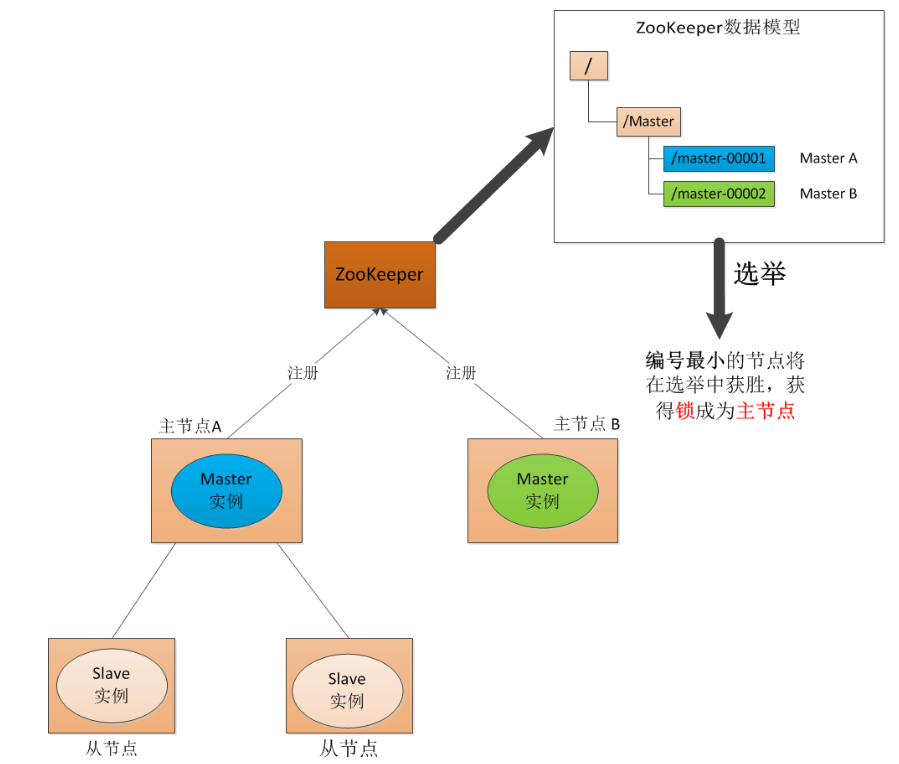
# ZooKeeper一致性原理

## ZooKeeper 的实现

### ZooKeeper处理单点故障

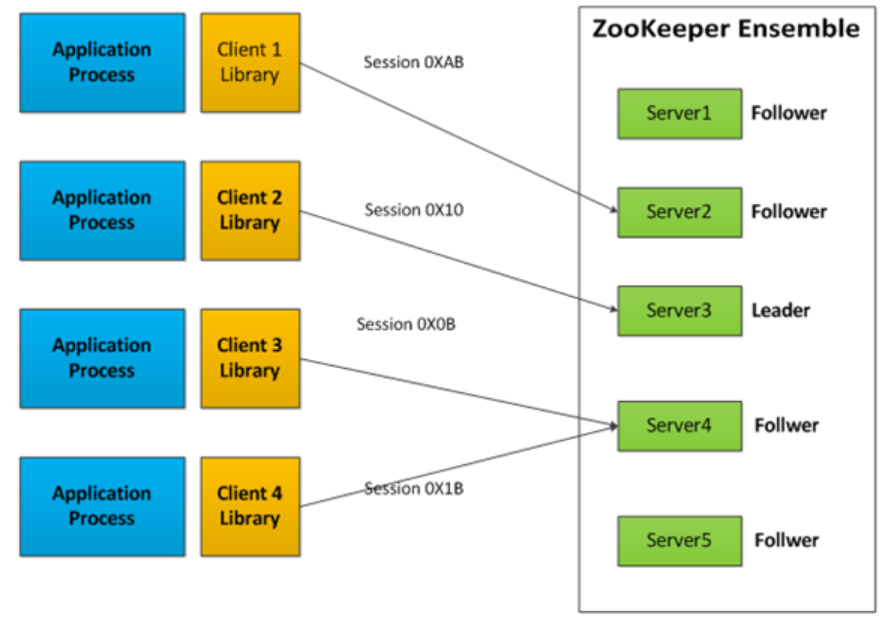
我们知道可以通过ZooKeeper对分布式系统进行Master选举，来解决分布式系统的**单点故障**



