什么是分布式系统

> 一个分布式系统是一些独立的计算机集合，但是对这个系统的用户来说，系统就像一台计算机一样。

分布式系统是一个硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统。简单来说就是一群独立计算机集合共同对外提供服务，但是对于系统的用户来说，就像是一台计算机在提供服务一样。分布式意味着可以采用更多的普通计算机（相对于昂贵的大型机）组成分布式集群对外提供服务。计算机越多，CPU、内存、存储资源等也就越多，能够处理的并发访问量也就越大。从分布式系统的概念中我们知道，各个主机之间通信和协调主要通过网络进行，所以分布式系统中的计算机在空间上几乎没有任何限制，这些计算机可能被放在不同的机柜上，也可能被部署在不同的机房中，还可能在不同的城市中，对于大型的网站甚至可能分布在不同的国家和地区。

分布式系统的特点

> 无论空间上如何分布，一个标准的分布式系统应该具有以下几个主要特征

分布性

组成分布式系统的多台计算单元在空间上可以随意分布和挪动

透明性

分布式系统可以隐藏其内部结构和实现细节，使用户感觉系统就像一个整体，这种透明性包括访问透明、位置透明、并发透明、复制透明等。

异构性

分布式系统中的计算机可能具有不同的硬件、操作系统和编程语言等异构特征。

对等性

分布式系统的计算单元都是对等的。

并发性

在一个计算机网络中，程序运行过程的并发性操作是非常常见的行为。例如同一个分布式系统中的多个节点，可能会并发地操作一些共享的资源，如何准确并高效地协调分布式并发操作也成为了分布式系统架构与设计中最大的挑战之一。

分布式系统面临的问题

1.缺乏全局时钟

在分布式系统中，很难定义两个事件究竟谁先谁后，原因就是因为分布式系统缺乏一个全局的时钟序列控制。

2.机器宕机

机器宕机是最常见的异常之一。在大型集群中每日宕机发生的概率为千分之一左右，在实践中，一台宕机的机器恢复的时间通常认为是24小时，一般需要人工介入重启机器。

3.网络异常

消息丢失，两片节点之间彼此完全无法通信，即出现了“网络分化”；消息乱序，有一定的概率不是按照发送时的顺序依次到达目的节点

4.分布式三态

如果某个节点向另一个节点发起RPC(Remote procedure call)调用，即某个节点A 向另一个节点B 发送一个消息，节点B 根据收到的消息内容完成某些操作，并将操作的结果通过另一个消息返回给节点A，那么这个RPC 执行的结果有三种状态：“成功”、“失败”、“超时（未知）”，称之为分布式系统的三态。

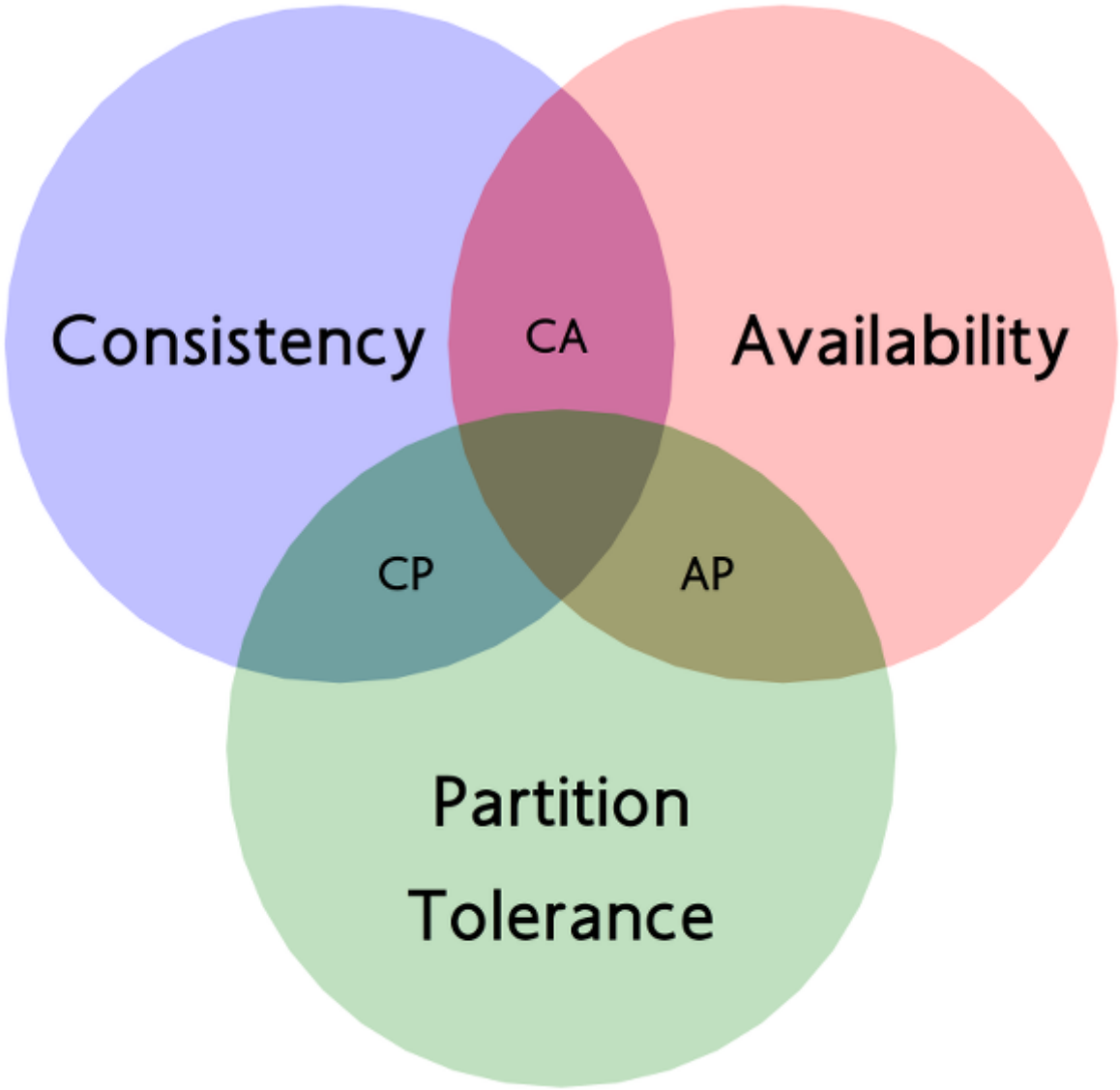
分布式理论基础

CAP

CAP 理论可以表述为，一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容忍性（Partition Tolerance）这三项中的两项。

