# 高可用问题的引入

## 简单并不总那么有效

假设你是个做蛋糕的，有个处理订单的系统，通过这个系统能够记录每一笔订单的详细情况并可以随时查询。用户可以通过手机App或者网页查询。如果你只有一台电脑，软件也没有bug，运行的很好，一切看起来那么正常。

但是有一天电脑硬盘坏了，虽然头天晚上做了备份，但是今天的订单全丢了。当然坏硬盘发生的概率往往没那么高，但是天热机器发生个风扇故障的概率要高的多。在这种情况下，即使之前的数据没丢，你也不能正常处理订单了。

## 高可用组合没看起来那么好

你有个朋友，给你建议上个高可用系统，双服务器M1和M2，只要其中一个能干活就可以。问题在哪呢？ 你的App或者浏览器，在发送时，可能发给M1，也可以发给M2，这种情况下，如果现在只剩下一个草莓蛋糕，张三和李四两个同学都通过App下单，但是分别是M1和M2处理的，那么可能两个人都认为自己订蛋糕成功了，但是去取的时候，有一个人发现没有了。显然，两个没有写作的服务器，不能干活。

你决定只有一个服务器能接收请求，张三李四都得找M1下订单。M1收到订单后，M1和M2一起处理同一个订单，做同样的操作，它们都处理完成才算成功。这下可靠性貌似提高了，比如M2挂了，M2发现没法发订单给小M2，M1就独立处理了。可是如果M2好好的，M1挂了呢？这个看起来也很简单: 直接让M2接订单并独立干活，不理M1不就完了吗？

问题在于：谁决定让M2独立干活，不理小M1？M2自己？还是店老板您呢？

显然不能是店老板，他没法一直在那看着。

那M2自己判断是不是就可以了？看起来好像对头，但是M2如何判断M1宕机了呢？在网络不稳定的情况下，**可能出现客户张三找到了M1，李四找到了M2。在每个订单被处理时，都认为另外一个服务器挂了，自己应该独立处理**。这个就是脑裂问题，关于这个问题，有个用的比较多的解释如下：

TODO: 脑裂问题示意图：

# What does "split-brain" mean?

"Split brain" is a condition whereby two or more computers or groups of computers lose contact with one another but still act as if the cluster were intact. This is like having two governments trying to rule the same country. If multiple computers are allowed to write to the same file system without knowledge of what the other nodes are doing, it will quickly lead to data corruption and other serious problems.

Split-brain is prevented by enforcing quorum rules (which say that no group of nodes may operate unless they are in contact with a majority of all nodes) and fencing (which makes sure nodes outside of the quorum are prevented from interfering with the cluster).

简言之，就是每个节点都认为对方挂了，自己应该多干活，结果就乱套了。如果这个系统是做饭馆餐桌预定的，就会出现大家都认为自己订到了某个包间的情况。。。

# Paxos的演变

前面已经说明了单个服务器所存在的单点故障问题，也说明了多个服务器下，网络分割可能带来的脑裂问题。

一个自然的想法是，如果我们有多个服务器，就不是某一个服务器说了算，而是把这些服务器组成一个委员会，多数都同意通过的才能形成决定。一旦形成决定，那么所有的服务器都的遵从这个决定。比如，决定第一个订单时张三的，第二个是李四的，王五的订单因为蛋糕没了失败。

**要考虑的因素：**

网络可能丢失数据、可能有很大延迟、可能产生分割。

上述问题可以归结为：各个服务器务必按照相同顺序处理订单。在其中一个服务器故障后，如果还有其他服务器可以继续进行处理，那么切换过程不能造成混乱。即使发生切换，对客户来说，必须跟单个服务器在做处理是一样的结果，就是可用性更高了一点而已。具体可以理解为：

各个服务器都执行相同的订单序列，例如:

1号订单，是张三的2个草莓蛋糕

2号订单，是李四的3个慕斯蛋糕

3号订单，是王五的2个巧克力奶昔

...