那么我们继续分析一下，ZooKeeper通过Master选举来帮助分布式系统解决**单点故障**，保证该系统中每时每刻只有一个Master为分布式系统提供服务。也就是说分布式的单点问题交给了ZooKeeper来处理，不知道大家此时有没有发现一个问题——"故障转移到了ZooKeeper身上"。大家看一下图就会发现，如果我们的ZooKeeper只用一台机器来提供服务，若这台机器挂了，那么该分布式系统就直接变成双Master模式了，那么我们在分布式系统中引入ZooKeeper也就失去了意义。那么这也就意味着，ZooKeeper在其**实现的过程**中要做一些可用性和恢复性的保证。这样才能让我们放心的以ZooKeeper为起点来构建我们的分布式系统，来达到节省成本和减少bug的目的。

### ZooKeeper运行模式

ZooKeeper服务有两种不同的运行模式。一种是"**独立模式**"(standalone mode)，即只有一个ZooKeeper服务器。这种模式较为简单，比较适合于测试环境，甚至可以在单元测试中采用，但是不能保证高可用性和恢复性。在生产环境中的ZooKeeper通常以"**复制模式**"(replicated mode)运行于一个计算机集群上，这个计算机集群被称为一个"**集合体**"(ensemble)



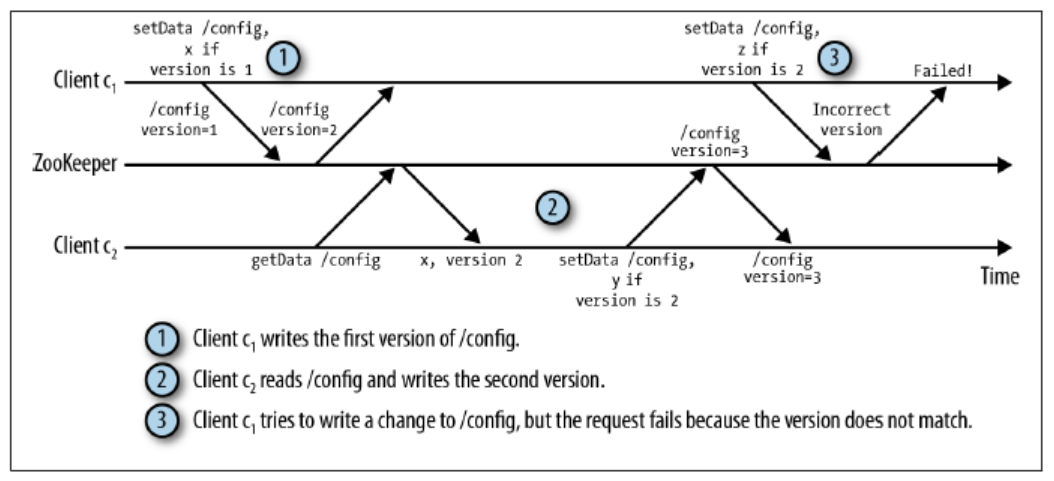
ZooKeeper通过**复制**来实现高可用性，只要集合体中**半数以上**的机器处于可用状态，它就能够提供服务。例如，在一个有5个节点的集合体中，每个Follower节点的数据都是Leader节点数据的副本，也就是说我们的每个节点的数据视图都是一样的，这样就可以有五个节点提供ZooKeeper服务。并且集合体中任意2台机器出现故障，都可以保证服务继续，因为剩下的3台机器超过了半数。

从概念上来说，ZooKeeper它所做的就是确保对Znode树的每一个修改都会被复制到集合体中超过半数的机器上。如果少于半数的机器出现故障，则最少有一台机器会保存最新的状态，那么这台机器就是我们的Leader。其余的副本最终也会更新到这个状态。如果Leader挂了，由于其他机器保存了Leader的副本，那就可以从中选出一台机器作为新的Leader继续提供服务。