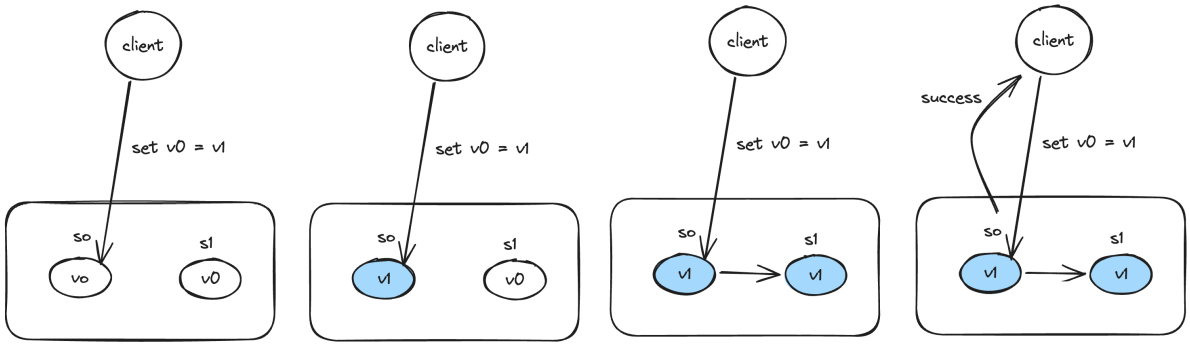
- 一致性是指“所有节点同时看到相同的数据”，即更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致，等同于所有节点拥有数据的最新版本。
- 可用性是指“任何时候，读写都是成功的”，即服务一直可用，而且是正常响应时间。我们平时会看到一些 IT 公司的对外宣传，比如系统稳定性已经做到 3 个 9、4 个 9，即 99.9%、99.99%，这里的 N 个 9 就是对可用性的一个描述，叫做 SLA，即服务水平协议。比如我们说月度 99.95% 的 SLA，则意味着每个月服务出现故障的时间只能占总时间的 0.05%，如果这个月是 30 天，那么就是 21.6 分钟。
- 分区容忍性具体是指“当部分节点出现消息丢失或者分区故障的时候，分布式系统仍然能够继续运行”，即系统容忍网络出现分区，并且在遇到某节点或网络分区之间网络不可达的情况下，仍然能够对外提供满足一致性和可用性的服务。

> 在分布式系统中，由于系统的各层拆分，P 是确定的，CAP 的应用模型就是 CP 架构和 AP 架构。分布式系统所关注的，就是在 Partition Tolerance 的前提下，如何实现更好的 A 和更稳定的 C。

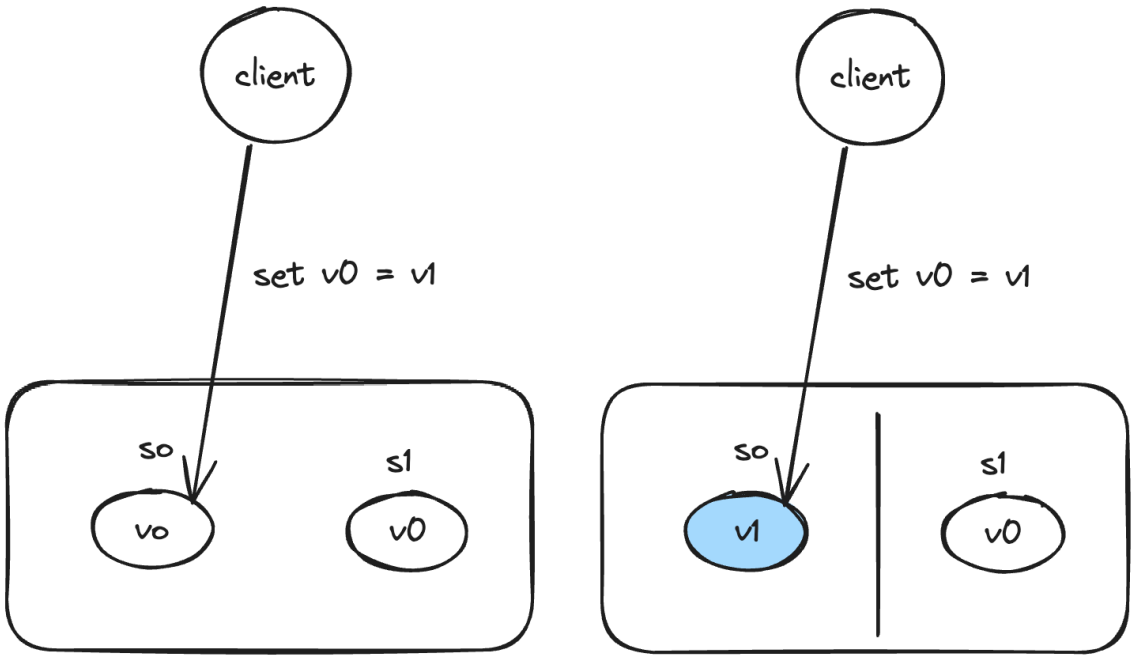


CAP 的证明

一般是采用反证法来证明 CAP



在上图中有2个节点 s0 和 s1 提供服务。当前节点的值都为V0。然后客户端设置值为v1,当前执行更新操作的是s0节点，它更新完自己的存储后
还需要同步去更新s1的值，更新完成后返回给客户端成功。这样客户端后续访问任意节点的时候都可以获取到最新的值v1。



假设s0和s1之前发生了分区故障，这个时候如果s0返回客户端成功。可用性保证了但是一致性保证不了，因为下次客户端从s1读取的是v0。
假设s0要保证一致性那么就不能返回成功给客户端，这样可用性就保证不了了。

