可靠分布式系统基础 Paxos 算法

Author	Taosheng Shi
WeChat Contact	data-lake
Mail Contact	tshshi@126.com
Organization	NOKIA
Document category	Distributed System
Document location	https://github.com/stone-note/articles

Version	Status	Date	Author	Description of changes
0.1	Draft	12/7/2017	Taosheng Shi	Initiate
0.2	Draft	DD-MM-YYYY	YourNameHere	TypeYourCommentsHere
1.0	Approved	DD-MM-YYYY	YourNameHere	TypeYourCommentsHere

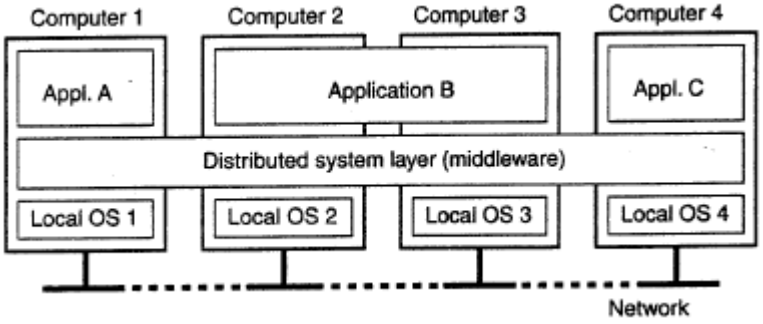
Contents

1 什么是分布式系统？

毋庸置疑，Internet 和 DNS 是两个典型的成功的分布式系统。那么，分布式系统是不是就是计算机网络？1990 年，Sun Microsystems 公司提出网络即是计算机（The network is the computer.），后来 google 提出数据中心即是计算机，现在有人提出云即是计算机。这些都试图从抽象的概念上总结分布式系统呈现的一致视图。《分布式系统：原理与范型》一书中对分布式系统的定义就是从用户的角度描述的：

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

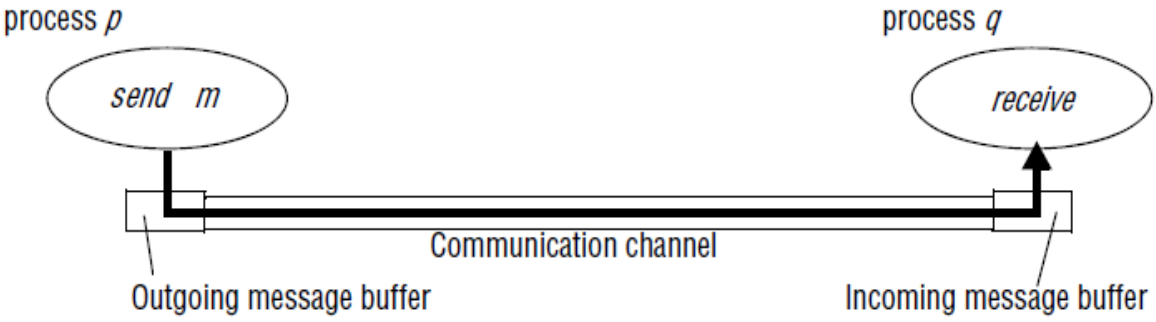
其强调的的结构是这样的：



《分布式系统：概念与设计》一书中给出的定义则更强调了消息传递的本质：

*A distributed system is one in which components located at networked computers communicate and coordinate their actions only**by passing messages.** This definition leads to the following especially significant characteristics of distributed systems:concurrency of components, lack of a global clock and independent failures ofcomponents。*

那么分布式系统一个基本模型将是这样的：

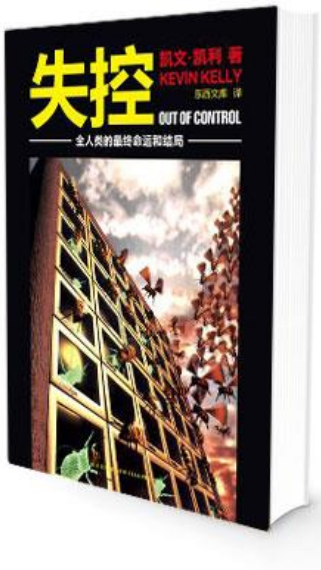


《分布式计算：原理，算法，系统》一书给出的定义则更接近自然界的本质：

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.