### ZooKeeper的读写机制

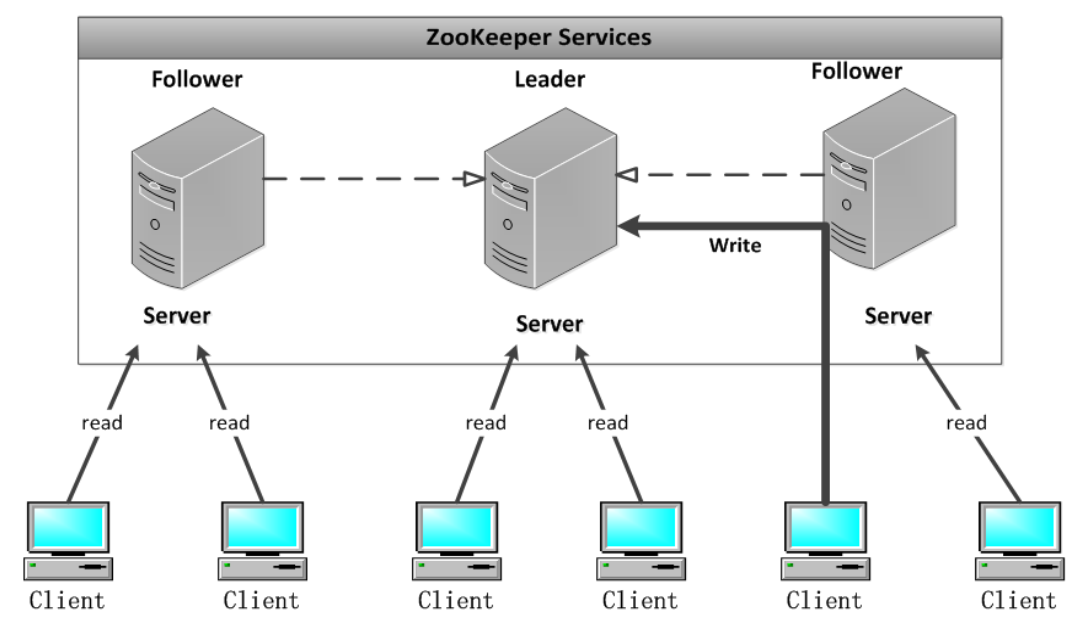
1. **概述**

**ZooKeeper**的**核心思想**是，提供一个非锁机制的Wait Free的用于分布式系统同步的核心服务。提供简单的文件创建、读写操作接口，其系统核心本身对**文件读写**并不提供加锁互斥的服务，但是提供基于版本比对的更新操作，客户端可以基于此自己实现加锁逻辑。

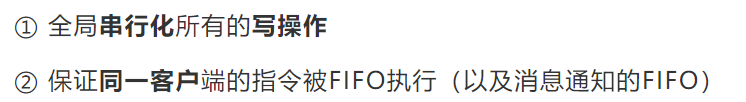


1. **ZK集群服务**

Zookeeper是一个由多个Server组成的集群，该集群有一个Leader，多个Follower。客户端可以连接任意ZooKeeper服务节点来读写数据，



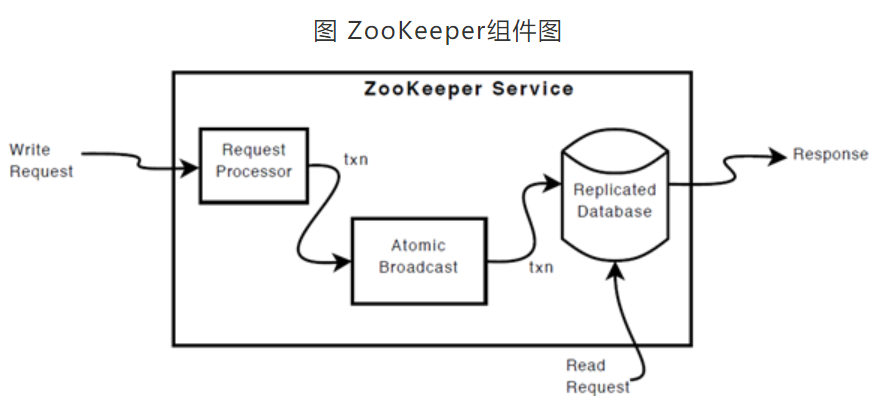
ZK集群中每个Server，都保存一份数据副本。Zookeeper使用简单的同步策略，通过以下两条基本保证来实现数据的一致性：



