# Zab协议

ZAB 协议是为分布式协调服务ZooKeeper专门设计的一种支持崩溃恢复的一致性协议。基于该协议，ZooKeeper 实现了一种主从模式的系统架构来保持集群中各个副本之间的数据一致性。

ZAB协议运行过程中，所有的客户端更新都发往Leader，Leader写入本地日志后再复制到所有的Follower节点。

一旦Leader节点故障无法工作，ZAB协议能够自动从Follower节点中重新选择出一个合适的替代者，这个过程被称为选主，选主也是ZAB协议中最为重要和复杂的过程。本文主要描述ZAB协议的选主过程以及在Zookeeper中的实现。

## 选主时机

**节点启动时**

每个节点启动的时候状态都是LOOKING，处于观望状态，接下来就是要进行选主了。

**Leader节点异常**

Leader节点运行后会周期性地向Follower发送心跳信息（称之为ping），如果一个Follower未收到Leader节点的心跳信息，Follower节点的状态会从FOLLOWING转变为LOOKING。

**多数Follower节点异常**

Leader节点也会检测Follower节点的状态，如果多数Follower节点不再响应Leader节点（可能是Leader节点与Follower节点之间产生了网络分区），那么Leader节点可能此时也不再是合法的Leader了，也必须要进行一次新的选主。

## 选主流程

### 基本概念

**election epoch**

这是分布式系统中极其重要的概念，由于分布式系统的特点，无法使用精准的时钟来维护事件的先后顺序，因此，Lampert提出的Logical Clock就成为了界定事件顺序的最主要方式。

分布式系统中以消息标记事件，所谓的Logical Clock就是为每个消息加上一个逻辑的时间戳。（逻辑时钟，用来判断是否为同一次选举。每调用一次选举函数，logicalclock自增1，并且在选举过程中如果遇到election比当前logicalclock大的值，就更新本地logicalclock的值。）在ZAB协议中，每个消息都被赋予了一个zxid，zxid全局唯一。zxid有两部分组成：高32位是epoch，低32位是epoch内的自增id，由0开始。每次选出新的Leader，epoch会递增，同时zxid的低32位清0。这其实像极了咱们古代封建王朝的君主更替，每一次的江山易主，君王更替。

**zxid**

每个消息的编号，在分布式系统中，事件以消息来表示，事件发生的顺序以消息的编号来标记。在ZAB协议中，这就是zxid。ZAB协议中，消息的编号只能由Leader节点来分配，这样的好处是我们就可以通过zxid来准确判断事件发生的先后，记住，是任意事件，这也是分布式系统中，由全局唯一的主节点来处理更新事件带来的极大好处。

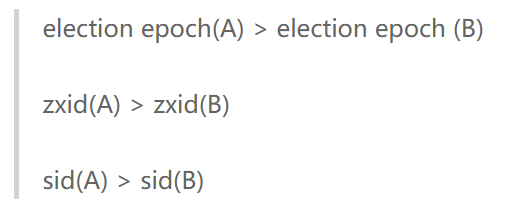
分布式系统运行的过程中，Leader节点必然会发生改变，一致性协议必须能够正确处理这种情况，保证在Leader发生变化的时候，新的Leader期间，产生的zxid必须要大于老的Leader时生成zxid。这就得通过上面说的epoch机制了，具体实现会在下面的选主过程中详细描述。