注意，张三是不是真的下订单时间比李四早，在这里没那么重要，上面的三个人他们完全可能是同时下的订单，也可以是张三下订单比李四晚，但是张三的手机用的是5G网络。但是我们不care这个，只要各个服务器都认为张三的订单是1号订单，李四的是2号订单......就可以了。当然，排在后面的订单有可能买不着哦。

为了保证可靠性，我们的目标是用多个服务器组建集群，并且保证在其中大部分机器正常工作时，服务不中断，保证一致性。当然，前提得是个自治的系统，不能靠店老板在那看着哪个服务器应该干活。

1. ==> prepare request(proposal number n)

2. <==, response(保证不accept比n小的，保护我已经accept的最大的proposal number accepted.val, accpeted.num)

3.==> accept request(proposal number n, val v)

4.<== accept (n, v)

acceptor需要持久化的部分： max\_responded\_pn (proposal number), max\_accepted\_pn, max\_accepted\_val.

requestor需要持久化的部分：last\_proposed\_pn, last\_proposed\_val，需要二者一致

P1, P2, P2a, P2b, P2c之间的关系，已经在visio图中有，但是不美观。

既然一个value被majority个accept后，就称为被chosen，如果一个acceptor能够accept多个value，那么会形成混乱，多个都被chosen。因此，每个acceptor只能接受一个value。

为什么要有requirement P1?

**P1**. An acceptor must accept the first proposal that it receives //首个必须接受

==>不考虑消息丢失等问题，如果只有一个proposer，只propose了一个value，我们也希望能成功。所以必须acceptor必须接受其收到的第一个proposal。

按照P1，可能存在多个proposal被并发，每个都被少数派accept，从而不能形成结论，即不能chose任何value。为了保证这种情况下能够继续，或者多轮proposal能最终成功，就要求每个acceptor能够允许接受多个proposal。

为了不混淆，需要对各个proposal进行编号，编号方法另外描述。

既然允许每个acceptor接受多个proposal，并且一个proposal一旦被majority个接受，那么其value就被选择。那么就可能有多个value被chosen，即多次形成决议。注意，majority个acceptor接受即认为形成决议，但是proposer可能并未及时收到决议，或者决议本身丢失了，导致新一轮proposal。为了保证多次可能的决议的一致性，这就要求：

**P2**. If a proposal with value *v* is chosen, then every higher-numbered proposal that is chosen has value *v*. **//****只有一个value能被选中**

满足P1的一个自然想法是：

**P2*a***. If a proposal with value *v* is chosen, then every higher-numbered proposal accepted by any acceptor has value *v*. **//只允许接受已被选中的v**(在已有value v被选中的情况下)

由于通信是异步的，一个刚刚醒来的proposer，根本不能知道那些value被accept了，但是其proposal的number可能更大。按照p1，acceptor需要接受它，但是可能违反了P2a。为此，对P2a进行加强。即为了同时满足p1和P2a,P2a变成：

**P2*b*.** If a proposal with value *v* is chosen, then every higher-numbered proposal issued by any proposer has value *v*.  **//只允许提议已被选中的v** (在已有value v被选中的情况下)

如何满足P2b的一段推导，很有意思：

原文下面一段，需要好好理解：

Combining this with the induction assumption, the hypothesis that *m* is chosen implies:

Every acceptor in *C* has accepted a proposal with number in *m . .* (*n −* 1), and every proposal with number in *m . .* (*n −* 1) accepted by any acceptor has value *v*.

