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
雅虎研究公司
{fpj,breed}@yahoo-inc.com

摘要

在本文中，我们描述了ZooKeeper，一个用于协调分布式应用进程的服务。由于ZooKeeper是关键基础设施的一部分，ZooKeeper旨在提供一个简单和高性能的内核，以便在客户端建立更复杂的协调原语。它将群组消息传递、共享寄存器和分布式锁服务的元素整合到一个重复的集中式服务中。ZooKeeper暴露的接口具有共享寄存器的免等待功能，以及类似于分布式文件系统的缓存失效的事件驱动机制，以提供一个简单而强大的协调服务。

ZooKeeper接口能够实现高性能的服务。除了无等待的特性外，ZooKeeper还为每个客户端提供了先进先出的请求执行保证，并为所有改变ZooKeeper状态的再请求提供线性化。这些设计上的决定使我们能够实现一个高性能的处理管道，读取请求由本地服务器来满足。我们表明，对于目标工作负载，即2:1到100:1的读写比，ZooKeeper可以每秒处理几万到几十万的事务。这种性能使ZooKeeper可以被客户端应用程序广泛使用。

1 简介

大规模的分布式应用需要不同形式的协调。配置是协调的最基本形式之一。在最简单的形式中，配置只是系统进程的操作参数列表，而更复杂的系统有动态配置参数。小组成员和领导者的选举在分布式系统中也很常见：经常有进程需要知道哪些其他进程是活的，这些进程负责什么。锁构成了一个强大的协调原素实施对关键再资源的相互排斥的访问。

协调的一个方法是为每个不同的协调需求开发服务。例如，亚马逊简单队列服务[3]就特别关注排队问题。其他服务已经被开发出来，专门用于领导者选举[25]和配置[27]。实现更强大的基本要素的服务可以用来实现不太强大的要素。例如，Chubby[6]是一个具有强大同步保证的锁服务。然后，锁可以被用来实现领导者的选举、群组成员等。

在设计我们的协调服务时，我们放弃了在服务器端实现特定的基元，而是选择了暴露一个API，使应用开发者能够实现他们自己的基元。这样的选择导致了*协调内核*的实现，它可以在不需要改变服务核心的情况下实现新的基元。这种方法使多种形式的协调适应应用程序的要求，而不是将开发者限制在一套固定的基元上。

在设计ZooKeeper的API时，我们摒弃了阻塞原语，如锁。协调服务的阻塞基元会导致缓慢或有问题的客户端对快速客户端的性能产生负面影响，以及其他问题。如果处理请求依赖于其他客户端的响应和故障检测，那么服务的实现本身就会变得更加复杂。因此，我们的系统，Zookeeper，实现了一个API，操纵简单的*无等待的*数据对象，像文件系统一样分层组织。事实上，ZooKeeper的API类似于其他文件系统的API，只看API的符号，ZooKeeper似乎是没有锁方法、打开和关闭的Chubby。然而，实现无等待的数据对象使ZooKeeper与基于阻塞基元（如锁）的系统有很大区别。尽管无等待属性对于每一个...

我们的系统具有良好的性能和容错性，但这还不足以实现共同管理。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（FIFO）客户端顺序*和*可写入（linearizable*）能够有效地实现服务，这足以实现我们应用所关心的协调原语。事实上，我们可以用我们的API为任何数量的进程实现共识，根据Herlihy的层次结构，Zoo- Keeper实现了一个通用对象[14]。

ZooKeeper服务包括一个使用复制的服务器群，以实现高可用性和高性能。它的高性能使由大量进程组成的应用程序能够使用这样一个协调内核来管理协同工作的所有方面。我们能够用一个简单的流水线结构来实现ZooKeeper，它允许我们在有成百上千个请求的情况下仍能实现低延迟。这样的流水线自然能够以先进先出的顺序执行来自单个客户端的操作。保证先进先出的客户端顺序使客户端能够异步地提交操作。有了异步操作，一个客户端就能在同一时间有多个未完成的操作。例如，当一个新的客户端成为领导者时，它必须对元数据进行管理和相应的更新，这种功能是可取的。如果没有多个未完成操作的可能性，初始化的时间可能是几秒钟，而不是几秒钟。

为了保证更新操作满足线性化能力，我们实现了一个基于领导的原子广播协议[23]，称为Zab[24]。然而，ZooKeeper应用程序的典型工作负载是由读操作主导的，因此扩展读吞吐量是可取的。在ZooKeeper中，服务器在本地处理读取操作，我们不使用Zab来完全排序。在客户端缓存数据是提高读取性能的一项重要技术。例如，对于一个进程来说，缓存当前领导者的标识符是非常有用的，而不是每次需要知道领导者的时候都去探测ZooKeeper。ZooKeeper使用观察机制，使客户端能够缓存数据而不需要直接管理客户端的缓存。通过这种机制，客户端可以观察某个数据对象的更新，并在更新时收到通知。Chubby直接对客户端缓存进行管理。它阻止更新，以验证所有缓存数据的客户端的缓存。在这种设计下，如果这些客户端中的任何一个速度慢或有问题，更新就会延迟。Chubby使用租约来防止一个有问题的客户无限期地阻塞系统。然而，租约只限制了缓慢或有问题的客户端的影响，而ZooKeeper手表则避免了这种影响。彻底的问题。

在本文中，我们讨论了我们的设计和实现。ZooKeeper的扩展。通过ZooKeeper，我们能够实现我们的应用程序所需要的所有协调原语，尽管只有写是可线性化的。为了验证我们的方法，我们展示了我们如何用ZooKeeper实现一些协调原语。

总而言之，在本文中，我们的主要贡献是。

协调内核。我们提出了一种具有宽松一致性保证的免等待协调服务，用于分布式系统中。特别是，我们描述了我们对*协调内核*的设计和实现，我们已经在许多关键的应用中使用该内核来实现各种协调技术。

协调配方。我们展示了如何使用ZooKeeper来构建更高级别的协调原语，甚至是分布式应用中经常使用的阻塞和强一致性原语。**协调的经验。**我们分享一些我们使用ZooKeeper的方法，并评估其每形成。

2 ZooKeeper服务

客户端通过客户端API使用ZooKeeper客户端库向ZooKeeper提交请求。除了通过客户端API提供ZooKeeper服务接口外，客户端库还负责管理客户端和ZooKeeper服务器之间的网络连接。

在本节中，我们首先提供了ZooKeeper服务的高层视图。然后我们讨论客户用来与ZooKeeper互动的API。

术语。在本文中，我们用*客户端*表示ZooKeeper服务的用户，用*服务器*表示提供ZooKeeper服务的进程，用*znode*表示ZooKeeper数据中的一个内存数据节点，它被组织在一个被称为*数据树*的分层命名空间里。我们还使用术语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为其客户提供了一组数据节点（znodes）的抽象，这些节点根据一个分层的名称空间来组织。这个层次结构中的节点是客户通过ZooKeeper API操作的数据对象。分层名称空间通常在文件系统中使用。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的方式，它可以更好地组织应用程序的元数据。要提到一个

