分布式锁服务ZooKeeper

zookeeper概述

针对分布式应用的分布式协作服务，zookeeper的开发动机就是为了减轻分布式应用从头开发协作服务的负担。

设计目标

简单。 允许多个分布的进程基于一个共享的，类似标准文件系统的树状名称空间进行协作。每个节点称作一个znode。

ZooKeeper is replicated

几个zookeeper集群包含多个zookeeper server, 称作一个ensemble。 这些server彼此都知道对方的存在。

需要维护的数据: 内存中的状态的镜像, 持久化存储中的事务日志和快照。

client和单个zookeeper server通信。client维护一个持久TCP连接，通过其发送请求, 获取响应和watch events,并发送心跳信息，如果到server的TCP连接中断, client将会连接到另外一个server。

ZooKeeper is ordered

zookeeper对每次更新进行一个计数器stamp以反映所有zookeeper事务的次序。后续的操作能够使用此次序来实现高级抽象，如同步原语。

ZooKeeper is fast

zookeeper在read操作上是非常快的。通常的应用中，读写操作比也在10:1左右。

数据模型和层级名称空间

节点和临时节点

有别于标准文件系统的是,zookeeper名称空间中的每个节点都可以关联数据到它本身或者它的子节点。就好比标准文件系统中的文件同时又可以充当目录。(zookeeper用于存储协作数据: 状态信息, 配置, 位置信息等, 所以存放在每个node的数据通常都比较小,一般在K级别)。我们以znode来描述zookeeper中的数据节点。

znodes维护一个stat结构，其中包括数据变更, 权限变更的版本信息，还有时间戳,以便于缓存有效性验证和协作更新。每当znode的数据变更时,版本号都会递增。例如, 每次client在获取数据时，它同时也获得了数据的版本信息。

存储在znode中的数据对read, write都是原子性操作的。read会获取znode中的所有字节, write会整个替换znode中的信息。每个znode都包含一个访问控制列表(ACL)以约束谁可以访问此节点。

zookeeper还有一个临时节点的概念。临时节点在创建它的session的生命周期内存活, 当其session终止时，此类节点将会被删除。临时节点在我们需要实现[tbd]时非常有用。

条件更新和监听器(watches)

zookeeper提供监听器的概念。 client可以在某个znode上设置watch。 当znode有变更时, 相关的watch会被触发或删除。一旦watch触发, client将会收到一个数据包以通知znode的变更。如果在client和zookeeper server之间的连接被中断了, client将会收到一个本地的通知。这些都能被用于[tbd]。

Guarantees

zookeeper在使用上非常简单高效。因为它的设计目标,是作为构建复杂服务类型，如同步, zookeeper提供的保证包括:

序列一致性: 数据更新会依照client发送的次序来进行。

原子性: 更新要么成功,要么失败。不存在部分结果。

唯一系统镜像: client总是会看到一致的视图，而不管它是连接到具体哪个zookeeper server。

可靠性: 一旦更新完成, 它会持续保存直到有另外的client重写。

及时: 客户端视图会在一定的时间间隔内进行更新。

Simple API

zookeeper的一个重要设计目标就是要提供简单的编程接口。所以,它仅仅提供如下操作:

create:在树种某个位置创建一个节点。

delete:删除一个节点。

exists:检查给定节点是否存在。

get data: 从一个节点读取数据。

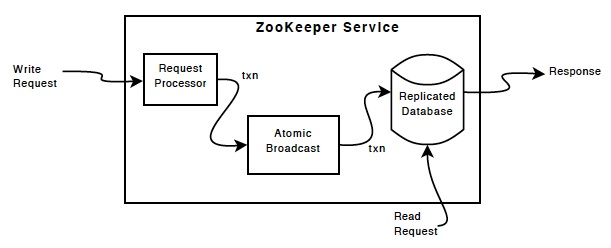
set data: 写数据到给定节点。

get children: 获取节点的子节点列表

sync: 等待知道数据被传输。

实现

如下展示了zookeeper服务的高层组件。



除了request processor, 组成zookeeper服务的每个server都会在本地备份其它组件的拷贝。

replicated database是一个包含整个数据树的内存数据库。更新被logged到磁盘以提供可恢复性,写操作先持久化到磁盘，然后再对内存数据库作变更。

每个zookeeper server都对client提供服务。client连接到具体的某一个server以提交请求。读操作依赖与每个server的本地数据库。改变服务状态的请求,写操作,由一致性协议来处理。

