云计算

中国科学院大学人工智能学院

唐海娜 hntang@ucas.ac.cn

2018年春季学期

提纲

- CAP和BASE理论
- Paxos
- Gossip

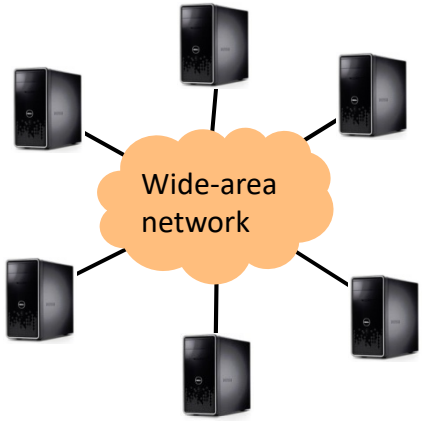
从集中式到分布式

MainFrame

Single-core machine

Multicore server

Cluster



Large-scale distributed system

More challenges

True concurrency

Network
Message passing
More failure modes (faulty nodes, ...)

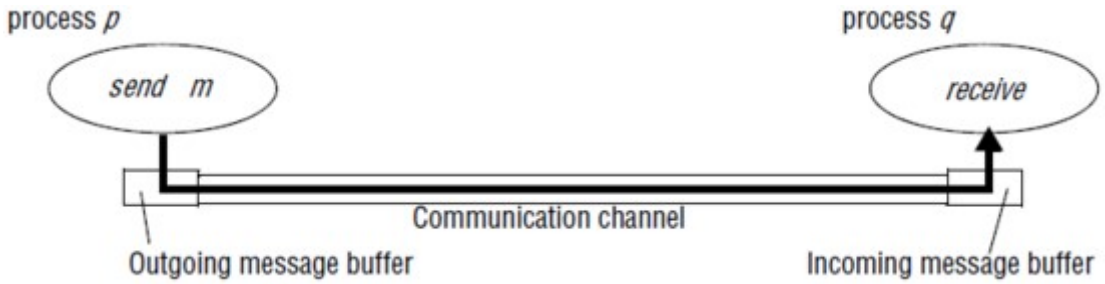
Wide-area network
Even more failure modes
Incentives, laws, ...

分布式系统

- 分布式系统是一个其硬件或软件组件分布在连网的计算机上，组件之间通过传递消息进行通信和动作协调的系统

  ……

- 分布式系统是若干独立计算机的集合，这些计算机对于用户来说就像是单个相关系统



分布式系统的特性

- 并发：在一个计算机网络中，执行并发程序是常见的行为。对共享资源并发访问的协调是重要问题
- 缺乏全局时钟：分布式系统通过消息传递来完成程序的协作，正确的协作往往取决于对发生时间的共识。网络上计算机与时钟同步所达到的准确性是有限的，没有一个正确时间的全局概念
- 故障独立性：计算机系统、网络都有可能出故障，网络故障导致网上互联计算机的隔离，针对各种故障提供高可用性；故障检测、容错、故障恢复

分布式系统的挑战：并发

- 由于缺乏全局时钟，网络上的计算机与时钟同步所达到的准确性是有限的，即没有一个正确时间的全局概念
- 而分布式系统中的并发执行是常见行为。代表公共资源的任何对象如何在并发资源中操作正确?
- 在分布式系统中,总会发生诸如机器宕机 或网络异常等情况,如何在一个可能发生上述异常的分布式系统中,快速且正确地在集群内部对某项决策达成一致?

Q: how do you control access to shared resources? how do multiple machines reach an agreement?

- 分布式事务；两段式提交;Paxos

分布式系统的挑战:容错

- 在分布式系统中,总会发生诸如机器宕机或网络异常等情况
- 如何在一个可能发生上述异常的分布式环境中,保证系统的正常运行?
- Q: how do you know if a machine has failed?
  - Failure detection
- Q: how do you program your system to operate continually even under failures?
  - Replication, gossiping

分布式系统的挑战：存储管理

- 大规模分布式系统中，如何高效访问数万台甚至数十万台主机上的数以百万计的资源？
  - 如何存：分配特定节点存储特定信息
  - 如何取：查询特定信息，定位到特定节点
- Q: how do you locate where things are and access them?
  - DHT

CAP THEROM

- 2000年，Eric Brewer教授提出了著名的CAP理论，后来Seth和Nancy证明了CAP理论的正确性。
- 一个分布式系统不可能同时满足这三个需求，最多只能同时满足两个。
  - 一致性(Consistency):系统在执行过某项操作后仍然处于一致的状态。在分布式系统中，更新操作执行成功后所有的用户都应该读取到最新的值，这样的系统被认为具有一致性
  - 可用性（Availability):每一个操作总是能够在一定的时间内返回结果，“一定时间内”是指，系统的结果必须在给定时间内返回，如果超时则被认为不可用
  - 分区容错性（Partition Tolerance):系统在存在网络分区的情况下，仍然可以正常工作。除了整个网络的故障外，其他故障都不能导致整个系统无法正确响应