给定的znode，我们使用文件系统路径的标准UNIX符号。例如，我们使用/A/B/C来表示通向znode C的路径，其中C有B作为它的父节点，B有A作为它的父节点。所有的节点都存储数据，所有的节点，除了短暂的节点，都可以有子节点。

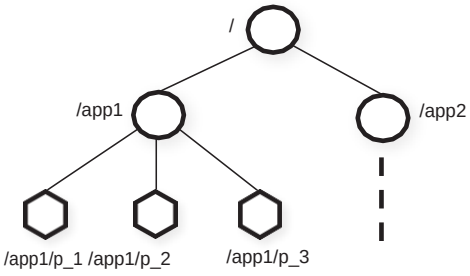

图1：ZooKeeper分层名称空间的说明。

客户端可以创建两种类型的znodes。

常规。客户端通过明确地创建和删除正规的Znodes来操纵它们。

短暂的。客户端创建这样的节点，他们可以明确地删除它们，或者在创建它们的会话终止时（故意的或由于失败）让系统自动删除它们。

此外，当创建一个新的znode时，客户可以设置一个*顺序*标志。在设置了顺序标志的情况下创建的节点有一个单调递增的计数器的值附加在它的名字上。如果n是新的Znode，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实现了观察，以允许客户端在在不需要轮询的情况下，及时收到变化的通知。当客户端发出一个设置了观察标志的读操作时，除了服务器承诺在返回的信息发生变化时通知客户端外，该操作的完成是正常的。监视是与会话相关的一次性触发器；一旦被触发或会话关闭，它们就会被取消注册。监视表明发生了变化，但不提供变化。例如，如果一个客户在"/foo "被改变两次之前，就已经获得了getData(''/foo'', true)，客户将获得一个观察事件，告诉客户"/foo "的数据已经改变。Session事件，比如连接丢失事件，也会被发送到watch callbacks，以便客户端知道watch事件可能被延迟。

数据模型。ZooKeeper的数据模型基本上是一个具有简化API的文件系统，只有完整的数据读写，或者是一个具有层次的键/值表。

chical keys.分层命名空间对于定位不同应用的命名空间的子树以及设置对这些子树的访问权限非常有用。我们还利用客户端的目录概念来建立更高层次的基元，我们将在第2.4节看到。与文件系统中的文件不同，Znodes不是为一般的数据存储而设计的。相反，Znodes映射到客户端应用程序的缩写，通常对应于用于协调目的的元数据。为了说明问题，在图1中我们有两个子树，一个是应用1（/app1），另一个是应用2（/app2）。应用1的子树实现了一个简单的组成员协议：每个客户进程p_i，在/app1下创建一个znode p_i，只要这个znode p_i持续存在，它就会一直存在。进程正在运行。

尽管znodes并不是为一般的数据存储而设计的，但ZooKeeper确实允许客户端存储一些信息，这些信息可用于分布式计算中的元数据或配置。例如，在一个基于领导者的应用中，对于一个刚刚开始的应用服务器来说，了解其他哪个服务器目前是领导者是很有用的。为了实现这个目标，我们可以让当前的领导者在znode空间的一个已知位置上写下这个信息。Znode也有相关的元数据，有时间戳和版本计数器，这使得客户端可以跟踪Znode的变化，并根据Znode的版本执行条件更新。

会话。客户端连接到ZooKeeper并启动一个会话。会话有一个相关的超时时间。如果客户端在超过该超时时间内没有收到任何来自其会话的信息，ZooKeeper就认为该客户端有问题。当客户端明确关闭会话句柄或ZooKeeper检测到客户端有问题时，会话就会结束。在一个会话中，客户端观察到一系列反映其操作执行情况的状态变化。会话使客户端能够在ZooKeeper集合中透明地从一个服务器移动到另一个服务器，并因此在ZooKeeper服务器之间持续存在。

2.2 客户端API

我们在下面介绍ZooKeeper API的一个相关子集，并讨论每个请求的语义。

create(path, data, flags)。创建一个znode路径名称为path，在其中存储data[]，并且返回新的znode的名称。flags使客户能够选择znode的类型：常规的、短暂的，并设置顺序标志。

delete(path, version)。如果该znode处于预期的版本，则删除该znode路径；**existence(path, watch)。**如果znode

路径名path存在，否则返回false。观察
标志使客户端能够设置一个

在znode上观看。

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

setData(path, data, version)。如果版本号是znode的当前版本，则将data[]写到znode路径。

getChildren(path, watch)。返回一个znode的子节点的名称集合。

sync(path)。等待所有在操作开始时等待的更新传播到客户端连接的服务器上。目前路径被忽略。

所有的方法都有一个同步和一个非同步的方法。通过API提供的慢性版本。当一个应用程序需要执行一个单一的ZooKeeper操作，并且没有并发的任务要执行时，它就会使用同步API，因此它进行必要的ZooKeeper调用并进行阻塞。然而，异步API使应用程序能够同时拥有多个未完成的ZooKeeper操作和其他任务的同时执行。ZooKeeper客户端保证每个操作的相关回调都被调用。

请注意，ZooKeeper不使用句柄来访问znode。每个请求都包括被操作的znode的完整路径。这种选择不仅简化了API（没有open()或close()方法），而且还消除了服务器需要维护的额外状态。

每一个更新方法都有一个预期的版本号，这就可以实现传统的更新。如果程序的实际版本号与预期的版本号不一致，更新就会以一个意外的版本错误而失败。如果版本号是-1，它就不执行版本检查。

2.3 ZooKeeper保证

ZooKeeper有两个基本的排序保证。

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

先进先出的客户顺序：来自一个特定客户的所有请求都按照客户发送的顺序执行。

请注意，我们对线性化的定义与Herlihy[15]最初提出的定义不同，我们称之为*A-线性化*（asynchronous linearizabil- ity）。在Herlihy对线性化的最初定义中，一个客户端在同一时间只能有一个未完成的操作（一个客户端是一个线程）。在我们的定义中，我们允许一个客户端有多个未完成的操作，同时，我们可以选择保证同一客户端的未完成操作没有特定的顺序，或者保证先进先出的顺序。我们选择后者作为我们的属性。需要注意的是，所有对可线性化对象成立的结果也对可A线性化对象成立，因为一个满足A线性化的系统也满足线性化。因为只有更新请求是可线性化的，ZooKeeper在每个副本中都会本地处理读取请求。这使得服务可以随着服务器的增加而线性地扩展到系统中。

为了了解这两种保证是如何相互作用的，请考虑以下情况。一个由若干进程组成的系统选出一个领导者来指挥工人进程。当一个新的领导者负责该系统时，它必须改变大量的配置参数，并在完成后通知其他进程。这样我们就有两个重要的要求。