**LOOKING/FOLLOWING/LEADING**

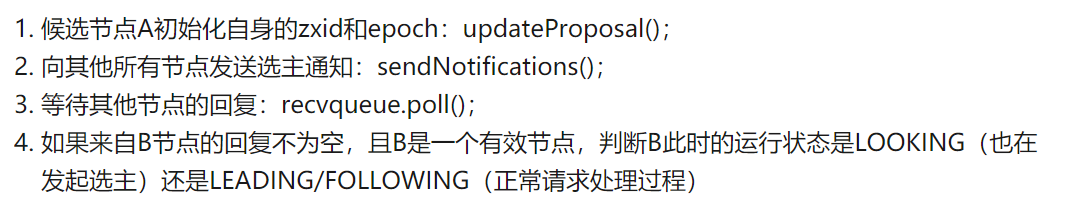
这三者描述了系统中节点的状态：

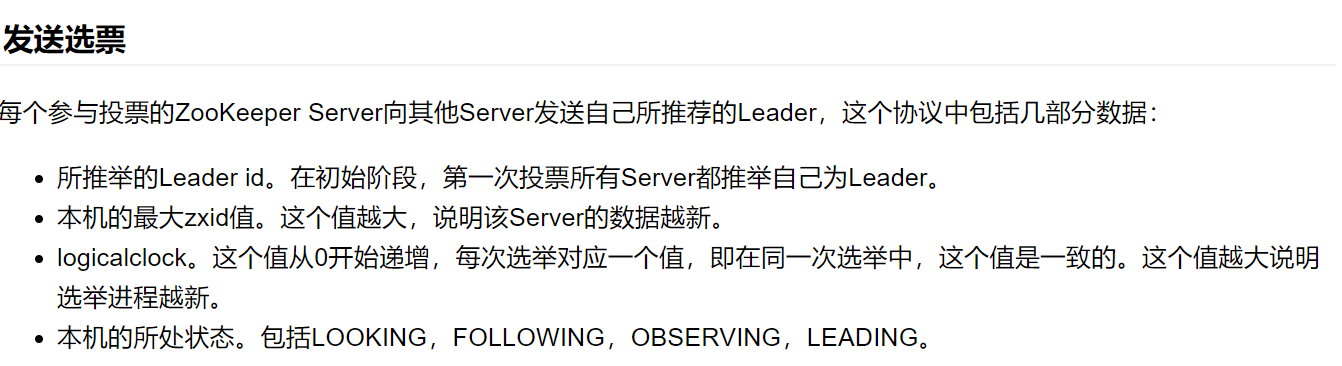
* LOOKING: 节点正处于选主状态，不对外提供服务，直至选主结束；
* FOLLOWING: 作为系统的从节点，接受主节点的更新并写入本地日志；
* LEADING: 作为系统主节点，接受客户端更新，写入本地日志并复制到从节点

