ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Grid
{phunt,mahadev}@yahoo-inc.com

Flavio P. Junqueira and Benjamin Reed
Yahoo! Research
{fpj,breed}@yahoo-inc.com

Abstract

在本文中，我们描述了 ZooKeeper，这是一种用于协调分布式应用程序进程的服务。由于 ZooKeeper 是关键基础设施的一部分，ZooKeeper 旨在提供一个简单且高性能的内核，用于在客户端构建更复杂的协调原语。它将来自组消息传递、共享寄存器和分布式锁服务的元素合并到一个复制的集中式服务中。 ZooKeeper 公开的接口具有共享寄存器的免等待方面，具有类似于分布式文件系统的缓存失效的事件驱动机制，以提供简单但功能强大的协调服务。

In this paper, we describe ZooKeeper, a service for coordinating processes of distributed applications. Since ZooKeeper is part of critical infrastructure, ZooKeeper aims to provide a simple and high performance kernel for building more complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, centralized service. The interface exposed by ZooKeeper has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems to provide a simple, yet powerful coordination service.

ZooKeeper 接口支持高性能服务实现。除了无等待属性之外，ZooKeeper 还为每个客户端提供请求的 FIFO 执行保证和所有更改 ZooKeeper 状态的请求的线性化。这些设计决策可以实现高性能处理管道，本地服务器可以满足读取请求。我们展示了目标工作负载，2:1 到 100:1 的读写比率，ZooKeeper 每秒可以处理数万到数十万个事务。这种性能允许 ZooKeeper 被客户端应用程序广泛使用。

The ZooKeeper interface enables a high-performance service implementation. In addition to the wait-free property, ZooKeeper provides a per client guarantee of FIFO execution of requests and linearizability for all requests that change the ZooKeeper state. These design decisions enable the implementation of a high performance processing pipeline with read requests being satisfied by local servers. We show for the target workloads, 2:1 to 100:1 read to write ratio, that ZooKeeper can handle tens to hundreds of thousands of transactions per second. This performance allows ZooKeeper to be used extensively by client applications.

1 Introduction

大规模分布式应用程序需要不同形式的协调。配置是最基本的协调形式之一。在最简单的形式中，配置只是系统进程的操作参数列表，而更复杂的系统具有动态配置参数。组成员身份和领导选举在分布式系统中也很常见：通常进程需要知道哪些其他进程还活着，以及这些进程负责什么。锁构成了一个强大的协调原语，它实现了对关键资源的互斥访问。

Large-scale distributed applications require different forms of coordination. Configuration is one of the most basic forms of coordination. In its simplest form, configuration is just a list of operational parameters for the system processes, whereas more sophisticated systems have dynamic configuration parameters. Group membership and leader election are also common in distributed systems: often processes need to know which other processes are alive and what those processes are in charge of. Locks constitute a powerful coordination primitive that implement mutually exclusive access to critical resources.

One approach to coordination is to develop services for each of the different coordination needs. For example, Amazon Simple Queue Service [3] focuses specifically on queuing. Other services have been developed specifically for leader election [25] and configuration [27]. Services that implement more powerful primitives can be used to implement less powerful ones. For example, Chubby [6] is a locking service with strong synchronization guarantees. Locks can then be used to implement leader election, group membership, etc.

一种协调方法是为每种不同的协调需求开发服务。例如，Amazon Simple Queue Service [3] 专门关注排队。其他服务已经专门为领导人选举 [25] 和配置 [27] 而开发。实现更强大原语的服务可用于实现功能较弱的原语。例如，Chubby [6] 是一种具有强同步保证的锁定服务。然后可以使用锁来实现领导者选举、组成员资格等。

When designing our coordination service, we moved away from implementing specific primitives on the server side, and instead we opted for exposing an API that enables application developers to implement their own primitives. Such a choice led to the implementation of a *coordination kernel* that enables new primitives without requiring changes to the service core. This approach enables multiple forms of coordination adapted to the requirements of applications, instead of constraining developers to a fixed set of primitives.

在设计协调服务时，我们不再在服务器端实现特定的原语，而是选择公开一个 API，使应用程序开发人员能够实现自己的原语。这样的选择导致了一个协调内核的实现，该内核支持新的原语，而无需更改服务核心。这种方法支持多种形式的协调，以适应应用程序的需求，而不是将开发人员限制在一组固定的原语中。

When designing the API of ZooKeeper, we moved away from blocking primitives, such as locks. Blocking primitives for a coordination service can cause, among other problems, slow or faulty clients to impact negatively the performance of faster clients. The implementation of the service itself becomes more complicated if processing requests depends on responses and failure detection of other clients. Our system, Zookeeper, hence implements an API that manipulates simple *wait-free* data objects organized hierarchically as in file systems. In fact, the ZooKeeper API resembles the one of any other file system, and looking at just the API signatures, ZooKeeper seems to be Chubby without the lock methods, open, and close. Implementing wait-free data objects, however, differentiates ZooKeeper significantly from systems based on blocking primitives such as locks.

在设计 ZooKeeper 的 API 时，我们远离了阻塞原语，例如锁。除其他问题外，阻塞协调服务的原语可能会导致客户端缓慢或故障，从而对更快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，则服务本身的实现会变得更加复杂。因此，我们的系统 Zookeeper 实现了一个 API，该 API 可以像在文件系统中一样操作分层组织的简单无等待数据对象。实际上，ZooKeeper API 类似于任何其他文件系统之一，仅查看 API 签名，ZooKeeper 似乎是 Chubby，没有锁定方法、打开和关闭。然而，实现无等待数据对象将 ZooKeeper 与基于阻塞原语（如锁）的系统显着区别开来。

Although the wait-free property is important for per-

尽管无等待属性对性能和容错很重要，但它对协调来说是不够的。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）客户端顺序和可线性化的写入可以有效地实现服务，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，ZooKeeper实现了一个通用对象[14]。

ZooKeeper 服务由一组使用复制来实现高可用性和性能的服务器组成。它的高性能使包含大量进程的应用程序能够使用这种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，它允许我们处理成百上千个未完成的请求，同时仍然实现低延迟。这样的管道自然能够以 FIFO 顺序从单个客户端执行操作。保证 FIFO 客户端顺序使客户端能够异步提交操作。使用异步操作，客户端一次可以有多个未完成的操作。这个特性是可取的，例如，当一个新的客户端成为领导者并且它必须操作元数据并相应地更新它时。在没有多个未完成操作的可能性的情况下，初始化时间可以是秒级而不是亚秒级。

为了保证更新操作满足线性化，我们实现了一个基于领导者的原子广播协议[23]，称为 Zab [24]。然而，ZooKeeper 应用程序的典型工作负载以读取操作为主，因此需要扩展读取吞吐量。在 ZooKeeper 中，服务器在本地处理读取操作，我们不使用 Zab 对它们进行完全排序。

在客户端缓存数据是提高读取性能的一项重要技术。例如，对于进程来说，缓存当前领导者的标识符是很有用的，而不是每次需要知道领导者时都去探查 ZooKeeper。 ZooKeeper 使用监视机制使客户端能够缓存数据，而无需直接管理客户端缓存。使用这种机制，客户端可以监视给定数据对象的更新，并在更新时接收通知。 Chubby 直接管理客户端缓存。它阻止更新以使缓存正在更改的数据的所有客户端的缓存无效。在这种设计下，如果这些客户端中的任何一个运行缓慢或出现故障，更新就会延迟。 Chubby 使用租约来防止有故障的客户端无限期地阻塞系统。然而，租约只限制了缓慢或故障客户端的影响，而 ZooKeeper 监视则完全避免了这个问题。

formance and fault tolerance, it is not sufficient for coordination. We have also to provide order guarantees for operations. In particular, we have found that guaranteeing both *FIFO client ordering* of all operations and *linearizable writes* enables an efficient implementation of the service and it is sufficient to implement coordination primitives of interest to our applications. In fact, we can implement consensus for any number of processes with our API, and according to the hierarchy of Herlihy, ZooKeeper implements a universal object [14].