- 当新的领导者开始进行改变时，我们不希望其他进程开始使用正在被改变的配置。
- 如果新的领导者在配置完全更新之前就死亡，我们不希望进程使用这个部分配置。

请注意，分布式锁，如Chubby提供的锁，将有助于满足第一个要求，但不足以满足第二个要求。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存在时才会使用该配置。新的领导者通过删除*ready*，更新各种配置节点，并创建*ready*来进行配置变更。所有这些变化都可以通过流水线和异步发布来快速更新配置状态。虽然一个改变操作的延迟是2毫秒，但如果一个新的领导者必须更新5000个不同的节点，如果请求是一个接一个地发出，将需要10秒；通过异步地发出请求，请求将花费不到一秒钟。由于排序保证，如果一个进程看到了*就绪的*znode，它也必须看到新的领导者所做的所有配置改变。如果新的领导者在*准备好的*znode创建之前死亡，其他进程知道配置还没有最终完成，就不会使用它。

上述方案仍有一个问题：如果一个进程在新的领导者开始进行改变之前看到*ready*存在，然后在改变进行时开始读取configuration，会发生什么？这个问题通过对通知的排序保证得到了解决：如果一个客户正在关注一个变化，客户将在看到变化后系统的新状态之前看到通知事件。

在客户端可以读取任何新的配置之前，将该变化告知客户端。

当客户端在ZooKeeper之外还有自己的通信通道时，会出现另一个问题。例如，考虑两个客户端*A*和*B*在ZooKeeper中拥有一个共享配置，并通过一个共享的通信渠道进行通信。如果*A*改变了ZooKeeper中的共享配置，并通过共享通信通道告诉*B*这一改变，*B*在重新读取配置时就会发现这一改变。如果*B*的ZooKeeper副本稍稍落后于*A*的，它可能看不到新的配置。利用上述保障措施，*B*可以在重新读取配置前发出写入指令，以确保它看到最新的信息。为了更有效地处理这种情况，ZooKeeper提供了同步请求：当紧随其后的是读，就构成了*慢速读*。同步使服务器在处理读之前应用所有悬而未决的写请求，而不需要完全写的开销。这个基元与ISIS[5]的`flush`基元的想法相似。

ZooKeeper还具有以下两个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，则服务是可用的；如果ZooKeeper服务成功地响应了一个变更请求，只要有一个法定的服务器最终能够恢复，该变更在任何数量的故障中都会持续。

2.4 基元的例子

在本节中，我们将展示如何使用ZooKeeper的API来实现更强大的原语。ZooKeeper服务对这些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，如组成员和配置管理，也是无需等待的。对于其他的，如会合，客户端需要等待一个事件。尽管ZooKeeper是无等待的，但我们可以用ZooKeeper实现高效的阻塞原语。ZooKeeper的排序保证允许对系统状态进行有效的推理，而手表允许有效的等待。

配置管理

ZooKeeper可以用来实现分布式应用中的动态配置。在其最简单的形式中，配置被存储在一个z节点中，即z_c。进程以z_c的完整路径名启动。启动的进程通过读取z_c并将观察标志设置为 "true "来获得其配置。如果z_c中的配置被更新，进程会被通知并读取新的配置，并再次将观察标志设置为真。

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有最新的信息。例如，如果一个观察z_c的进程被通知到z_c的变化，在它对z_c发出读取之前，z_c又有三个变化，该进程就不会再收到三个通知事件。这并不影响进程的行为，因为这三个事件只是通知进程它已经知道的事情：它拥有的z_c的信息是过时的。

会合

在分布式系统中，有时并不总是先验地清楚最终的系统配置是什么样子的。例如，一个客户可能想启动一个主进程和几个工作进程，但启动进程是由调度器完成的，所以客户不知道提前的信息，如广告和端口，它可以给工作进程连接到主进程。我们用ZooKeeper处理这种情况，使用一个交会节点z_r，这是一个由客户端创建的节点。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。当主进程启动时，它在z_r中填入它正在使用的地址和端口的信息。当工作者启动时，他们读取z_r，并将watch设置为true。如果z_r还没有被填入，工人会等待z_r被更新时的通知。如果z_r是一个短暂的节点，主进程和工作进程可以观察z_r是否被删除，并在客户端结束时进行自我清理。

小组成员资格

我们利用短暂节点的优势来实现小组成员资格。具体来说，我们利用短暂节点允许我们看到创建该节点的会话的状态这一事实。我们首先指定一个znode，z_g，以代表该组。当该组的一个进程成员开始时，它在z_g下创建一个短暂的子节点。如果每个进程有一个唯一的名字或标识符，那么这个名字就被用作子节点的名字；否则，该进程用SEQUENTIAL标志创建z节点，以获得一个唯一的名字分配。进程可以将进程信息放在子znode的数据中，例如，进程使用的地址和端口。

在z_g下创建了子znode之后，这个过程就正常开始了。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在z_g之前就会被自动删除。

进程可以通过简单地列出z_g的子节点来获得组信息。如果一个进程想监视组成

员的变化，该进程可以将观察标志设置为真，并在收到变化通知时刷新组信息（也就是将观察标志设置为真的方式）。

简单锁

尽管ZooKeeper不是一个锁服务，但它可以用来实现锁。使用ZooKeeper的应用程序通常使用根据其需求定制的同步原语，如上图所示。这里我们展示了如何用ZooKeeper实现锁，以表明它可以实现各种一般的同步原语。

最简单的锁的实现使用"锁文件"。锁由一个znode表示。为了获得一个锁，一个客户试图用EPHEMERAL标志来创建指定的znode。如果创建成功，客户端就持有该锁。否则，客户端可以通过设置观察标志来读取znode，以便在当前领导者死亡时获得通知。当客户端死亡或显式删除该节点时，它将释放锁。其他正在等待锁的客户端一旦观察到znode被删除，就会再次尝试获得一个锁。

虽然这种简单的锁定协议是有效的，但它确实有一些问题。首先，它受到羊群效应的影响。如果有许多客户等待获得一个锁，当锁被释放时，他们都会争夺这个锁，尽管只有一个客户可以获得这个锁。其次，它只实现了独占锁。下面的两个基元显示了如何克服这两个问题。

没有羊群效应的简单锁 我们定义一个锁znode *l*来实现这种锁。直观地说，我们把所有请求锁的客户排成一排，每个客户按照请求到达的顺序获得锁。因此，希望获得锁的客户做如下工作。

锁定

```
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中刚好在n之前排序的z节点
5 如果存在(p, true)，则等待观察事件。
6 转到2
```

解锁

```
1 删除(n)
```

在Lock的第1行使用SEQUENTIAL标志，命令客户端尝试获取锁，并对所有其他尝试进行重新排序。如果客户端的znode在第3行有最低的序列号，则客户端持有该锁。否则，客户端等待删除已经拥有锁的节点或将在该客户端的节点之前收到锁的节点。通过只监视在客户端znode之前的znode，我们避免了羊群效应，只在锁被释放或锁请求被放弃时唤醒一个进程。一旦被客户端监视的znode离开，客户端必须检查它是否现在持有锁。(之前的锁请求可能已经被放弃了，有一个序列号较低的znode仍在等待或持有该锁)。