Distributed systems havebeen in existence since the start of the universe. From a school of fish to a flock of birds and entire ecosystems of microorganisms, there is communication among mobile intelligent agents in nature.

《失控：全人类的最终命运和结局》一书描述分布式系统已经超越了机器和软件的范围，着重描述人造和天生的混合以及人类社会，生物群落，自组织，涌现，超级有机体等概念。作者认为人类社会是由各种人和机器组成的大型分布式系统（第 22.6 节）。
书中有大量的新颖的观点，值得一读。



2 分布式系统一致性的需求

2.1 需求定义

1. Safety
    - Only a value that has been proposed may be chosen.
    - Only a single value is chosen.
    - A node never learns that a value has been chosen unless it actually has been.
2. Liveness
    - Some proposed value is eventually chosen.
    - If a value has been chosen, a node can eventually learn the value.

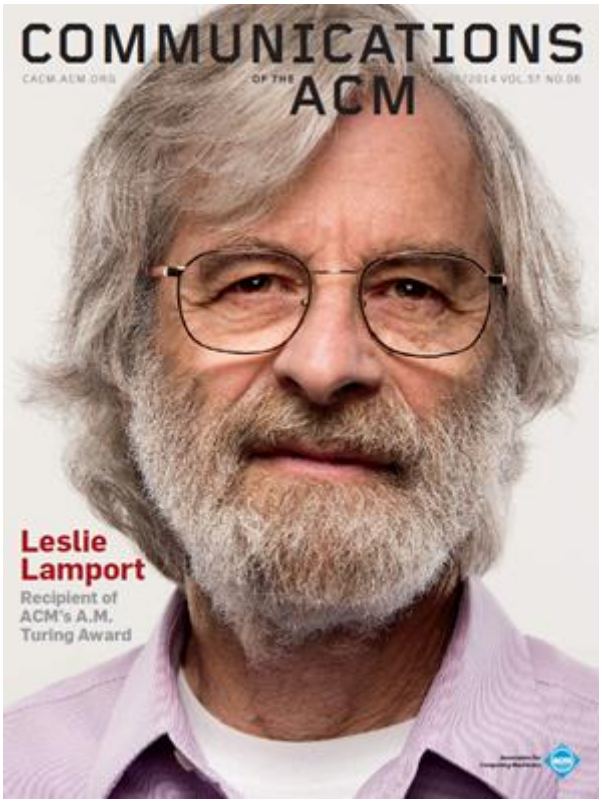
为什么安全性和活性是一致性算法的需求呢？因为一致性和活性是所有分布式算法正确性的的等价描述。LESLIE LAMPORT 在其 1977 年的论文《Proving the Correctness of Multiprocess Programs》中第一次描述了安全性和活性，后来成为构建分布式系统的标准。

Correctness == Safety and liveness

2.2 离开主题

在这里想稍微离开主题一下，是关于 Leslie Lamport。Leslie Lamport，has been awarded the 2013 Association for Computing Machinery A.M. Turing Award，for “imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems, in which several autonomous computers communicate with each other by passing messages”.他使分布式计算系统看起来混乱的行为变得清晰、定义明确且具有连贯性，在该系统下，多台自主计算机之间可以相互通信（参见 CSDN 报道）。个人认为：这是一个迟到的图灵奖，Leslie Lamport 的基础理论和算法大量用于现在互联网公司的分布式系统中（google,hadoop,yahoo!,amzon,facebook 等等），几乎所有涉及分布式一致性问题的系统都要引用他的 paxos 算法。Google 的论文 chubby 中写道：**Indeed, all working protocols for asynchronous consensus we have so far encountered have Paxos at their core.**迄今为止，未见其他计算机大牛在分布式系统领域的贡献比他多。ACM 的报道以《General Agreement》为题，明喻其分布式系统一致算法，暗语他的理论和算法已经成为分布式平台的基础设施，并最终得到世界的一致承认。他提出和解决了分布式系统大量基础问题，包括逻辑时钟（解决分布式系统时序问题，思想来源于爱因斯坦的相对论），安全性和活性（并发系统的正确性验证）问题，拜占庭将军问题（分布式系统在任意故障的环境下如何达成一致），paxos 算法

（解决分布式系统的一致性问题）等。 Leslie Lamport 目前工作在微软，是微软第五位获得图灵奖的科学家。Leslie Lamport 的文章都值得一读，参考他的 HOME PAGE。