所有的读请求由Zk Server 本地响应，所有的更新请求将转发给Leader，由Leader实施。

1. **ZK组件**

ZK组件，如图所示。ZK组件除了**请求处理器**（Request Processor）以外，组成ZK服务的每一个Server会复制这些组件的副本。



**ReplicatedDatabase**是一个内存数据库，它包含了整个Data Tree。为了恢复，更新会被记录到磁盘，并且写在被应用到内存数据库之前，先被序列化到磁盘。

每一个ZK Server，可服务于多个Client。Client可以连接到一台Server，来提交请求。**读请求**，由每台Server数据库的本地副本来进行服务。改变服务器的状态的写请求，需要通过一致性协议来处理。

作为一致性协议的一部分，来自Client的所有写请求，都要被转发到一个单独的Server，称作Leader。ZK集群中其他Server 称作Follower，负责接收Leader发来的提议消息，并且对消息转发达成一致。消息层处理leader失效，同步Followers和Leader。

ZooKeeper使用自定义的原子性消息协议。由于消息传送层是原子性的，ZooKeeper能够保证本地副本不产生分歧。当leader收到一个写请求，它会计算出当写操作完成后系统将会是什么状态，接着将之转变为一个捕获状态的事务。

1. **ZK性能**

ZooKeeper被应用程序广泛使用，并有数以千计的客户端同时的访问它，所以我们需要高吞吐量。我们为ZooKeeper 设计的工作负载的读写比例是 2：1以上。然而我们发现，ZooKeeper的高写入吞吐量，也允许它被用于一些写占主导的工作负载。ZooKeeper通过每台Server上的本地ZK的状态副本，来提供高读取吞吐量。因此，**容错性**和**读吞吐量**是以添加到该服务的服务器数量为尺度。**写吞吐量**并不以添加到该服务的机器数量为尺度。

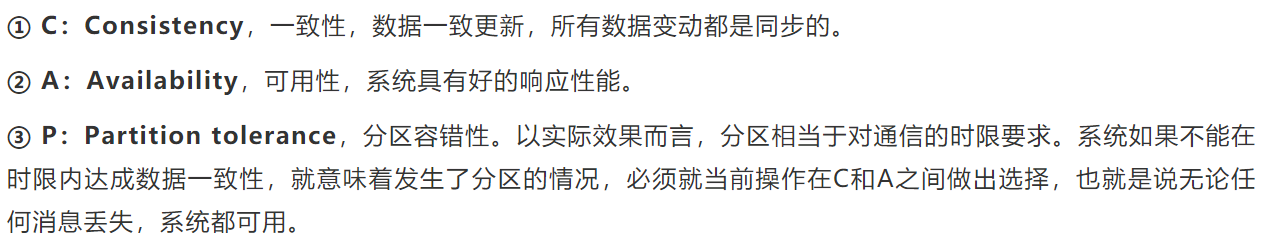
## ZooKeeper的保证

经过上面的分析，我们知道要保证ZooKeeper服务的高可用性就需要采用**分布式模式**，来冗余数据写多份，写多份带来一致性问题，一致性问题又会带来性能问题，那么就此陷入了无解的死循环。那么在这，就涉及到了我们分布式领域的著名的CAP理论，在这就简单的给大家介绍一下，关于CAP的详细内容请看我的《分布式理论》

### CAP理论

1. **理论概述**

分布式领域中存在CAP理论：



该理论已被**证明**：任何分布式系统只可同时满足两点，无法三者兼顾。 因此，将精力浪费在思考如何设计能满足三者的完美系统上是愚钝的，应该根据应用场景进行适当取舍。

1. **一致性分类**

一致性是指从系统外部读取系统内部的数据时，在一定约束条件下相同，即数据变动在系统内部各节点应该是同步的。根据一致性的强弱程度不同，可以将一致性级别分为如下几种：

**① 强一致性**（strong consistency）。任何时刻，任何用户都能读取到最近一次成功更新的数据。

**② 单调一致性**（monotonic consistency）。任何时刻，任何用户一旦读到某个数据在某次更新后的值，那么就不会再读到比这个值更旧的值。也就是说，可获取的数据顺序必是单调递增的。

**③ 会话一致性**（session consistency）。任何用户在某次会话中，一旦读到某个数据在某次更新后的值，那么在本次会话中就不会再读到比这个值更旧的值。会话一致性是在单调一致性的基础上进一步放松约束，只保证单个用户单个会话内的单调性，在不同用户或同一用户不同会话间则没有保障。