释放一个锁就像删除代表锁请求的znode *n*一样简单。通过使用在创建时使用EPHEMERAL标志，崩溃的进程将自动清理任何锁请求或释放它们可能拥有的任何锁。

总而言之，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被唤醒，因为每个znode正好被另一个客户端监视，所以我们不会有羊群效应。
2. 没有轮询或超时。
3. 由于我们实现了锁的方式，我们可以通过浏览ZooKeeper的数据看到锁争夺的数量，打破锁，并调试锁的问题。

读/写锁

为了实现读/写锁，我们稍微改变了锁的程序，有单独的读锁和写锁程序。解锁程序与全局锁的情况相同。

写锁

```
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中刚好在n之前排序的z节点
5 如果存在(p, true)，等待事件发生
6 转到2
```

阅读锁定

```
1 n = create(l + "/read-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 如果在C中没有写出低于n的znodes，退出
4 p = 在C中写下z节点的顺序，正好在n之前
5 如果存在(p, true)，等待事件发生
6 转到3
```

这个锁的程序与之前的锁略有不同。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，因为只有早期的写锁节点才能阻止客户端获得读锁。当有几个客户端在等待一个读锁，并在序列号较低的"写"节点被删除时得到通知时，我们可能会出现"羊群效应"；事实上，这是一个需要的行为，所有这些读客户端应该被释放，因为他们现在可能拥有锁。

双重屏障

双重屏障使客户能够同步计算的开始和结束。当有足够多的进程（由障碍物的阈值定义）加入障碍物时，进程就会开始他们的计算，一旦完成就会离开障碍物。在ZooKeeper中，我们用一个znode来表示一个屏障，称为*b*。每个进程*p*在进入时通过创建一个znode作为*b*的子节点来注册*b*，并在准备离开时取消注册--
重新移动这个子节点。当*b*的子节点数量超过屏障阈值时，进程可以进入屏障。当所有的进程都移除它们的子节点时，进程可以离开屏障。我们使用手表来有效地等待进入和

退出条件要得到满足。为了进入，进程观察是否存在一个准备好的*b*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。要离开时，进程观察一个特定的子节点是否消失，只有在该子节点被移除后才检查退出条件。

3 ZooKeeper应用程序

现在我们描述一些使用ZooKeeper的应用程序，并简要说明它们是如何使用的。我们用**黑体字**显示每个例子的主要内容。

抓取服务

抓取是搜索引擎的一个重要部分，雅虎抓取了数十亿的网络文件。抓取服务（FS）是雅虎爬虫的一部分，它目前正在生产中。从本质上讲，它有主进程来指挥页面获取过程。主进程向取件者提供配置，取件者则回信告知其状态和健康状况。在FS中使用ZooKeeper的主要优点是可以从主站的故障中恢复，保证故障中的可用性，以及将客户与服务器解耦，允许他们通过从ZooKeeper中读取它们的状态来指导他们重新寻找健康的服务器。因此，FS主要使用ZooKeeper来管理**配置元数据**，尽管它也使用ZooKeeper来选举主站（**领导者选举**）。

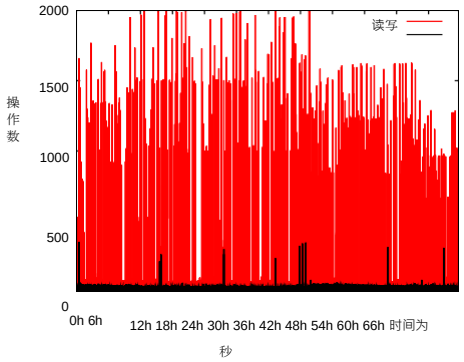

图2：一台ZK服务器的工作负载与获取服务。每个点代表一秒钟的样本。

图2显示了FS使用的ZooKeeper服务器在三天内的读写流量。为了生成这张图，我们计算了这一时期每一秒的操作数，每一个点都对应于该秒的操作数。我们观察到，与写流量相比，读流量要高得多。在速率高于每秒1,000个操作的时期，读：写的比例在10:1和100:1之间变化。这个工作负载中的读操作是`getData()`、`getChildren()`和`existence()`，其发生率依次增加。

Katta Katta [17] 是一个分布式索引器，使用ZooKeeper
进行协调，它是一个非雅虎应用的例子。Katta使用分片来划分索引的工作。一个主服务器将分片分配给从服务器并跟踪进度。从属服务器可能会失败，所以主服务器必须在从属服务器来往时重新分配负载。主服务器也可能失败，所以其他服务器必须准备好在失败的情况下接管。Katta使用ZooKeeper来跟踪从属服务器和主服务器的状态（**组成员**），并处理主服务器的故障切换（**领导者选举**）。Katta还使用ZooKeeper来跟踪和传播对从服务器的碎片分配（**配置管理**）。

雅虎信息代理

雅虎信息代理（YMB）是一个分布式发布-订阅系统。该系统管理着数以千计的主题，客户可以向其发布和接收信息。这些主题分布在一组服务器中，以提供可扩展性。每个主题都使用主备份方案进行复制，确保消息被复制到两台机器上，以确保可靠的消息传递。组成YMB的服务器使用一个无共享的分布式架构，这使得协调对于正确的操作至关重要。YMB使用ZooKeeper来管理主题的分配（**配置元数据**），处理系统中机器的故障（**故障检测**和**组成员**），并控制系统运行。

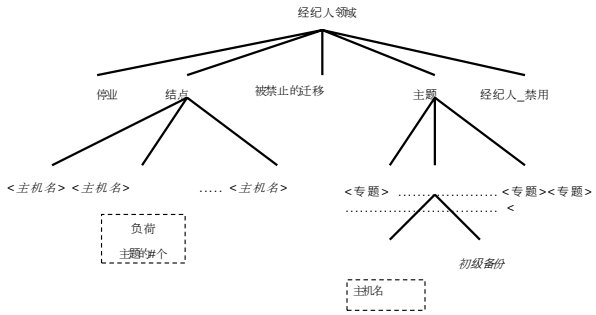

图3：ZooKeeper中雅虎消息代理（YMB）的结构布局。

图3显示了YMB的znode数据布局的一部分。每个代理域都有一个叫做`nodes`的znode，它为组成YMB服务的每个活动服务器都有一个短暂的znode。每个YMB服务器在`nodes`下创建一个短暂的znode，其负载和status信息通过ZooKeeper提供组成员和状态信息。诸如禁止关闭和迁移的节点由组成服务的所有服务器监控，并允许对YMB进行集中控制。主题目录为YMB管理的每个主题都有一个子节点。这些主题节点有一个子节点，表示以下

内容

每个主题的主服务器和备份服务器，以及该主题的用户。主服务器和备份服务器的节点不仅允许服务器发现负责一个主题的服务器，而且还管理**领导者的选举**和服务器崩溃。

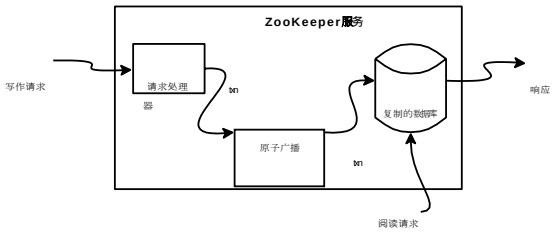

图4：ZooKeeper服务的组成部分。

4 ZooKeeper的实施

ZooKeeper通过在组成服务的每个服务器上复制ZooKeeper数据来提供高可用性。我们假设服务器会因崩溃而失败，而这些有问题的服务器后来可能会恢复。图4显示了ZooKeeper服务的高级组件。当重新收到一个请求时，服务器准备执行该请求（重新寻求处理器）。如果这样的请求需要服务器之间的协调（写请求），那么他们就会使用一个协议协议（一个原子广义的实现），最后服务器将变化提交给Zoo-
Keeper数据库，该数据库在集合的所有服务器中完全复制。在读取请求的情况下，服务器只需读取本地数据库的状态并生成对请求的再响应。