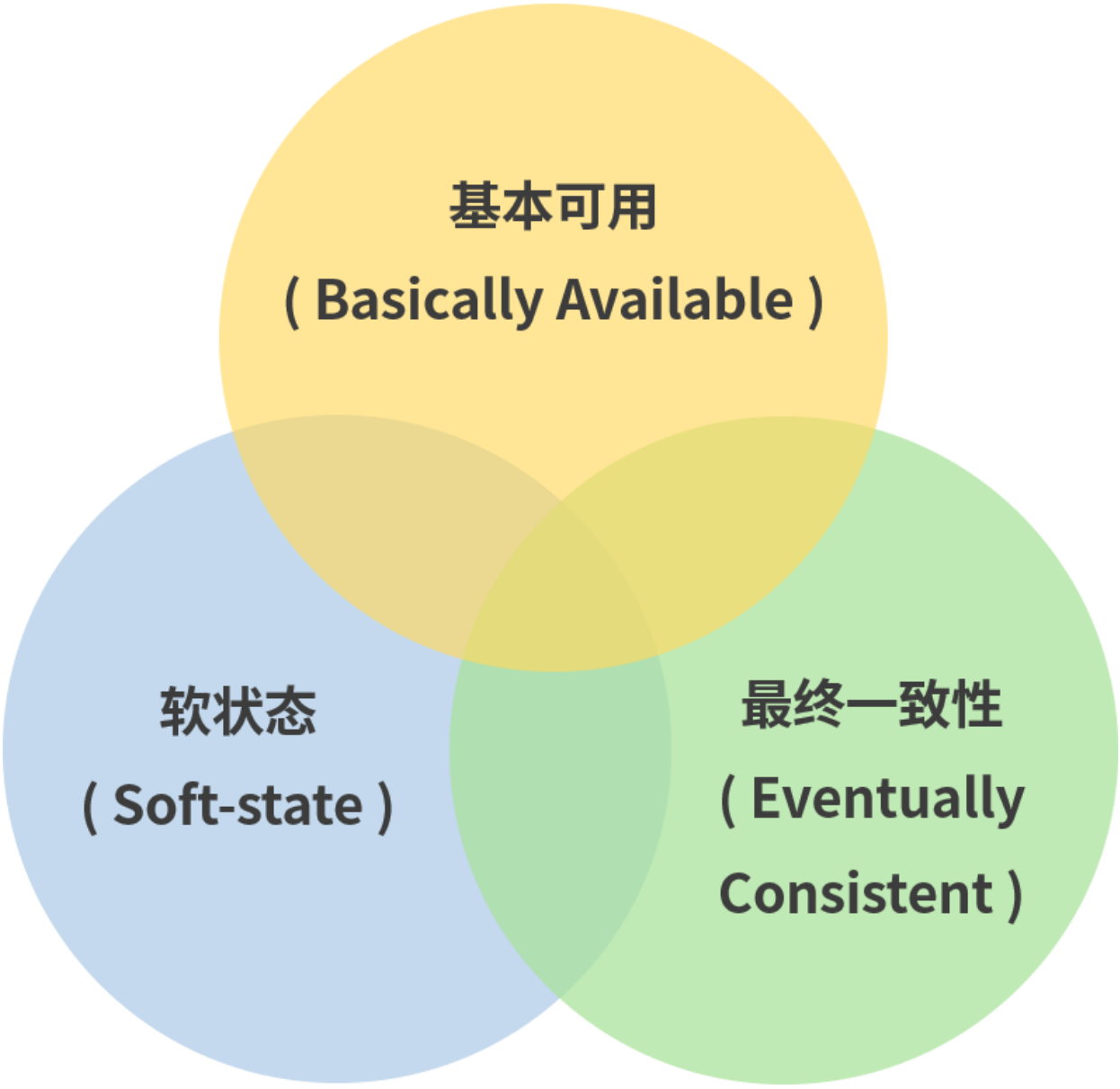
ACID 和 BASE

什么是 ACID

ACID是传统关系型数据库事务的四个特性，其中的四个字母分别代表：Atomicity, Consistency, Isolation, Durability
原子性(Atomicity): 指所有在事务中的操作要么都成功，要么都不成功，所有的操作都不可分割，没有中间状态。一旦某一步执行失败，就会全部回滚到初始状态。
一致性(Consistency): 在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。这表示

写入的资料必须完全符合所有的预设约束、触发器、级联回滚等
隔离性(Isolation)：多个事务并发执行时，每个事务都应该感觉不到其他事务的存在，每个事务的操作应该与其他事务的操作相互隔离，避免数据的冲突。数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括未提交读（Read uncommitted）、提交读（read committed）、可重复读（repeatable read）和串行化（Serializable）。
持久性(Durability)：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失
具有ACID特性的数据库系统，可以保证在写入或更新数据时，事务是正确可靠的。ACID的目标是保证数据的正确性和一致性。

什么是 BASE



核心思想是即使无法做到强一致性（Strong Consistency，CAP的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写。
BASE理论是对CAP中一致性和可用性权衡的结果，是对大规模互联网系统分布式实践的总结。

Basically Available(基本可用)
Basically Available 是指系统出现不可预知的严重故障(如服务器宕机，部分网络中断)时，允许损失部分可用性，即保证核心可用。

Soft state(软状态/柔性事务)
Soft State 是指允许系统存在中间状态，而该中间状态不会影响系统的整体可用性。Soft State 允许系统在多个不同的组件之间存在不一致。

Eventual Consistency(最终一致性)
Eventually Consistency 是指系统经过一定时间后，最终能够达到一致的状态。Soft state只是一个临时状态，对一个系统而来，如果一直是Soft state，那么就是一个错误。所以，对于支持Soft state的系统，仅应在一个时间期限是Soft state。在期限过后，应当保证系统的一致性。也即实现最终一致性。这个时间期限取决于网络延时、系统负载、数据复制、方案设计等因素。

ACID 和 BASE 的区别是什么

1.定义与来源不同

ACID是传统的数据库事务属性，包括原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)和持久性(Durability)。它主要用于关系型数据库。而BASE来源于分布式应用，代表Basically AvAllable、Soft state、Eventually consistent，强调系统的可用性和最终一致性。

