ZooKeeper：互联网规模系统的无等待协调

Patrick Hunt 和 Mahadev Konar
雅虎网格
{*phunt*,mahadev}@yahoo-inc.com

Flavio P. Junqueira 和 Benjamin Reed 雅虎
研究院
{*fpj*,breed}@yahoo-inc.com

摘要

在本文中，我们介绍了用于协调分布式应用程序进程的服务 ZooKeeper。由于 ZooKeeper 是关键基础设施的一部分，因此 ZooKeeper 的目标是为在客户端构建更复杂的协调基元提供一个简单、高性能的内核。它将群组消息传递、共享寄存器和分布式锁服务等元素整合到一个可重复的集中式服务中。Zoo Keeper 提供的接口具有共享寄存器的免等待特性，以及类似于分布式文件系统缓存失效的事件驱动机制，从而提供简单而强大的协调服务。

ZooKeeper 接口可实现高性能服务。除了免等待特性外，ZooKeeper 还为每个客户端提供了先进先出（FIFO）执行请求的保证，并为所有改变 ZooKeeper 状态的再请求提供了线性化。有了这些设计上的改进，就能实现高性能的处理流水线，由本地服务器来满足读取请求。我们展示了在读写比为 2:1 到 100:1 的目标工作负载下，ZooKeeper 每秒可处理数万到数十万个事务。这样的性能使 ZooKeeper 能够被客户端应用程序广泛使用。

1 导言

大规模分布式应用需要不同形式的协调。配置是最基本的协调形式之一。最简单的配置只是系统进程的运行参数列表，而更复杂的系统则具有动态配置参数。小组成员和组长选举在分布式系统中也很常见：进程往往需要知道哪些其他进程还在运行，以及这些进程负责什么。锁是一种强大的协调工具

这些系统实现了对关键再资源的互斥访问。

一种协调方法是为每种不同的协调需求开发服务。例如，亚马逊简单队列服务 [3] 专门针对队列问题。还有一些服务是专门为领导者选举 [25] 和配置 [27] 而开发的。实现功能更强大的原语的服务可以用来实现功能较弱的原语。例如，Chubby [6] 是一种具有强大同步保证的锁定服务。锁可以用来实现领导者选举、群成员等功能。

在设计协调服务时，我们放弃了在服务器端实现特定基元的做法，而是选择公开一个应用程序接口（API），让应用开发人员能够实现他们自己的基元。这种选择促成了*协调内核*的实现，它可以在不改变服务核心的情况下实现新的基元。通过这种方法，可以根据应用程序的要求实现多种形式的协调，而不是将开发人员限制在一套固定的基元中。

在设计 ZooKeeper 的应用程序接口时，我们摒弃了锁等阻塞原语。除其他问题外，协调服务的阻塞原语还可能导致速度慢或有故障的客户端对速度快的客户端的性能产生负面影响。如果处理请求依赖于其他客户端的响应和故障检测，那么服务本身的实现就会变得更加复杂。因此，我们的 Zookeeper 系统实现了一种应用程序接口（API），可操作简单的*空闲等待*数据对象，就像文件系统中的分级组织一样。事实上，ZooKeeper 的应用程序接口与任何其他文件系统的应用程序接口都很相似，而且仅从应用程序接口的符号来看，ZooKeeper 似乎就是没有锁方法、打开和关闭的 Chubby。不过，ZooKeeper 实现了免等待数据对象，这让它与基于锁等阻塞原语的系统有了显著区别。虽然无等待属性对于每

但这还不足以实现共同排序。我们还必须为操作提供顺序保证。特别是，我们发现保证所有操作的*先进先出客户端排序*和*可线性写入可以*高效地实现服务，这足以实现我们的应用所关心的协调原语。事实上，我们可以用我们的应用程序接口为任意数量的进程实现共识，而且根据赫里希的层次结构，Zoo- Keeper 实现了一个通用对象[14]。

ZooKeeper 服务由一组服务器组成，这些服务器通过复制实现高可用性和高性能。ZooKeeper 的高性能使包含大量进程的应用程序能够使用这样一个协调内核来管理协调的各个方面。我们能够使用简单的流水线架构来实现 ZooKeeper，这种架构允许我们处理成百上千个未处理请求，同时还能实现较低的延迟。这样的流水线自然能让单个客户端以先进先出的顺序执行操作。保证先进先出的客户端顺序可以让客户端以异步方式提交操作。有了异步操作，客户端就能同时执行多个未完成操作。例如，当一个新客户机成为领导者时，它必须对元数据进行相应的管理和更新，这种功能就非常理想。如果没有多个未完成的操作，初始化时间可能会达到秒级，而不是亚秒级。

为了保证更新操作满足线性化要求，我们实施了一种基于领导者的原子广播协议[23]，称为 Zab[24]。不过，ZooKeeper 应用程序的典型工作负载主要是读操作，因此需要扩展读吞吐量。在 ZooKeeper 中，服务器在本地处理读操作，我们不使用 Zab 对其进行完全排序。在客户端缓存数据是提高读取性能的一项重要技术。例如，一个进程可以缓存当前领导者的标识符，而不用每次都向 ZooKeeper 询问领导者的信息。ZooKeeper 使用一种观察机制，使客户端能够缓存数据，而无需直接管理客户端缓存。通过这种机制，客户端可以监视给定数据对象的更新，并在更新时收到通知。Chubby 可直接管理客户端缓存。它会阻止更新，以验证所有缓存被更改数据的客户端的缓存。在这种设计下，如果其中任何一个客户端速度较慢或出现故障，更新就会延迟。Chubby 使用租约来防止故障客户端无限期地阻塞系统。不过，租约只能限制慢速或故障客户端的影响，而 ZooKeeper 手表则能避免这完全是个问题。

在本文中，我们将讨论我们的设计和实现方法。

ZooKeeper 的应用。有了 ZooKeeper，我们就能实现应用程序所需的所有协调原语，即使只有写入是可线性化的。为了验证我们的方法，我们将展示如何利用 ZooKeeper 实现一些协调原语。

综上所述，本文的主要贡献在于

协调内核：我们为分布式系统提出了一种具有宽松一致性保证的免等待协调服务。我们特别介绍了*协调内核的*设计和实现，我们已在许多关键应用中使用该*内核*实现了各种协调技术。

协调秘诀：我们展示了如何利用 ZooKeeper 构建更高级别的协调基元，甚至是分布式应用中常用的阻塞和强一致性基元。**协调经验：**我们分享了一些使用 ZooKeeper 的方法，并评估了其每性能。

2 ZooKeeper 服务

客户端使用 ZooKeeper 客户端库，通过客户端 API 向 ZooKeeper 提交请求。除了通过客户端 API 提供 ZooKeeper 服务接口外，客户端库还管理客户端与 ZooKeeper 服务器之间的网络连接。

在本节中，我们将首先介绍 ZooKeeper 服务的高级视图。然后，我们将讨论客户端与 ZooKeeper 交互所使用的应用程序接口。

术语。在本文中，我们用*客户端来*表示 ZooKeeper 服务的用户，用*服务器*来表示提供 ZooKeeper 服务的进程，用 *znode* 来表示 ZooKeeper *数据*中的内存数据节点。我们还用更新和写入来指代任何修改数据树状态的操作。客户端在连接到 ZooKeeper 时会建立一个*会话，*并获得一个会话句柄，通过它来发出请求。

2.1 服务概述

ZooKeeper 为其客户端提供了一组数据节点（znodes）的抽象概念，这些节点按照层次名称空间进行组织。这个层次结构中的 znodes 是数据对象，客户可通过 ZooKeeper API 对其进行操作。分层名称空间通常用于文件系统。这是一种理想的数据对象组织方式，因为用户已经习惯了这种抽象方式，而且它还能更好地组织应用程序元数据。要引用一个

对于给定的 znode，我们使用标准的 UNIX 符号来表示文件系统路径。例如，我们使用 `/A/B/C` 表示到 znode `C` 的路径，其中 `C` 的父节点是 `B`，而 `B` 的父节点是 `A`。所有 z 节点都存储数据，除了短暂的 z 节点外，所有 z 节点都可以有子节点。