复制的数据库是一个*内存*数据库，包含了整个数据树。默认情况下，树上的每个节点最多存储1MB的数据，但这个最大值是一个配置参数，在特定情况下可以改变。为了保证可恢复性，我们有效地将更新的数据记录到磁盘上，并强制写入磁盘介质，然后再应用到内存数据库上。事实上，就像Chubby[8]一样，我们保留了一个已提交操作的重放日志（在我们的例子中，是一个写前日志），并生成了内存数据库的定期快照。

每个ZooKeeper服务器都为客户提供服务。客户端正好连接到一个服务器来提交其请求。正如我们前面指出的，读取请求是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，即写请求，则由协议协议处理。

作为协议的一部分，写请求被转发给一个服务器，称为*领导者*[1]。其余的ZooKeeper服务器，称为*跟随者*，接受由领导者的状态变化组成的消息提议，并就状态变化达成一致。

4.1 请求处理器

由于信息传递层是原子性的，我们保证本地副本永远不会出现分歧，尽管在任何时候，在
时间，一些服务器可能比其他服务器应用更多的事务。与客户发送的请求不同，事务是*空转的*。当领导者收到一个写请求时，它计算当写被应用时系统的状态是什么，并将其转换为一个捕捉这个新状态的事务。该事务
讨论范围内。

[1]作为协议协议的一部分，领导者和追随者的细节不在本文的

由于可能存在尚未应用于数据库的外置事务，因此必须计算状态。例如，如果客户端做了一个有条件的setData，并且请求中的版本号与被更新的znode的未来版本号相匹配，服务会生成一个setDataTXN，其中包含新数据、新版本号和更新的时间戳。如果发生错误，例如版本号不匹配或被更新的znode不存在，则会生成一个错误TXN。

4.2 原子广播

所有更新ZooKeeper状态的请求都被转发给领导者。领导执行请求，并通过Zab[24]这个原子广播协议广播ZooKeeper状态的变化。收到客户端请求的服务器在传递相应的状态变化时对客户端进行响应。Zab使用去掉故障的简单多数四分法来决定提案，所以Zab以及ZooKeeper只有在大多数服务器都正确的情况下才能工作（*即*，有$2f + 1$个服务器，我们可以容忍f个故障）。

为了实现高吞吐量，ZooKeeper试图保持请求处理管道的完整。它可能在处理管道的不同部分有成千上万的请求。由于状态的改变取决于对先前状态改变的应用，Zab提供了比常规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化按照它们被发送的顺序传递，并且在领导者广播自己的变化之前，来自先前领导者的所有变化都会被传递给既定的领导者。

有一些实现细节简化了我们的实现，并给我们带来了出色的性能。我们使用TCP进行传输，因此消息的顺序由网络来维持，这使我们能够简化我们的实现。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。

记忆数据库，这样我们就不必把信息两次写到磁盘上。

在正常的操作中，Zab确实按顺序准确地传递了所有的消息，但是由于Zab没有持久地记录每一个被传递的消息的ID，Zab可能在恢复期间重新传递一个消息。因为我们使用的是idempotent事务，所以只要是按顺序传递，多次传递是可以接受的。事实上，ZooKeeper要求Zab至少重新交付在最后一次快照开始后交付的所有消息。

4.3 复制的数据库

每个副本在内存中都有一份ZooKeeper状态的副本。当ZooKeeper服务器从崩溃中恢复时，它需要恢复这个内部状态。在服务器运行一段时间后，重放所有交付的消息来恢复状态会花费太多时间，所以ZooKeeper使用定期快照，只要求重新交付快照开始后的消息。我们称ZooKeeper快照为*模糊快照*，因为我们不锁定ZooKeeper状态来进行快照；相反，我们对树进行深度扫描，以原子方式读取每个Znode的数据和元数据并将其写入磁盘。由于产生的模糊快照可能已经应用了快照生成过程中的一些子集的状态变化，所以结果可能不符合ZooKeeper在任何时间点的状态。然而，由于状态变化是等价的，只要我们按顺序应用状态变化，我们就可以应用它们两次。

例如，假设在ZooKeeper的数据树中，两个节点`/foo`和`/goo`的值分别为`f1`和`g1`，并且在模糊快照开始时都处于版本1，下面的状态变化流以（`transactionType, path, value, newversion`）的形式到来。

```
(SetDataTXN, /foo, f2, 2)
(SetDataTXN, /goo, g2, 2)
(SetDataTXN, /foo, f3, 3)
```

在处理这些状态变化后，`/foo`和`/goo`的值分别为`f3`和`g2`，版本为3和2。然而，模糊快照可能记录了`/foo`和`/goo`的值为`f3`和`g1`，版本分别为3和1，这并不是ZooKeeper数据树的有效状态。如果服务器崩溃了，用这个快照恢复，Zab重新传送状态变化，得到的状态对应于崩溃前的服务状态。

4.4 客户端与服务器之间的互动

当服务器处理一个写请求时，它也会发送和清除与任何观察相关的通知。响应该更新。服务器按顺序处理写，不同时处理其他写或读。这确保了通知的严格连续。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。

读取请求在每个服务器上进行本地处理。每个读请求都被处理并被标记为一个*zxid*，该*zxid*与服务器看到的最后一笔交易有关。这个*zxid*定义了读取请求的部分顺序，并与写入请求的顺序相呼应。通过本地处理读取，我们获得了出色的读取性能，因为它只是本地服务器上的一个内存操作，没有磁盘活动或协议运行。这一设计选择是实现我们对以读为主的工作负载的卓越性能目标的关键。

使用快速读取的一个缺点是不能保证读取操作的优先顺序。也就是说，一个读操作可能会返回一个陈旧的值，即使最近对同一个节点的更新已经被提交。并非所有的应用都需要优先顺序，但对于需要优先顺序的应用，我们已经实现了 同步 。这个基元是异步执行的，并由领导者在所有悬而未决的写入其local副本后排序。为了保证一个给定的读操作重新变成最新的更新值，客户端在读操作的后面调用 同步 。客户端操作的FIFO顺序保证和 同步 的全局保证使得读取操作的结果能够反映 同步 发出前发生的任何变化。在我们的实现中，我们不需要原子地广播 同步 ，因为我们使用的是基于领导者的算法，我们只是把同步操作放在领导者和服务器之间的请求队列的末尾，执行对 同步 的调用。为了使这一方法奏效，跟随者必须确定领导者仍然是领导者。如果有未决的事务提交，那么服务器就不会怀疑领导者。如果挂起的队列是空的，领导者需要发出一个空事务来提交，并在该事务之后下令 同步 。这有一个很好的特性，即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，超时的设置使领导者在追随者放弃他们之前意识到他们不是领导者，所以我们不发布空交易。