2.3 隐喻

1. Safety
   Properties that state that nothing bad ever happens
2. Liveness
   Properties that state that something good eventually happens

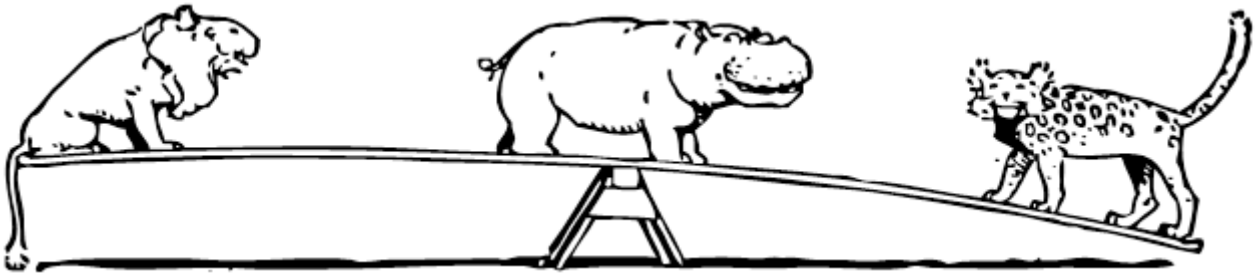
我查阅了最初的文章，未有耐心推敲枯燥的数学证明。在这里搜集一些有趣的例子分享一下：

1. 比如说你参加一门考试，那么在这个事件中你如何通过安全性和活性保证你的正确性呢？
   Safety : 如果你参加考试，不应该失败
   Liveness : 你应该最终通过考试
2. 交叉路口的红绿灯在汽车通过的时候如何保证正确性呢？
   Safety : 只有一个方向能有绿灯
   Liveness : 没有给方向最终都会有一个绿灯
3. 整数排序：
   Safety :（本例中也称为局部正确性）：如果算法终止，输出必须是排序的整数。
   Liveness :（本例中也称为终止性）：最终算法将会终止。例如，将会以输出作为响应。

一些有趣的描述：

1. Safety 是由无可挽回的事情刻画的。这个事情从来不会发生…
   Safety 的违反：是在有限的时间内发生；
   Safety 的满足：是在无限的时间内发生；
   如果一个系统什么也不做，那么它是满足安全性的。
2. Liveness 则是由没可能违反的事情刻画的。最终，当什么时候…
   Liveness 的满足：在有限的时间内结束；
   Liveness 的违反：在无限的时间内结束；
   如果一个系统什么都做，那么它是满足活性的。
3. 有 Liveness 的地方就有希望。
4. 如何判断一个属性是属于 Liveness 还是 Safety 呢？
   是不是有一个有限的时间点，通过一个系统的接口观察到属性 P 被违反了？
   如果答案是 YES，那么 P 就是 Safety。
   有时候，Liveness 是违反有限时间的，当系统进入死循环和死锁时，通过系统的接口是观察不到的。

Safety 的通俗理解是：没有坏的事情发生。同时出现两个方向的红灯或绿灯。如果车钥匙扭到点火的位置，车子必须启动。程序能够运行。钟表能够滴答。如果一个用户发出访问临界区的请求，那么必须被通过。Liveness 的通俗理解：美好的事情最终会发生。红灯最终会变绿。如果车钥匙扭到点火的位置，车子最终将会启动。程序最终将会终止。钟表会无限滴答。如果一个用户发出访问临界区的请求，那么最终请求会被通过。那么，Safety 就是保证没有坏的事情发生，而 Liveness 就是保证好的事情最终都会发生。每一个分布式系统都会不同程度上保证这两个属性，一些强一点，而另一些弱一点。这是一种平衡。比如说，原子操作就是最大程度的保证了 Safety，却丧失了 Liveness。原子操作在多个操作同时到达时不会相互干扰，但是在网络分区的情况下无能为力。而今天的分布式系统中广泛采用最终一致性。最终一致性是在一段时间之后，所有的参与者都会对同一个值达成一致。那意味着美好的事情最终会发生。但是，最终一致性无法保证 Safety，把最终一致性看作 Liveness（最终）和 Safety（一致性）的组合，那么我们可以提出这样的问题：系统中间状态的哪一个值最终被采纳？在所有参与者达成一致前，系统返回什么值呢？
看来，Safety 和 Liveness 有那么一点普适意义的感觉，这对我们的系统设计有什么指导意义呢？



3 分布式系统故障模型

3.1 分布式系统面对的主要问题

How to reach consensus/data consistency in distributed system that can tolerate non-malicious failures?