我认为这里面的C，可能只是个泛指。对于不同的proposal number，即*m . .* (*n −* 1)，对应的C可能是不同的。但是都有个majority集合。

Since any set *S* consisting of a majority of acceptors contains at least one member of *C* (这句话是很重要的过渡。前面没有提到S，为什么？), we can conclude that a proposal numbered *n* has value *v* by ensuring that the following invariant is maintained:

**P2*c***. For any *v* and *n*, if a proposal with value *v* and number *n* is issued, then there is a set *S* consisting of a majority of acceptors such that either (a) no acceptor in *S* has accepted any proposal numbered less than *n*, or (b) *v* is the value of the highest-numbered proposal among all proposals numbered less than *n* accepted by the acceptors in *S*. **//****总是提议能看到的PN最高的value**

实际上，这个不变式就是为了保证proposal number变化时，*v*的不变性。即要么S中任意一个都没有accept过小于n的proposal，这时候肯定没有形成一致，即没有一个value被accept了；要么v就是S中acceptor接受过的proposal number最高prosoal对应的value。

如何从P2c推出能达到P2b?

如果在PN 为m时，v被选中了，那么表示有个majority C，全部accept了v。

我们要论证按照P2C这个不变式，在PN为m+1.. n时，一定能会propose v。

在确定PN=m+1的proposal 的value x时，我们找到了某个majority，S中的最大value，那么S一定包含C中某个acceptor，因为二者都是majority。而m是最大值，所以一定选择对应m的value，即v。

在PN=m+2..n区间，考虑任意的PN i，由于在PN属于[m, i-1]的proposal 的value都是v，而S必然至少有一个与C重合，因此查到的PN最大的value，仍然是v。

highest-numbered 为什么重要？

首先，既然说highest，那么就隐含着，大家accept的proposal number可能是不同的。

其次，不存在number x, x > highest-number，且被majority个接受了。否则S中一定有至少一个接受了x，矛盾。

我理解，这样好实现，在后面，直接让acceptor承诺，不accept小于n的proposal。

注意，在讨论前面这些条件时，并没有牵涉到prepare 和 accept两个阶段，我认为是按照就一个阶段讨论的。

# 论文笔记

## 一些问题

对于响应只读请求这块，大家分别用了不同的方法。Lamport的make simple没有提及，raft是让leader不断获取peon的heartbeat保证自己活着，ceph让各个peon在拥有lease等条件下响应读请求。

有没有一个通用的办法来解决呢？

https://www.cs.cmu.edu/~dga/papers/leases-socc2014.pdf

这个里面不需要时钟完全同步，但是要求时钟频率接近。此文已下载。

Replicas do not synchronize clocks, but their clock rates are assumed to be similar, such that a modest guard time can account for clock drift over a short interval.

参考文档：https://simbo1905.wordpress.com/2014/11/09/paxos-and-read-consistency/

Paxos And Read Consistency

其中提到的另外一篇：

"How to Build a Highly Available System Using Consensus"

专利：

Read-only operations processing in a paxos replication system

## 其他参考

https://understandingpaxos.wordpress.com/ 这个应该不错

## 一些细节问题

### Lease干什么的？

Ceph的文档里面，不容易看出来。但是从raft的paper看，lease不是授权follower回答只读请求，相反，是让leader确认自己还是个leader，不会在被夺权后，还以为自己是个leader，从而回答旧的信息。

Ceph的lease，含义是否跟raft的paper说的一样，还需要仔细理解。

### multi-paxos

https://www.zhihu.com/question/36648084

这个里面，郁白说multi-paxos不限制日志之间的顺序或者依赖关系，各个instance之间是完全独立的。

http://zhuanlan.zhihu.com/likai/20417442

这里说的幽灵问题，如果是两阶段，先prepare了，再propose。如果B和C都没有6，那么后面7-10，其value就是空的，并不是之前第一轮时A想提议的value。

作者的这个例子中，到底是否允许多个pending的request? 如果允许，那么没什么奇怪的，没执行完成不算事！另外，即使有多个并发，向client确认执行完成的过程，是否严格按照顺序？ 如果按照顺序确认，也没问题的。 5-20这些请求，应用要求按照什么顺序执行？到底有没有要求?

其实在"Paxos Made Simple"里面有一段，我认为已经回答了这个问题。但是上面知乎网页里面所说，应该不是paxos，而是Raft:

The new leader, being a learner in all instances of the consensus algorithm, should know most of the commands that have already been chosen.

...