ZooKeeper服务器按照先进先出的顺序处理来自客户端的请求。响应包括响应所涉及的*zxid*。即使在没有活动的间隔期间，心跳信息也包括客户端连接到的服务器最后看到的*zxid*。如果客户端连接到一个新的服务器，新的服务器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ZooKeeper数据的查看至少和客户端的查看一样近。如果客户端的视图比服务器的视图更新，则

服务器不会重新建立与客户端的会话，直到服务器跟上为止。由于客户端只看到已经复制到大多数ZooKeeper服务器上的变化，因此可以保证客户端能够找到另一个拥有最新系统视图的服务器。这种行为对于保证持久性非常重要。

为了检测客户端会话的失败，ZooKeeper使用超时。如果其他服务器在会话超时内没有收到任何来自客户端会话的信息，领导者就会确定已经发生了故障。如果客户端足够频繁地发送requests，那么就不需要发送任何其他消息。否则，客户端会在活动少的时期发送心跳信息。如果客户端不能与服务器通信以发送请求或心跳，它会连接到不同的ZooKeeper服务器以重新建立会话。为了防止会话超时，ZooKeeper客户端库在会话闲置$s/3$ ms后发送心跳信息，如果$2s/3$ ms内没有服务器的消息，则切换到新的服务器，其中s是会话超时，单位是毫秒。

5 评价

我们在一个由50台服务器组成的集群上进行了所有的评估。每台服务器有一个Xeon双核2.1GHz处理器，4GB内存，千兆以太网和两个SATA硬盘。我们把下面的讨论分成两部分：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的故障的吞吐量变化进行了基准测试。我们改变了组成ZooKeeper服务的服务器的数量，但始终保持客户端的数量不变。为了模拟大量的客户，我们用35台机器来模拟250个同时进行的客户。

我们有一个ZooKeeper服务器的Java实现，以及Java和C客户端[2]。在这些体验中，我们使用了Java服务器，它被配置为记录在一个专用磁盘上并在另一个磁盘上进行快照。我们的基准客户端使用异步的Java客户端API，每个客户端至少有100个请求未完成。每个请求包括对1K数据的读或写。我们没有显示其他操作的基准，因为所有修改状态的操作的性能大致相同，而非修改状态的操作的性能，除了同步，也大致相同。(同步操作的性能与轻量级写操作的性能接近，因为请求必须是在一个小时内完成。

[2]该实现在http://hadoop. apache.org/zookeeper上公开提供。

到领导者，但不会被广播）。客户端每隔300 *毫秒*发送一次已完成操作的计数，我们每隔6 *秒*进行一次采样。为了防止内存溢出，服务器会节制系统中并发的重新请求的数量。ZooKeeper使用请求节流来保持服务器不被淹没。在这些实验中，我们将ZooKeeper服务器配置为最多有2,000个总请求在处理中。

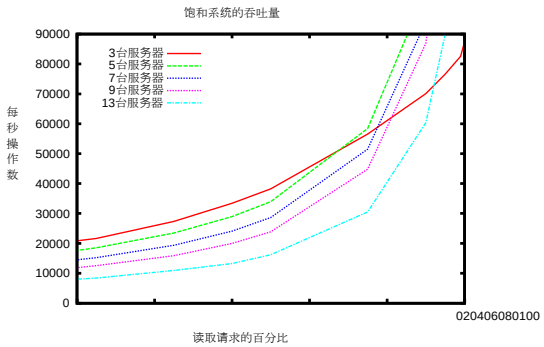

图5：当读和写的比例变化时，饱和系统的吞吐量性能。

服务员	100%**阅读**	0%的**阅读量**
13	460k	8k
9	296k	12k
7	257k	14k
5	165k	18k
3	87k	21k

表1：饱和系统的极端的吞吐量性能。

在图5中，我们显示了当我们改变读和写请求的比例时的吞吐量，每条曲线对应于提供ZooKeeper服务的不同数量的服务器。表1显示了在读取负载的极端情况下的数字。读取吞吐量比写入吞吐量高，因为读取不使用原子广播。该图还显示，服务器的数量对广播协议的性能也有负面影响。从这些图表中，我们观察到系统中的服务器数量不仅影响到服务可以处理的故障数量，而且还影响到服务可以处理的工作负荷。请注意，三台服务器的曲线在60%左右与其他服务器相交。这种情况并不是三台服务器配置所独有的，由于本地读取的平行性，所有的配置都会发生这种情况。然而，在图中的其他配置中无法观察到这种情况，因为我们为可读性设置了Y轴最大吞吐量的上限。

有两个原因导致写请求比读请求耗时更长。首先，写请求必须经过原子广播，这需要一些额外的处理

并增加了请求的延时。写入请求处理时间较长的另一个原因是，服务器必须确保在向领导者发送确认信息之前将事务记录在非易失性存储中。原则上，这个要求是过分的，但对于我们的生产系统，我们用性能来换取可靠性，因为ZooKeeper构成了应用的基础事实。我们使用更多的服务器来容忍更多的故障。我们通过将ZooKeeper的数据分割成多部分来提高写入量。

尖端的ZooKeeper组合。Gray等人[12]曾观察到复制和分区之间的这种性能权衡。

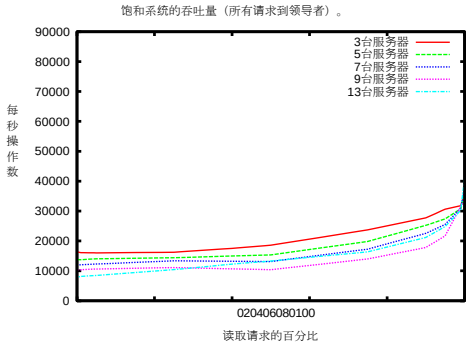

图6: 饱和系统的吞吐量，变化的是
当所有客户连接到领导者时，读与写的比例。

ZooKeeper能够通过在组成服务的服务器之间分配负载来实现如此高的吞吐量。我们可以分配负载，因为我们有宽松的一致性保证。胖胖的客户端将所有的请求都指向领导者。图6显示了如果我们不利用这种放松，强迫客户只连接到领导者，会发生什么。正如预期的那样，对于以读为主的工作负载，吞吐量要低得多。

但即使是以写为主的工作负载，吞吐量也较低。为客户提供服务所造成的额外的CPU和网络负载影响了领导者协调建议广播的能力，这反过来又大大影响了整体写性能。

原子广播协议完成了系统的大部分工作，因此对ZooKeeper的性能限制比其他组件更大。图7显示了原子广播组件的吞吐量。为了衡量其性能，我们通过直接在领导者处生成事务来模拟客户，所以没有客户连接或客户请求和回复。在最大的吞吐量下，原子广播组件成为CPU的约束。理论上，图7的性能将与ZooKeeper的性能相匹配，100%写入。然而，ZooKeeper的客户端通信、ACL检查和请求到事务的连接都是由CPU控制的。