3.2 系统模型

分布式系统书籍中，首推《分布式系统：概念与设计》。所以我更倾向于依赖消息传递而实现的分布式系统。因为这是一种 share-nothing 的结构，而消息传递本身是无状态的（类似于 TCP 协议的状态绑定是协议的使用，而不是协议本身的性质）。share-nothing 便于线性扩展，无状态便于故障恢复（failover click click，failback，failsafe，failstop，failfast，failsilent）。对比一下 RedHat 的 GFS 和 NFS 协议，可以深入理解一下无状态的概念。
GFS 是基于共享存储的集群文件系统，而 NFS 准确来说只是一个兼容 posix 语义的文件共享系统。GFS 中各个 node 是全对称（peer to peer），但需要中心的锁服务器并发和协定，在文件数量很大和并发操作很多的情况下，特别 node 的故障和恢复的情况下，不出问题都很难。而 NFS 的实现是无状态的，且经过多年工业级检验，比较稳定。
NFS is a stateless protocol. This means that the file server stores no per-client information, and there are no NFS "connections". For example, NFS has no operation to open a file, since this would require the server to store state information (that a file is open; what its file descriptor is; the next byte to read; etc). Instead, NFS supports a Lookup procedure, which converts a filename into a file handle. This file handle is an unique, immutable identifier, usually an i-node number, or disk block address. NFS does have a Read procedure, but the client must specify a file handle and starting offset for every call to Read. Two identical calls to Read will yield the exact same results. If the client wants to read further in the file, it must call Read with a larger offset.
Issues of state：
One of the design decisions made when designing a network filesystem is

determining what part of the system will track the files that each client has open, information referred to generically as "state." A server that does not record the status of files and clients is said to be stateless; one that does is stateful. Both approaches have been used over the years, and both have benefits and drawbacks.
Stateful servers keep track of all open files across the network. This mode of operation introduces many layers of complexity (more than you might expect) and makes recovery in the event of a crash far more difficult. When the server returns from a hiatus, a negotiation between the client and server must occur to reconcile the last known state of the connection. Statefulness allows clients to maintain more control over files and facilitates the management of files that are opened in read/write mode.
On a stateless server, each request is independent of the requests that have preceded it. If either the server or the client crashes, nothing is lost in the process. Under this design, it is painless for servers to crash or reboot, since no context is maintained. However, it's impossible for the server to know which clients have opened files for writing, so the server cannot manage concurrency.
在这样的系统中，进程实体和通信信道都可能发生故障。

3.3 遗漏故障

3.3.1 进程遗漏故障

迄今为止，除了超时机制外，未见更加有效的进程故障判断方法。在异步系统中，超时只能表明没有响应，其原因肯那个是崩溃了，也可以是消息还没有到达。

3.3.2 通信遗漏故障

通信的故障发生在三个区域：发送方的缓冲区，通信信道和接收方的缓冲区。

3.4 拜占庭故障

这是可能出现的最坏的故障，此时可能发生任何类型错误，包括假冒的节点或者是被黑客攻陷的节点等。关于这个问题，Leslie Lamport 在《The Byzantine Generals Problem》第一次描述了这个问题。非正式的来说，是这样的：

- three or more generals are to agree to attack or to retreat.

- One, the commander, issues the order. The others, lieutenants to the commander, must decide whether to attack or retreat.
- But one or more of the generals may be ‘treacherous’ – that is, faulty.
- If the commander is treacherous, he proposes attacking to one general and retreating to another.
- If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.

Zookeeper 论文中说：To date, we have not observed faults in production that would have been prevented using a fully Byzantine fault-tolerant protocol.