CAP THEROM

- 一致性(Consistency):
- 可用性（Availability):
- 分区容错性（Partition Tolerance):

节点断电，硬盘故障
管理员配置错误、错删数据
软件Bug
网络拥塞、丢包
黑客入侵
火灾、地震

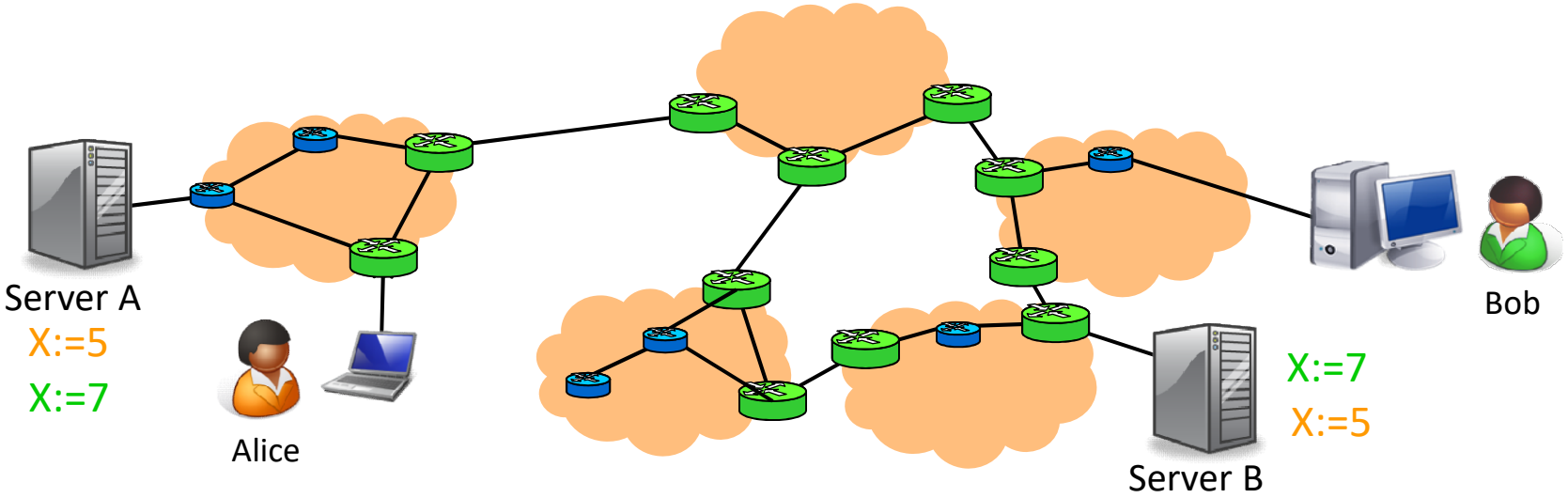
CAP THEROM

- 一致性(Consistency):
- 可用性（Availability): Cloud data center need to be very responsive—Cache
- 分区容错性（Partition Tolerance):

Amazon/Google 延迟每增加500ms，收益就会下跌20%
Amazon每增加1ms的延迟，收入损失$6M/year

CAP THEROM

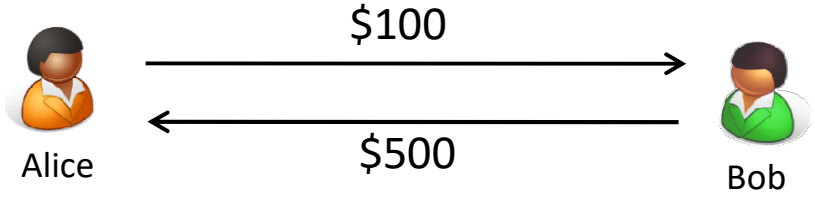
- 一致性(Consistency):all nodes see same data at any time, or reads return latest written value by any client.
  - BANK; Book flight
  - Online Video; Number of Units available; Online Map
- 可用性（Availability):
- 分区容错性（Partition Tolerance):



Giving up one of {C, A, P}

- Give up partitions
  - 把所有数据放在一台服务器或一个Rack的集群上
  - **可扩展性受影响**
- Give up availability
  - 通过加锁机制使服务等待一定时间
  - 遇到网络分区使服务等待一定时间
  - **响应时间受影响**
- Give up consistency
  - 最终一致性（*Eventually consistent*）
  - If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.
  - Examples: DNS, web caching, Amazon Dynamo, Hadoop HBase, CouchDB