**④** **最终一致性**（eventual consistency）。用户只能读到某次更新后的值，但系统保证数据将最终达到完全一致的状态，只是所需时间不能保障。

**⑤ 弱一致性**（weak consistency）。用户无法在确定时间内读到最新更新的值。

### ZooKeeper与CAP理论

我们知道ZooKeeper也是一种分布式系统，它在一致性上有人认为它提供的是一种强一致性的服务（通过sync操作），也有人认为是单调一致性（更新时的大多说概念），还有人为是最终一致性（顺序一致性），反正各有各的道理这里就不在争辩了。然后它在分区容错性和可用性上做了一定折中，这和CAP理论是吻合的。ZooKeeper从以下几点保证了数据的一致性

**① 顺序一致性**

来自任意特定客户端的更新都会按其发送顺序被提交。也就是说，如果一个客户端将Znode z的值更新为a，在之后的操作中，它又将z的值更新为b，则没有客户端能够在看到z的值是b之后再看到值a（如果没有其他对z的更新）。

**② 原子性**

每个更新要么成功，要么失败。这意味着如果一个更新失败，则不会有客户端会看到这个更新的结果。

**③ 单一系统映像**

一个客户端无论连接到哪一台服务器，它看到的都是同样的系统视图。这意味着，如果一个客户端在同一个会话中连接到一台新的服务器，它所看到的系统状态不会比在之前服务器上所看到的更老。当一台服务器出现故障，导致它的一个客户端需要尝试连接集合体中其他的服务器时，所有滞后于故障服务器的服务器都不会接受该连接请求，除非这些服务器赶上故障服务器。

**④ 持久性**

一个更新一旦成功，其结果就会持久存在并且不会被撤销。这表明更新不会受到服务器故障的影响。

## ZooKeeper原理

### 原理概述

Zookeeper的核心是原子广播机制，这个机制保证了各个server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是**恢复模式**和**广播模式。**