2.一致性原则不同

ACID注重实时一致性，确保事务执行后数据处于一致状态。BASE则强调最终一致性，允许系统中的数据短暂地不一致，但最终会达到一致状态。

3.可用性与容错性差异

ACID强调数据的准确性和完整性，可能牺牲了部分可用性。BASE更注重系统的可用性，即使在部分节点失败或网络问题的情况下也能提供服务

4.应用场景不同

ACID常用于对事务完整性和数据一致性要求高的场合，如银行、航空预订等。而BASE更适用于大型分布式系统，如社交网络、电商等，它们需要高可用性和伸缩性。

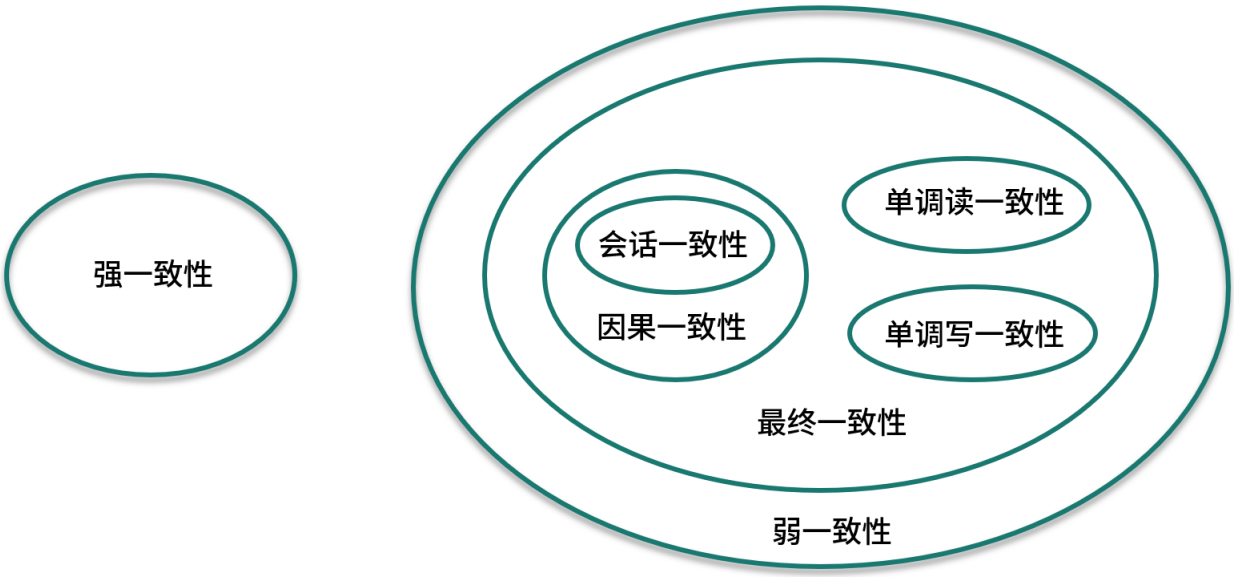
分布式系统一致性

全局时钟和逻辑时钟

分布式系统解决了传统单体架构的单点问题和性能容量问题，另一方面也带来了很多新的问题，其中一个问题就是多节点的时间同步问题：不同机器上的物理时钟难以同步，导致无法区分在分布式系统中多个节点的事件时序。 没有全局时钟，绝对的内部一致性是没有意义的，一般来说，我们讨论的一致性都是外部一致性，而外部一致性主要指的是多并发访问时更新过的数据如何获取的问题。 和全局时钟相对的，是逻辑时钟，逻辑时钟描绘了分布式系统中事件发生的时序，是为了区分现实中的物理时钟提出来的概念。

一般情况下我们提到的时间都是指物理时间，但实际上很多应用中，只要所有机器有相同的时间就够了，这个时间不一定要跟实际时间相同。更进一步解释：如果两个节点之间不进行交互，那么它们的时间甚至都不需要同步。 因此问题的关键点在于节点间的交互要在事件的发生顺序上达成一致，而不是对于时间达成一致。 逻辑时钟的概念也被用来解决分布式一致性问题。

一致性模型

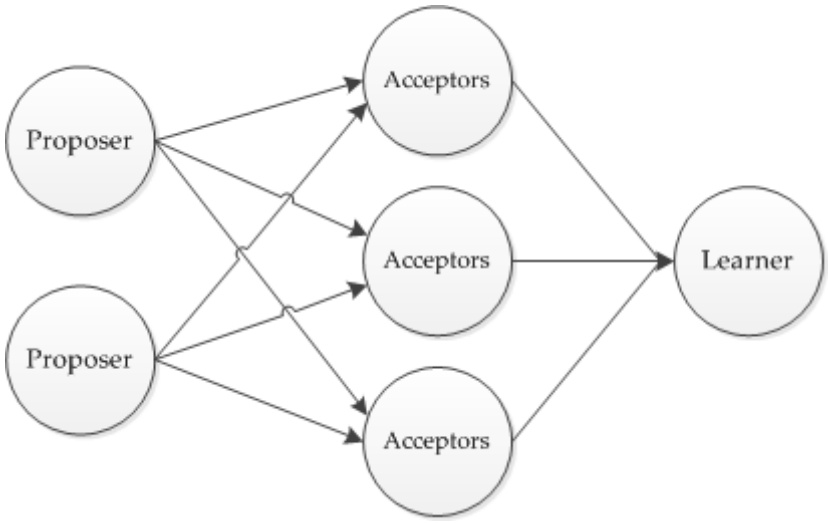


强一致性共识算法

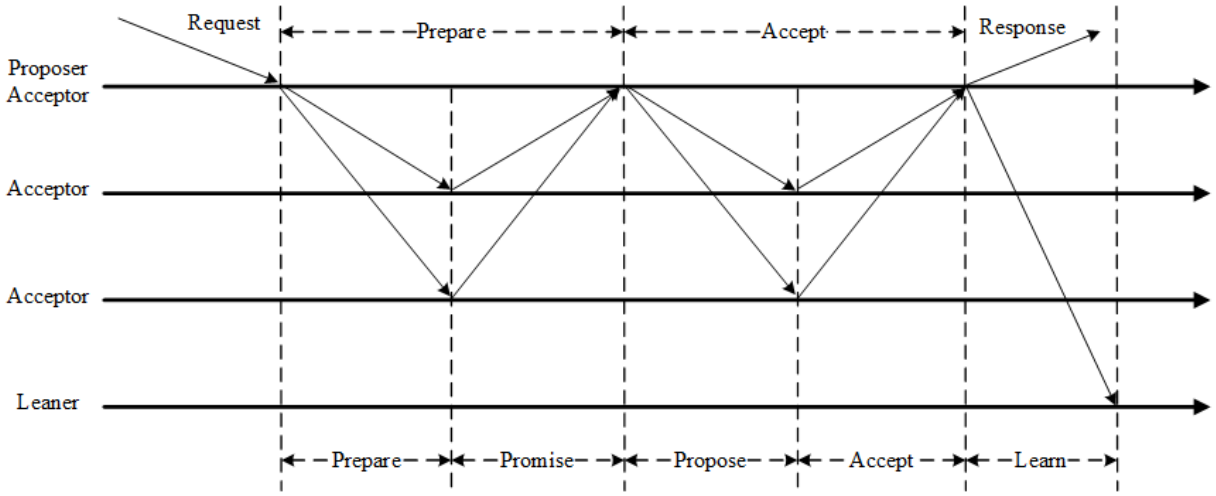
paxos

basic paxos

一个或多个提议进程 (Proposer) 可以发起提案 (Proposal)，Paxos算法使所有提案中的某一个提案，在所有进程中达成一致。系统中的多数派同时认可该提案，即达成了一致。最多只针对一个确定的提案达成一致。
Paxos将系统中的角色分为提议者 (Proposer)，决策者 (Acceptor)，和最终决策学习者 (Learner):
Proposer: 提出提案 (Proposal)。Proposal信息包括提案编号 (Proposal ID) 和提议的值 (Value)。
Acceptor：参与决策，回应Proposers的提案。收到Proposal后可以接受提案，若Proposal获得多数Acceptors的接受，则称该Proposal被批准。
Learner：不参与决策，从Proposers/Acceptors学习最新达成一致的提案（Value）。