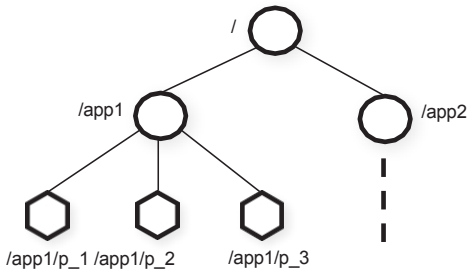


图 1：ZooKeeper 分级名称空间示意图。

客户可以创建两种类型的 znode：

常规：客户端通过显式创建和删除来操作常规 znode；

短暂：客户端创建此类 znode 后，会明确删除它们，或者让系统在创建它们的会话终止时（故意或由于故障）自动删除它们。

此外，在创建新的 znode 时，客户端可以设置*顺序*标志。在设置了序列标志后创建的节点，其名称后会附加一个单调递增的计数器值。如果 *n* 是新的 znode，而 *p* 是父 znode，那么 *n* 的序列值永远不会小于在 `p` 下创建的任何其他序列 znode 名称中的值。

ZooKeeper 实现了手表功能，允许客户端在不需要轮询的情况下及时接收变化通知。当客户端发出设置了监视标志的读取操作时，除了服务器承诺在返回的信息发生变化时通知客户端外，其他操作都会正常完成。监视是与会话相关联的一次性触发器；一旦触发或会话关闭，就不再注册。监视表明发生了变化，但不提供变化。例如，如果客户端在 `"/foo "` 被更改两次之前执行了 `getData(''/foo'', true)`，客户端将收到一个监视事件，告诉客户端"/foo "的数据已经更改。连接丢失事件等中断事件也会发送到监视回调，以便客户端知道监视事件可能会延迟。

数据模型ZooKeeper 的数据模型本质上是一个文件系统，具有简化的应用程序接口（API），只能进行全数据读取和写入，或者是一个键/值表，具有分层的数据结构。

分层名称空间分层名称空间可用于定位不同应用程序名称空间的子树，以及设置这些子树的访问权限。我们还利用客户端的目录概念来构建更高层次的基元，我们将在第 2.4 节中看到这一点。与文件系统中的文件不同，znodes 并不是为一般数据存储而设计的。相反，znodes 映射的是客户端应用程序的节段，通常与用于协调目的的元数据相对应。举例说明，在图 1 中，我们有两个子树，一个是应用程序 1 的子树（/app1），另一个是应用程序 2 的子树（/app2）。应用 1 的子树实现了一个简单的群成员协议：每个客户进程 p_i 在 /app1 下创建一个 znode p i，只要该客户进程在 /app2 下创建了一个 znode p i，该 znode 就会一直存在。进程正在运行。

虽然 ZooKeeper 并不是为一般数据存储而设计的，但它确实允许客户端存储一些可用于分布式计算中的元数据或配置的信息。例如，在基于领导者的应用中，对于刚刚开始工作的应用服务器来说，了解当前哪个服务器是领导者是非常有用的。为了实现这一目标，我们可以让当前的领导者将这一信息写入 znode 空间中的已知位置。znode 还具有与时间戳和版本计数器相关联的元数据，这使得客户端可以跟踪 znode 的变化，并根据 znode 的版本执行相应的更新。

会话客户端连接到 ZooKeeper 并启动会话。会话有一个相关的超时时间。如果客户端在超时后仍未从会话中接收到任何信息，ZooKeeper 就会认为该客户端有问题。当客户端显式关闭会话句柄或 ZooKeeper 检测到客户端有问题时，会话就会结束。在会话期间，客户端会观察到一系列状态变化，这些变化反映了客户端操作的执行情况。会话能让客户端在 ZooKeeper 集群中以透明方式从一个服务器移动到另一个服务器，从而在不同的 ZooKeeper 服务器之间持续存在。

2.2 客户端 API

下面我们将介绍 ZooKeeper API 的相关子集，并讨论每个请求的语义。 **create(path, data, flags)**：创建一个 znode 的路径名，并在其中存储数据[]，以及返回新 znode 的名称。flags 使客户端能够选择 znode 的类型：常规、短暂，并设置顺序标志；

delete(path, version)：删除 如果该 znode 处于预期版本，则删除该 znode **路径**；

exists(path, watch)： **如果 znode 处于**预期版本，则返回 true：如果 znode 则返回 false。观察标记可让客户端设置

在 znode 上进行监视；

getData(path, watch)： 返回与 znode 相关的数据和元数据，如版本信息。`watch` 标志的作用与 `exists()` 相同，只是如果 znode 不存在，Zoo- Keeper 不会设置 watch；

setData(path,数据,版本)： 如果版本号是 znode 的当前版本，则向 znode `path` 写入 `data[]`；

getChildren(path, watch)： 返回 znode 的子节点名称集；

sync(路径)： 等待操作开始时所有待处理的更新传播到客户端连接的服务器。路径目前被忽略。

所有方法都有同步和异步两种。同步版本可通过应用程序接口（API）获得。当应用程序需要执行单个 ZooKeeper 操作且没有并发任务要执行时，它就会使用同步 API，进行必要的 ZooKeeper 调用并阻塞。而异步应用程序接口（asynchronous API）则能让应用程序同时执行多个未完成的 ZooKeeper 操作和其他任务。ZooKeeper 客户端保证，每个操作的相应回调都会被调用。

请注意，ZooKeeper 不使用句柄来访问 znode。相反，每个请求都包含被操作znode的完整路径。这一选择不仅简化了应用程序接口（没有 `open()` 或 `close()` 方法），还消除了服务器需要维护的额外状态。

每种更新方法都有一个预期的版本号，这样就能实现相应的更新。如果zn-ode的实际版本号与预期版本号不符，更新就会失败，并出现意外版本错误。如果版本号为-1，则不执行版本检查。

2.3 ZooKeeper 保证

ZooKeeper 有两种基本的排序保证：

可线性化写入： 所有更新 ZooKeeper 状态的请求都是可序列化的，并尊重先验性；

先进先出的客户顺序： 来自某个客户的所有请求都按照客户发送的顺序执行。

请注意，我们对可线性化的定义与 Herlihy [15] 最初提出的定义不同，我们称之为 *A-linearizability*（异步可线性化）。在 Her- lihy 最初提出的线性化定义中，一个客户端一次只能有一个未完成的操作（一个客户端就是一个线程）。在我们的定义中，我们允许一个

如果客户端有多个未完成操作，我们可以选择保证同一客户端的未完成操作没有特定顺序，或者保证先进先出顺序。对于我们的特性，我们选择后者。需要注意的是，所有对可线性化对象成立的结果对 A-可线性化对象同样成立，因为满足 A-可线性化的系统也满足可线性化。因为只有更新请求是可线性化的，所以 ZooKeeper 会在每个副本本地处理读取请求。这样，当系统中增加服务器时，服务就可以线性扩展。

要了解这两种保证是如何相互作用的，请考虑以下情况。一个由多个进程组成的系统会选出一个领导者来指挥工作进程。当新的领导者掌管系统时，它必须更改大量的配置参数，并在完成更改后通知其他进程。这样，我们就有了两个重要的要求：

- 当新领导者开始进行更改时，我们不希望其他进程开始使用正在更改的配置；
- 如果新领导者在配置完全更新前死亡，我们不希望进程使用这个部分配置。

需要注意的是，分布式锁（如 Chubby 提供的锁）有助于满足第一项要求，但不足以满足第二项要求。在 ZooKeeper 中， 新的领导者可以指定一个路径作为*就绪*的 znode；其他进程只有在该 znode 存在时才会使用配置。新领导者通过删除*就绪*、更新各种配置 znode 和创建*就绪来*更改配置。所有这些更改都可以流水线方式异步发布，以快速更新配置状态。虽然更改操作的延迟时间约为 2 毫秒，但如果一个新的领导者必须更新 5000 个不同的 Z 节点，那么如果一个接一个地发出请求，将需要 10 秒钟；而如果异步发出请求，所需时间将少于一秒。由于存在排序保证，如果进程看到*已就绪*的 znode，它也必须看到新领导者所做的所有配置更改。如果新的领导者在创建*就绪*的 znode 之前死亡，其他进程就会知道配置尚未最终确定，并不会使用它。