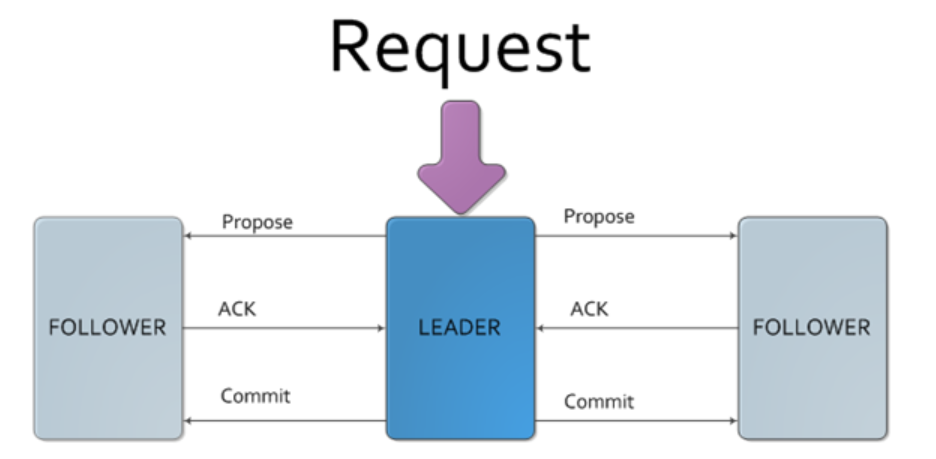
**(1) 恢复模式**

当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和server具有相同的系统状态。

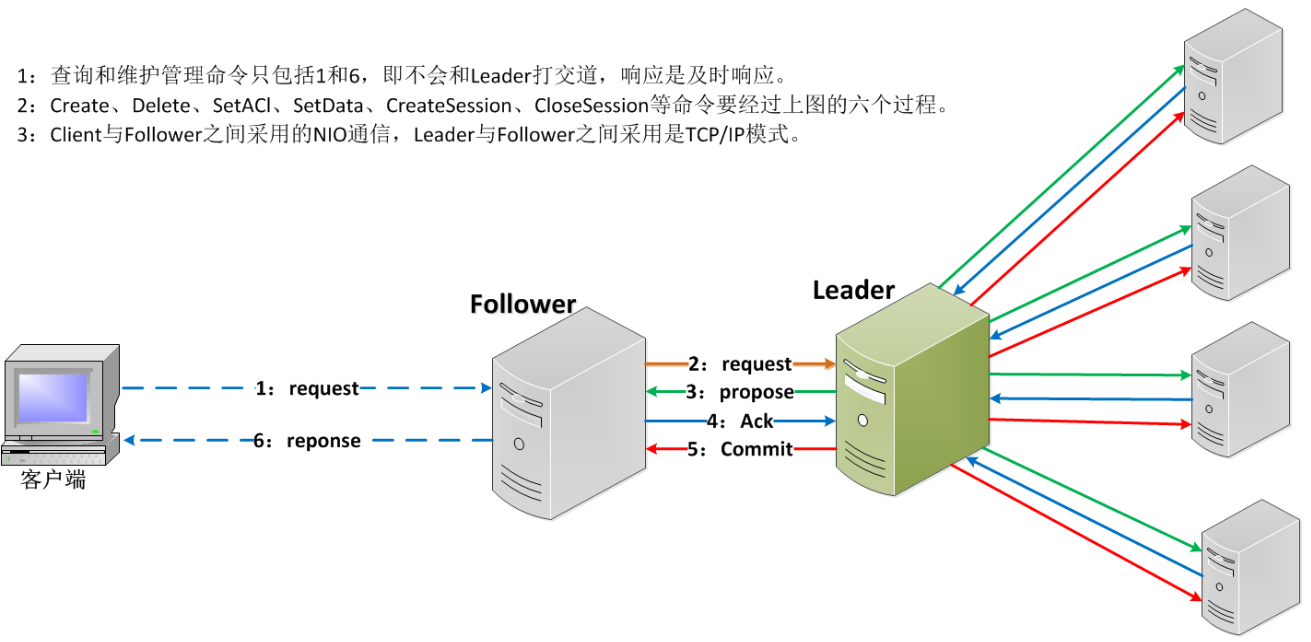
**(2) 广播模式**

一旦Leader已经和多数的Follower进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个Server加入ZooKeeper服务中，它会在恢复模式下启动，发现Leader，并和Leader进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper服务一直维持在Broadcast状态，直到Leader崩溃了或者Leader失去了大部分的Followers支持。

**Broadcast模式**极其类似于分布式事务中的2pc（two-phrase commit **两阶段提交**）：即Leader提起一个决议，由Followers进行投票，Leader对投票结果进行计算决定是否通过该决议，如果通过执行该决议（事务），否则什么也不做。



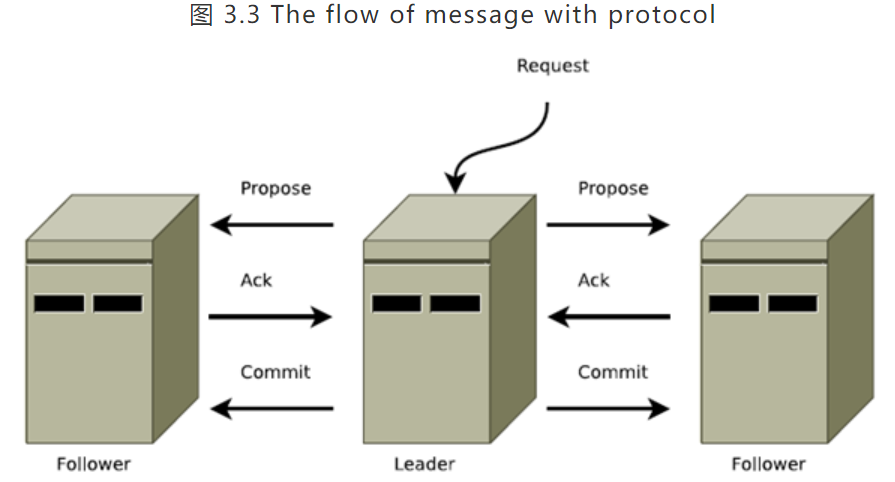
在**广播模式**ZooKeeper Server会接受Client请求，所有的写请求都被转发给**领导者**，再由领导者将更新广播给**跟随者**。当半数以上的跟随者已经将修改**持久化**之后，领导者才会提交这个更新，然后客户端才会收到一个更新成功的响应。这个用来达成共识的协议被设计成具有原子性，因此每个修改要么成功要么失败。



### Zab协议详解

#### **广播模式**

**广播模式**类似一个简单的**两阶段提交**：Leader发起一个请求，收集选票，并且最终提交，图3.3演示了我们协议的消息流程。我们可以简化该两阶段提交协议，因为我们并没有"aborts"的情况。followers要么确认Leader的Propose，要么丢弃该Leader的Propose。没有"aborts"意味着，只要有指定数量的机器确认了该Propose，而不是等待所有机器的回应。