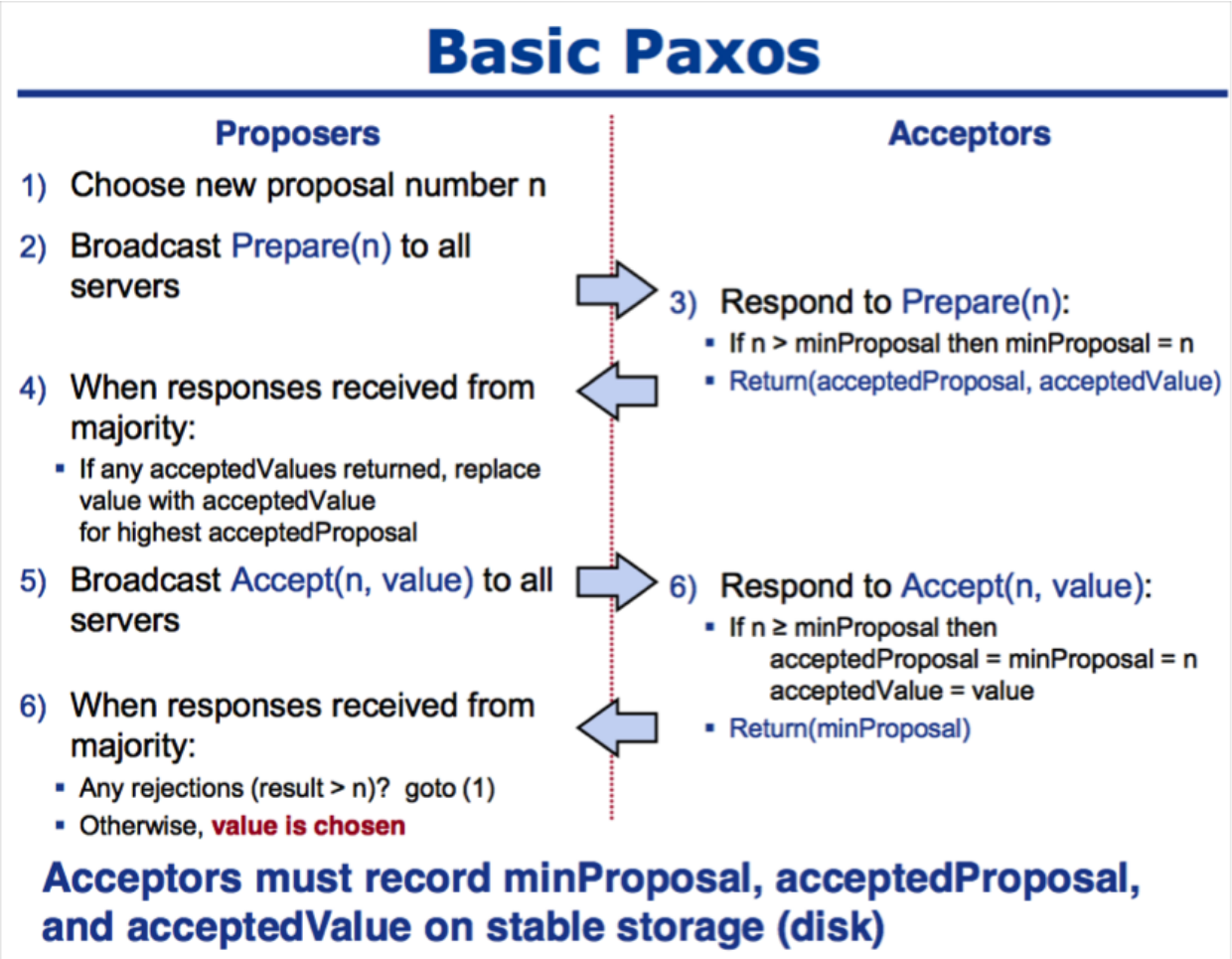
Paxos算法通过一个决议分为两个阶段（Learn阶段之前决议已经形成）：
第一阶段：Prepare阶段。Proposer向Acceptors发出Prepare请求，Acceptors针对收到的Prepare请求进行Promise承诺。
第二阶段：Accept阶段。Proposer收到多数Acceptors承诺的Promise后，向Acceptors发出Propose请求，Acceptors针对收到的Propose请求进行Accept处理。
第三阶段：Learn阶段。Proposer在收到多数Acceptors的Accept之后，标志着本次Accept成功，决议形成，将形成的决议发送给所有Learners。

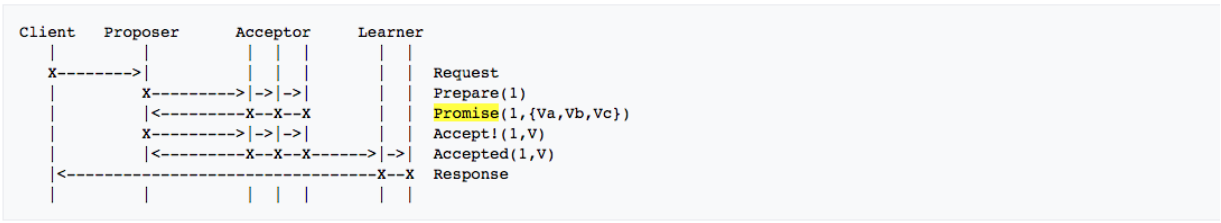


Paxos算法流程中的每条消息描述如下：
Prepare: Proposer生成全局唯一且递增的Proposal ID (可使用时间戳加Server ID)，向所有Acceptors发送Prepare请求，这里无需携带提案内容，只携带Proposal ID即可。
Promise: Acceptors收到Prepare请求后，做出“两个承诺，一个应答”。

> 两个承诺：
>
> 1. 不再接受Proposal ID小于等于（注意：这里是<= ）当前请求的Prepare请求。
> 2. 不再接受Proposal ID小于（注意：这里是< ）当前请求的Propose请求。
>
> 一个应答：
> 不违背以前作出的承诺下，回复已经Accept过的提案中Proposal ID最大的那个提案的Value和Proposal ID，没有则返回空值。
>
> Propose: Proposer 收到多数Acceptors的Promise应答后，从应答中选择Proposal ID最大的提案的Value，作为本次要发起的提案。如果所有应答的提案Value均为空值，则可以自己随意