上述方案仍然存在一个问题：如果一个进程在新的领导者开始进行变更之前看到*准备就绪*，然后在变更进行时开始读取配置，会发生什么情况？这个问题可以通过通知的排序保证来解决：如果客户端正在关注变化，那么客户端会在看到变化后的系统新状态之前看到通知事件。因此，如果读取*准备就绪*的 znode 的进程请求获得该 znode 的变更通知，它将会看到一个通知信息--"....."。

在客户端读取任何新配置之前，将更改通知客户端。

当客户端除了 ZooKeeper 之外还有自己的通信通道时，就会出现另一个问题。例如，考虑两个客户端 *A* 和 *B*，它们在 ZooKeeper 中拥有共享配置，并通过共享通信频道进行通信。如果 *A* 更改了 ZooKeeper 中的共享配置，并通过共享通信通道将更改信息告知 *B*，那么 *B* 在重新读取配置时就会看到更改信息。如果 *B* 的 ZooKeeper 副本稍稍落后于 *A 的* ZooKeeper 副本，它可能看不到新配置。利用上述保证，*B* 可以在重新读取配置之前发出写入命令，确保看到最新信息。为了更有效地处理这种情况，Zoo- Keeper 提供了同步请求：当紧随读取之后时，就构成了*慢速*读取。同步会导致服务器在处理读取之前应用所有待处理的写入请求，而不会产生完全写入的开销。这个基元与 ISIS 的 `flush` 基元（`flush` primitive）理念相似[5]。

ZooKeeper 还具有以下两种有效性和持久性保证：如果大多数 ZooKeeper 服务器处于活动状态并进行通信，则服务可用；如果 ZooKeeper 服务成功响应了更改请求，则只要有足够数量的服务器能够恢复，该更改就会在发生故障后继续存在。

2.4 基元示例

在本节中，我们将展示如何使用 ZooKeeper API 来实现更强大的原语。ZooKeeper 服务对这些功能更强大的原语一无所知，因为它们完全是在客户端使用 ZooKeeper 客户端 API 实现的。一些常见的原语（如群成员资格和配置管理）也是无需等待的。而对于其他原语，如会合，客户端则需要等待事件发生。尽管 ZooKeeper 是免等待的，但我们仍可使用 ZooKeeper 实现高效的阻塞基元。ZooKeeper 的排序保证允许高效推理系统状态，而手表则允许高效等待。

配置管理 ZooKeeper 可用于在分布式应用中实现动态配置。在最简单的形式中，配置存储在一个 z 节点（z_c）中。进程启动时使用的是 z_c 的完整路径名。启动进程通过读取 z_c 并将观察标志设为 true 来获取配置。如果 z_c 中的配置被更新，进程会收到通知并读取新配置，同时再次将监视标志设为 true。

请注意，在这个方案中，和其他大多数使用手表的方案一样，手表是用来确保一个进程已经

最新信息。例如，如果监视 z_c 的进程收到了 z_c 的变更通知，而在它对 z_c 发出读取之前，z_c 又发生了三次变更，那么该进程就不会再收到三次通知事件。这不会影响进程的行为，因为这三个事件只是通知进程它已经知道的事情：它所掌握的 z_c 的信息已经过时。

会合 有时，在分布式系统中，最终的系统配置并不总是先验的。例如，客户端可能想要启动一个主进程和多个工作进程，但启动进程的工作是由调度程序来完成的，因此客户端无法提前知道工作进程可以连接到主进程的广告和端口等信息。我们在使用 Zoo- Keeper 时会使用一个会合节点（rendezvous znode，z_r）来处理这种情况，该节点是由客户端创建的。客户端将 z_r 的完整路径名作为主进程和工作进程的启动参数。主进程启动时，会在 z_r 中填写它正在使用的地址和端口。当工作者进程启动时，它们会读取 z_r，并将 watch 设置为 true。如果 z_r 尚未填入，则工作者会等待 z_r 更新时收到通知。如果 z_r 是一个短暂节点，主进程和工作进程就会观察 z_r 是否被删除，并在客户端结束时清理自己。

群成员资格 我们利用短暂节点来实现群成员资格。具体来说，我们利用短暂节点允许我们查看创建节点的会话状态这一事实。我们首先指定一个 znode（z_g）来代表组。当组中的一个进程成员启动时，它会在 z_g 下创建一个短暂子 znode。如果每个进程都有一个唯一的名称或标识符，那么该名称就会被用作子 znode 的名称；否则，进程在创建 znode 时就会使用 SEQUENTIAL 标志来获得唯一的名称分配。进程可在子 znode 的数据中加入进程信息，例如进程使用的地址和端口。

在 z_g 下创建子 znode 后，进程将正常启动。它不需要做任何其他事情。如果进程失败或结束，代表它的 znode 会自动从 z_g 中移除。

进程可以通过简单地列出 z_g 的子进程来获取群组信息。如果进程想监控群组成员的变化，可以将监视标志设为 true，并在收到变化通知时刷新群组信息（也就是将监视标志设为 true）。

简单锁 虽然 ZooKeeper 并不是一个锁服务，但它可以用来实现锁。使用 ZooKeeper 的应用程序通常会根据自身需要使用同步原语，如上图所示。在这里，我们将展示如何使用 ZooKeeper 实现锁，以说明它可以实现各种通用的同步原语。

最简单的锁实现方式是使用 "锁文件"。锁由一个 znode 表示。为获得锁，客户端会尝试创建带有 EPHEMERAL 标志的指定 znode。如果创建成功，客户端就持有锁。否则，客户端可以通过设置观察标记来读取 zn-ode，以便在当前领导者死亡时得到通知。客户端在死亡或删除 znode 时会释放锁。其他正在等待锁的客户端一旦发现 znode 被删除，就会再次尝试获取锁。

这种简单的锁定协议虽然有效，但也存在一些问题。首先，它存在羊群效应。如果有很多客户端在等待获取锁，那么当锁被释放时，即使只有一个客户端可以获取锁，它们也会争抢。其次，它只能实现排他性锁定。下面两个原语展示了如何克服这两个问题。

无群集效应的简单锁 我们定义了一个锁 znode *l* 来实现这种锁。直观地说，我们将所有请求锁的客户端排成一排，每个客户端按照请求到达的顺序获取锁。因此，希望获得锁的客户端会做如下操作：

锁

```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 如果 n 是 C 中最低的 z 节点，则退出
4 p = C 中在 n 之前排序的 z 节点
5 如果 exists(p, true) 等待观察事件
6 第 2 项
```

解锁

```
1 删除(n)
```

在锁的第 1 行中使用 SEQUENTIAL 标志，会对客户端获取锁的尝试进行排序，并对所有其他尝试进行排序。如果客户端的 zn 节点在第 3 行的序列号最低，则客户端持有锁。否则，客户端将等待删除拥有锁或将在该客户端的 znode 之前获得锁的 zn-ode。通过只监视客户端 znode 之前的 znode，我们只在锁被释放或锁请求被放弃时唤醒一个进程，从而避免了羊群效应。一旦被客户端监视的 znode 消失，客户端必须检查它现在是否持有锁。(上一个锁请求可能已被放弃，而另一个序列号更低的 znode 仍在等待或持有该锁）。

释放锁就像删除表示锁请求的 zn-ode *n* 一样简单。 通过使用

在创建进程时使用 EPHEMERAL 标志，崩溃的进程将自动清理所有锁请求或释放所有锁。

总之，这种锁定方案有以下优点：

1. 移除一个 znode 只会导致一个客户端唤醒，因为每个 znode 都会被另外一个客户端监视，所以我们不会产生羊群效应；
2. 没有轮询或超时；
3. 由于我们采用了加锁的方式，因此我们可以通过浏览 ZooKeeper 的数据来查看锁竞争的数量、断开锁以及调试加锁问题。

读/写锁 为了实现读/写锁，我们对锁定过程稍作改动，将读锁和写锁过程分开。解锁程序与全局锁相同。

写入锁

```
1  n = create(l + "/write-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  如果 n 是 C 中最低的 z 节点，则退出
4  p = C 中在 n 之前排序的 z 节点
5  如果 exists(p, true) 等待事件发生
6  第 2 项
```