广播协议在所有的通讯过程中使用TCP的FIFO信道，通过使用该信道，使保持有序性变得非常的容易。通过FIFO信道，消息被有序的deliver。只要收到的消息一被处理，其顺序就会被保存下来。

Leader会广播已经被deliver的Proposal消息。在发出一个Proposal消息前，Leader会分配给Proposal一个单调递增的唯一id，称之为**zxid**。因为Zab保证了因果有序，所以递交的消息也会按照zxid进行排序。广播是把Proposal封装到消息当中，并添加到指向Follower的输出队列中，通过FIFO信道发送到 Follower。当Follower收到一个Proposal时，会将其写入到磁盘，可以的话进行批量写入。一旦被写入到磁盘媒介当中，Follower就会发送一个ACK给Leader。 当Leader收到了指定数量的ACK时，Leader将广播commit消息并在本地deliver该消息。当收到Leader发来commit消息时，Follower也会递交该消息。

需要注意的是， 该简化的两阶段提交自身并不能解决Leader故障，所以我们 添加恢复模式来解决Leader故障。

#### **恢复模式**

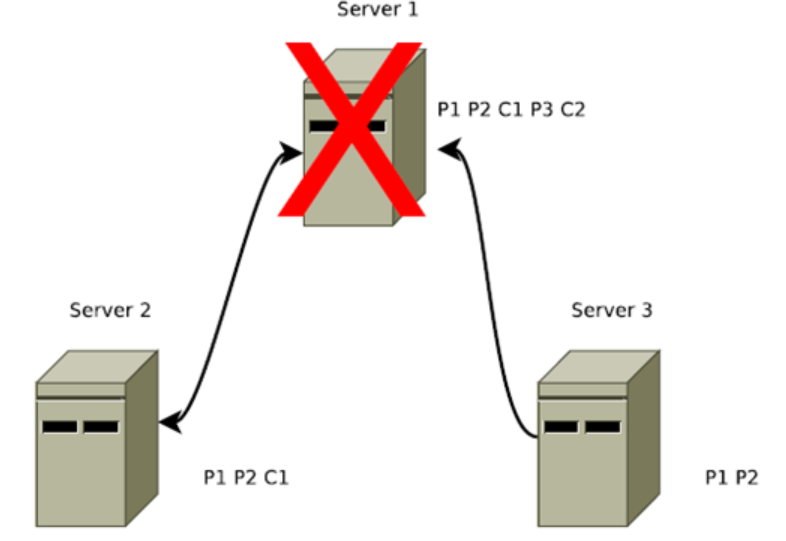
1. **恢复阶段概述**

正常工作时Zab协议会一直处于广播模式，直到Leader故障或失去了指定数量的Followers。为了保证进度，恢复过程中必须选举出一个新Leader，并且最终让所有的Server拥有一个正确的状态。对于Leader选举，需要一个能够成功高几率的保证存活的算法。Leader选举协议，不仅能够让一个Leader得知它是leader，并且有指定数量的Follower同意该决定。如果Leader选举阶段发生错误，那么Servers将不会取得进展。最终会发生超时，重新进行Leader选举。在我们的实现中，**Leader选举**有两种不同的实现方式。如果有指定数量的Server正常运行，快速选举的完成只需要几百毫秒。

1. **恢复阶段的保证**

**第一条：**

**若一条消息在一台机器上被deliver，那么该消息必须将在每台机器上deliver，即使那台机器故障了**。例如，出现了这样**一种情况**：Leader发送了commit消息，但在该commit消息到达其他任何机器之前，Leader发生了故障。也就是说，只有Leader自己收到了**commit**消息。如图3.4中的**C2**。因为Leader已经deliver了该C2消息，client能够在消息中看到该事务的结果。所以该事务必须能够在其他所有的Server中deliver，最终使得client看到了一个一致性的服务视图。



**第二条：**

**一个被skip的消息，必须仍然需要被skip。**例如，发生了这样**一种情况：**Leader发送了propose消息，但在该propose消息到达其他任何机器之前，Leader发生了故障。也就是说，只有Leader自己收到了propose消息。如图3.4中的P3所示。在图3.4中没有任何一个server能够看到**3号提议**，所以在图3.5中当server 1恢复时他需要在系统恢复时丢弃三号提议P3。