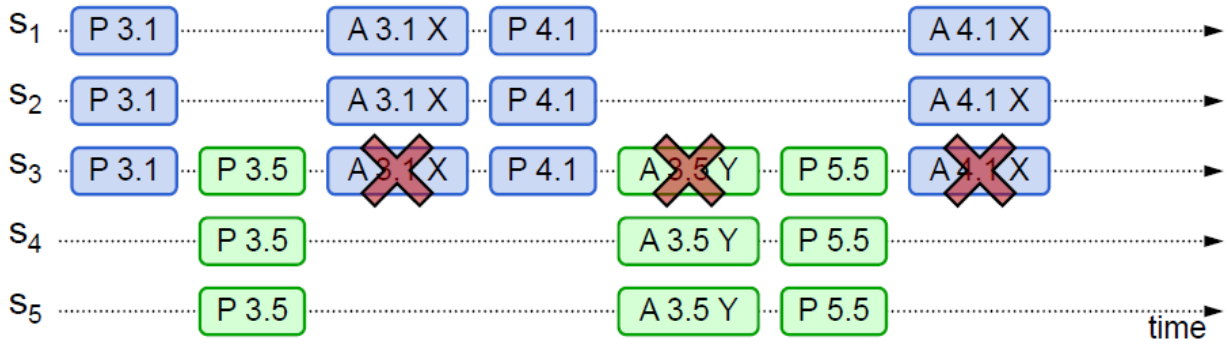
> 决定提案Value。然后携带当前Proposal ID，向所有Acceptors发送Propose请求。
> Accept: Acceptor收到Propose请求后，在不违背自己之前作出的承诺下，接受并持久化当前Proposal ID和提案Value。
> Learn: Proposer收到多数Acceptors的Accept后，决议形成，将形成的决议发送给所有Learners。





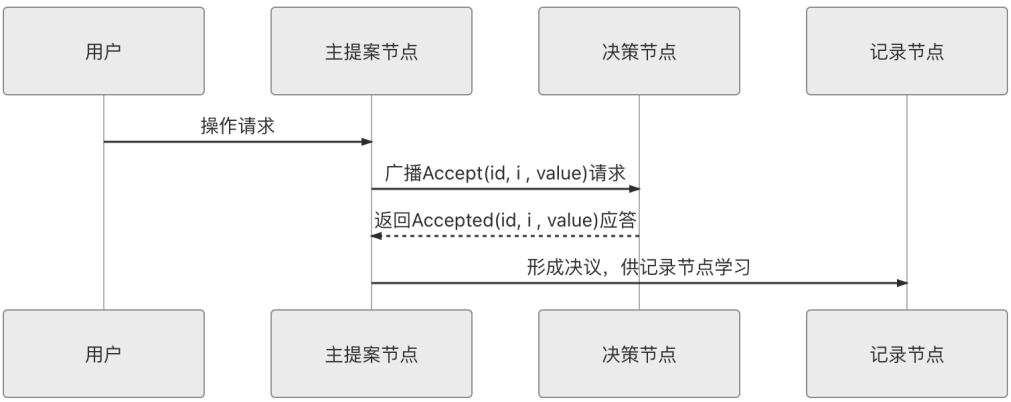
Here, V is the last of (Va, Vb, Vc).

paxos活锁 两个Proposers交替Prepare成功，而Accept失败，形成活锁（Livelock）。



multi Paxos

Multi Paxos 对 Basic Paxos 的核心改进是增加了“选主”的过程，提案节点会通过定时轮询（心跳），确定当前网络中的所有节点里是否存在有一个主提案节点，一旦没有发现主节点存在，节点就会在心跳超时后使用 Basic Paxos 中定义的准备、批准的两轮网络交互过程，向所有其他节点广播自己希望竞选主节点的请求，希望整个分布式系统对“由我作为主节点”这件事情协商达成一致共识，如果得到了决策节点中多数派的批准，便宣告竞选成功。当选主完成之后，除非主节点失联之后发起重新竞选，否则从此往后，就只有主节点本身才能够提出提案。此时，无论哪个提案节点接收到客户端的操作请求，都会将请求转发给主节点来完成提案，而主节点提案的时候，也就无需再次经过准备过程，因为可以视作是经过选举时的那一次准备之后，后续的提案都是对相同提案 ID 的一连串的批准过程。也可以通俗理解为选主过后，就不会再有其他节点与它竞争，相当于是处于无并发的环境当中进行的有序操作，所以此时系统中要对某个值达成一致，只需要进行一次批准的交互即可



raft

Raft 集群中每个节点都处于以下三种角色之一：

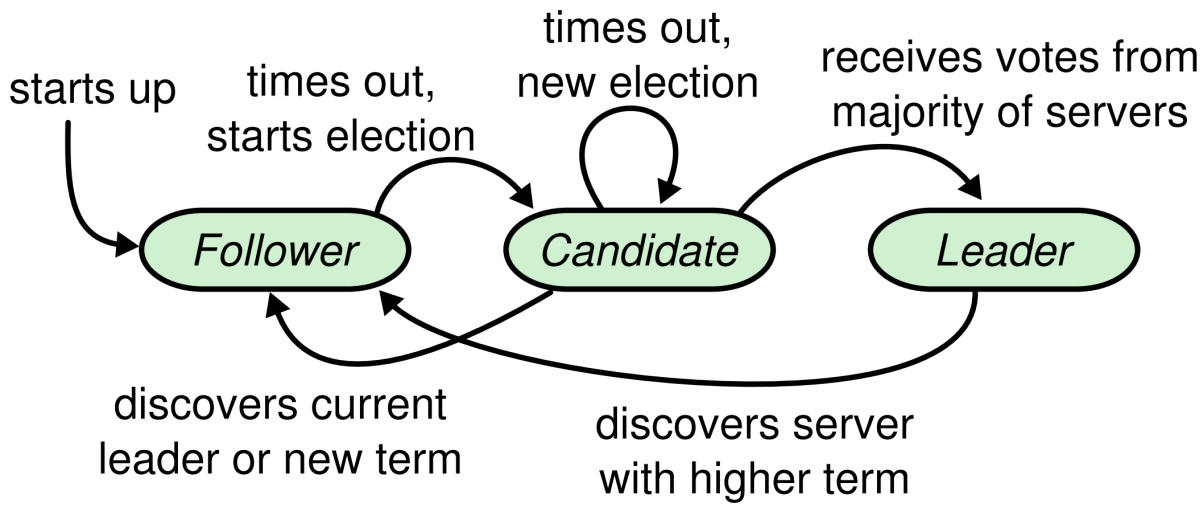
> Leader: 所有请求的处理者，接收客户端发起的操作请求，写入本地日志后同步至集群其它节点。
> Follower: 请求的被动更新者，从 leader 接收更新请求，写入本地文件。如果客户端的操作请求发送给了 follower，会首先由 follower 重定向给 leader。
> Candidate: 如果 follower 在一定时间内没有收到 leader 的心跳，则判断 leader 可能已经故障，此时启动 leader election 过程，本节点切换为 candidate 直到选主结束。