节点B判断确定A可以成为主，那么节点B就投票给节点A，判断的依据是：



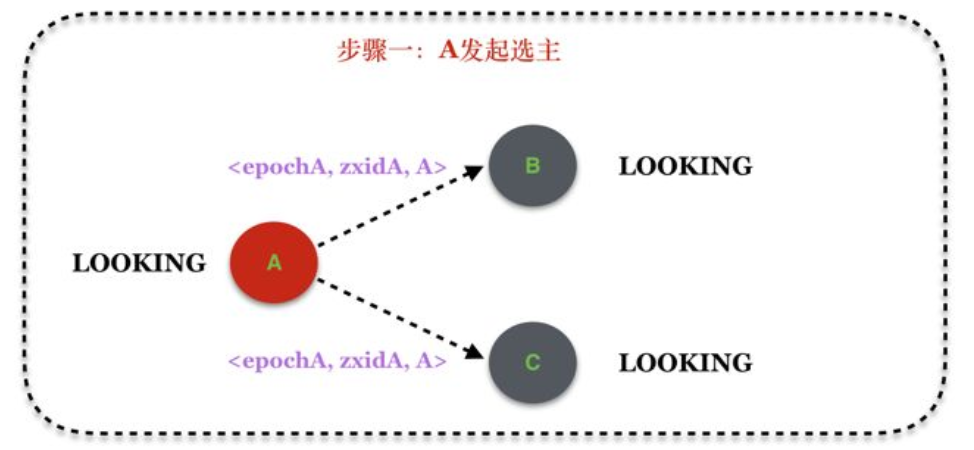
### 主流程



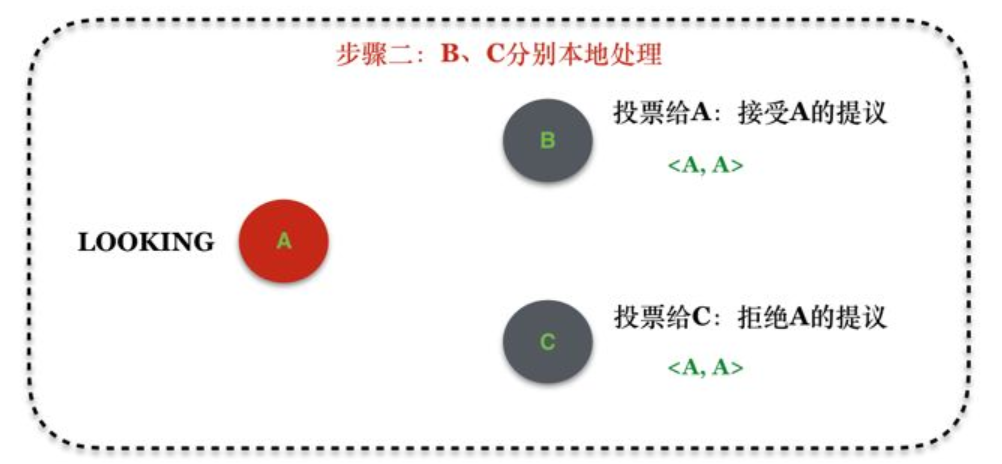


#### 情况1：投票节点是LOOKING状态

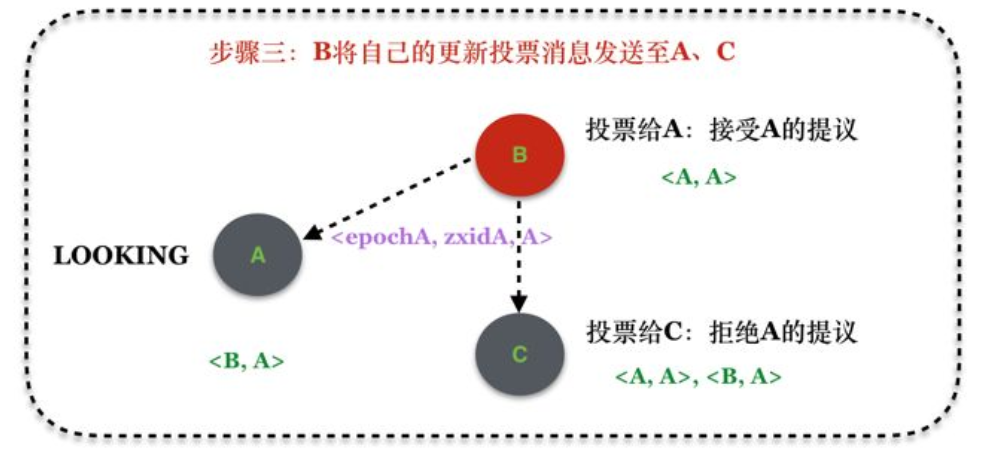
**STEP 1**：处于LOOKING状态的A发起一次选主请求，并将请求广播至B、C节点，而此时B、C也恰好处于LOOKING状态：