The ZooKeeper service comprises an ensemble of servers that use replication to achieve high availability and performance. Its high performance enables applications comprising a large number of processes to use such a coordination kernel to manage all aspects of coordination. We were able to implement ZooKeeper using a simple pipelined architecture that allows us to have hundreds or thousands of requests outstanding while still achieving low latency. Such a pipeline naturally enables the execution of operations from a single client in FIFO order. Guaranteeing FIFO client order enables clients to submit operations asynchronously. With asynchronous operations, a client is able to have multiple outstanding operations at a time. This feature is desirable, for example, when a new client becomes a leader and it has to manipulate metadata and update it accordingly. Without the possibility of multiple outstanding operations, the time of initialization can be of the order of seconds instead of sub-second.

To guarantee that update operations satisfy linearizability, we implement a leader-based atomic broadcast protocol [23], called Zab [24]. A typical workload of a ZooKeeper application, however, is dominated by read operations and it becomes desirable to scale read throughput. In ZooKeeper, servers process read operations locally, and we do not use Zab to totally order them.

Caching data on the client side is an important technique to increase the performance of reads. For example, it is useful for a process to cache the identifier of the current leader instead of probing ZooKeeper every time it needs to know the leader. ZooKeeper uses a watch mechanism to enable clients to cache data without managing the client cache directly. With this mechanism, a client can watch for an update to a given data object, and receive a notification upon an update. Chubby manages the client cache directly. It blocks updates to invalidate the caches of all clients caching the data being changed. Under this design, if any of these clients is slow or faulty, the update is delayed. Chubby uses leases to prevent a faulty client from blocking the system indefinitely. Leases, however, only bound the impact of slow or faulty clients, whereas ZooKeeper watches avoid the problem altogether.

In this paper we discuss our design and implementation of ZooKeeper. With ZooKeeper, we are able to implement all coordination primitives that our applications require, even though only writes are linearizable. To validate our approach we show how we implement some coordination primitives with ZooKeeper.

To summarize, in this paper our main contributions are:

Coordination kernel: We propose a wait-free coordination service with relaxed consistency guarantees for use in distributed systems. In particular, we describe our design and implementation of a *coordination kernel*, which we have used in many critical applications to implement various coordination techniques.

Coordination recipes: We show how ZooKeeper can be used to build higher level coordination primitives, even blocking and strongly consistent primitives, that are often used in distributed applications.

Experience with Coordination: We share some of the ways that we use ZooKeeper and evaluate its performance.

2 The ZooKeeper service

Clients submit requests to ZooKeeper through a client API using a ZooKeeper client library. In addition to exposing the ZooKeeper service interface through the client API, the client library also manages the network connections between the client and ZooKeeper servers.

In this section, we first provide a high-level view of the ZooKeeper service. We then discuss the API that clients use to interact with ZooKeeper.

Terminology. In this paper, we use *client* to denote a user of the ZooKeeper service, *server* to denote a process providing the ZooKeeper service, and *znode* to denote an in-memory data node in the ZooKeeper data, which is organized in a hierarchical namespace referred to as the *data tree*. We also use the terms update and write to refer to any operation that modifies the state of the data tree. Clients establish a *session* when they connect to ZooKeeper and obtain a session handle through which they issue requests.

2.1 Service overview

ZooKeeper provides to its clients the abstraction of a set of data nodes (znodes), organized according to a hierarchical name space. The znodes in this hierarchy are data objects that clients manipulate through the ZooKeeper API. Hierarchical name spaces are commonly used in file systems. It is a desirable way of organizing data objects, since users are used to this abstraction and it enables better organization of application meta-data. To refer to a

在本文中，我们讨论了 ZooKeeper 的设计和实现。使用 ZooKeeper，我们能够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，我们展示了如何使用 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：我们提出了一种在分布式系统中使用的具有宽松一致性保证的无等待协调服务。特别是，我们描述了协调内核的设计和实现，我们在许多关键应用程序中使用它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，甚至是阻塞和强一致的原语，这些原语经常用于分布式应用程序中。

协调经验：我们分享了一些使用 ZooKeeper 并评估其性能的方式。

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 公开 ZooKeeper 服务接口外，客户端库还管理客户端和 ZooKeeper 服务器之间的网络连接。在本节中，我们首先提供 ZooKeeper 服务的高级视图。然后我们讨论客户端用来与 ZooKeeper 交互的 API。

术语。在本文中，我们用client表示ZooKeeper服务的用户，server表示提供ZooKeeper服务的进程，znode表示ZooKeeper数据中的内存数据节点，它被组织在一个层次化的命名空间中，称为作为数据树。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 并获取会话句柄时建立会话，通过该句柄发出请求。

ZooKeeper 向其客户端提供一组数据节点（znode）的抽象，这些节点根据分层名称空间进行组织。此层次结构中的 znode 是客户端通过 ZooKeeper API 操作的数据对象。分层名称空间通常用于文件系统。这是一种组织数据对象的理想方式，因为用户已经习惯了这种抽象，并且它可以更好地组织应用程序元数据。

给定 znode，我们使用标准的 UNIX 符号表示文件系统路径。例如，我们使用 /A/B/C 来表示 znode C 的路径，其中 C 以 B 作为其父节点，B 以 A 作为其父节点。所有 znode 都存储数据，并且除了临时 znode 之外的所有 znode 都可以有子节点。

客户端可以创建两种类型的 znode： 常规：客户端通过显式创建和删除常规 znode 来操作它们；短暂的：客户端创建这样的 znode，他们要么显式删除它们，要么让系统在创建它们的会话终止时自动删除它们（故意或由于失败）。

此外，在创建新的znode时，客户机可以设置顺序标志。使用顺序标志集创建的节点的名称后面附加了一个单调递增的计数器值。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的名称中的值。

ZooKeeper实现了监视功能，允许客户端在不需要轮询的情况下及时接收更改通知。当客户机发出设置了监视标志的读取操作时，该操作将正常完成，但服务器承诺在返回的信息发生更改时通知客户机。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，它们将被注销。监视表明发生了变化，但不提供变化。例如，如果客户机在“/foo”更改两次之前发出getData（“/foo”，true），则客户机将收到一个监视事件，告诉客户机“/foo”的数据已更改。会话事件（如连接丢失事件）也会被发送到监视回调，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带有简化API的文件系统，只有完整的数据读写，或者是一个带有层次结构的键/值表-

given znode, we use the standard UNIX notation for file system paths. For example, we use /A/B/C to denote the path to znode C, where C has B as its parent and B has A as its parent. All znodes store data, and all znodes, except for ephemeral znodes, can have children.

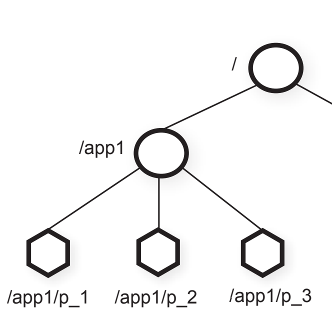

Figure 1: Illustration of ZooKeeper hierarchical name space.

There are two types of znodes that a client can create:

Regular: Clients manipulate regular znodes by creating and deleting them explicitly;

Ephemeral: Clients create such znodes, and they either delete them explicitly, or let the system remove them automatically when the session that creates them terminates (deliberately or due to a failure).

Additionally, when creating a new znode, a client can set a *sequential* flag. Nodes created with the sequential flag set have the value of a monotonically increasing counter appended to its name. If n is the new znode and p is the parent znode, then the sequence value of n is never smaller than the value in the name of any other sequential znode ever created under p.

ZooKeeper implements watches to allow clients to receive timely notifications of changes without requiring polling. When a client issues a read operation with a watch flag set, the operation completes as normal except that the server promises to notify the client when the information returned has changed. Watches are one-time triggers associated with a session; they are unregistered once triggered or the session closes. Watches indicate that a change has happened, but do not provide the change. For example, if a client issues a getData(‘‘/foo’’, true) before “/foo” is changed twice, the client will get one watch event telling the client that data for “/foo” has changed. Session events, such as connection loss events, are also sent to watch callbacks so that clients know that watch events may be delayed.

Data model. The data model of ZooKeeper is essentially a file system with a simplified API and only full data reads and writes, or a key/value table with hierarchical keys. The hierarchal namespace is useful for allocating subtrees for the namespace of different applications and for setting access rights to those subtrees. We also exploit the concept of directories on the client side to build higher level primitives as we will see in section 2.4.