Suppose that the outcome of these executions determine the value to be proposed in instances 135 and 140, but leaves the proposed value unconstrained in all other instances.

...

Instead, we let it fill the gap immediately by proposing, as commands 136 and 137, a special “noop” command that leaves the state unchanged.

其实如果考虑超时，即使不发生leader更替，下面两个问题依然存在: 1) 如果过一段时间查一次，肯定会出现不一致问题，即开始查不到，后来查到了；2) 超时到底client认为失败，重新发送相同的命令。 我理解这个应该不是paxos本身的问题，而是跟client之间的协调问题。

PAXOS的一致性，是指保证的是多个Server之间的状态修改顺序相同！至于多个并发的client操作到底谁先执行，这个超出范畴！

郁白的博客中，[liutongxuan](http://www.zhihu.com/people/liutongxuan" \t "_blank)的注释，已经说明了新leader当选后要做的工作，在其开始提供服务前，会有一段状态同步时间的，不是直接对client提供查询。

另外，在“Implementing Replicated Logs with Paxos”一文中，有下面描述。这个能避免幽灵问题吗？

1) Leader does not respond until command has been chosen for log entry and executed by leader’s state machine。

2) If request times out (e.g., leader crash):

I.Client reissues command to some other server；

II.Eventually redirected to new leader；

III. Retry request with new leader.

### Number跟Value什么关系？为什么要两个东西？

分别对应request和accept两个phase，参见made simple一文。 注意这个number，虽然在多个proposer并发并冲突时，可能一直增长，但是仍然属于一个instance以内。number被majority个acceptor接受后，才可以进入accept阶段。

### 为什么要分两个phase? 不能直接一次把value跟着proposal也发出来吗？在什么情况下会导致什么问题？

这里把作者文章中，从单阶段演变到多阶段的过程进行梳理，得出两阶段的由来。

P1. An acceptor must accept the first proposal that it receives.

P1 and the requirement that a value is chosen only when it is accepted by a majority of acceptors => 为了保证能成功，每个acceptor可能需要接受多个proposal =>可以允许 多个proposal被choosen，但是被choosen(不是accept)的proposal的 value必须要相同=>(原文中的P2)每个被choosen的value，要跟上第一个被choosen的value相同=> (原文中的P2a) 如果有V被choosen，那么后续所有被accept（不是choosen）的proposal，其value必须也是V => (原文中的P2b)如果有被choosen的value V，那么后续发送的number更大的proposal，其value必须是V => 用归纳法证明P2b，需要在证明中保证的不变式，为P2c =>由于P2c中部分条件属于未来，难以预测，但是我们可以很容易获取或者废掉过去的东西，用更高的proposal number 废掉所有更低的，因此有了2 phases。

P2a是P2的充分条件。

P2b是P2a的一个加强，是同时考虑P1和P2a得到的条件。因为考虑了“Because communication is asynchronous, a proposal could be chosen with some particular acceptor *c* never having received any proposal. Suppose a new proposer “wakes up” and issues a higher-numbered proposal with a different value. P1 requires *c* to accept this proposal, violating P2*a*”。

Acceptor *c*可能没有参与或者收到前面已经被choosen的value V(proposal n)，这时一个刚醒来的proposer发了一个proposal k，对于*c*来说，按照P1要求，必须accept proposal k，按照P2a，只能accept value等于V的请求。因此，这就要求proposal k的value必须等于V，而刚醒来的proposer此时并不知道V。

至于k是多少？我的理解： 有些是给每个propser分配一个固定的id，这个id作为proposal number的低位。这样就可能很容易出现一个proposal一上来就来了个大的proposal number。为什么不能让刚醒来的proposer知道上一个被接受的proposal number和 value ==>实际上可能某个value正在被accept，但是我们不确定，没法去等待它完成，也不知道是否存在这个value。

P2c怎么来的？ 是用归纳法证明P2b过程中需要的一个维持的一个不变式，这个是归纳法常有的一个方法。

can therefore satisfy P2*b* by maintaining the invariance of P2*c*. To maintain the invariance of P2*c*, a proposer that wants to issue a proposal numbered *n* must learn the highest-numbered proposal with number less than *n*, if any, that has been or will be accepted by each acceptor in some majority of acceptors. Learning about proposals already accepted is easy enough; predicting future acceptances is hard. Instead of trying to predict the future, the proposer controls it by extracting a promise that there won’t be any such acceptances. In other words, the proposer requests that the acceptors not accept any more proposals numbered less than *n*. This leads to the following algorithm for issuing proposals.