4 分布式系统的复制技术

基于所有的分布式系统书籍都会提到复制技术以及它的两个需求：性能和高可用性。也就是说，通过使用复制技术，可以提供就近的数据访问和故障时的切换。当然，复制也是产生一致性问题的根源。

关于复制和一致性的探讨参考我另一篇拙文《CAP 理论与分布式系统设计》。

5 一致性问题概述

5.1 一致性问题演化

一致性问题源于 70 年代末的数据库系统：那个时候的目标是实现 distribution transparency—that is, to the user of the system it appears as if there is only one system instead of a number of collaborating systems.而采用的策略是：Many systems during this time took the approach that it was better to fail the complete system than to break this transparency。
90 年代中期，互联网开始发展。此时人们认为可用性比一致性更重要，但也在思考是否应该在一致性和可用性之间平衡。
2000 年，在 Principles of Distributed Computing(PODC)会议上，Eric Brewer 在其 Towards Robust Distributed Systems 论文中提出来 CAP 理论，阐述了 data consistency, system availability, and tolerance to network partition 分布式系统中的三个属性在任何时候都只能实现两个。两年后，猜想得到证明 Seth Gilbert and Nancy Lynch：Gilbert, S. and

Lynch, N. Brewer’ s conjecture and the feasibility of consistent, available, partition-tolerant Web services. ACM SIGACT News 33, 2 (2002).其含义是一个不允许分区的系统可以通过事务协议实现数据一致性和可用性。这就要求客户机和服务器在同一个环境中，并且故障对于两者是整体性的，对于客户机就不会出现分区。但是在更大的分布式系统中，分区必然会出现，一致性和可用性就不能同时保证。这样就只有两个选择：放松一致性获得可用性和保证一致性牺牲可用性。系统的使用者（例如开发人员）需要知悉系统的行为。如果系统强调了一致性，开发者将会面临系统不可用的情况，例如无法写入，此时开发者需要处理写失败的情况（暂存本地还是一直挂起？）。如果系统强调了可用性，开发者要明确读取数据不一定是最新的，有大量的应用采用这样的模型，允许少量的陈旧数据而工作的很好。
在事务处理领域，一致性被定义为：ACID properties (atomicity, consistenconsistency,isolation, durability) 可以认为是一致性的另外一种类型或描述。
In ACID, consistency relates to the guarantee that when a transaction is finished the database is in a consistent state;
In ACID-based systems, this kind of consistency is often the responsibility of the developer writing the transaction but can be assisted by the database managing integrity constraints.

5.2 一致性：客户端和服务器

我们可以从客户端和服务器两个角度来看一致性问题。从客户端的角度也是从用户的角度，就是说你如何看待数据的更新？对于用户来说，服务器就是一个黑盒，但是必须提供数据的持久性和可用性。客户端的一致性特别强调用户多个无关的进程或者线程如何看待存储系统中数据对象，具体来说就是当一个更新操作完成之后，对于后续的进程有什么影响。比如：

- 强一致性：当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。
- 弱一致性：系统并不保证续进程或者线程的访问都会返回最新的更新过的值。在系统返回之前需要满足一定的条件，更新操作完成之后到访问返回最新值之间的时间窗口称为不一致窗口。
- 最终一致性：弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。DNS 是一个典型的最终一致性系统。
- 最终一致性模型的变种：
    - 因果一致性：如果 A 进程在更新之后向 B 进程通知更新的完成，那么 B 的访问操作将会返回更新的值。如果没有因果关系的 C 进程将会遵循最终一致性的规则。

    - 读己所写一致性：因果一致性的特定形式。一个进程总可以读到自己更新的数据。
    - 会话一致性：读己所写一致性的特定形式。进程在访问存储系统同一个会话内，系统保证该进程读己之所写。
- 单调读一致性：如果一个进程已经读取到一个特定值，那么该进程不会读取到该值以前的任何值。
- 单调写一致性：系统保证对同一个进程的写操作串行化。

我们熟悉的一致性保证是强一致性，因果一致性，单调读一致性，单调一致性。
从服务器的角度是从实现的角度，即更新的数据流是如何流过系统？系统为这些更新提供了哪些保证？

6 Paxos 算法

6.1 Paxos 算法中的角色

Classes of agents:

- Proposers
- Acceptors
- Learners

A node can act as more than one clients (usually 3).

6.2 算法要点

6.2.1 Phase 1 (prepare)

A proposer selects a proposal number n and sends a prepare request with number n to majority of acceptors.
If an acceptor receives a prepare request with number n greater than that of any prepare request it saw, it responses YES to that request with a promise not to accept any more proposals numbered less than n and include the highest-numbered proposal (if any) that it has accepted.