Unlike files in file systems, znodes are not designed for general data storage. Instead, znodes map to abstractions of the client application, typically corresponding to meta-data used for coordination purposes. To illustrate, in Figure 1 we have two subtrees, one for Application 1 (/app1) and another for Application 2 (/app2). The subtree for Application 1 implements a simple group membership protocol: each client process p_i creates a znode p_i under /app1, which persists as long as the process is running.

Although znodes have not been designed for general data storage, ZooKeeper does allow clients to store some information that can be used for meta-data or configuration in a distributed computation. For example, in a leader-based application, it is useful for an application server that is just starting to learn which other server is currently the leader. To accomplish this goal, we can have the current leader write this information in a known location in the znode space. Znodes also have associated meta-data with time stamps and version counters, which allow clients to track changes to znodes and execute conditional updates based on the version of the znode.

Sessions. A client connects to ZooKeeper and initiates a session. Sessions have an associated timeout. ZooKeeper considers a client faulty if it does not receive anything from its session for more than that timeout. A session ends when clients explicitly close a session handle or ZooKeeper detects that a clients is faulty. Within a session, a client observes a succession of state changes that reflect the execution of its operations. Sessions enable a client to move transparently from one server to another within a ZooKeeper ensemble, and hence persist across ZooKeeper servers.

2.2 Client API

We present below a relevant subset of the ZooKeeper API, and discuss the semantics of each request.

create(path, data, flags): Creates a znode with path name path, stores data[] in it, and returns the name of the new znode. flags enables a client to select the type of znode: regular, ephemeral, and set the sequential flag;

delete(path, version): Deletes the znode path if that znode is at the expected version;

exists(path, watch): Returns true if the znode with path name path exists, and returns false otherwise. The watch flag enables a client to set a

分层名称空间对于为不同应用程序的名称空间分配子树以及设置对这些子树的访问权限非常有用。我们还利用客户端目录的概念来构建更高级别的原语，如第2.4节所示。

与文件系统中的文件不同，znode不是为通用数据存储而设计的。相反，znode映射到客户机应用程序的抽象，通常对应于用于协调目的的元数据。为了说明这一点，在图1中我们有两个子树，一个用于应用程序1 (/app1)，另一个用于应用程序2 (/app2)。应用程序1的子树实现了一个简单的组成员协议:每个客户端进程pi在/app1下创建一个znode pi，只要进程在运行，这个znode pi就会持续存在。

虽然znode并不是为通用数据存储而设计的，但ZooKeeper确实允许客户端存储一些信息，这些信息可以用于元数据或分布式计算中的配置。例如，在基于leader的应用程序中，对于刚刚开始了解其他服务器当前是leader的应用服务器来说，这是很有用的。为了实现这一目标，我们可以让当前leader将此信息写入znode空间中的一个已知位置。znode还将元数据与时间戳和版本计数器相关联，这允许客户端跟踪对znode的更改，并根据znode的版本执行有条件的更新。

客户端连接到ZooKeeper并发起会话。会话有一个相关的超时。如果超过这个超时时间，ZooKeeper没有从它的会话中收到任何消息，则认为客户端故障。当客户端显式关闭会话句柄或ZooKeeper检测到客户端故障时，会话结束。在会话中，客户机观察反映其操作执行情况的一系列状态更改。会话使客户端能够在ZooKeeper集合中透明地从一台服务器移动到另一台服务器，从而在ZooKeeper服务器之间保持一致。

我们在下面展示ZooKeeper API的相关子集，并讨论每个请求的语义。

create（path，data，flags）：创建一个路径名为path的znode，在其中存储数据[]，并返回新znode的名称。flags允许客户端选择znode的类型：常规、临时，并设置顺序标志；

delete（path，version）：如果znode为预期版本，则删除znode路径；

exists(path, watch)：如果路径名为 path 的 znode 存在，则返回 true，否则返回 false。 watch 标志使客户端能够在 znode 上设置监视；

getData（path，watch）：返回与znode关联的数据和元数据，如版本信息。watch标志的工作方式与exists（）的工作方式相同，只是ZooKeeper在znode不存在时不设置watch；

setData（路径、数据、版本）：如果版本号是znode的当前版本，则将数据[]写入znode路径；

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

;sync(path):在操作开始时等待所有更新，并将其传播到客户端连接的服务器。该路径当前被忽略

所有方法都通过 API 提供同步和异步版本。当应用程序需要执行单个 ZooKeeper 操作并且没有并发任务要执行时，应用程序会使用同步 API，因此它会进行必要的 ZooKeeper 调用和阻塞。然而，异步 API 使应用程序能够同时执行多个未完成的 ZooKeeper 操作和其他并行执行的任务。ZooKeeper 客户端保证每个操作的相应回调按顺序调用。

请注意，ZooKeeper 不使用句柄来访问 znode。相反，每个请求都包含正在操作的 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。

每个更新方法都有一个预期的版本号，它可以实现条件更新。如果 znode 的实际版本号与预期的版本号不匹配，则更新将失败并出现意外的版本错误。如果版本号为-1，则不进行版本检查。

ZooKeeper 有两个基本的顺序保证：
线性化写入：所有更新 ZooKeeper 状态的请求都是可序列化的并且尊重优先级；
FIFO 客户端顺序：来自给定客户端的所有请求都按照客户端发送的顺序执行。

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-linearizability（异步线性化）。在 Herlihy 对线性化的原始定义中，客户端一次只能有一个未完成的操作（客户端是一个线程）。

watch on the znode;

getData(path, watch): Returns the data and meta-data, such as version information, associated with the znode. The `watch` flag works in the same way as it does for `exists()`, except that ZooKeeper does not set the watch if the znode does not exist;

setData(path, data, version): Writes `data[]` to znode `path` if the version number is the current version of the znode;

getChildren(path, watch): Returns the set of names of the children of a znode;

sync(path): Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to. The path is currently ignored.

All methods have both a synchronous and an asynchronous version available through the API. An application uses the synchronous API when it needs to execute a single ZooKeeper operation and it has no concurrent tasks to execute, so it makes the necessary ZooKeeper call and blocks. The asynchronous API, however, enables an application to have both multiple outstanding ZooKeeper operations and other tasks executed in parallel. The ZooKeeper client guarantees that the corresponding callbacks for each operation are invoked in order.

Note that ZooKeeper does not use handles to access znodes. Each request instead includes the full path of the znode being operated on. Not only does this choice simplifies the API (no `open()` or `close()` methods), but it also eliminates extra state that the server would need to maintain.

Each of the update methods take an expected version number, which enables the implementation of conditional updates. If the actual version number of the znode does not match the expected version number the update fails with an unexpected version error. If the version number is -1, it does not perform version checking.

2.3 ZooKeeper guarantees

ZooKeeper has two basic ordering guarantees:

Linearizable writes: all requests that update the state of ZooKeeper are serializable and respect precedence;

FIFO client order: all requests from a given client are executed in the order that they were sent by the client.

Note that our definition of linearizability is different from the one originally proposed by Herlihy [15], and we call it *A-linearizability* (asynchronous linearizability). In the original definition of linearizability by Herlihy, a client is only able to have one outstanding operation at a time (a client is one thread). In ours, we allow a client to have multiple outstanding operations, and consequently we can choose to guarantee no specific order for outstanding operations of the same client or to guarantee FIFO order. We choose the latter for our property. It is important to observe that all results that hold for linearizable objects also hold for A-linearizable objects because a system that satisfies A-linearizability also satisfies linearizability. Because only update requests are A-linearizable, ZooKeeper processes read requests locally at each replica. This allows the service to scale linearly as servers are added to the system.

To see how these two guarantees interact, consider the following scenario. A system comprising a number of processes elects a leader to command worker processes. When a new leader takes charge of the system, it must change a large number of configuration parameters and notify the other processes once it finishes. We then have two important requirements:

- As the new leader starts making changes, we do not want other processes to start using the configuration that is being changed;
- If the new leader dies before the configuration has been fully updated, we do not want the processes to use this partial configuration.