**STEP 2**：B、C节点处理A的选主消息，其中，B接受A的提议，C拒绝A的提议，伴随着A的选主消息的一个额外收获是B和C此时都获得了A节点选主的结果，记录该信息，作为后续判断大家是否达成一致的标准。



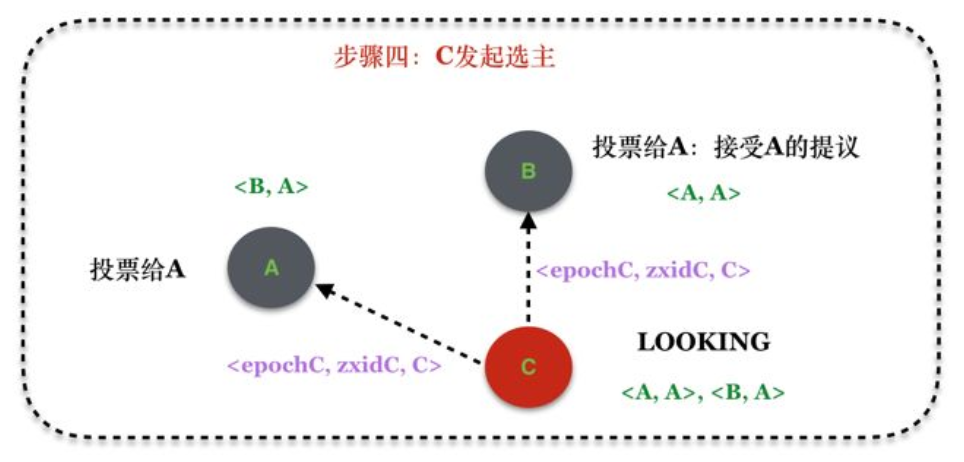
**STEP 3**：B将处理结果通知A、C



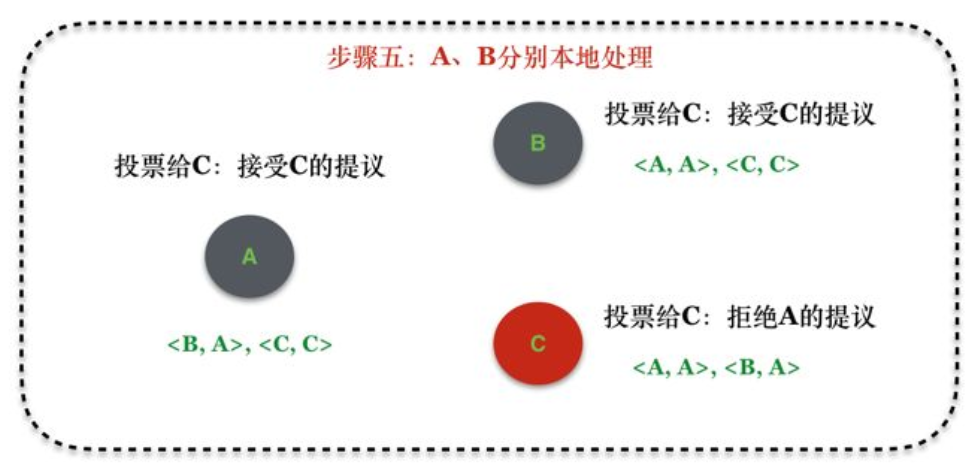
说明：

* 因为B更新了自己的投票，从投票给自己变成投票给A，因此根据协议的定义，需要将该消息扩散出去。而C由于拒绝了A的提议，因此，无需扩散消息；
* B将消息扩散给A和C的同时，A和C也就了解了B的投票信息，可以更新本地的投票信息表，例如上面经过B的扩散后，A知道了B节点的投票信息，C知道了A和B节点的投票信息。