阅读锁

```
1  n = create(l + "/read-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  如果 C 中没有低于 n 的写 z 节点，则退出
4  p = 在 C 中写入在 n 之前排序的 z 节点
5  如果 exists(p, true) 等待事件发生
6  第 3 项
```

此锁定程序与之前的锁定略有不同。写锁只在命名上有所不同。由于读锁可以共享，第 3 行和第 4 行略有不同，因为只有较早的写锁节点才能阻止客户端获得读锁。如果有多个客户端在等待读锁，并在序列号较低的 "写 "节点被删除时收到通知，那么我们可能会出现 "羊群效应"；事实上，这是一种理想的行为，所有这些读客户端都应该被释放，因为他们现在可能已经拥有了锁。

双屏障 双屏障可让客户端同步计算的开始和结束。当有足够多的进程（由屏障阈值定义）加入屏障时，进程就会开始计算，并在计算结束后离开屏障。在 ZooKeeper 中，我们用一个 znode（称为 *b*）来表示屏障。每个进程 *p* 在进入屏障时，都会通过创建一个 znode 作为 *b* 的子节点来注册 *b*，并在准备离开时取消注册，重新移动子节点。当 *b* 的子节点数量超过屏障阈值时，进程就可以进入屏障。当所有进程都移走其子节点后，进程就可以离开屏障。我们使用手表来有效地等待进入和离开障碍。

退出条件必须满足。要进入时，进程会观察是否存在 *b* 的准备就绪的子节点，该子节点将由导致子节点数量超过障碍阈值的进程创建。要退出时，进程会密切关注某个子节点的消失，只有在该子节点被删除后才会检查退出条件。

3 ZooKeeper 应用程序

现在，我们将介绍一些使用 ZooKeeper 的应用程序，并简要说明它们是如何使用 ZooKeeper 的。我们用**粗体**显示每个示例的主要内容。

抓取服务 抓取是搜索引擎的重要组成部分，雅虎抓取了数十亿的网络文档。抓取服务（FS）是雅虎抓取程序的一部分，目前已投入使用。从根本上说，它拥有指挥页面抓取过程的主进程。主进程为抓取器提供配置，抓取器则回传其状态和健康状况。在 FS 中使用 ZooKeeper 的主要优点是，可以从主进程的故障中恢复、保证故障后的可用性，以及将客户端与服务器解耦，使客户端只需从 ZooKeeper 中读取服务器的状态，就能将其重新搜索导向健康的服务器。因此，FS 主要使用 ZooKeeper 来管理**配置元数据**，尽管它也使用 Zoo Keeper 来选举主服务器（**领导者选举**）。

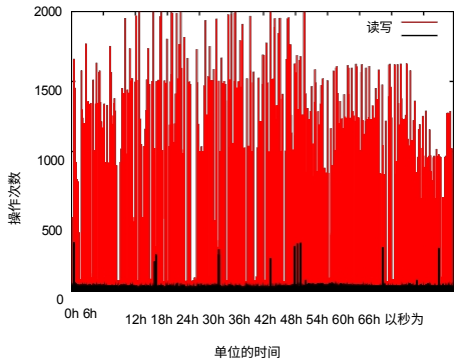


图 2：一台 ZK 服务器使用获取服务的工作量。每个点代表一秒钟的样本。

Katta Katta [17] 是一个使用 Zoo- Keeper 进行协调的分布式索引器，它是一个非乐虎国际手机版下载应用的例子。Katta 使用分片来划分索引工作。主服务器将分片分配给从服务器，并跟踪进度。从服务器可能会出现故障，因此主服务器必须根据从服务器的去留重新分配负载。主服务器也可能出现故障，因此其他服务器必须做好准备，以防万一。Katta 使用 ZooKeeper 来跟踪从属服务器和主服务器的状态（**群成员**），并处理主服务器的故障切换（**领导者选举**）。Katta 还使用 ZooKeeper 跟踪和传播从服务器的分片分配（**配置管理**）。

雅虎消息代理 雅虎消息代理（YMB）是一个分布式发布-订阅系统。该系统管理着成千上万的主题，客户可以向这些主题发布消息，也可以从这些主题接收消息。这些主题分布在一组服务器中，以提供可扩展性。每个主题都采用主备方案进行复制，确保信息复制到两台服务器上，从而保证信息传递的可靠性。组成 YMB 的服务器采用无共享分布式架构，因此协调对于正确运行至关重要。YMB 使用 ZooKeeper 来管理主题（**配置元数据**）的分布，处理系统中机器的故障（**故障检测**和**群成员资格**），并控制系统的运行。

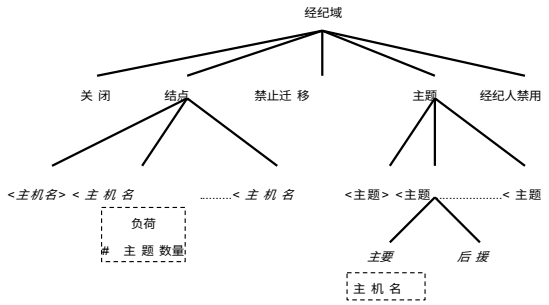


图 3：ZooKeeper 中雅虎消息代理 (YMB) 结构的布局

图 2 显示了 FS 使用的 Zoo- Keeper 服务器在三天内的读写流量。为生成此图，我们计算了这段时间内每秒的操作次数，每个点对应该秒内的操作次数。我们发现，读取流量远高于写入流量。在速率高于每秒 1,000 次操作的期间，读写比率介于 10:1 和 100:1 之间。该工作负载中的读操作依次为 `getData()`、`getChildren()` 和 `exists()`。

图 3 显示了 YMB 的部分 znode 数据布局。每个代理域都有一个名为 "节点 "的 znode，每个组成 YMB 服务的活动服务器都有一个短暂的 znode。每个 YMB 服务器都会在节点下创建一个短暂的 znode，并通过 ZooKeeper 提供负载和状态信息，包括群组成员和状态信息。禁止关闭和迁移等节点由组成服务的所有服务器监控，从而实现对 YMB 的集中控制。每个由 YMB 管理的主题都有一个子 Znode。这些主题 Znode 的子 Znode 表示

每个主题的主服务器和备份服务器以及该主题的订阅者。主服务器和备份服务器 znode 不仅能让服务器发现负责某个主题的服务器，还能管理**领导者选举**和服务器崩溃。

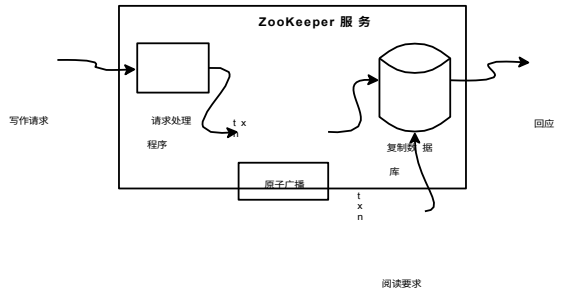


图 4：ZooKeeper 服务的组成部分。

4 ZooKeeper 实现

ZooKeeper 通过在组成服务器的每台服务器上复制 ZooKeeper 数据来提供高可用性。我们假设服务器会因崩溃而发生故障，而这些故障服务器随后可能会恢复。图 4 显示了 ZooKeeper 服务的高级组件。

收到请求后，服务器会准备执行请求（请求处理器）。如果该请求需要服务器之间的协调（写入请求），那么它们就会使用协议协议（原子宽投的实现），最后服务器会将更改提交到 Zoo Keeper 数据库中，该数据库会在集合的所有服务器之间完全复制。在读取请求的情况下，服务器只需读取本地数据库的状态，然后生成对请求的响应。

复制数据库是一个*内存*数据库，包含整个数据树。默认情况下，数据树中的每个 znode 最多存储 1MB 的数据，但这个最大值是一个配置参数，在特定情况下可以更改。为了保证可恢复性，我们将上载数据有效地记录到磁盘上，并在写入内存数据库之前强制写入磁盘介质。事实上，与 Chubby [8] 一样，我们也会保留已提交操作的重放日志（在我们由领导者的状态变化组成的信息提议，并就状态变化达成一致。

4.1 请求处理程序

由于消息传递层是原子层，我们保证本地副本永远不会出现分歧，尽管在
某些服务器可能比其他服务器应用了更多的事务。与客户端发送的请求不同，事务是*惰性的*。当领导者收到写入请求时，它会计算写入时系统的状态，并将其转换为事务，以捕捉新状态。事务
的例子中是先写日志），并定期生成内存数据库快照。