6.2.2 Phase 2 (accept)

If the proposer receives a response YES to its prepare requests from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a values v which is the value of the highest-numbered proposal among the responses.
If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

6.2.3 达到一致性的标准

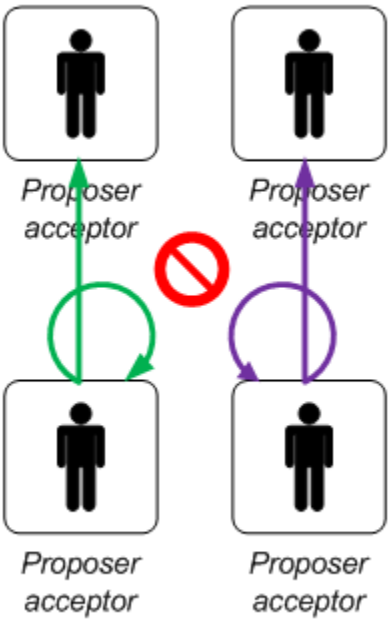
A value is chosen at proposal number n iff majority of acceptor accept that value in phase 2 of the proposal number.

6.3 算法分析

6.3.1 P1

An acceptor must accept the first proposal that it receives.

1. 首先要明确的是，必须有多个 acceptor，如果系统内只有一个 acceptor，单点故障，系统的状态将彻底丢失；
2. 算法核心，多数原则：如果采用多个 acceptors，“真理”就掌握在多数人手中。
3. P1 阶段蕴含的意思很明显：acceptor 必须接受他收到的的第一个 proposal（这样保证系统内只有一个 proposal（只提出了一个提案）的情况，这也是为什么作者引入 P1 阶段的原因）
4. P1 阶段存在的问题：多个 proposers 同时提出不同的 proposals，每一个 acceptor 都接受一个 proposal，此时也不能达成一致。情形如下：



这样的结果是 2 对于 2，平手。问题在于既要满足 P1 又要满足多数原则，则必然要求一个 acceptor 能够接受多个 proposal。所以需要 P2 阶段。

5. 开始 P2 阶段之前，需要说明一个 proposal 是偏序的，也就是说 Proposals = same value + different(may be) proposal number，因为不同的 proposals 有不同的 numbers，所谓偏序就是 value 不变，number 是全序。这样做是为了保证一个 acceptor 能够接受多个 proposal 时，提案是唯一的（其值是代表一个提案的）。Number 具体意义取决于实现，可以是权重，也可以是时间+权重。全序的 number 是保证算法 safety property 的根本。一个 proposal 表示如下：

proposal number	Value（提案）

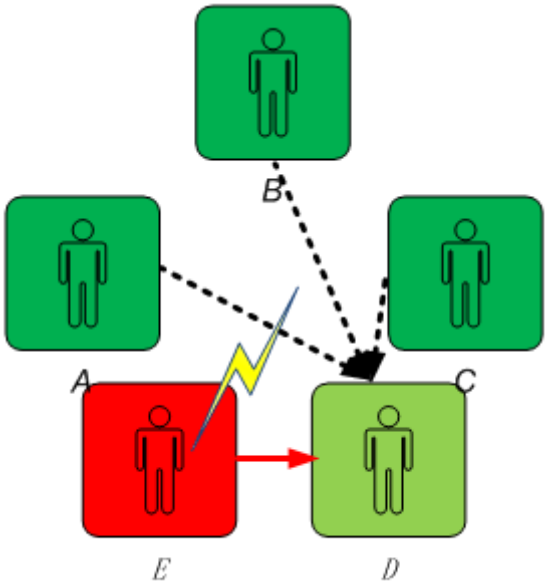
6.3.2 P2

If a proposal with value v is chosen, then every higher-numbered proposal,that is chosen has value v.

有了全序的 number，就能够保证 P2。如果一致已经达成 v，后续能够被接受的提案必须都有 v。一个 proposal 必须至少被一个 acceptor 接受，所以 **P2*a*** 可以满足 **P2**。

> P2a . If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v.

P2a 处理不了下面的情况：ABC 已经达成一致，C 没有收到任何的 proposal，由于消息的异步，一致的 proposal 还没有通知到它。然后一个曾经断链或者关机的节点加入系统，并提出一个更 higher-numbered 带有新 value 的 proposal 给 D，根据 P1，D 必须接受该 proposal，与 **P2*a*** 矛盾。所以用 **P2*b*** 对 **P2*a*** 加强。



> P2b . If a proposal with value v is chosen, then every higher-numbered proposalissued by any proposer has value v.