Observe that distributed locks, such as the locks provided by Chubby, would help with the first requirement but are insufficient for the second. With ZooKeeper, the new leader can designate a path as the *ready* znode; other processes will only use the configuration when that znode exists. The new leader makes the configuration change by deleting *ready*, updating the various configuration znodes, and creating *ready*. All of these changes can be pipelined and issued asynchronously to quickly update the configuration state. Although the latency of a change operation is of the order of 2 milliseconds, a new leader that must update 5000 different znodes will take 10 seconds if the requests are issued one after the other; by issuing the requests asynchronously the requests will take less than a second. Because of the ordering guarantees, if a process sees the *ready* znode, it must also see all the configuration changes made by the new leader. If the new leader dies before the *ready* znode is created, the other processes know that the configuration has not been finalized and do not use it.

The above scheme still has a problem: what happens if a process sees that *ready* exists before the new leader starts to make a change and then starts reading the configuration while the change is in progress. This problem is solved by the ordering guarantee for the notifications: if a client is watching for a change, the client will see the notification event before it sees the new state of the system after the change is made. Consequently, if the process that reads the *ready* znode requests to be notified of changes to that znode, it will see a notification inform-

在我们的方案中，我们允许一个客户端有多个未完成的操作，因此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保证 FIFO 顺序。我们选择后者作为我们的财产。重要的是要观察到适用于可线性化对象的所有结果也适用于可线性化对象，因为满足 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 在每个副本本地处理读取请求。这允许服务随着服务器添加到系统而线性扩展。

要了解这两个保证如何相互作用，请考虑以下场景。一个由多个进程组成的系统会选举一个领导者来指挥工作进程。当一个新的领导者接管系统时，它必须更改大量的配置参数，并在完成后通知其他进程。然后我们有两个重要的要求：

- 当新领导开始进行更改时，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之前死亡，我们不希望进程使用这个部分配置。

请注意，分布式锁（例如 Chubby 提供的锁）有助于满足第一个需求，但不足以满足第二个需求。使用 ZooKeeper，新的 leader 可以指定一条路径作为准备好的 znode；其他进程将仅在该 znode 存在时使用该配置。新领导者通过删除就绪、更新各种配置 znode 和创建就绪来进行配置更改。所有这些更改都可以流水线化并异步发布，以快速更新配置状态。尽管更改操作的延迟大约为 2 毫秒，但如果请求一个接一个地发出，则必须更新 5000 个不同 znode 的新领导者将需要 10 秒；通过异步发出请求，请求将花费不到一秒钟的时间。由于排序保证，如果一个进程看到准备好的 znode，它也必须看到新领导者所做的所有配置更改。如果新的leader在准备好的znode创建之前就死掉了，其他进程就知道配置还没有最终确定，不会使用它。

上述方案还有一个问题：如果一个进程在新的领导者开始进行更改之前看到准备就绪，然后在更改进行时开始读取配置，会发生什么情况。这个问题是通过通知的顺序保证来解决的：如果客户端正在监视更改，客户端将在更改后看到系统的新状态之前看到通知事件

ing the client of the change before it can read any of the new configuration.

Another problem can arise when clients have their own communication channels in addition to ZooKeeper. For example, consider two clients A and B that have a shared configuration in ZooKeeper and communicate through a shared communication channel. If A changes the shared configuration in ZooKeeper and tells B of the change through the shared communication channel, B would expect to see the change when it re-reads the configuration. If B's ZooKeeper replica is slightly behind A's, it may not see the new configuration. Using the above guarantees B can make sure that it sees the most up-to-date information by issuing a write before re-reading the configuration. To handle this scenario more efficiently ZooKeeper provides the `sync` request: when followed by a read, constitutes a *slow read*. `sync` causes a server to apply all pending write requests before processing the read without the overhead of a full write. This primitive is similar in idea to the `flush` primitive of ISIS [5].

ZooKeeper also has the following two liveness and durability guarantees: if a majority of ZooKeeper servers are active and communicating the service will be available; and if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover.

2.4 Examples of primitives

In this section, we show how to use the ZooKeeper API to implement more powerful primitives. The ZooKeeper service knows nothing about these more powerful primitives since they are entirely implemented at the client using the ZooKeeper client API. Some common primitives such as group membership and configuration management are also wait-free. For others, such as rendezvous, clients need to wait for an event. Even though ZooKeeper is wait-free, we can implement efficient blocking primitives with ZooKeeper. ZooKeeper's ordering guarantees allow efficient reasoning about system state, and watches allow for efficient waiting.

Configuration Management ZooKeeper can be used to implement dynamic configuration in a distributed application. In its simplest form configuration is stored in a znode, z_c. Processes start up with the full pathname of z_c. Starting processes obtain their configuration by reading z_c with the watch flag set to true. If the configuration in z_c is ever updated, the processes are notified and read the new configuration, again setting the watch flag to true.

Note that in this scheme, as in most others that use watches, watches are used to make sure that a process has the most recent information. For example, if a process watching z_c is notified of a change to z_c and before it can issue a read for z_c there are three more changes to z_c, the process does not receive three more notification events. This does not affect the behavior of the process, since those three events would have simply notified the process of something it already knows: the information it has for z_c is stale.

Rendezvous Sometimes in distributed systems, it is not always clear a priori what the final system configuration will look like. For example, a client may want to start a master process and several worker processes, but the starting processes is done by a scheduler, so the client does not know ahead of time information such as addresses and ports that it can give the worker processes to connect to the master. We handle this scenario with ZooKeeper using a rendezvous znode, z_r, which is an node created by the client. The client passes the full pathname of z_r as a startup parameter of the master and worker processes. When the master starts it fills in z_r with information about addresses and ports it is using. When workers start, they read z_r with watch set to true. If z_r has not been filled in yet, the worker waits to be notified when z_r is updated. If z_r is an ephemeral node, master and worker processes can watch for z_r to be deleted and clean themselves up when the client ends.

Group Membership We take advantage of ephemeral nodes to implement group membership. Specifically, we use the fact that ephemeral nodes allow us to see the state of the session that created the node. We start by designating a znode, z_g to represent the group. When a process member of the group starts, it creates an ephemeral child znode under z_g. If each process has a unique name or identifier, then that name is used as the name of the child znode; otherwise, the process creates the znode with the SEQUENTIAL flag to obtain a unique name assignment. Processes may put process information in the data of the child znode, addresses and ports used by the process, for example.

After the child znode is created under z_g the process starts normally. It does not need to do anything else. If the process fails or ends, the znode that represents it under z_g is automatically removed.

Processes can obtain group information by simply listing the children of z_g. If a process wants to monitor changes in group membership, the process can set the watch flag to true and refresh the group information (always setting the watch flag to true) when change notifications are received.

Simple Locks Although ZooKeeper is not a lock service, it can be used to implement locks. Applications using ZooKeeper usually use synchronization primitives tailored to their needs, such as those shown above. Here we show how to implement locks with ZooKeeper to show that it can implement a wide variety of general synchronization primitives.

The simplest lock implementation uses “lock files”. The lock is represented by a znode. To acquire a lock, a client tries to create the designated znode with the EPHEMERAL flag. If the create succeeds, the client holds the lock. Otherwise, the client can read the znode with the watch flag set to be notified if the current leader dies. A client releases the lock when it dies or explicitly deletes the znode. Other clients that are waiting for a lock try again to acquire a lock once they observe the znode being deleted.

While this simple locking protocol works, it does have some problems. First, it suffers from the herd effect. If there are many clients waiting to acquire a lock, they will all vie for the lock when it is released even though only one client can acquire the lock. Second, it only implements exclusive locking. The following two primitives show how both of these problems can be overcome.