任期

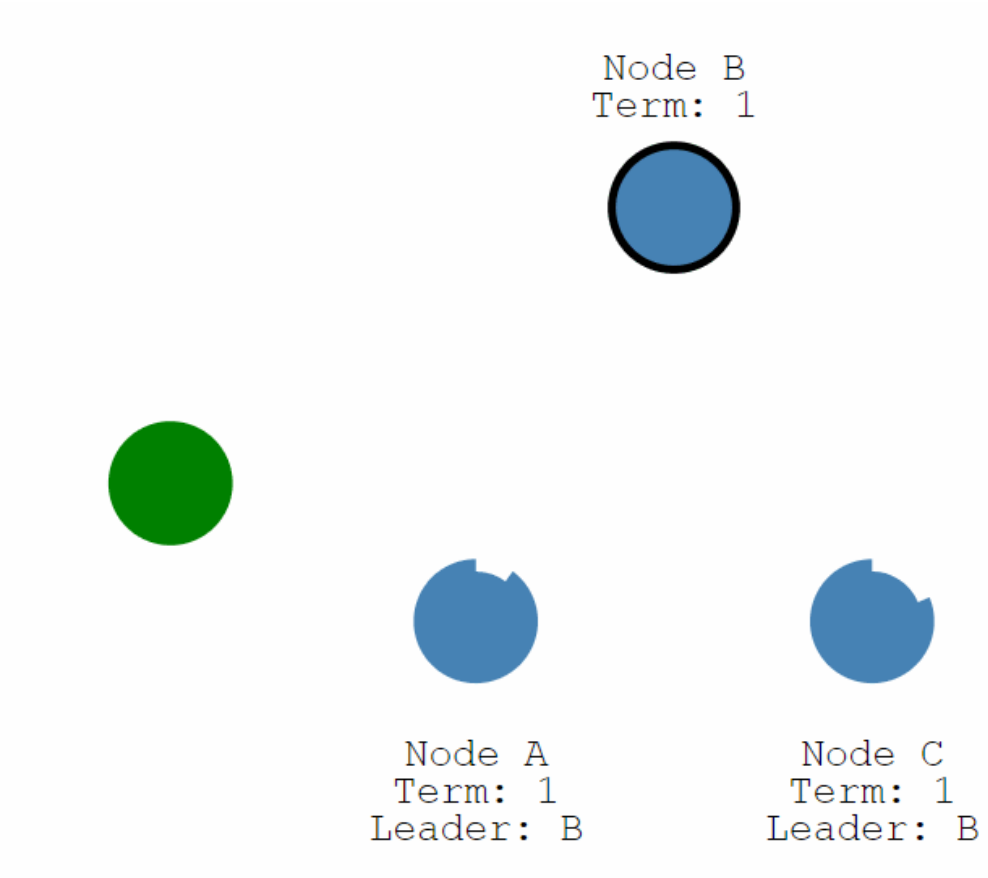
每开始一次新的选举，称为一个任期（term），每个 term 都有一个严格递增的整数与之关联。
每当 candidate 触发 leader election 时都会增加 term，如果一个 candidate 赢得选举，他将在本 term 中担任 leader 的角色。但并不是每个 term 都一定对应一个 leader，有时候某个 term 内会由于选举超时导致选不出 leader，这时 candicate 会递增 term 号并开始新一轮选举。
节点间通过 RPC 来通信，主要有两类 RPC 请求：

- RequestVote RPCs: 用于 candidate 拉票选举。
- AppendEntries RPCs: 用于 leader 向其它节点复制日志以及同步心跳。

leader选举



日志复制



zab

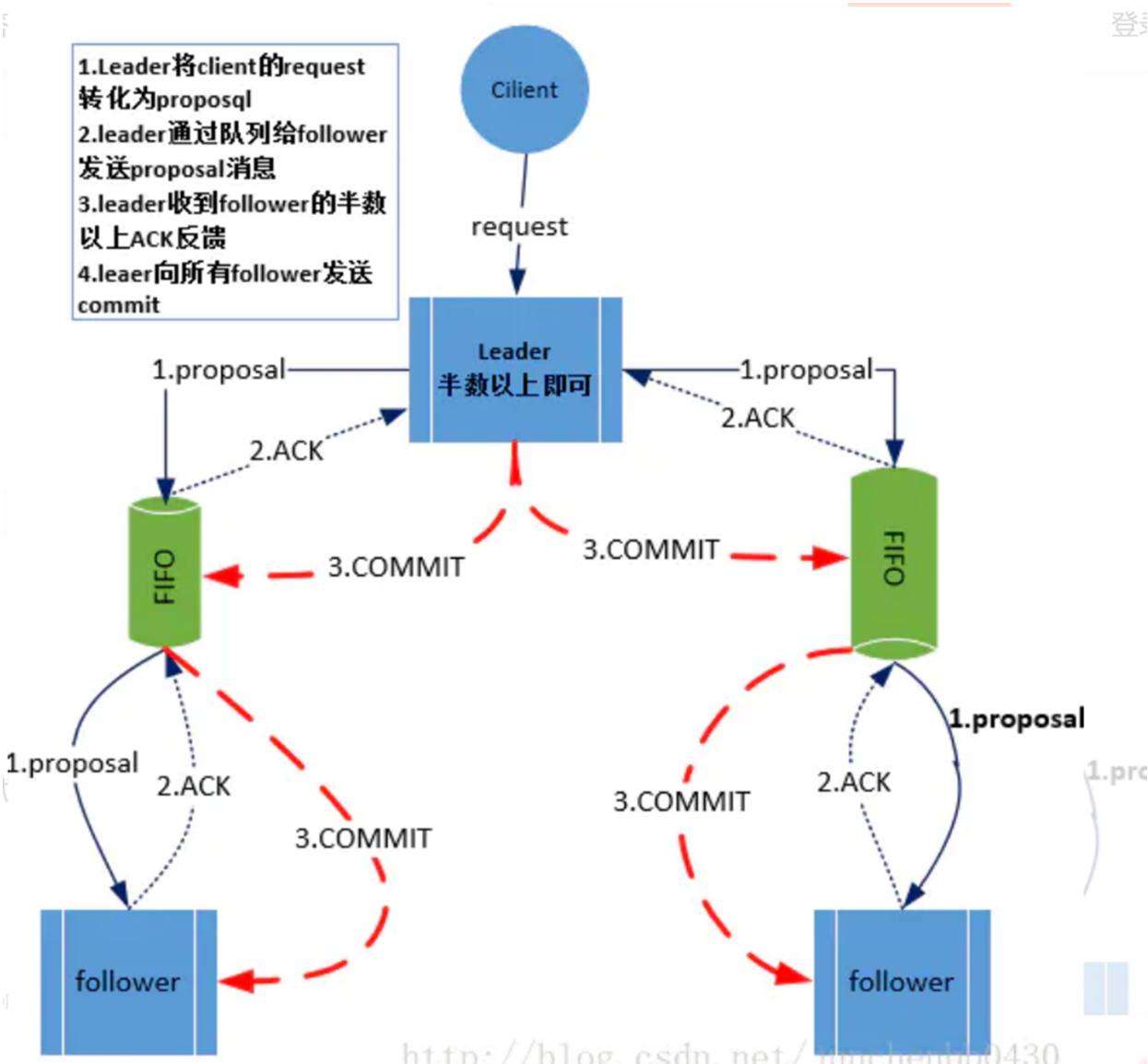
> Zab协议 的全称是 Zookeeper Atomic Broadcast （Zookeeper原子广播）。 Zookeeper 是通过 Zab 协议来保证分布式事务的最终一致性。
> ZAB协议中主要有三个角色，分别是:

Leader： 集群中唯一的写请求处理者
Follower：能够接收客户端的请求，如果是读请求则可以自己处理，如果是写请求则要转发给 Leader
Observer： 就是没有选举权和被选举权的 Follower

崩溃恢复

1、当leader出现问题，zab协议进入崩溃恢复模式，并且选举出新的leader。当新的leader选举出来以后，如果集群中已经有过半机器完成了leader服务器的状态同（数据同步），退出崩溃恢复，进入消息广播模式。
2、当新的机器加入到集群中的时候，如果已经存在leader服务器，那么新加入的服务器就会自觉进入崩溃恢复模式，找到leader进行数据同步。

消息广播



几种强一致算法的总结

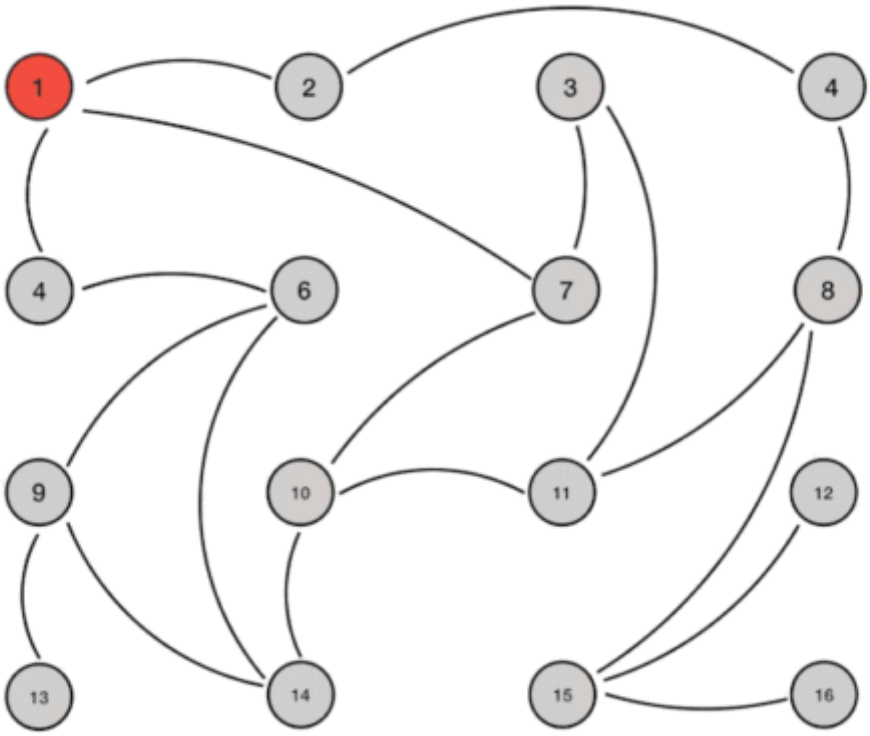
~	paxos	raft	zab
算法复杂度	高	低	中
选主	无选主	选主	选主

最终一致性

DNS
gossip

gossip

相比 Paxos、Raft 等算法，Gossip 的过程十分简单，它可以看作是以下两个步骤的简单循环：
如果有某一项信息需要在整个网络中所有节点中传播，那从信息源开始，选择一个固定的传播周期（譬如 1 秒），随机选择它相连接的 k 个节点（称为 Fan-Out）来传播消息。
每一个节点收到消息后，如果这个消息是它之前没有收到过的，将在下一个周期内，选择除了发送消息给它的那个节点外的其他相邻 k 个节点发送相同的消息，直到最终网络中所有节点都收到了消息，尽管这个过程需要一定时间，但是理论上最终网络的所有节点都会拥有相同的消息。



一致性迷思

1. 多副本的一致性
   指的的多副本之间数据是否是一致的
2. 一致性hash
   是一种解决数据分配的算法，提升的是系统的可用性和性能，并没有解决分布式系统中的一致性问题
3. CAP理论的一致性
   从客户端看来，分布式系统任意一个节点都可以提供相同的数据返回
4. ACID里的一致性
   更多的是事务前后符合数据库的约束，比如薪水字段是大于0的。不能某次事务后该字段出现了负数。

一致性 hash

> 一致性Hash算法并不是分布式一致性算法，它是一种用于解决分布式系统中数据分片问题的算法。在分布式系统中，数据通常被分散存储在不同的节点上，为了保证系统的可用性和性能，需要将数据分为多个分片，并将它们分布在不同的节点上。一致性Hash算法通过将每个节点映射到一个环上，将数据映射到环上的一个位置，然后将数据存储在该位置所对应的节点上，从而实现了数据的分片和分布式存储。虽然一致性Hash算法可以提高分布式系统的可用性和性能，但它并没有解决分布式系统中的一致性问题。

一致性Hash算法是个经典算法，Hash环的引入是为解决单调性(Monotonicity)的问题；虚拟节点的引入是为了解决平衡性(Balance)问题

分布式系统常见知识点

- 说说一致性哈希算法
- 知道cap和base是什么
- 为什么cap不能兼得
- raft协议过程是怎么样的

参考文档

分布式系统中的一致性模型，以及事务
技术文章摘抄
分布式理论笔记
Java全栈知识体系
凤凰架构
raft算法实现原理详解
谈谈分布式一致性机制
一致性协议
一致性算法视频
Raft算法和ZAB算法

Paxos算法详解
深入浅出paxos
Zab协议详解-分布式系统
深入浅出 Zookeeper 中的 ZAB 协议
分布式系统之ZAB协议
ZAB协议选主过程详解