其实 P2a 是从 acceptor 端控制不接受不是已经达成一致的那个 v，其实 P2b 是从 proposer 端控制不接受不是已经达成一致的那个 v。
P2b 不易实现，所以又有了 P2c。

> P2c. 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 thatEither (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal amongAll proposals numbered less than n accepted by the acceptors in S.

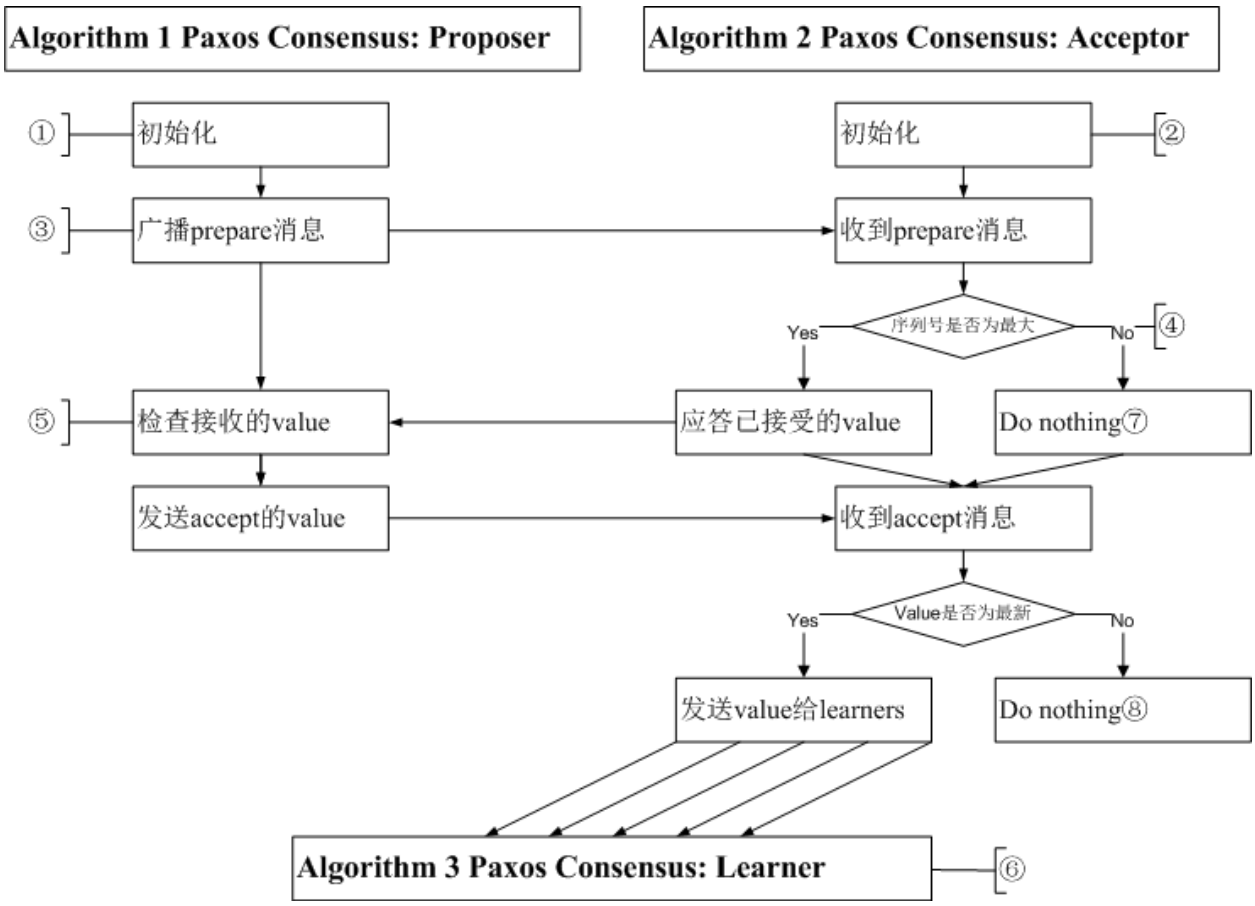
多数派要么没有接受过，要么当前是 the highest-numbered proposal。关键是偏序的 number，每次都要去拿号。去遍询当前的状态。