Simple Locks without Herd Effect We define a lock znode l to implement such locks. Intuitively we line up all the clients requesting the lock and each client obtains the lock in order of request arrival. Thus, clients wishing to obtain the lock do the following:

Lock

```
1 n = create(l + “/lock-”, EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

Unlock

```
1 delete(n)
```

The use of the SEQUENTIAL flag in line 1 of `Lock` orders the client’s attempt to acquire the lock with respect to all other attempts. If the client’s znode has the lowest sequence number at line 3, the client holds the lock. Otherwise, the client waits for deletion of the znode that either has the lock or will receive the lock before this client’s znode. By only watching the znode that precedes the client’s znode, we avoid the herd effect by only waking up one process when a lock is released or a lock request is abandoned. Once the znode being watched by the client goes away, the client must check if it now holds the lock. (The previous lock request may have been abandoned and there is a znode with a lower sequence number still waiting for or holding the lock.)

Releasing a lock is as simple as deleting the znode n that represents the lock request. By using the EPHEMERAL flag on creation, processes that crash will automatically cleanup any lock requests or release any locks that they may have.

In summary, this locking scheme has the following advantages:

1. The removal of a znode only causes one client to wake up, since each znode is watched by exactly one other client, so we do not have the herd effect;
2. There is no polling or timeouts;
3. Because of the way we have implemented locking, we can see by browsing the ZooKeeper data the amount of lock contention, break locks, and debug locking problems.

Read/Write Locks To implement read/write locks we change the lock procedure slightly and have separate read lock and write lock procedures. The unlock procedure is the same as the global lock case.

Write Lock

```
1  n = create(l + “/write-”, EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

Read Lock

```
1  n = create(l + “/read-”, EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if no write znodes lower than n in C, exit
4  p = write znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 3
```

This lock procedure varies slightly from the previous locks. Write locks differ only in naming. Since read locks may be shared, lines 3 and 4 vary slightly because only earlier write lock znodes prevent the client from obtaining a read lock. It may appear that we have a “herd effect” when there are several clients waiting for a read lock and get notified when the “write-” znode with the lower sequence number is deleted; in fact, this is a desired behavior, all those read clients should be released since they may now have the lock.

Double Barrier Double barriers enable clients to synchronize the beginning and the end of a computation. When enough processes, defined by the barrier threshold, have joined the barrier, processes start their computation and leave the barrier once they have finished. We represent a barrier in ZooKeeper with a znode, referred to as b. Every process p registers with b – by creating a znode as a child of b – on entry, and unregisters – removes the child – when it is ready to leave. Processes can enter the barrier when the number of child znodes of b exceeds the barrier threshold. Processes can leave the barrier when all of the processes have removed their children. We use watches to efficiently wait for enter and exit conditions to be satisfied. To enter, processes watch for the existence of a `ready` child of *b* that will be created by the process that causes the number of children to exceed the barrier threshold. To leave, processes watch for a particular child to disappear and only check the exit condition once that znode has been removed.

3 ZooKeeper Applications

We now describe some applications that use ZooKeeper, and explain briefly how they use it. We show the primitives of each example in **bold**.

The Fetching Service Crawling is an important part of a search engine, and Yahoo! crawls billions of Web documents. The Fetching Service (FS) is part of the Yahoo! crawler and it is currently in production. Essentially, it has master processes that command page-fetching processes. The master provides the fetchers with configuration, and the fetchers write back informing of their status and health. The main advantages of using ZooKeeper for FS are recovering from failures of masters, guaranteeing availability despite failures, and decoupling the clients from the servers, allowing them to direct their request to healthy servers by just reading their status from ZooKeeper. Thus, FS uses ZooKeeper mainly to manage **configuration metadata**, although it also uses ZooKeeper to elect masters (**leader election**).

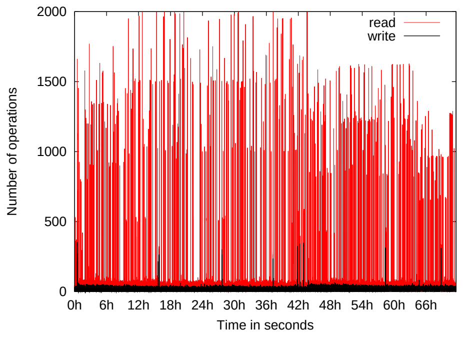

Figure 2: Workload for one ZK server with the Fetching Service. Each point represents a one-second sample.

Figure 2 shows the read and write traffic for a ZooKeeper server used by FS through a period of three days. To generate this graph, we count the number of operations for every second during the period, and each point corresponds to the number of operations in that second. We observe that the read traffic is much higher compared to the write traffic. During periods in which the rate is higher than $1,000$ operations per second, the read:write ratio varies between 10:1 and 100:1. The read operations in this workload are `getData()`, `getChildren()`, and `exists()`, in increasing order of prevalence.

Katta Katta [17] is a distributed indexer that uses ZooKeeper for coordination, and it is an example of a non-Yahoo! application. Katta divides the work of indexing using shards. A master server assigns shards to slaves and tracks progress. Slaves can fail, so the master must redistribute load as slaves come and go. The master can also fail, so other servers must be ready to take over in case of failure. Katta uses ZooKeeper to track the status of slave servers and the master (**group membership**), and to handle master failover (**leader election**). Katta also uses ZooKeeper to track and propagate the assignments of shards to slaves (**configuration management**).

Yahoo! Message Broker Yahoo! Message Broker (YMB) is a distributed publish-subscribe system. The system manages thousands of topics that clients can publish messages to and receive messages from. The topics are distributed among a set of servers to provide scalability. Each topic is replicated using a primary-backup scheme that ensures messages are replicated to two machines to ensure reliable message delivery. The servers that makeup YMB use a shared-nothing distributed architecture which makes coordination essential for correct operation. YMB uses ZooKeeper to manage the distribution of topics (**configuration metadata**), deal with failures of machines in the system (**failure detection** and **group membership**), and control system operation.

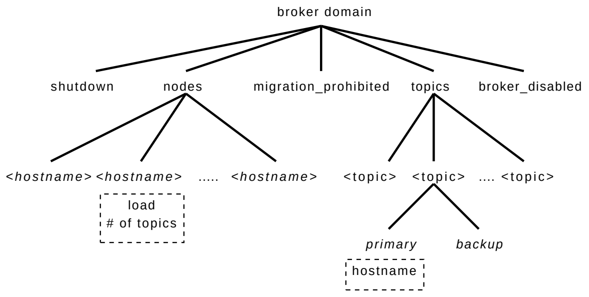

Figure 3: The layout of Yahoo! Message Broker (YMB) structures in ZooKeeper

Figure 3 shows part of the znode data layout for YMB. Each broker domain has a znode called `nodes` that has an ephemeral znode for each of the active servers that compose the YMB service. Each YMB server creates an ephemeral znode under `nodes` with load and status information providing both group membership and status information through ZooKeeper. Nodes such as `shutdown` and `migration_prohibited` are monitored by all of the servers that make up the service and allow centralized control of YMB. The `topics` directory has a child znode for each topic managed by YMB. These topic znodes have child znodes that indicate the primary and backup server for each topic along with the subscribers of that topic. The `primary` and `backup` server znodes not only allow servers to discover the servers in charge of a topic, but they also manage **leader election** and server crashes.

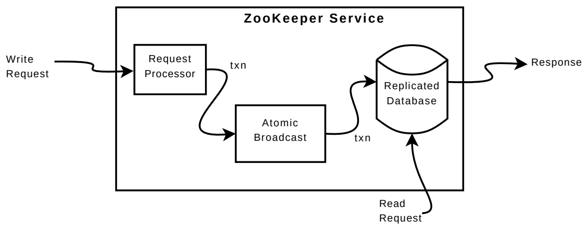

Figure 4: The components of the ZooKeeper service.

4 ZooKeeper Implementation

ZooKeeper provides high availability by replicating the ZooKeeper data on each server that composes the service. We assume that servers fail by crashing, and such faulty servers may later recover. Figure 4 shows the high-level components of the ZooKeeper service. Upon receiving a request, a server prepares it for execution (request processor). If such a request requires coordination among the servers (write requests), then they use an agreement protocol (an implementation of atomic broadcast), and finally servers commit changes to the ZooKeeper database fully replicated across all servers of the ensemble. In the case of read requests, a server simply reads the state of the local database and generates a response to the request.

The replicated database is an *in-memory* database containing the entire data tree. Each znode in the tree stores a maximum of 1MB of data by default, but this maximum value is a configuration parameter that can be changed in specific cases. For recoverability, we efficiently log updates to disk, and we force writes to be on the disk media before they are applied to the in-memory database. In fact, as Chubby [8], we keep a replay log (a write-ahead log, in our case) of committed operations and generate periodic snapshots of the in-memory database.

Every ZooKeeper server services clients. Clients connect to exactly one server to submit its requests. As we noted earlier, read requests are serviced from the local replica of each server database. Requests that change the state of the service, write requests, are processed by an agreement protocol.

As part of the agreement protocol write requests are forwarded to a single server, called the *leader*[1]. The rest of the ZooKeeper servers, called *followers*, receive message proposals consisting of state changes from the leader and agree upon state changes.

