zookeeper一致性协议ZAB即paxos算法

问题导读

1.什么是ZAB？

2.Zookeeper是否可以理解为一个小存储系统？

3.吞吐量很高时，磁盘的IO便成为系统瓶颈如何解决？

ZAB：ZooKeeper的Atomic Broadcast协议，能够保证发给各副本的消息顺序相同。

Zookeeper使用了一种称为Zab（ZookeeperAtomic Broadcast）的协议作为其一致性复制的核心，其特点为高吞吐量、低延迟、健壮、简单，但不过分要求其扩展性。

Zookeeper的实现是有Client、Server构成，Server端提供了一个一致性复制、存储服务，Client端会提供一些具体的语义，比如分布式锁、选举算法、分布式互斥等。从存储内容来说，Server端更多的是存储一些数据的状态，而非数据内容本身，因此Zookeeper可以作为一个小文件系统使用。数据状态的存储量相对不大，完全可以全部加载到内存中，从而极大地消除了通信延迟。

Server可以Crash后重启，考虑到容错性，Server必须“记住”之前的数据状态，因此数据需要持久化，但吞吐量很高时，磁盘的IO便成为系统瓶颈，其解决办法是使用缓存，把随机写变为连续写。

考虑到Zookeeper主要操作数据的状态，为了保证状态的一致性，Zookeeper提出了两个安全属性（Safety Property）：

- 全序（Total order）：如果消息a在消息b之前发送，则所有Server应该看到相同的结果
- 因果顺序（Causal order）：如果消息a在消息b之前发生（a导致了b），并被一起发送，则a始终在b之前被执行。

为了保证上述两个安全属性，Zookeeper使用了TCP协议和Leader。通过使用TCP协议保证了消息的全序特性（先发先到），通过Leader解决了因果顺序问题：先到Leader的先执行。因为有了Leader，Zookeeper的架构就变为：Master-Slave模式，但在该模式中Master（Leader）会Crash，因此，Zookeeper引入了Leader选举算法，以保证系统的健壮性。归纳起来Zookeeper整个工作分两个阶段：

- Atomic Broadcast
- Leader选举

1.Atomic Broadcast

同一时刻存在一个Leader节点，其他节点称为“Follower”，如果是更新请求，如果客户端连接到Leader节点，则由Leader节点执行其请求；如果连接到Follower节点，则需转发请求到Leader节点执行。但对读请求，Client可以直接从Follower上读取数据，如果需要读到最新数据，则需要从Leader节点进行，Zookeeper设计的读写比例是2：1。

Leader通过一个简化版的二段提交模式向其他Follower发送请求，但与二段提交有两个明显的不同之处：
• 因为只有一个Leader，Leader提交到Follower的请求一定会被接受（没有其他Leader干扰）
• 不需要所有的Follower都响应成功，只要一个多数派即可
通俗地说，如果有2f+1个节点，允许f个节点失败。因为任何两个多数派必有一个交集，当Leader切换时，通过这些交集节点可以获得当前系统的最新状态。如果没有一个多数派存在（存活节点数小于f+1）则，算法过程结束。但有一个特例：
如果有A、B、C三个节点，A是Leader，如果B Crash，则A、C能正常工作，因为A是Leader，A、C还构成多数派；如果A Crash则无法继续工作，因为Leader选举的多数派无法构成。

2. Leader Election

Leader选举主要是依赖Paxos算法，Leader选举遇到的最大问题是，”新老交互“的问题，新Leader是否要继续老Leader的状态。这里要按老Leader Crash的时机点分几种情况：
1. 老Leader在COMMIT前Crash（已经提交到本地）
2. 老Leader在COMMIT后Crash，但有部分Follower接收到了Commit请求

第一种情况，这些数据只有老Leader自己知道，当老Leader重启后，需要与新Leader同步并把这些数据从本地删除，以维持状态一致。
第二种情况，新Leader应该能通过一个多数派获得老Leader提交的最新数据，老Leader重启后，可能还会认为自己是Leader，可能会继续发送未完成的请求，从而因为两个Leader同时存在导致算法过程失败，解决办法是把Leader信息加入每条消息的id中，Zookeeper中称为zxid，zxid为一64位数字，高32位为leader信息又称为epoch，每次leader转换时递增；低32位为消息编号，Leader转换时应该从0重新开始编号。通过zxid，Follower能很容易发现请求是否来自老Leader，从而拒绝老Leader的请求。

因为在老Leader中存在着数据删除（情况1），因此Zookeeper的数据存储要支持补偿操作，这也就需要像数据库一样记录log。

3. Zab与Paxos

Zab的作者认为Zab与paxos并不相同，Zab就是Paxos的一种简化形式，Paxos保证不了全序顺序。
这里首要一点是Paxos的一致性不能达到ZooKeeper的要求。举个例子：假设一开始Paxos系统中的leader是P1，他发起了两个事务<t1, v1>（表示序号为t1的事务要写的值是v1）和<t2, v2>，过程中挂了。新来个leader是P2，他发起了事务<t1, v1'>。而后又来个新leader是P3，他汇总了一下，得出最终的执行序列<t1, v1'>和<t2, v2>，即P2的t1在前，P1的t2在后。

分析为什么不满足ZooKeeper需求：
ZooKeeper是一个树形结构，很多操作都要先检查才能确定能不能执行，比如P1的事务t1可能是创建节点“/a”，t2可能是创建节点“/a/aa”，只有先创建了父节点“/a”，才能创建子节点“/a/aa”。而P2所发起的事务t1可能变成了创建“/b”。这样P3汇总后的序列是先创建“/b”再创建“/a/aa”，由于“/a”还没建，创建“a/aa”就搞不定了。

解决方案：
为了保证这一点，ZAB要保证同一个leader的发起的事务要按顺序被apply，同时还要保证只有先前的leader的所有事务都被apply之后，新选的leader才能在发起事务。

ZAB的核心思想：形象的说就是保证任意时刻只有一个节点是leader，所有更新事务由leader发起去更新所有复本（称为follower），更新时用的就是两阶段提交协议，只要多数节点prepare成功，就通知他们commit。各follower要按当初leader让他们prepare的顺序来apply事务。因为ZAB处理的事务永远不会回滚，ZAB的2PC做了点优化，多个事务只要通知zxid最大的那个commit，之前的各follower会统统commit。
这里有几个关键点：
1、leader所follower之间通过心跳来检测异常；
2、检测到异常之后的节点若试图成为新的leader，首先要获得大多数节点的支持，然后从状态最新的节点同步事务，完成后才可正式成为leader发起事务；
3、区分新老leader的关键是一个会一直增长的epoch；

4.结束

本文只是想从协议、算法的角度分析Zookeeper，而非分析其源码实现，因为Zookeeper版本的变化，文中描述的场景或许已找不到对应的实现。