每台 ZooKeeper 服务器都为客户端提供服务。客户端只连接到一台服务器来提交请求。如前所述，读取请求由每个服务器数据库的本地副本提供服务。改变服务状态的请求，即写入请求，则由协议协议处理。

作为协议的一部分，写入请求被转发给一台服务器，称为*领导者*[1]。其余的 ZooKeeper 服务器（称为*跟随者*）则接收写入请求。

1 作为协议规程的一部分，领导者和追随者的详细情况不在本文讨论范围之内。

由于可能存在尚未应用到数据库的现有事务，因此必须计算数据库状态。例如，如果客户端执行了有条件的 `setData`，且请求中的版本号与要更新的 znode 的未来版本号相匹配，服务就会生成一个 `setDataTXN`，其中包含新数据、新版本号和更新的时间戳。如果出现错误，如版本号不匹配或要更新的 znode 不存在，则会生成 `errorTXN`。

4.2 原子广播

所有更新 ZooKeeper 状态的请求都会转发给领导者。领导者执行请求，并通过原子广播协议 Zab [24] 广播 ZooKeeper 状态的变化。收到客户端请求的服务器会在发送相应的状态变化时回应客户端。Zab 使用去错误简单多数法定人数来决定提议，因此 Zab 以及 ZooKeeper 只有在多数服务器正确的情况下才能工作（*即* $2f + 1$ 个服务器，我们可以容忍 f *次*失败）。

为了实现高吞吐量，ZooKeeper 会尽量保持请求处理管道满载。在处理管道的不同部分可能会有成千上万个请求。由于状态变化取决于先前状态变化的应用，Zab 提供了比普通原子广播更强的顺序保证。更具体地说，Zab 保证领导者广播的变更会按照发送的顺序进行传递，并且在已建立的领导者广播自己的变更之前，会将之前领导者的所有变更传递给它。

有一些实现细节简化了我们的实现，并为我们带来了卓越的性能。我们使用 TCP 进行传输，因此消息顺序由网络保持，这让我们可以简化实现过程。我们使用 Zab 选择的领导者作为 ZooKeeper 的领导者，这样创建事务的进程也会提出事务。我们使用日志来跟踪提议，并将其作为 ZooKeeper 中的写前日志。

内存数据库，这样我们就不必将信息两次写入磁盘。

在正常运行过程中，Zab 会按顺序准确地传递所有信息一次，但由于 Zab 不会持久地记录每条已传递信息的 ID，因此在恢复过程中，Zab 可能会重新传递一条信息。因为我们使用的是幂等事务，所以只要按顺序传递，就可以接受多次传递。事实上，ZooKeeper 要求 Zab 至少重新传递在上次快照开始后传递的所有信息。

4.3 复制数据库

每个副本的内存中都有一份 ZooKeeper 状态副本。当 ZooKeeper 服务器从崩溃中恢复时，它需要恢复内部状态。在服务器运行一段时间后，重放所有已发送的消息来恢复状态会耗时过长，因此 ZooKeeper 使用周期性快照，只要求重新发送快照开始后的消息。我们称 ZooKeeper 快照为*模糊快照*，因为我们在拍摄快照时不会锁定 ZooKeeper 的状态；相反，我们会对树进行深度扫描，原子式读取每个 Zn- ode 的数据和元数据，并将它们写入磁盘。由于生成的模糊快照可能应用了快照生成过程中传递的状态变化的某些子集，因此其结果可能与 ZooKeeper 在任何时间点的状态都不一致。不过，由于状态变化是幂等的，只要我们按顺序应用状态变化，就可以应用两次。

例如，假设在 ZooKeeper 数据树中，两个节点 `/foo` 和 `/goo` 的值分别为 `f1` 和 `g1`，在模糊快照开始时，这两个节点都处于版本 1，下面的状态变化流以⟨transactionType、`path`、`value`、`new-version`⟩的形式到达：

⟨*SetDataTXN*, `/foo`, `f2`, *2*⟩
⟨*SetDataTXN*, `/goo`, `g2`, *2*⟩
⟨*SetDataTXN*, `/foo`, `f3`, *3*⟩

处理完这些状态变化后，`/foo` 和 `/goo` 的值分别为 `f3` 和 `g2`，版本分别为 3 和 2。然而，模糊快照可能记录了 `/foo` 和 `/goo` 的值 `f3` 和 `g1`，版本分别为 3 和 1，这不是 ZooKeeper 数据树的有效状态。如果服务器崩溃并使用该快照恢复，Zab 重新传递状态更改，则生成的状态与服务器崩溃前的状态一致。

4.4 客户端与服务器的交互

当服务器处理写入请求时，它还会发送和清除与任何手表相关的通知。

的更新。服务器按顺序处理写入，不会同时处理其他写入或读取。这确保了通知的严格连续性。请注意，服务器在本地处理通知。只有客户端连接的服务器才会跟踪和触发该客户端的通知。

读取请求由每个服务器本地处理。每个读取请求都会被处理并标记一个 *zxid*，该 zxid 与服务器看到的最后一个事务相关。该 *zxid* 定义了读取请求的部分顺序，并与写入请求重合。通过本地处理读取，我们可以获得出色的读取性能，因为这只是本地服务器上的内存内操作，无需运行磁盘活动或协议协议。这一设计选择是实现我们的目标--在读取占主导地位的工作负载中实现卓越性能的关键。

使用快速读取的一个缺点是无法保证读取操作的优先顺序。也就是说，读操作可能会返回一个陈旧的值，即使同一 znode 的最新更新已经提交。并非所有应用都需要优先顺序，但对于需要优先顺序的应用，我们实现了同步。这个基元是异步执行的，由领导者在向其下级副本写入所有待定内容后排序。为保证读取操作能重新读取最新更新的值，客户端会在读取操作后调用同步。客户端操作的先进先出顺序保证与同步的全局保证一起，使读取操作的结果能够反映同步发出之前发生的任何变化。在我们的实现中，由于我们使用的是基于领导者的算法，因此不需要原子式广播同步，我们只需将同步操作放在领导者和服务器之间请求队列的末尾，然后调用同步。为了使同步操作有效，跟随者必须确定领导者仍然是领导者。如果有提交的待处理事务，服务器就不会怀疑领导者。如果待处理队列为空，则领导者需要发布一个空事务来提交，并在该事务之后下令同步。这样做的好处是，当领导者处于负载状态时，不会产生额外的广播流量。在我们的实现中，超时的设置是为了让领导者在追随者抛弃他们之前意识到自己不是领导者，因此我们不会发布空事务。

ZooKeeper 服务器以先进先出的顺序处理来自客户端的请求。响应包括响应相对的 *zxid*。即使是在无活动期间的心跳信息，也包括客户端所连接的服务器看到的最后一个*zxid*。如果客户端连接到新的服务器，新服务器会通过检查客户端的最后一个*zxid*和它的最后一个*zxid*，确保它对Zoo- Keeper数据的视图至少和客户端的视图一样新。如果客户端的视图比服务器的视图更新，则

在服务器跟上之前，服务器不会与客户端重新建立会话。由于客户端只能看到复制到大多数 ZooKeeper 服务器上的更改，因此可以保证客户端能找到另一个拥有系统最新视图的服务器。这种行为对于保证持久性非常重要。

ZooKeeper 使用超时来检测客户端会话失败。如果在会话超时时间内，其他服务器没有收到任何来自客户端会话的信息，领导者就会判定会话失败。如果客户端发送再请求的频率足够高，那么就不需要发送任何其他信息。否则，客户端会在活动较少期间发送心跳信息。如果客户端无法与服务器通信以发送请求或心跳信息，它就会连接到另一台 ZooKeeper 服务器，重新建立会话。为了防止会话超时，ZooKeeper 客户端库会在会话空闲 s/3 毫秒后发送心跳信息，如果在 2s/3 毫秒（其中 *s* 是会话超时时间，单位为毫秒）内没有服务器的消息，就会切换到新的服务器。