4.1 Request Processor

Since the messaging layer is atomic, we guarantee that the local replicas never diverge, although at any point in time some servers may have applied more transactions than others. Unlike the requests sent from clients, the transactions are *idempotent*. When the leader receives a write request, it calculates what the state of the system will be when the write is applied and transforms it into a transaction that captures this new state. The future state must be calculated because there may be outstanding transactions that have not yet been applied to the database. For example, if a client does a conditional `setData` and the version number in the request matches the future version number of the znode being updated, the service generates a `setDataTXN` that contains the new data, the new version number, and updated time stamps. If an error occurs, such as mismatched version numbers or the znode to be updated does not exist, an `errorTXN` is generated instead.

4.2 Atomic Broadcast

All requests that update ZooKeeper state are forwarded to the leader. The leader executes the request and broadcasts the change to the ZooKeeper state through Zab [24], an atomic broadcast protocol. The server that receives the client request responds to the client when it delivers the corresponding state change. Zab uses by default simple majority quorums to decide on a proposal, so Zab and thus ZooKeeper can only work if a majority of servers are correct (*i.e.*, with $2f + 1$ server we can tolerate f failures).

To achieve high throughput, ZooKeeper tries to keep the request processing pipeline full. It may have thousands of requests in different parts of the processing pipeline. Because state changes depend on the application of previous state changes, Zab provides stronger order guarantees than regular atomic broadcast. More specifically, Zab guarantees that changes broadcast by a leader are delivered in the order they were sent and all changes from previous leaders are delivered to an established leader before it broadcasts its own changes.

There are a few implementation details that simplify our implementation and give us excellent performance. We use TCP for our transport so message order is maintained by the network, which allows us to simplify our implementation. We use the leader chosen by Zab as the ZooKeeper leader, so that the same process that creates transactions also proposes them. We use the log to keep track of proposals as the write-ahead log for the in-

[1] Details of leaders and followers, as part of the agreement protocol, are out of the scope of this paper.

memory database, so that we do not have to write messages twice to disk.

During normal operation Zab does deliver all messages in order and exactly once, but since Zab does not persistently record the id of every message delivered, Zab may redeliver a message during recovery. Because we use idempotent transactions, multiple delivery is acceptable as long as they are delivered in order. In fact, ZooKeeper requires Zab to redeliver at least all messages that were delivered after the start of the last snapshot.

4.3 Replicated Database

Each replica has a copy in memory of the ZooKeeper state. When a ZooKeeper server recovers from a crash, it needs to recover this internal state. Replaying all delivered messages to recover state would take prohibitively long after running the server for a while, so ZooKeeper uses periodic snapshots and only requires redelivery of messages since the start of the snapshot. We call ZooKeeper snapshots *fuzzy snapshots* since we do not lock the ZooKeeper state to take the snapshot; instead, we do a depth first scan of the tree atomically reading each znode's data and meta-data and writing them to disk. Since the resulting fuzzy snapshot may have applied some subset of the state changes delivered during the generation of the snapshot, the result may not correspond to the state of ZooKeeper at any point in time. However, since state changes are idempotent, we can apply them twice as long as we apply the state changes in order.

For example, assume that in a ZooKeeper data tree two nodes `/foo` and `/goo` have values `f1` and `g1` respectively and both are at version 1 when the fuzzy snapshot begins, and the following stream of state changes arrive having the form ⟨`transactionType, path, value, new-version`⟩:

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

After processing these state changes, `/foo` and `/goo` have values `f3` and `g2` with versions 3 and 2 respectively. However, the fuzzy snapshot may have recorded that `/foo` and `/goo` have values `f3` and `g1` with versions 3 and 1 respectively, which was not a valid state of the ZooKeeper data tree. If the server crashes and recovers with this snapshot and Zab redelivers the state changes, the resulting state corresponds to the state of the service before the crash.

4.4 Client-Server Interactions

When a server processes a write request, it also sends out and clears notifications relative to any watch that corresponds to that update. Servers process writes in order and do not process other writes or reads concurrently. This ensures strict succession of notifications. Note that servers handle notifications locally. Only the server that a client is connected to tracks and triggers notifications for that client.

Read requests are handled locally at each server. Each read request is processed and tagged with a *zxid* that corresponds to the last transaction seen by the server. This *zxid* defines the partial order of the read requests with respect to the write requests. By processing reads locally, we obtain excellent read performance because it is just an in-memory operation on the local server, and there is no disk activity or agreement protocol to run. This design choice is key to achieving our goal of excellent performance with read-dominant workloads.

One drawback of using fast reads is not guaranteeing precedence order for read operations. That is, a read operation may return a stale value, even though a more recent update to the same znode has been committed. Not all of our applications require precedence order, but for applications that do require it, we have implemented `sync`. This primitive executes asynchronously and is ordered by the leader after all pending writes to its local replica. To guarantee that a given read operation returns the latest updated value, a client calls `sync` followed by the read operation. The FIFO order guarantee of client operations together with the global guarantee of `sync` enables the result of the read operation to reflect any changes that happened before the `sync` was issued. In our implementation, we do not need to atomically broadcast `sync` as we use a leader-based algorithm, and we simply place the `sync` operation at the end of the queue of requests between the leader and the server executing the call to `sync`. In order for this to work, the follower must be sure that the leader is still the leader. If there are pending transactions that commit, then the server does not suspect the leader. If the pending queue is empty, the leader needs to issue a null transaction to commit and orders the `sync` after that transaction. This has the nice property that when the leader is under load, no extra broadcast traffic is generated. In our implementation, timeouts are set such that leaders realize they are not leaders before followers abandon them, so we do not issue the null transaction.

ZooKeeper servers process requests from clients in FIFO order. Responses include the *zxid* that the response is relative to. Even heartbeat messages during intervals of no activity include the last *zxid* seen by the server that the client is connected to. If the client connects to a new server, that new server ensures that its view of the ZooKeeper data is at least as recent as the view of the client by checking the last *zxid* of the client against its last *zxid*. If the client has a more recent view than the server, the server does not reestablish the session with the client until the server has caught up. The client is guaranteed to be able to find another server that has a recent view of the system since the client only sees changes that have been replicated to a majority of the ZooKeeper servers. This behavior is important to guarantee durability.

To detect client session failures, ZooKeeper uses timeouts. The leader determines that there has been a failure if no other server receives anything from a client session within the session timeout. If the client sends requests frequently enough, then there is no need to send any other message. Otherwise, the client sends heartbeat messages during periods of low activity. If the client cannot communicate with a server to send a request or heartbeat, it connects to a different ZooKeeper server to re-establish its session. To prevent the session from timing out, the ZooKeeper client library sends a heartbeat after the session has been idle for $s/3$ ms and switch to a new server if it has not heard from a server for $2s/3$ ms, where s is the session timeout in milliseconds.

5 Evaluation

We performed all of our evaluation on a cluster of 50 servers. Each server has one Xeon dual-core 2.1GHz processor, 4GB of RAM, gigabit ethernet, and two SATA hard drives. We split the following discussion into two parts: throughput and latency of requests.

5.1 Throughput

To evaluate our system, we benchmark throughput when the system is saturated and the changes in throughput for various injected failures. We varied the number of servers that make up the ZooKeeper service, but always kept the number of clients the same. To simulate a large number of clients, we used 35 machines to simulate 250 simultaneous clients.

We have a Java implementation of the ZooKeeper server, and both Java and C clients[2]. For these experiments, we used the Java server configured to log to one dedicated disk and take snapshots on another. Our benchmark client uses the asynchronous Java client API, and each client has at least 100 requests outstanding. Each request consists of a read or write of 1K of data. We do not show benchmarks for other operations since the performance of all the operations that modify state are approximately the same, and the performance of nonstate modifying operations, excluding `sync`, are approximately the same. (The performance of `sync` approximates that of a light-weight write, since the request must go to the leader, but does not get broadcast.) Clients send counts of the number of completed operations every $300ms$ and we sample every $6s$. To prevent memory overflows, servers throttle the number of concurrent requests in the system. ZooKeeper uses request throttling to keep servers from being overwhelmed. For these experiments, we configured the ZooKeeper servers to have a maximum of $2,000$ total requests in process.

