和 Webmon[16] 更依赖于应用级的标注，而 X-Trace[12]、Pinpoint[9] 和 Magpie[3] 则侧重对库和中间件的修改。Dapper 更接近后者。Dapper 与 Pinpoint、X-Trace 以及最新版本的 Magpie 类似，使用全局 ID 将分布式系统不同部分的相关事件关联起来。同样和这些系统类似，Dapper 把性能测量隐藏在通用软件模块中，尝试避免标注应用程序。Magpie 放弃使用全局 ID，就不用处理正确传播全局 ID 带来的挑战，而是为每个应用写入事件模式（event schema）并显式地描述事件之间的关系。我们不清楚 schema 在实践中实现透明性到底有多有效。X-Trace 的核心标注需求比 Dapper 更有雄心，不仅在节点边界收集跟踪，还在节点内部不同软件层级间收集跟踪。而我们对于性能测量低损

耗的要求迫使我们不能采用这种模式，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。Dapper 跟踪仍然能通过可选的应用标注来扩展。

9 总结

本文介绍了 Google 生产环境下的分布式系统跟踪平台 Dapper，并汇报了我们开发和使用 Dapper 的经验。Dapper 部署在 Google 的几乎所有系统上，使得大型系统得以被跟踪而无需修改应用程序，同时没有明显的性能影响。通过 Dapper 主跟踪用户界面的受欢迎程度可以看出 Dapper 对开发团队和运维团队的实用性，本文通过一些使用场景的例子也阐明了 Dapper 的实用性，甚至有些使用场景 Dapper 的设计者都未曾预料到。

据我们所知，本文是第一篇汇报一个大型的生产环境下的分布式系统跟踪框架的论文。实际上我们主要的贡献源于这样一个事实：我们汇报回顾的系统已经被使用超过两年了。我们发现，决定结合最小化应用透明的跟踪功能以及对程序员提供简单的 API 来增强跟踪是非常值得的。

我们相信，Dapper 比之前基于标注的分布式跟踪系统达到了更高的应用级透明性，只需要很少的人工干预。虽然这也归功于我们计算部署的一定程度上的同质性，但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的充分条件，我们希望能够对更异质的环境下的解决方案有所帮助。

最后，通过把 Dapper 跟踪仓库开放给内部开发者，促使了更多分析工具的产生，而仅仅由 Dapper 团队封闭地独自开发肯定产生不了这么多工具，这大大提高了设计和实现的成就。

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 Dis- tributed 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 Enter- prise Network Services Via Inference of Multi-level De- pendencies. In *Proceedings of SIGCOMM*, 2007.

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

[4] L. A. Barroso, J. Dean, and U. Ho ¨lzle. 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. Wal- lach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gru- ber. Bigtable: A Distributed Storage System for Struc- tured Data. In *Proceedings of the 7th USENIX Sympo- sium 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 In- ternational Conference on Dependable Systems and Net- works*, 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 Sympo- sium on Microarchitecture*, 1997.

[12] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Sto- ica. 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 Debug- ging 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. Web- Mon: A Performance Profiler for Web Transactions. In *E-Commerce Workshop*, 2002.