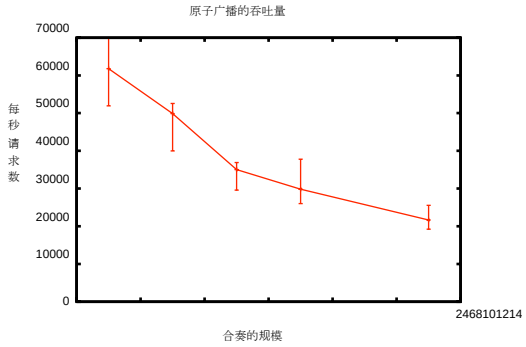

图7：孤立的原子广播组件的平均吞吐量。误差条表示最小值和最大值。

版本都需要CPU。对CPU的争夺降低了ZooKeeper的吞吐量，大大低于孤立的原子广播组件。因为ZooKeeper是一个重要的生产组件，到目前为止，我们对ZooKeeper的开发重点是正确性和健壮性。通过消除额外的拷贝、同一对象的多重序列化、更有效的内部数据结构等，有很多机会可以大幅提高性能。

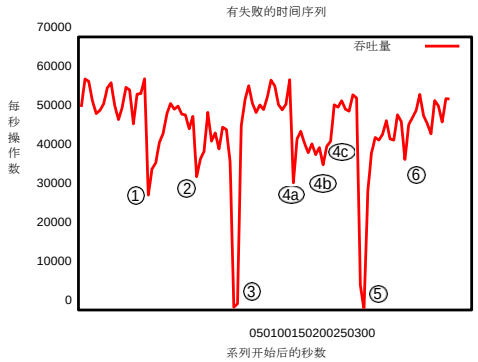

图8：故障时的吞吐量。

为了显示系统随着时间的推移在注入故障时的行为，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30%，这是我们预测的工作负载的一个保守的比例。我们定期地杀死一些服务器进程。图8显示了系统吞吐量随时间变化的情况。图中标记的事件如下。

1. 追随者的失败和恢复。
2. 不同追随者的失败和恢复。
3. 领导人的失败。
4. 前两分的两个追随者（a、b）失败，在第三分（c）恢复。
5. 领导人的失败。

6. 领导人的恢复。

从这张图中有几个重要的观察。首先，如果跟随者失败并迅速恢复，那么ZooKeeper能够在失败的情况下保持高吞吐量。一个追随者的失败并不会预先阻止服务器形成一个法定人数，而且只减少了服务器在失败前所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，足以预先阻止吞吐量大幅下降。在我们的观察中，ZooKeeper选举一个新的领导者需要不到*200ms*。因此，尽管服务器在几分之一秒内停止提供请求，但由于我们的采样周期为几秒钟，所以我们没有观察到吞吐量为零的情况。第三，即使追随者需要更多的时间来重新覆盖，一旦他们开始处理请求，ZooKeeper也能够再次提高吞吐量。在事件1、2和4之后，我们没有恢复到全部的吞吐量水平，原因之一是客户机只有在与跟随者的连接中断时才会切换跟随者。因此，在事件4之后，客户并没有重新分配自己，直到事件3和5的领导者失败。在实践中，这种不平衡随着时间的推移，随着客户的到来和离开而自行解决。

5.2 请求的延时

为了评估请求的延迟，我们创建了一个以Chubby基准[6]为模型的基准标记。我们创建了一个工作进程，简单地发送一个创建，等待它完成，发送一个新节点的异步删除，然后开始下一个创建。我们相应地改变工人的数量，在每次运行中，我们让每个工人创建50,000个节点。我们通过将完成的创建请求的数量除以所有工作者完成的总时间来计算吞吐量。

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

表2：每秒钟处理的创建请求。表2显示了我们基准测试的结果。创建这些请求包括1K的数据，而不是Chubby基准中的5字节，以更好地与我们的预期使用相吻合。即使有这些较大的请求，ZooKeeper的吞吐量也比Chubby公布的吞吐量高3倍多。单个ZooKeeper工作者基准的吞吐量表明，3个服务器的平均请求延迟为1.2ms，9个服务器为1.4ms。

	# 客户的数量		
# 障碍物的数量	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：以秒为单位的障碍物实验。每点是每个客户在五次运行中完成的平均时间。

5.3 障碍物的性能

在这个实验中，我们按顺序执行一些障碍，以评估用ZooKeeper实现的基元的性能。对于一定数量的障碍物b，每个客户端首先进入所有b个障碍物，然后依次离开所有b个障碍物。由于我们使用第2.4节的双壁垒算法，一个客户端首先等待所有其他客户端执行`enter()`程序，然后再进入下一个调用（与`leave()`类似）。

我们在表3中报告了我们的实验结果。在这个实验中，我们有50个、100个和200个客户端连续进入若干个障碍，$b \in \{200, 400, 800, 1600\}$。尽管一个应用程序可以有成千上万的ZooKeeper客户端，但通常只有一个更小的子集参与每个协调操作，因为客户端通常是根据应用程序的具体情况分组的。

从这个实验中观察到的两个有趣的现象是，处理所有障碍的时间随着障碍数量的增加而大致增加，这表明对数据树的同一部分的协同访问并没有产生任何意外的延迟，而且延迟的增加与客户的数量成正比。这是一个不使ZooKeeper服务饱和的顺序。事实上，我们观察到，即使客户以锁步方式进行，在所有情况下，障碍操作（进入和离开）的吞吐量都在每秒1,950到3,100次之间。在ZooKeeper操作中，这相当于每秒10,700到17,000次操作的吞吐量值。由于在我们的实现中，读和写的比例为4:1（80%的读操作），我们的基准代码使用的吞吐量与ZooKeeper可以实现的原始吞吐量（根据图5，超过40,000）相比要低得多。这是由于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，缓解分布式应用中的进程协调问题。为了实现这个目标，它的设计采用了以前的协调服务、容错系统、分布式算法和文件系统的思想。

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一个分布式锁服务[13]，并在计算机集群中共享信息[19]。最近，Chubby提出了一个为分布式应用管理咨询锁的系统[6]。Chubby分享了ZooKeeper的几个目标。它也有一个类似文件系统的接口，并且它使用一个协议来保证副本的一致性。然而，ZooKeeper不是一个锁服务。它可以被客户端用来实现锁，但它的API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。ZooKeeper客户端可以使用他们的本地副本来提供数据和管理手表，因为它的一致性模型要比Chubby宽松得多。这使得ZooKeeper能够提供比Chubby更高的性能，使应用程序能够更广泛地使用ZooKeeper。

文献中已经提出了一些容错系统，目的是为了减轻建立容错的分布式应用的问题。一个早期的系统是ISIS[5]。ISIS系统将抽象的类型规范转化为容错的分布式对象，从而使容错机制对用户透明。Horus[30]和Ensemble[31]是由ISIS发展而来的系统。ZooKeeper接纳了ISIS的虚拟同步概念。最后，Totem在一个利用局域网硬件广播的架构中保证了消息传递的总顺序[22]。ZooKeeper可以在各种网络拓扑结构下工作，这促使我们依靠服务器进程之间的TCP连接，而不假设任何特殊的拓扑结构或硬件特征。我们也没有公开ZooKeeper内部使用的任何精简的通信。