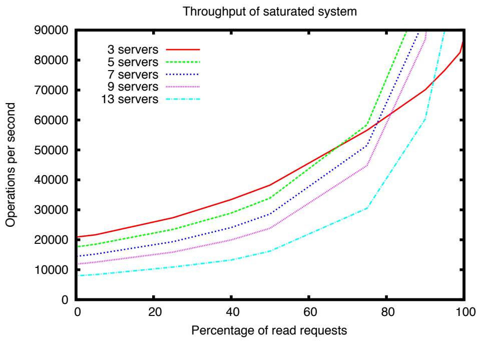

Figure 5: The throughput performance of a saturated system as the ratio of reads to writes vary.

Servers	100% **Reads**	0% **Reads**
13	460k	8k
9	296k	12k
7	257k	14k
5	165k	18k
3	87k	21k

Table 1: The throughput performance of the extremes of a saturated system.

In Figure 5, we show throughput as we vary the ratio of read to write requests, and each curve corresponds to a different number of servers providing the ZooKeeper service. Table 1 shows the numbers at the extremes of the read loads. Read throughput is higher than write throughput because reads do not use atomic broadcast. The graph also shows that the number of servers also has a negative impact on the performance of the broadcast protocol. From these graphs, we observe that the number of servers in the system does not only impact the number of failures that the service can handle, but also the workload the service can handle. Note that the curve for three servers crosses the others around 60%. This situation is not exclusive of the three-server configuration, and happens for all configurations due to the parallelism local reads enable. It is not observable for other configurations in the figure, however, because we have capped the maximum y-axis throughput for readability.

There are two reasons for write requests taking longer than read requests. First, write requests must go through atomic broadcast, which requires some extra processing

[2] The implementation is publicly available at `http://hadoop.apache.org/zookeeper`.

and adds latency to requests. The other reason for longer processing of write requests is that servers must ensure that transactions are logged to non-volatile store before sending acknowledgments back to the leader. In principle, this requirement is excessive, but for our production systems we trade performance for reliability since ZooKeeper constitutes application ground truth. We use more servers to tolerate more faults. We increase write throughput by partitioning the ZooKeeper data into multiple ZooKeeper ensembles. This performance trade off between replication and partitioning has been previously observed by Gray *et al.* [12].

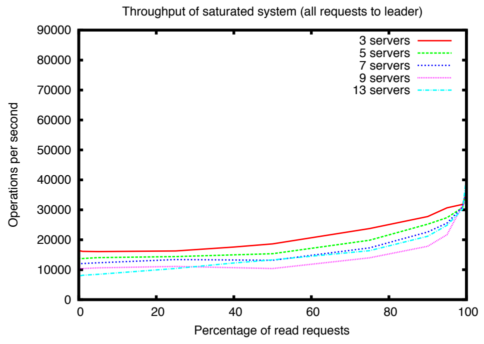

Figure 6: Throughput of a saturated system, varying the ratio of reads to writes when all clients connect to the leader.

ZooKeeper is able to achieve such high throughput by distributing load across the servers that makeup the service. We can distribute the load because of our relaxed consistency guarantees. Chubby clients instead direct all requests to the leader. Figure 6 shows what happens if we do not take advantage of this relaxation and forced the clients to only connect to the leader. As expected the throughput is much lower for read-dominant workloads, but even for write-dominant workloads the throughput is lower. The extra CPU and network load caused by servicing clients impacts the ability of the leader to coordinate the broadcast of the proposals, which in turn adversely impacts the overall write performance.

The atomic broadcast protocol does most of the work of the system and thus limits the performance of ZooKeeper more than any other component. Figure 7 shows the throughput of the atomic broadcast component. To benchmark its performance we simulate clients by generating the transactions directly at the leader, so there is no client connections or client requests and replies. At maximum throughput the atomic broadcast component becomes CPU bound. In theory the performance of Figure 7 would match the performance of ZooKeeper with 100% writes. However, the ZooKeeper client communication, ACL checks, and request to transaction conversions all require CPU. The contention for CPU lowers ZooKeeper throughput to substantially less than the atomic broadcast component in isolation. Because ZooKeeper is a critical production component, up to now our development focus for ZooKeeper has been correctness and robustness. There are plenty of opportunities for improving performance significantly by eliminating things like extra copies, multiple serializations of the same object, more efficient internal data structures, etc.

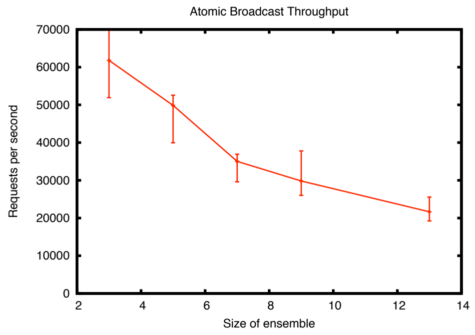

Figure 7: Average throughput of the atomic broadcast component in isolation. Error bars denote the minimum and maximum values.

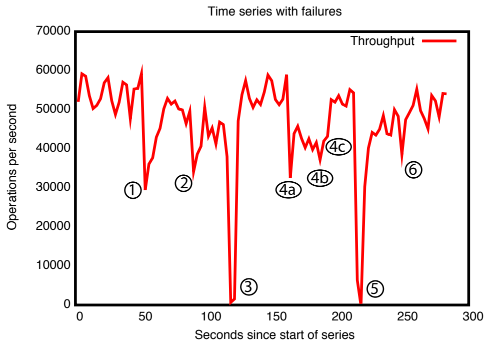

Figure 8: Throughput upon failures.

To show the behavior of the system over time as failures are injected we ran a ZooKeeper service made up of 5 machines. We ran the same saturation benchmark as before, but this time we kept the write percentage at a constant 30%, which is a conservative ratio of our expected workloads. Periodically we killed some of the server processes. Figure 8 shows the system throughput as it changes over time. The events marked in the figure are the following:

1. Failure and recovery of a follower;
2. Failure and recovery of a different follower;
3. Failure of the leader;
4. Failure of two followers (a, b) in the first two marks, and recovery at the third mark (c);
5. Failure of the leader.

6. Recovery of the leader.

There are a few important observations from this graph. First, if followers fail and recover quickly, then ZooKeeper is able to sustain a high throughput despite the failure. The failure of a single follower does not prevent servers from forming a quorum, and only reduces throughput roughly by the share of read requests that the server was processing before failing. Second, our leader election algorithm is able to recover fast enough to prevent throughput from dropping substantially. In our observations, ZooKeeper takes less than $200ms$ to elect a new leader. Thus, although servers stop serving requests for a fraction of second, we do not observe a throughput of zero due to our sampling period, which is on the order of seconds. Third, even if followers take more time to recover, ZooKeeper is able to raise throughput again once they start processing requests. One reason that we do not recover to the full throughput level after events 1, 2, and 4 is that the clients only switch followers when their connection to the follower is broken. Thus, after event 4 the clients do not redistribute themselves until the leader fails at events 3 and 5. In practice such imbalances work themselves out over time as clients come and go.

5.2 Latency of requests

To assess the latency of requests, we created a benchmark modeled after the Chubby benchmark [6]. We create a worker process that simply sends a create, waits for it to finish, sends an asynchronous delete of the new node, and then starts the next create. We vary the number of workers accordingly, and for each run, we have each worker create 50,000 nodes. We calculate the throughput by dividing the number of create requests completed by the total time it took for all the workers to complete.

	Number of servers			
Workers	3	5	7	9
1	776	748	758	711
10	2074	1832	1572	1540
20	2740	2336	1934	1890

Table 2: Create requests processed per second.

Table 2 show the results of our benchmark. The create requests include 1K of data, rather than 5 bytes in the Chubby benchmark, to better coincide with our expected use. Even with these larger requests, the throughput of ZooKeeper is more than 3 times higher than the published throughput of Chubby. The throughput of the single ZooKeeper worker benchmark indicates that the average request latency is 1.2ms for three servers and 1.4ms for 9 servers.

	# of clients		
# of barriers	**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

Table 3: Barrier experiment with time in seconds. Each point is the average of the time for each client to finish over five runs.

5.3 Performance of barriers

In this experiment, we execute a number of barriers sequentially to assess the performance of primitives implemented with ZooKeeper. For a given number of barriers b, each client first enters all b barriers, and then it leaves all b barriers in succession. As we use the double-barrier algorithm of Section 2.4, a client first waits for all other clients to execute the `enter()` procedure before moving to next call (similarly for `leave()`).