**STEP 4**：C同时也发起选主



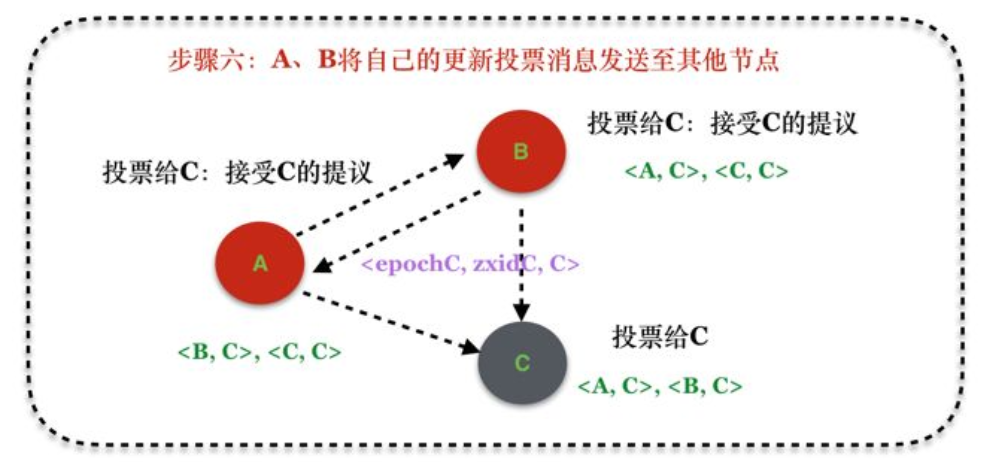
**STEP 5**：A、B分别处理C的选主请求



说明：

* 这里A和B判断得出C是最合适的Leader，因此A和B都更新自己的候选Leader为C，同时由于C的消息，A和B都更新自身维护的投票信息，增加C的投票信息

**STEP 6**：A、B将更新后的信息扩散到其他节点



说明：

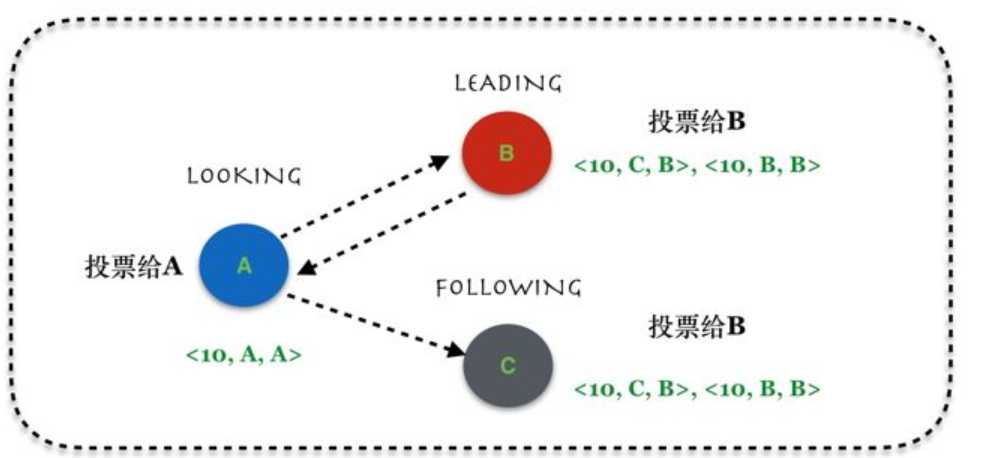
* 因为在第五步中A和B分别将自己的候选Leader变成了C，因此需要将该信息通知到其他节点，其他节点在收到新的投票信息后会更新本地的投票信息列表，如上图。

**STEP 7**: 选主结束

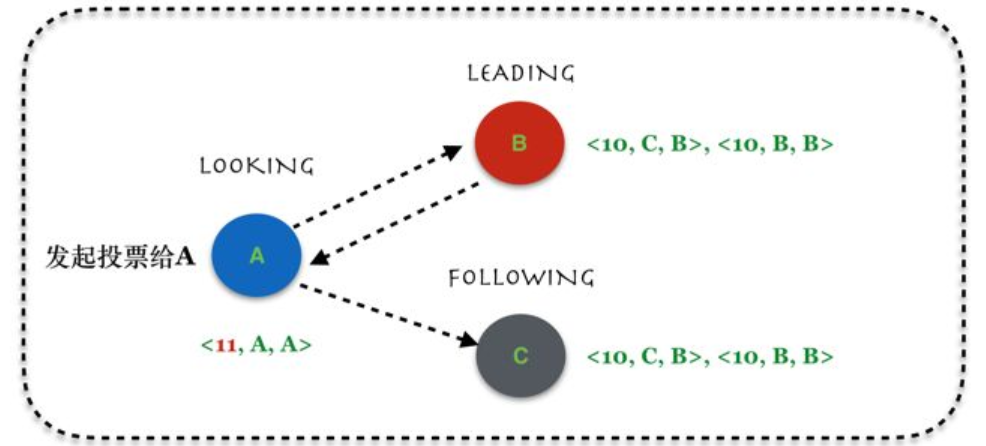
#### 情形2：投票节点是FOLLOWING/LEADING状态

* 如果LogicalClock相同，将数据保存早recvset，如果Sender宣称自己是Leader，那么判断是不是半数以上的服务器都选举它，如果是设置角色并退出选举。
* 否则，这是一条与当前LogicalClock不符合的消息，说明在另一个选举过程中已经有了选举结果，于是将该选举结果加入到OutOfElection集合中，根据OutOfElection来判断是否可以结束选举，如果可以也是保存LogicalClock，更新角色，退出选举。

logical clock相同可能是因为出现这种情况：A、B同时发起选主，此时他们的election epoch可能相同，如果B率先完成了选主过程（B可能变成了Leader，也有可能B选择了其他节点为Leader），但是A还在选主过程中，此时如果B收到了A的选主消息，那么B就将自己的选主结果和自己的状态（LEADING/FOLLOWING）连同自己的election epoch回复给A，对于A来说，它收到了一个来自选主完成的节点B的election epoch相同的回复，便有了上面的第一种情况；