5 评估

我们在一个由 50 台服务器组成的集群上进行了所有评估。每台服务器都有一个至强双核 2.1GHz 处理器、4GB 内存、千兆以太网和两个 SATA 硬盘。我们将下面的讨论分为两部分：请求的吞吐量和延迟。

5.1 吞吐量

为了评估我们的系统，我们对系统饱和时的吞吐量以及各种注入故障时的吞吐量变化进行了基准测试。我们改变了组成 ZooKeeper 服务的服务器数量，但始终保持客户端数量不变。为了模拟大量客户端，我们使用 35 台机器同时模拟 250 个客户端。

我们有一个 ZooKeeper 服务器的 Java 实现，以及 Java 和 C 客户端[2]。在这些实验中，我们使用的 Java 服务器配置为在一个专用磁盘上记录日志，并在另一个磁盘上拍摄快照。我们的基准客户端使用异步 Java 客户端 API，每个客户端至少有 100 个未处理请求。每个请求包括 1K 数据的读取或写入。我们没有显示其他操作的基准，因为所有修改状态的操作性能大致相同，而不修改状态的操作（不包括同步）性能也大致相同。（同步的性能近似于轻量级写入，因为请求必须

[2] 该实现可在 http://hadoop. apache.org/zookeeper 上公开获取。

但不会被广播）。客户端每 *300* 毫秒发送一次已完成操作的计数，我们每 *6* 秒采样一次。为防止内存溢出，服务器会对系统中的并发再请求数量进行节流。ZooKeeper 使用请求节流来防止服务器不堪重负。在这些实验中，我们将 ZooKeeper 服务器配置为最多处理 2,000 个总请求。

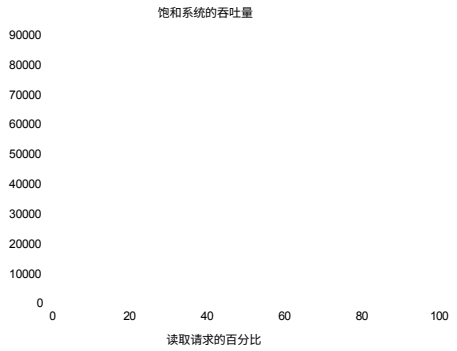


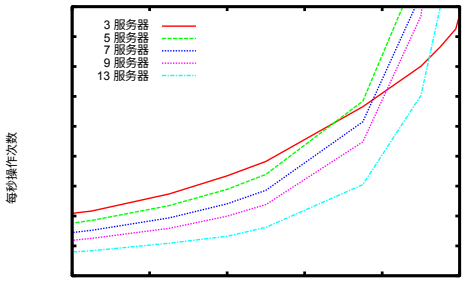
图 5：读写比例变化时饱和系统的吞吐量性能。

服务器	100% **阅读**	0% **阅读次数**
13	460k	8k
9	296k	12k
7	257k	14k
5	165k	18k
3	87k	21k

表 1：饱和系统极端情况下的吞吐量性能。

图 5 显示了读写请求比例变化时的吞吐量，每条曲线对应提供 ZooKeeper 服务的不同服务器数量。表 1 显示了读取负载极端情况下的数据。由于读取不使用原子广播，因此读取吞吐量高于写入吞吐量。图表还显示，服务器数量也会对广播协议的性能产生负面影响。从这些图表中我们可以看出，系统中服务器的数量不仅会影响服务可处理的故障数量，还会影响服务可处理的工作量。请注意，三台服务器的曲线在 60% 左右与其他服务器的曲线交叉。这种情况并非三台服务器配置所独有，由于本地读取的并行性，所有配置都会出现这种情况。不过，图中的其他配置无法观察到这种情况，因为为了便于阅读，我们对 y 轴的最大吞吐量设置了上限。

写入请求比读取请求耗时长有两个原因。首先，写请求必须经过原子广播，这需要一些额外的处理



并增加请求的延迟。写入请求处理时间较长的另一个原因是，服务器在向领导者发送回执之前，必须确保将事务记录到非易失性存储区。在实际应用中，这一要求是过高的，但对于我们的生产系统来说，由于 ZooKeeper 构成了应用的基本事实，因此我们用性能来换取可靠性。我们使用更多的服务器来容忍更多的故障。我们通过将 ZooKeeper 数据划分为多个分区来提高写入吞吐量。

复制和分区之间的性能折衷。Gray 等人[12]曾观察到复制与分区之间的这种性能折衷。

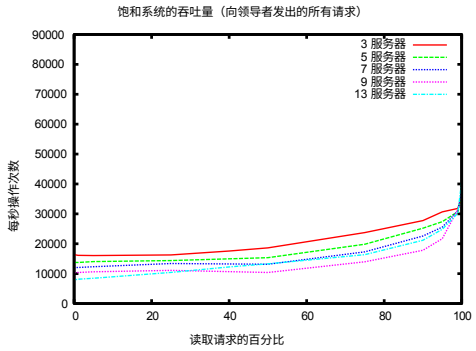


图 6：饱和系统的吞吐量，当所有客户端都连接到领导者时，改变读写比例。

ZooKeeper 能够实现如此高的吞吐量，靠的是在组成服务器的各个服务器之间分配负载。我们之所以能分散负载，是因为我们放宽了一致性保证。Chubby 客户端会将所有请求指向领导者。图 6 显示了如果我们不利用这种松散性，而迫使客户端只连接到领导者时的情况。不出所料，以读取为主的工作负载的吞吐量要低得多、

但即使对于以写入为主的工作负载，吞吐量也较低。为客户端提供服务所造成的额外 CPU 和网络负载会影响领导者协调建议广播的能力，进而严重影响整体写入性能。

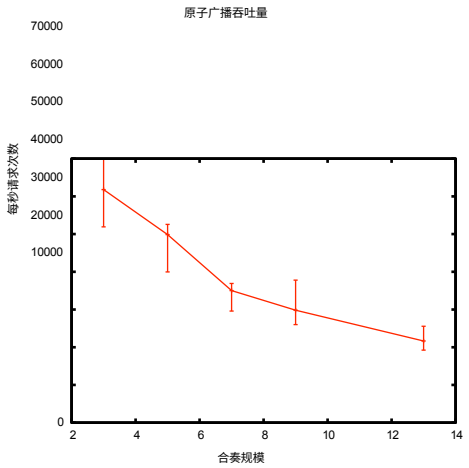
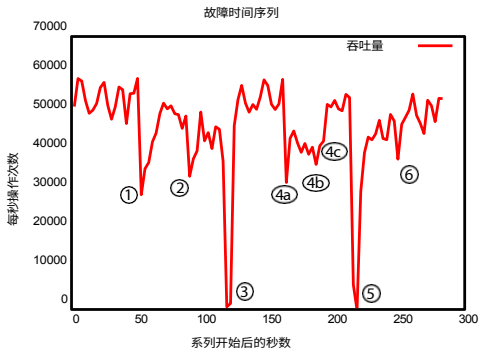


图 7：独立原子广播组件的平均吞吐量。误差条表示最小值和最大值。

所有版本都需要 CPU。对 CPU 的争夺使 ZooKeeper 的吞吐量大大低于独立的原子广播组件。由于 ZooKeeper 是一个关键的生产组件，到目前为止，我们对 ZooKeeper 的开发重点一直是正确性和鲁棒性。通过消除额外的副本、同一目标的多重序列化、更高效的内部数据结构等，我们有很多机会大幅提高性能。



原子广播协议承担了系统的大部分工作，因此对 Zoo- Keeper 性能的限制比其他任何组件都要大。图 7 显示了原子广播组件的吞吐量。为了测试其性能，我们通过直接在领导者处生成事务来模拟客户端，

因此不存在客户端连接或客户端请求和回复。在最大吞吐量时，原子广播组件会受到 CPU 的限制。理论上，图 7 的性能与 ZooKeeper 100% 写入的性能相当。然而，ZooKeeper 的客户端通信、ACL 检查以及请求与事务的连接都需要 CPU 来处理。

图 8：故障时的吞吐量。

为了展示系统在故障注入后的长期表现，我们运行了一个由 5 台机器组成的 ZooKeeper 服务。我们运行了与之前相同的饱和基准，但这次我们将写入比例保持在 恒定的 30%，这是我们预期工作负载的保守比例。我们定期杀死一些服务器进程。图 8 显示了随时间变化的系统吞吐量。图中标记的事件如下：