We report the results of our experiments in Table 3. In this experiment, we have 50, 100, and 200 clients entering a number b of barriers in succession, $b \in \{200, 400, 800, 1600\}$. Although an application can have thousands of ZooKeeper clients, quite often a much smaller subset participates in each coordination operation as clients are often grouped according to the specifics of the application.

Two interesting observations from this experiment are that the time to process all barriers increase roughly linearly with the number of barriers, showing that concurrent access to the same part of the data tree did not produce any unexpected delay, and that latency increases proportionally to the number of clients. This is a consequence of not saturating the ZooKeeper service. In fact, we observe that even with clients proceeding in lock-step, the throughput of barrier operations (enter and leave) is between 1,950 and 3,100 operations per second in all cases. In ZooKeeper operations, this corresponds to throughput values between 10,700 and 17,000 operations per second. As in our implementation we have a ratio of reads to writes of 4:1 (80% of read operations), the throughput our benchmark code uses is much lower compared to the raw throughput ZooKeeper can achieve (over 40,000 according to Figure 5). This is due to clients waiting on other clients.

6 Related work

ZooKeeper has the goal of providing a service that mitigates the problem of coordinating processes in distributed applications. To achieve this goal, its design uses ideas from previous coordination services, fault tolerant systems, distributed algorithms, and file systems.

We are not the first to propose a system for the coordination of distributed applications. Some early systems propose a distributed lock service for transactional applications [13], and for sharing information in clusters of computers [19]. More recently, Chubby proposes a system to manage advisory locks for distributed applications [6]. Chubby shares several of the goals of ZooKeeper. It also has a file-system-like interface, and it uses an agreement protocol to guarantee the consistency of the replicas. However, ZooKeeper is not a lock service. It can be used by clients to implement locks, but there are no lock operations in its API. Unlike Chubby, ZooKeeper allows clients to connect to any ZooKeeper server, not just the leader. ZooKeeper clients can use their local replicas to serve data and manage watches since its consistency model is much more relaxed than Chubby. This enables ZooKeeper to provide higher performance than Chubby, allowing applications to make more extensive use of ZooKeeper.

There have been fault-tolerant systems proposed in the literature with the goal of mitigating the problem of building fault-tolerant distributed applications. One early system is ISIS [5]. The ISIS system transforms abstract type specifications into fault-tolerant distributed objects, thus making fault-tolerance mechanisms transparent to users. Horus [30] and Ensemble [31] are systems that evolved from ISIS. ZooKeeper embraces the notion of virtual synchrony of ISIS. Finally, Totem guarantees total order of message delivery in an architecture that exploits hardware broadcasts of local area networks [22]. ZooKeeper works with a wide variety of network topologies which motivated us to rely on TCP connections between server processes and not assume any special topology or hardware features. We also do not expose any of the ensemble communication used internally in ZooKeeper.

One important technique for building fault-tolerant services is state-machine replication [26], and Paxos [20] is an algorithm that enables efficient implementations of replicated state-machines for asynchronous systems. We use an algorithm that shares some of the characteristics of Paxos, but that combines transaction logging needed for consensus with write-ahead logging needed for data tree recovery to enable an efficient implementation. There have been proposals of protocols for practical implementations of Byzantine-tolerant replicated state-machines [7, 10, 18, 1, 28]. ZooKeeper does not assume that servers can be Byzantine, but we do employ mechanisms such as checksums and sanity checks to catch non-malicious Byzantine faults. Clement *et al.* discuss an approach to make ZooKeeper fully Byzantine fault-tolerant without modifying the current server code base [9]. To date, we have not observed faults in production that would have been prevented using a fully Byzantine fault-tolerant protocol. [29].

Boxwood [21] is a system that uses distributed lock servers. Boxwood provides higher-level abstractions to applications, and it relies upon a distributed lock service based on Paxos. Like Boxwood, ZooKeeper is a component used to build distributed systems. ZooKeeper, however, has high-performance requirements and is used more extensively in client applications. ZooKeeper exposes lower-level primitives that applications use to implement higher-level primitives.

ZooKeeper resembles a small file system, but it only provides a small subset of the file system operations and adds functionality not present in most file systems such as ordering guarantees and conditional writes. ZooKeeper watches, however, are similar in spirit to the cache callbacks of AFS [16].

Sinfonia [2] introduces *mini-transactions*, a new paradigm for building scalable distributed systems. Sinfonia has been designed to store application data, whereas ZooKeeper stores application metadata. ZooKeeper keeps its state fully replicated and in memory for high performance and consistent latency. Our use of file system like operations and ordering enables functionality similar to mini-transactions. The znode is a convenient abstraction upon which we add watches, a functionality missing in Sinfonia. Dynamo [11] allows clients to get and put relatively small (less than 1M) amounts of data in a distributed key-value store. Unlike ZooKeeper, the key space in Dynamo is not hierarchal. Dynamo also does not provide strong durability and consistency guarantees for writes, but instead resolves conflicts on reads.

DepSpace [4] uses a tuple space to provide a Byzantine fault-tolerant service. Like ZooKeeper DepSpace uses a simple server interface to implement strong synchronization primitives at the client. While DepSpace's performance is much lower than ZooKeeper, it provides stronger fault tolerance and confidentiality guarantees.

7 Conclusions

ZooKeeper takes a wait-free approach to the problem of coordinating processes in distributed systems, by exposing wait-free objects to clients. We have found ZooKeeper to be useful for several applications inside and outside Yahoo!. ZooKeeper achieves throughput values of hundreds of thousands of operations per second for read-dominant workloads by using fast reads with watches, both of which served by local replicas. Although our consistency guarantees for reads and watches appear to be weak, we have shown with our use cases that this combination allows us to implement efficient and sophisticated coordination protocols at the client even though reads are not precedence-ordered and the implementation of data objects is wait-free. The wait-free property has proved to be essential for high performance.

Although we have described only a few applications, there are many others using ZooKeeper. We believe such a success is due to its simple interface and the powerful abstractions that one can implement through this interface. Further, because of the high-throughput of ZooKeeper, applications can make extensive use of it, not only course-grained locking.

Acknowledgements

We would like to thank Andrew Kornev and Runping Qi for their contributions to ZooKeeper; Zeke Huang and Mark Marchukov for valuable feedback; Brian Cooper and Laurence Ramontianu for their early contributions to ZooKeeper; Brian Bershad and Geoff Voelker made important comments on the presentation.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. 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: A new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of the 21st ACM symposium on Operating systems principles*, New York, NY, 2007.

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

[4] A. N. Bessani, E. P. Alchieri, M. Correia, and J. da Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference - EuroSys 2008*, Apr. 2008.

[5] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *SOSP '85: Proceedings of the 10th ACM symposium on Operating systems principles*, New York, USA, 1985. ACM Press.

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing (PODC)*, Aug. 2007.

[9] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the 22 nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. 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 highly available key-value store. In *SOSP '07: Proceedings of the 21st ACM symposium on Operating systems principles*, New York, NY, USA, 2007. ACM Press.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of SIGMOD '96*, pages 173–182, New York, NY, USA, 1996. ACM.

[13] A. Hastings. Distributed lock management in a transaction processing environment. In *Proceedings of IEEE 9th Symposium on Reliable Distributed Systems*, Oct. 1990.

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

[15] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *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. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1), 1988.

[17] Katta. Katta - distribute lucene indexes in a grid. `http://katta.wiki.sourceforge.net/`, 2008.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[19] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. Vaxclusters (extended abstract): a closely-coupled distributed system. *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. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[22] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, C. Lingley-Papadopoulos, and T. Archambault. The totem system. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995.

[23] S. Mullender, editor. *Distributed Systems, 2nd edition*. ACM Press, New York, NY, USA, 1993.

[24] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *LADIS '08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–6, New York, NY, USA, 2008. ACM.

[25] N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *DSN*, 2008.

[26] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), 1990.

[27] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. ACMS: The Akamai configuration management system. In *NSDI*, 2005.

[28] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.

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

[30] R. van Renesse and K. Birman. Horus, a flexible group communication systems. *Communications of the ACM*, 39(16), Apr. 1996.

[31] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Software - Practice and Experience*, 28(5), July 1998.