构建容错服务的一个重要技术是状态机复制[26]，而Paxos[20]是一种算法，能够有效实现异步系统的复制状态机。我们使用的算法与Paxos的一些特征相同，但它将共识所需的交易日志与数据树恢复所需的写前日志结合起来，以实现高效的实施。已经有一些关于实际实现拜占庭容忍复制状态机的协议建议[7, 10, 18, 1, 28]。ZooKeeper并不假设服务器是拜占庭的，但我们确实采用了诸如校验和和理智检查等机制来捕捉非恶意的拜占庭故障。Clement等人讨论了一种使ZooKeeper完全拜占庭式容错的方法，而无需修改当前的服务器代码库[9]。到目前为止，我们还没有观察到使用完全拜占庭容错协议所能防止的生产故障。[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，它依赖于一个基于Paxos的分布式锁服务。与Boxwood一样，ZooKeeper也是一个用于建立分布式系统的组件。然而，ZooKeeper有高性能的要求，并且更广泛地用于客户端应用程序。ZooKeeper提供了低级别的基元，应用程序用它来实现高级别的基元。

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个小子集，并增加了大多数文件系统不具备的功能，如排序保证和有条件写入。然而，ZooKeeper手表在精神上与AFS[16]的缓存回调相似。

Sinfonia[2]引入了*迷你交易*，这是一种构建可扩展的分布式系统的新范式。Sinfonia被设计用来存储应用数据，而ZooKeeper则存储应用元数据。ZooKeeper将其状态完全复制并保存在内存中，以获得高性能和一致的延迟。我们使用类似于文件系统的操作和排序，使功能类似于小型交易。znode是一个方便的抽象，我们在此基础上添加了手表，这是Sinfonia中缺少的功能。Dynamo[11]允许客户在一个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的键空间不是分层的。Dynamo也不为写入提供强大的耐久性和一致性保证，而是在读取时解决冲突。

DepSpace[4]使用元组空间来提供一个拜占庭容错的服务。像ZooKeeper一样，DepSpace使用一个简单的服务器接口，在客户端实现强大的同步化基元。虽然DepSpace的性能比ZooKeeper低得多，但它提供了更强的容错和保密性保证。

7 结论

ZooKeeper采用了一种无等待的方法来解决分布式系统中协调进程的问题，它向客户公开无等待的对象。我们发现ZooKeeper对雅虎内部和外部的一些应用很有用。ZooKeeper通过使用快速读取和观察，实现了每秒数十万次的操作，这些操作都是由本地副本提供的，从而实现了以读为主的工作负载。尽管我们对读和看的一致性保证似乎很弱，但我们的用例表明，这种组合允许我们在客户端实现高效和复杂的协调协议，即使读不是按优先级排序的，数据对象的实现是无等待的。事实证明，无等待的特性对高性能是至关重要的。

虽然我们只描述了几个应用，但还有很多其他的应用在使用ZooKeeper。我们相信这样的成功是由于其简单的接口和强大的抽象，人们可以通过这个接口实现。此外，由于ZooKeeper的高吞吐量，应用程序可以广泛使用它，而不仅仅是历程锁定。

鸣谢

我们要感谢Andrew Kornev和Runping Qi对ZooKeeper的贡献；感谢Zeke Huang和Mark Marchukov的宝贵反馈；感谢Brian Cooper和Laurence Ramontianu对ZooKeeper的早期贡献；Brian Bershad和Geoff Voelker对报告提出了重要意见。

参考文献

[1] M.Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie.可扩展的拜占庭容错服务。In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 59-74, New York, NY, USA, 2005.ACM.

[2] M.Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis.Sinfonia:构建可扩展分布式系统的新范式。In *SOSP '07: Proceedings of the 21st ACM symposium on Operating Systems Principles*, New York, NY, 2007.

[3] 亚马逊。亚马逊简单队列服务。`http://aws.amazon.com/sqs/`, 2008.

[4] A.N. Bessani, E. P. Alchieri, M. Correia, and J. da Silva Fraga.Depspace:一个杂乱无章的容错协调服务。在*第三届ACM SIGOPS/EuroSys欧洲系统会议论文集-EuroSys 2008*，2008年4月。

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

[6] M.Burrows.用于松散耦合的分布式系统的Chubby锁服务。*在第七届ACM/USENIX操作系统设计与实施研讨会（OSDI）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上线了。一个工程角度。*在第26届ACM分布式计算原理（PODC）年度研讨会上*，2007年8月。

[9] A.Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche.UpRight集群服务。*在第22届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, and L. Shira.Hq replication:一种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。亚马逊的高可用键值存储。在*SOSP '07: 第21届ACM运营系统原理研讨会论文集*，美国纽约，2007。ACM出版社。

[12] J.Gray, P. Helland, P. O'Neil, and D. Shasha.复制的危险和解决方案。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.In *Proceedings of IEEE 9th Symposium on Reliable Distributed Systems*, October. 1990.

[14] M.Herlihy.Wait-free synchronization.*ACM Transactions on Programming Languages and Systems*, 13(1), 1991.

[15] M.Herlihy and J. Wing.Linearizability: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和性能。*ACM Trans.Comput.Syst.*, 6(1), 1988.

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

[18] R.Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong。Zyzzyva：推测的拜占庭容错。*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）：一个紧密耦合的分布式系统。*SIGOPS Oper.Syst.Rev.*, 19(5), 1985.

[20] L.Lamport.The part-time parliament.*ACM Transactions on Computer Systems*, 16(2), May 1998.

[21] J.MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L.周晓明.Boxwood:抽象是存储基础设施的基础。In *Proceedings of the 6th ACM/USENIX Sympo- sium on Operating Systems Design and Implementation（OSDI）*, 2004.

[22] L.Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, C. Lingley-Papadopoulos, and T. Archambault.图腾系统。在*第25届国际容错计算研讨会的论文集中*，1995年6月。

[23] S.Mullender，编辑。*分布式系统，第二版*。ACM Press, New York, NY, USA, 1993.

[24] B.Reed and F. P. Junqueira.一个简单的完全有序的广投协议。在*LADIS '08:第二届大规模分布式系统和中间件研讨会论文集》*，第1-6页，美国纽约，2008。ACM.

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

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

[27] A.Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein。ACMS。Akamai的配置管理系统。在*NSDI*，2005年。

[28] A.Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno：最终一致的拜占庭-故障容忍。在*NSDI'09。第六届USENIX网络系统设计与实施研讨会论文集*，第169-184页，美国加州伯克利，2009年。USENIX协会。

[29] 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 and K. Birman.Horus,

一个灵活的群体通信系统.*Communications of the ACM*, 39(16), Apr. 1996.

[31] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D.Karr.使用合集构建适应性系统.*软件 - 实践与经验*，28（5），1998年7月。