在此段之前，说的都是value跟number一块发的，没有分phase。

为什么搞定P2c是困难的，要预测的future是什么呢？是：“可能要被接受的proposal”

To maintain the invariance of P2*c*, a proposer that wants to issue a proposal numbered *n* must learn the highest-numbered proposal with number less than *n*, if any, that has been or will be accepted by each acceptor in some majority of acceptors.

知道已经被choosen的比较好办，但是预测谁会被choose比较难。因此，选择不做预测。

仅仅不做预测是不行的，因为矛盾还会产生。那怎么办呢？我直接说：你别接受比我小的了，或者比我小的全部废掉。这就是用更高的proposal number的原因(proposal number更高的不一定就是加1，可以大很多，这个跟instance number是不同的)。

计算按照上述协议，很容易想到两个proposer不断地增加自己的proposal number，互相废掉对方，始终不能达成一致的情况。因此：

To guarantee progress, a distinguished proposer must be selected as the only one to try issuing proposals.

### 下面在proposal阶段捎带回v的机制，在raft中没有，是不是raft有啥问题？

then it sends an *accept* request to each of those acceptors for a proposal numbered *n* with a value *v*, where *v* is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals

### Choosen的含义？

A value is chosen when a single proposal with that value has been accepted by a majority of the acceptors. In that case, we say that the prop osal (as well as its value) has been chosen.

### Proposal number如何做到unique?

文中有：”assuming unique proposal numbers”

### 刚刚醒来的proposer如何知道下一个Instance id? 如何获取前面其没有参与的状态更新？

Make code中间有些内容，关于*prepare* message to the acceptors consisting of *< i*, *b >，*i就是instacnce id。

可以肯定，proposer或者leader proposer是知道哪些instance id 已经被choose，哪些正在进行中的。提到noop的那一节，就是为了获取。

在Ceph的实现中，一段时间内不仅leader固定，quorum成员也固定。新的成员要加入，则需要重新走bootstrap，包含同步和重新选举过程。注意在ceph中，大家互相信任别人的last\_committed，而不是互相怀疑，并且只允许一个未committed状态的。

在raft中的同步，则不是如此的。文档中的figure 7说明相关问题。

### Leader是个啥？

Leader Proposer，不是acceptor哦。

### 新当选的Leader，怎么能知道下一个instance id是什么？

The new leader, being a learner in all instances of the consensus algorithm, should know most of the commands that have already been chosen.

实际上是说，新当选的之前是个learner，不可能啥都不知道。即使有一部分instance没有完成phase 2，也有noop可以帮助搞定。

### 大家怎么知道某个value被choose了？

“**Learning a Chosen Value**”一节专门说这个，但是最后一句” If a learner needs to know whether a value has been chosen, it can have a proposer issue a proposal, using the algorithm described above.”，不知道指啥algorithm?

### Acceptor, proposer, learner是独立的吗？

我感觉它建议的实现是一体的，但是理论上没有说不能分开。

In its consensus algorithm, each process plays the role of proposer, acceptor, and learner

### 如何避免client的同一个命令，被多次执行？

由于一个leader可能在执行完成某个client命令后crash，没来得及通知client，那么client可能重试。重试可能导致同一命令被执行两次。因此，需要有方法避免。

这个方法来自raft的paper: The solution is for clients to assign unique serial numbers to every command. Then, the state machine tracks the latest serial number processed for each client, along with the associated response. If it receives a command whose serial number has already been executed, it responds immediately without re-executing the request.