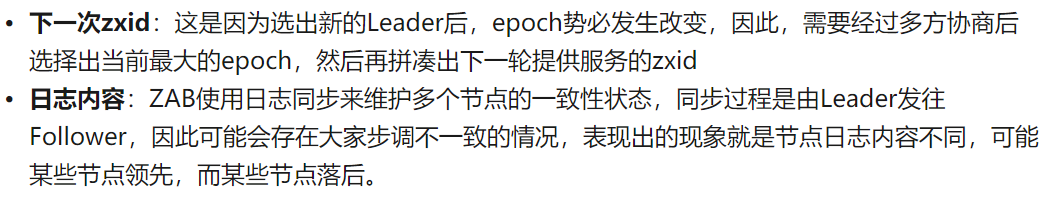


logical clock不相同可能是因为新增了一个节点或者某个节点出现了网络隔离导致其触发一次新的选主，然后系统中其他节点状态依然正常，此时发起选主的节点由于要递增其logical clock，必然会导致其logical clock要大于其他正常节点的logical clock（当然也可能小于，考虑一个新上线节点触发选主，其logical clock从1开始计算）。因此就出现了上面的第二种情况，如下图：



## Leader/Follower信息同步

选出了Leader还不算完，根据ZAB协议定义，在真正对外提供服务之前还需要一个信息同步的过程。具体来说，Leader和Follower之间需要同步以下信息：



### Epoch协商

选主过程结束后，接下来就是多数派节点协商出一个最大的epoch（但如果是采用FastLeaderElection算法的话，选出来的Leader其实就拥有了最大的epoch）