1. 追随者的失败与恢复；
2. 不同追随者的失败和恢复；
3. 领导者的失败；
4. 前两分中的两个追随者（a、b）失败，第三分（c）恢复；
5. 领导者的失败。

6. 领导者的复苏。

从这张图中可以观察到几个重要现象。首先，如果跟随者发生故障并迅速恢复，那么尽管发生故障，ZooKeeper 仍能维持较高的吞吐量。单个跟随者的故障不会导致服务器无法形成法定人数，而只会降低吞吐量，降低的幅度大致相当于该服务器在故障前正在处理的读取请求的份额。其次，我们的领导者选举算法能够快速恢复，足以防止吞吐量大幅下降。在我们的实验中，ZooKeeper 选举新领导者的时间不到 *200* 毫秒。因此，尽管服务器停止服务请求的时间只有几分之一秒，但由于我们的采样周期只有几秒，因此我们并没有观察到吞吐量为零的情况。第三，即使跟随者需要更多时间重新覆盖，ZooKeeper 也能在开始处理请求后再次提高吞吐量。在事件 1、2 和 4 发生后，我们的吞吐量未能完全恢复，原因之一是客户端只有在与追随者的连接中断时才会切换追随者。因此，在事件 4 之后，客户不会重新分配自己，直到领导者在事件 3 和 5 出现故障。实际上，这种不平衡会随着时间的推移，随着客户的来来往往而逐渐消除。

5.2 请求延迟

为了评估请求的延迟，我们仿照 Chubby 基准[6]创建了一个基准。我们创建了一个工作进程，该进程只需发送创建请求，等待创建完成后异步删除新节点，然后开始下一次创建。我们相应地改变工作者的数量，每次运行时，我们让每个工作者创建 50,000 个节点。我们用完成的创建请求数除以所有工作者完成创建所需的总时间来计算吞吐量。

	服务器数量			
工人	3	5	7	9
1	776	748	758	711
10	2074	1832	1572	1540
20	2740	2336	1934	1890

表 2：每秒处理的创建请求。

表 2 显示了我们的基准测试结果。创建的请求包括 1K 数据，而不是 Chubby 基准中的 5 字节，这样更符合我们的预期用途。即使请求量较大，ZooKeeper 的吞吐量也比 Chubby 公布的吞吐量高出 3 倍多。单个 ZooKeeper 工作者基准的吞吐量表明，三台服务器的平均请求延迟为 1.2 毫秒，而 Chubby 的平均请求延迟为 1.*5* 毫秒。
9 台服务器 1.4 毫秒。

	# 客户数量		
# 障碍数量	50	100	200
200	9.4	19.8	41.0
400	16.4	34.1	62.0
800	28.9	55.9	112.1
1600	54.0	102.7	234.4

表 3：以秒为单位的障碍实验时间。每个点都是每个客户端完成五次运行的平均时间。

5.3 障碍物的性能

在本实验中，我们连续执行了多个关卡，以评估使用 ZooKeeper 实现的基元性能。对于给定的屏障数 b，每个客户端首先进入所有 b 个屏障，然后依次离开所有 b 个屏障。由于我们使用的是第 2.4 节中的双屏障算法，一个客户端会先等待所有其他客户端执行 `enter()` 过程，然后再执行下一次调用（`leave()`过程也是如此）。

表 3 汇报了我们的实验结果。
在本实验中，我们让 50、100 和 200 个客户依次进入数量为 b 的障碍，$b\in$ 为 1。
{200, 400, 800, 1600}.虽然应用程序可以有
虽然 ZooKeeper 客户端数以千计，但参与每次协调操作的子集往往要小得多，因为客户端通常是根据应用程序的具体情况分组的。

从这个实验中可以观察到两个有趣的现象：处理所有障碍物的时间随着障碍物数量的增加而大致提前，这表明对数据树相同部分的并行访问不会造成任何意想不到的延迟；同时，延迟也会随着客户端数量的增加而成正比增加。这说明 ZooKeeper 服务并未达到饱和状态。事实上，我们观察到，即使客户端步调一致，障碍操作（进入和离开）的吞吐量在所有情况下都在每秒 1,950 到 3,100 次之间。在 ZooKeeper 操作中，这相当于每秒 10,700 到 17,000 次操作的吞吐量。在我们的实施过程中，读取与写入的比例为 4:1（80% 的读取操作），因此与 ZooKeeper 可达到的原始吞吐量（图 5 显示超过 40,000 次）相比，我们的基准代码所使用的吞吐量要低得多。这是因为客户端在等待其他客户端。

6 相关工作

ZooKeeper 的目标是提供一种服务，缓解分布式应用程序中的进程协调问题。为实现这一目标，ZooKeeper 的设计采用了以往协调服务、容错系统、分布式算法和文件系统的理念。

我们并不是第一个提出分布式应用协调系统的人。一些早期的系统为事务性应用提出了分布式锁服务[13]，并为计算机集群共享信息提出了分布式锁服务[19]。最近，Chubby 提出了一种为分布式应用程序管理咨询锁的系统 [6]。Chubby 与 Zoo- Keeper 有着相同的目标。它也有一个类似文件系统的界面，并使用协议协议来保证副本的一致性。不过，ZooKeeper 并不是一个锁服务。客户端可以用它来实现锁，但它的应用程序接口中没有锁操作。与 Chubby 不同，ZooKeeper 允许客户端连接到任何 ZooKeeper 服务器，而不仅仅是领导者。由于 ZooKeeper 的一致性模型比 Chubby 宽松得多，因此 ZooKeeper 客户端可以使用它们的本地副本来提供数据和管理手表。这样，ZooKeeper 就能提供比 Chubby 更高的性能，让应用程序能更广泛地使用 ZooKeeper。

文献中已经提出了容错系统，目的是缓解构建容错分布式应用程序的问题。早期的一个系统是 ISIS [5]。ISIS 系统将抽象类型规范转化为容错分布式对象，从而使容错机制对用户透明。Horus [30] 和 Ensemble [31] 是由 ISIS 演化而来的系统。ZooKeeper 采用了 ISIS 的虚拟同步概念。最后，Totem 在利用局域网硬件广播的架构中保证了信息传递的总顺序[22]。Zoo- Keeper 适用于多种网络拓扑结构，这促使我们依赖服务器进程之间的 TCP 连接，而不假定任何特殊的拓扑结构或硬件特性。此外，我们也不公开 ZooKeeper 内部使用的任何通讯方式。

构建容错服务的一项重要技术是状态机复制[26]，Paxos[20]是一种能够高效实现异步系统状态机复制的算法。我们使用的算法与 Paxos 的某些特性相同，但结合了共识所需的事务日志记录和数据树恢复所需的先写日志记录，从而实现了高效的实施。关于拜占庭耐受复制状态机的实际实施协议，已经有一些建议[7, 10, 18, 1, 28]。ZooKeeper 并不假定服务器可能是拜占庭的，但我们确实采用了诸如校验和与完整性检查等机制来捕捉非恶意的拜占庭故障。克莱门特 *等人*讨论了在不修改当前服务器代码基础的情况下使 ZooKeeper 实现完全拜占庭容错的方法 [9]。迄今为止，我们尚未在生产过程中观察到使用完全拜占庭容错协议可以避免的故障。[29].