作为一致性协议的一部分，所有的client写请求被提交到专门的一个leader server。 其余的server,被称为followers,从leader接收消息,并对消息的传递达成一致。

消息层负责替换失效leader并同步followers。

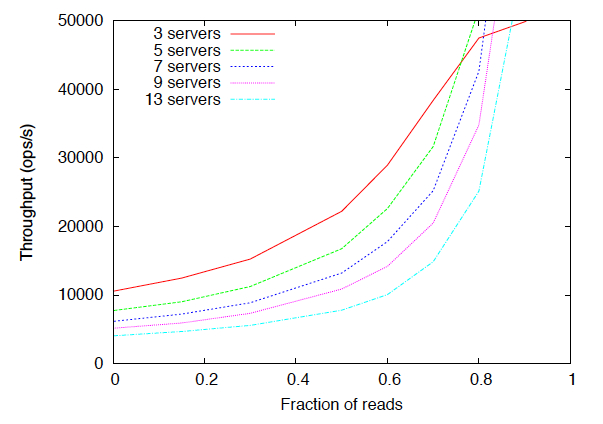
zookeeper使用自定义的原子消息传递协议。因为消息传递层是原子性的,zookeeper能够确保本地备份不会出现分歧。

使用

zookeeper的编程接口特意做的非常简单。在此之上，你可以实现更高层次的操作, 如同步原语, 组管理, 所有权等等。

性能

zookeeper被设计用于高性能场景。



可靠性

下图的benchmarks同时也表明zookeeper是可靠的。 [Reliability in the Presence of Errors](http://hadoop.apache.org/zookeeper/docs/current/zookeeperOver.html#fg_zkPerfReliability)展示了一个zookeeper部署如何应对各种失效。 在图中标示的事件定义如下:

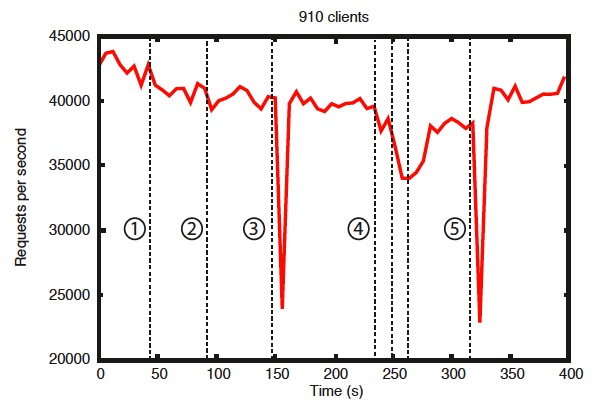
1.follower的失效和恢复。

2. 不同的follower失效和恢复。

3. leader的失效。

4.两个follower同时失效和恢复。

5. 另一个leader失效。



这张图中有一些重要的观测值。首先，如果followers失效并快速恢复,zookeeper可以持续保持高吞吐而不受影响;最重要的是，leader推举算法允许系统快速恢复以防止吞吐大幅度下降。在我们的观测中,zookeeper只花费不到200ms来推举一个新的leader;第三点,当follower恢复并开始处理请求时,zookeeper的吞吐也会回升。

zookeeper已经在很多工业级项目中被[成功运用](http://wiki.apache.org/hadoop/ZooKeeper/PoweredBy)。在Yahoo!, 它被用于Yahoo! Message Broker以提供协作和失效恢复服务, Yahoo! Message Broker是一个高效的发布/订阅系统,其管理着用于备份和数据迁移的主题。 zookeeper还被用于Yahoo!爬虫的抓取服务,在此它同样提供了失效恢复机制。许多Yahoo!的广告系统也使用zookeeper来提供可靠服务。