6.4 算法的本质

论文《Enhanced Paxos Commit for Transactions on DHTs》讨论了 paxos 算法的一些本质含义，对于工程实现有很好的借鉴意义。比如多个角色的重合，固定 acceptors 数量（不能在运行时增加），acceptors 作为分布式内存，一致性的达成是由 learner 来宣布等。

- Each process may take the role of a proposer, an acceptor, or a learner, or any combination thereof.
- A proposer attempts to get a consensus on a value.This value is either its own proposal or the resulting value of a previously achieved consensus.
- The acceptors altogether act as a collective memory on the consensus status achieved so far. The number of acceptors must be known in advance and must not increase during runtime, as it defines the size of the majority set m required to be able to achieve consensus.
- The decision, whether a consensus is reached, is announced by a learner.
- Proposers trigger the protocol by initiating a new round.Acceptors react on requests from proposers. By holding the current state of accepted proposals, the acceptors collectively provide a distributed, fault-tolerant memory for the consensus.In essence, a majority of acceptors together ' know' whether an agreement is already achieved, while the proposers are necessary to trigger the consensus process and to ' read' the distributed memory.
- Each round is marked by a distinct round number r. Round numbers are used as a mean of decentralized tokens. The protocol does not limit the number of concurrent proposers:There may be multiple proposers at the same time with different round numbers r. The proposer with the highest r holds the token for achieving consensus. Only messages with the highest round number ever seen by each acceptor, will be processed by that acceptor. All others will be ignored. If at any round, a majority of the acceptors accepted a proposal with value v, it will again be chosen by all subsequent rounds. This ensures the validity and integrity properties.
- The algorithm can be split into two phases: (1) an information gathering phase to check whether there was already an agreement in previous rounds,and (2) a consolidation phase to distribute the consensus to a majority of acceptors and thereby to agree on the decision. In the best case, consensus may be achieved in a single round. In the worst case, the decision may be arbitrarily long delayed by interleaving proposers with successively increasing round numbers (token stealing by each other).

6.5 算法实现



①proposer 的初始化：选择任意一个目前为止最大的整数设置为序列号 r，开始的时候，任意的的序列号都是可以的，唯一的限制是必须是在多个 proposers 中保证唯一（一种可行的方法是使用 proposer 的标识作为一部分）。如果恰好一个序列号比前一个序列号小，那么该序列号对应的 propose 会被认为是过时的而被忽略。

②acceptors 的初始化：响应的序列号 r = 0;已接受的序列号 r = 0;选择的 vaule 为 NULL。

③proposer 发送带有序列号 r 的 prepare(r)消息，广播给所有 acceptors，然后开始超时计时。如果在超时值时间内没有收到大多数 acceptors 的应答消息，就是用更大的序列号和一个增大的超时值重新开始新一轮的 prepare。

④如果 acceptors 收到的序列号是迄今为止最大的序列号，那么保存序列号并发送确认消息。

⑤proposer 检查从 acceptors 那里获得的最新的 vaule，如果仍旧为 NULL，则 proposer 自己 propose 一个值；如果不为 NULL，则接受最新的 vaule。

⑥如果 learner 从多数 acceptors 那里获得 valure，一致达成于 value。

⑦对于过时的序列号，为了帮助 proposer 更快的（使用更大的序列号）发起新一轮的 propose，acceptors 用已经接受的序列号来响应 proposer。

⑧如果 proposer 接受的 value 所对应的序列号过时，则 acceptors 发送拒绝的消息给 proposer。

6.6 异常处理

- proposer 的崩溃：会有新的进程替代 proposer 的角色，新的 proposer 需要首先需要从 acceptors 那里获得已经达成的一致信息（如果有的话）。
- acceptors 角色：acceptors 并不知道一致性状态是否达成，它们必须永远运行，时刻准备着接收新的 propose 消息，如果有更高序列号的 propose 到达，acceptors 需要接受和存储新的值。一个改进是可以在一致性状态达成之后，acceptors 的角色或者进程终止。
- acceptors 在接受一个 value 之后，广播给所有的 learners 的消息量过大：只广播给 learners 一个子集。
- 如果所有的 learners 都没有接收到 value：可以让该 node 重新发一起一次 propose 过程（learner 可以和 proposer 为同一个 node）。
- 理想的情况下，一轮的通信就可以达成一致。最坏的情况是出现活锁：两个 proposer 的 propose 互相不断的增大序列号，不断的 propose。这样需要选举出来一个 proposer 作为 leader,这是不是成了鸡生蛋的问题了？