Boxwood [21] 是一个使用分布式锁服务器的系统。Boxwood 为应用程序提供了更高层次的抽象，它依赖于基于 Paxos 的分布式锁服务。与 Boxwood 一样，ZooKeeper 也是用于构建分布式系统的组件。不过，ZooKeeper 对性能有较高要求，在客户端应用程序中的使用更为广泛。ZooKeeper 提供了较低级的基元，应用程序可利用这些基元实现较高级的基元。

ZooKeeper 类似于一个小型文件系统，但它只提供了文件系统操作的一小部分，并增加了大多数文件系统不具备的功能，如排序保证和条件写入。不过，ZooKeeper 手表在精神上类似于 AFS 的缓存回调[16]。

Sinfonia [2] 引入了*迷你交易*，这是构建可扩展分布式系统的一种新模式。Sinfonia 的设计目的是存储应用程序数据，而 ZooKeeper 则存储应用程序元数据。ZooKeeper 将其状态完全复制到内存中，以实现高性能和一致的延迟。我们使用类似文件系统的操作和排序方式，实现了类似迷你交易的功能。znode 是一个方便的抽象概念，我们在其上添加了 Sinfonia 所缺少的手表功能。Dynamo [11] 允许客户端在分布式键值存储中获取和放置相对较小（小于 1M）的数据量。与 ZooKeeper 不同，Dynamo 中的键值空间不是分层的。Dynamo 也不为写入提供强大的持久性和一致性保证，而是在读取时解决冲突。

DepSpace [4] 使用元组空间提供拜占庭容错服务。与 ZooKeeper 一样，DepSpace 也使用简单的服务器接口，在客户端实现强大的同步原语。虽然 DepSpace 的性能比 ZooKeeper 低得多，但它提供了更强的容错性和保密性保证。

7 结论

ZooKeeper 通过向客户端公开无等待对象，采用无等待方法来解决分布式系统中的进程协调问题。我们发现 ZooKeeper 在雅虎内部和外部的多个应用中都非常有用。ZooKeeper 通过使用由本地副本提供服务的快速读取和手表，在以读取为主的工作负载中实现了每秒数十万次的吞吐量。尽管我们对读取和监视的一致性保证似乎很弱，但我们通过使用案例证明，这种组合允许我们在客户端实施高效而复杂的协调协议，即使读取不按优先级排序，而且数据对象的实施是无等待的。事实证明，无等待特性对高性能至关重要。

尽管我们只介绍了几个应用，但使用 ZooKeeper 的应用还有很多。我们认为，ZooKeeper 之所以如此成功，是因为它的接口简单，而且可以通过这个接口实现强大的抽象功能。此外，由于 ZooKeeper 的高吞吐量，应用程序可以广泛使用它，而不仅仅是过程粒度锁定。

致谢

我们要感谢 Andrew Kornev 和齐润平对 ZooKeeper 的贡献；感谢 Zeke Huang 和 Mark Marchukov 的宝贵反馈；感谢 Brian Cooper 和 Laurece Ramontianu 对 ZooKeeper 的早期贡献；感谢 Brian Bershad 和 Geoff Voelker 对演示文稿提出的重要意见。

参考资料

[1] M.Abd-El-Malek、G. R. Ganger、G. R. Goodson、M. K. Reiter 和 J. J. Wylie。可故障扩展的拜占庭容错服务。*SOSP '05: 第二十届 ACM 操作系统原理研讨会论文集》*，第 59-74 页，美国纽约州纽约市，2005 年。ACM.

[2] M.阿奎莱拉、A. 莫森特、M. 沙、A. 维奇和 C. 卡拉马诺里斯。Sinfonia：构建可扩展分布式系统的新范式。*SOSP '07: 第 21 届 ACM 操作系统原理研讨会论文集*，纽约州纽约市，2007 年。

[3] 亚马逊。Amazon simple queue service. `http://aws.amazon.com/sqs/`, 2008.

[4] A.N. Bessani、E. P. Alchieri、M. Correia 和 J. da Silva Fraga。Depspace：拜占庭容错协调服务。*第三届 ACM SIGOPS/EuroSys 欧洲系统会议 - EuroSys 2008 论文集》*，2008 年 4 月。

[5] K.P. Birman.ISIS 系统中的复制与容错。In *SOSP '85: Proceedings of the 10th ACM symposium on Oper- ating systems principles*, New York, USA, 1985.ACM Press.

[6] M.Burrows.松耦合分布式系统的 Chubby 锁服务。*第 7 届 ACM/USENIX 操作系统设计与实现（OSDI）研讨会论文集*，2006 年。

[7] M.Castro 和 B. Liskov.实用的拜占庭容错和主动恢复。*ACM Transactions on Computer Systems*, 20(4), 2002.

[8] T.Chandra, R. Griesemer, and J. Redstone.Paxos 实时运行：工程视角。*第 26 届 ACM 分布式计算原理 (PODC) 年度研讨会论文集*，2007 年 8 月。

[9] A.Clement、M. Kapritsos、S. Lee、Y. Wang、L. Alvisi、M. Dahlin 和 T. Riche。UpRight 集群服务。*ACM 操作系统原理研讨会（SOSP）论文集*，2009 年 10 月。

[10] J.Cowling、D. Myers、B. Liskov、R. Rodrigues 和 L. Shira。Hq replication：用于拜占庭容错的混合法定人数协议。In *SOSP '07: Proceedings of the 21st ACM symposium on Operating Systems principles*, New York, NY, USA, 2007.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels.Dynamo：Amazons 高可用键值存储。在

SOSP '07: Proceedings of the 21st ACM symposium on Operat- ing systems principles, New York, NY, USA, 2007.ACM Press.

[12] J.Gray、P. Helland、P. O'Neil 和 D. Shasha。复制的危险与解决方案。In *Proceedings of SIGMOD '96*, pages 173-182, New York, NY, USA, 1996.ACM.

[13] A.Hastings.事务处理环境中的分布式锁管理。*电气和电子工程师学会第九届可靠分布式系统研讨会论文集》*，1990 年 10 月。

[14] M.Herlihy.无等待同步*ACM 编程语言与系统交互》*，13(1)，1991 年。

[15] M.Herlihy and J. Wing.线性化：并发对象的正确性条件。*ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[16] J. H. Howard、M. L. Kazar、S. G. Menees、D. A. Nichols、M.Satyanarayanan, R. N. Sidebotham, and M. J. West.分布式文件系统的规模与性能。*ACM Trans.Com- put.*6(1), 1988.

[17] Katta.Katta - 在网格中分发 lucene 索引。`http://katta.wiki.sourceforge.net/`，2008 年。

[18] R.Kotla、L. Alvisi、M. Dahlin、A. Clement 和 E. Wong。Zyzzyva：投机性拜占庭容错。*SIGOPS Oper.Syst.Rev.*, 41(6):45-58, 2007.

[19] N.P. Kronenberg, H. M. Levy, and W. D. Strecker.Vaxclusters（扩展摘要）：紧密耦合的分布式系统。*SIGOPS Oper.Syst.Rev.*, 19(5), 1985.

[20] L.Lamport.兼职议会》。*ACM Transactions on Computer Systems*, 16(2), May 1998.

[21] J.MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L.Zhou.黄杨木：抽象作为存储基础架构的基础。*第 6 届 ACM/USENIX 操作系统设计与实现（OSDI）研讨会论文集*，2004 年。

[22] L.Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, C. Lingley- Papadopoulos, and T. Archambault.图腾系统。*第 25 届容错计算国际研讨会论文集*，1995 年 6 月。

[23] S.Mullender, editor.*分布式系统》，第 2 版*。ACM Press, New York, NY, USA, 1993.

[24] B.Reed 和 F. P. Junqueira.一个简单的完全有序广投协议In *LADIS '08：大型分布式系统与中间件第二次研讨会论文集》*，第 1-6 页，美国纽约，2008 年。ACM.

[25] N.Schiper 和 S. Toueg.动态系统的稳健轻量级稳定领导者选举服务。*DSN*, 2008.

[26] F.B. Schneider.使用状态机方法实现容错服务：A tutorial.*ACM Computing Surveys*, 22(4), 1990.

[27] A.Sherman、P. A. Lisiecki、A. Berkheimer 和 J. Wein。ACMS：Akamai 配置管理系统。见 *NSDI*，2005 年。

[28] A.Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis.Zeno：最终一致的拜占庭容错。在 *NSDI'09：第 6 届 USENIX 网络系统设计与实现研讨会论文集》*，第 169-184 页，美国加利福尼亚州伯克利，2009 年。USENIX 协会。

[29] Y.Y. J. Song, F. Junqueira, and B. Reed. 怀疑论者的 BFT。`http://www.net.t-labs.tu-berlin.de/~petr/BFTW3/abstracts/talk-abstract.pdf`.

[30] R. van Renesse 和 K. Birman.Horus, a flexible group communication systems.*ACM 通信》*，39（16），1996 年 4 月。

[31] R. van Renesse、K. Birman、M. Hayden、A. Vaysburd 和 D.Karr.利用集合构建自适应系统。*软件 - 实践与经验》*，28（5），1998 年 7 月。