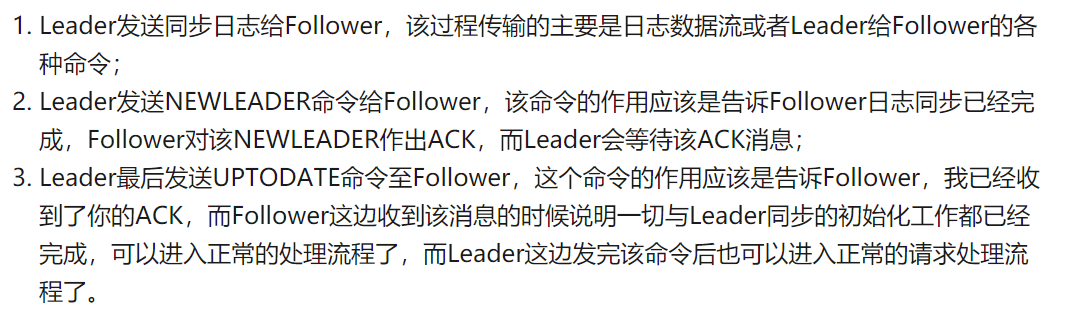
### 日志同步

选主结束后，接下来需要在Leader和Follower之间同步日志，根据ZAB协议定义，这个同步过程可能是Leader流向Follower。

对比的原理是将Follower的最新的日志zxid和Leader的已经提交的日志zxid对比，会有以下几种可能：



即使是一个日志同步过程也要经历以下几个同步过程：



使用日志实现分布式系统一致性的方案中，日志代表了系统中发生的事件，而日志存在两种状态：

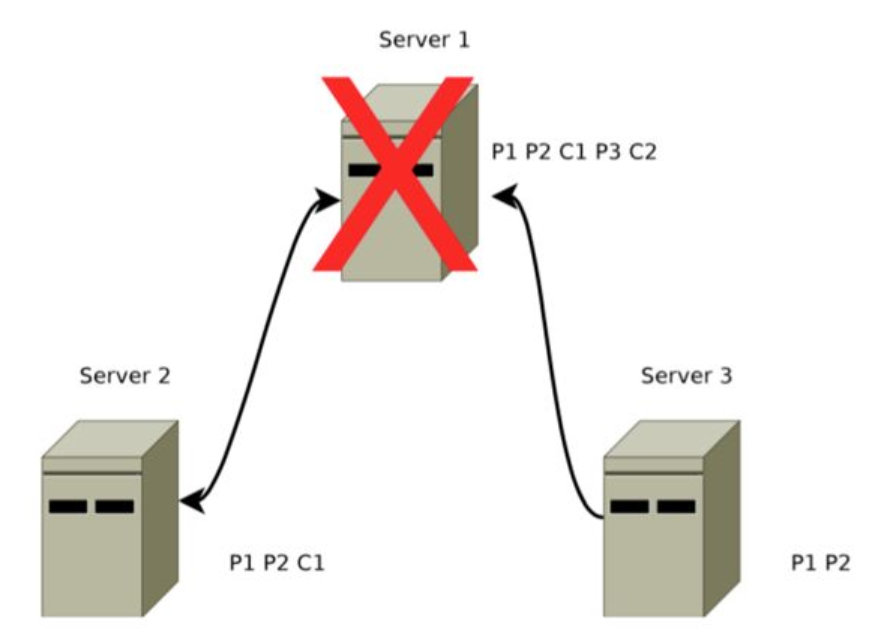
* 发起（Proposal）：日志已经被记录在Leader/Follower的日志文件中，相当于节点已经记录了该事件；
* 提交（Commit）：一旦事件被多数节点记录，Leader节点便提交该日志，即处理事件。事件被处理完成后，Leader才会给予客户端答复，后续，Leader节点同样会将该Commit命令通知Follower节点。

一旦日志被提交，那么在客户端看来事件已经被系统处理，那该事件产生的状态就不能凭空消失，因此，在选主协议中最重要的两点保证是：

* 已经被处理的消息不能丢
* 被丢弃的消息不能再次出现

#### 已经被处理的消息不能丢

这一情况会出现在以下场景：当 leader 收到合法数量 follower 的 ACKs 后，就向各个 follower 广播 COMMIT 命令，同时也会在本地执行 COMMIT 并向连接的客户端返回「成功」。但是如果在各个 follower 在收到 COMMIT 命令前 leader 就挂了，导致剩下的服务器并没有执行都这条消息。



如图，消息 1 的 COMMIT命令C1在 Server1（Leader）和 Server2（Follower）上执行了，但是在Server3收到消息C1之前Server1便挂了，客户端很可能已经收到消息C1已经成功执行的回复，协议需要保证重新选主后，C1消息不会丢失。

为了实现该目的，Zab选主时使用以下的策略：

选择拥有 proposal 最大值（即 zxid 最大） 的节点作为新的 Leader。

由于所有提案被COMMIT 之前必须有大多数量的 Follower ACK，即大多数服务器已经将该 proposal写入日志文件。因此，新选出的Leader如果满足是大多数节点中proposal最多的，它就必然存有所有被COMMIT消息的proposal。

接下来，新Leader与Follower 建立先进先出的队列， 先将自身有而Follower缺失的proposal发送给 它，再将这些 proposal的COMMIT命令发送给 Follower，这便保证了所有的Follower都保存了所有的 proposal、所有的Follower 都处理了所有的消息。

通过以上策略，能保证已经被处理的消息不会丢

#### 被丢弃的消息不能再次出现

这一情况会出现在以下场景：当Leader 接收到消息请求生成 proposal后就挂了，其他Follower 并没有收到此proposal，因此新选出的Leader中必然不含这条消息。 此时，假如之前挂了的Leader 重新启动并注册成了Follower，它要与新的Leader保持一致，就必须要删除自己上旧的proposal。

Zab 通过巧妙的设计 zxid 来实现这一目的。一个 zxid 是64位，高 32 是纪元（epoch）编号，每经过一次 Leader选举产生一个新的Leader，其epoch 号 +1。低 32 位是消息计数器，每接收到一条消息这个值 +1，新Leader 选举后这个值重置为 0。

这样设计的目的是即使旧的Leader 挂了后重启，它也不会被选举为Leader，因为此时它的zxid 肯定小于当前的新Leader。另外，当旧的Leader 作为Follower提供服务，新的Leader也会让它将所有多余未被COMMIT的proposal清除。