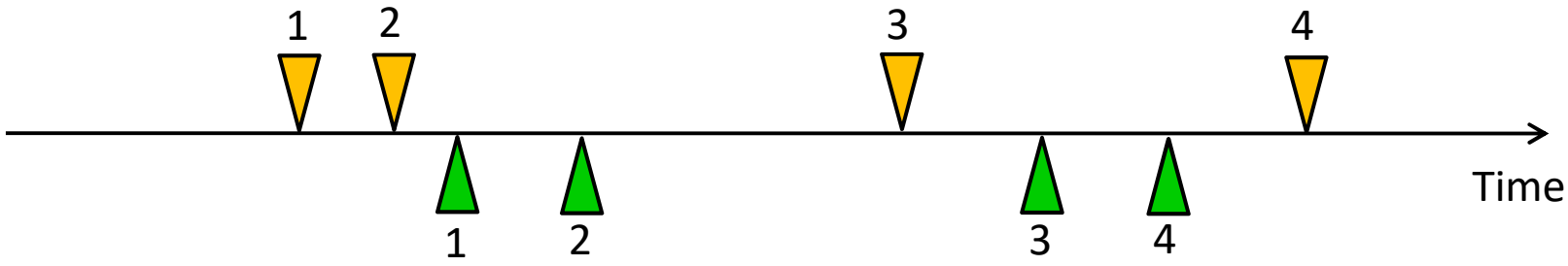
并发 & 同步




```
1) B=Balance(Bob)
2) A=Balance(Alice)
3) SetBalance(Bob,B+100)
4) SetBalance(Alice,A-100)
```

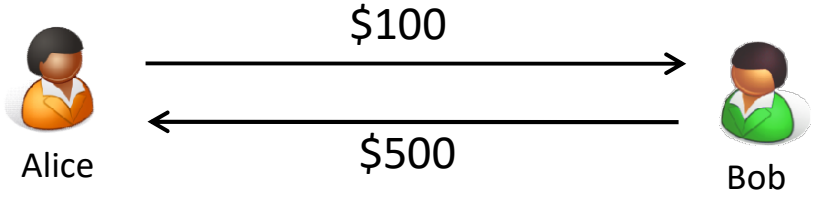
```
1) A=Balance(Alice)
2) B=Balance(Bob)
3) SetBalance(Alice,A+500)
4) SetBalance(Bob,B-500)
```

- 这两段代码同时执行会出现什么情况?



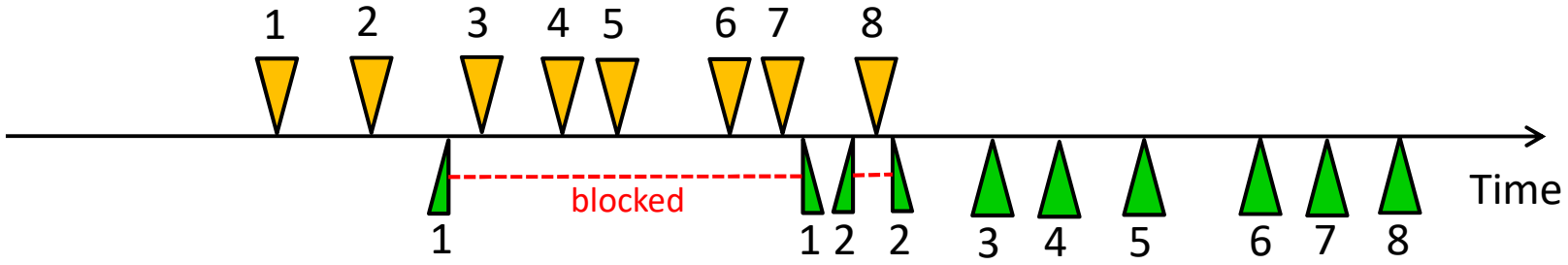
Alice's balance:	$200	$200	$700	$700	$100
Bob's balance:	$800	$900	$900	$300	$300

Locking helps!



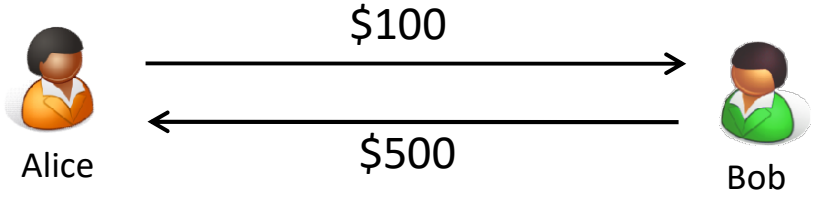

```
1) LOCK(Bob)
2) LOCK(Alice)
3) B=Balance(Bob)
4) A=Balance(Alice)
5) SetBalance(Bob,B+100)
6) SetBalance(Alice,A-100)
7) UNLOCK(Alice)
8) UNLOCK(Bob)
```

```
1) LOCK(Alice)
2) LOCK(Bob)
3) A=Balance(Alice)
4) B=Balance(Bob)
5) SetBalance(Alice,A+500)
6) SetBalance(Bob,B-500)
7) UNLOCK(Bob)
8) UNLOCK(Alice)
```



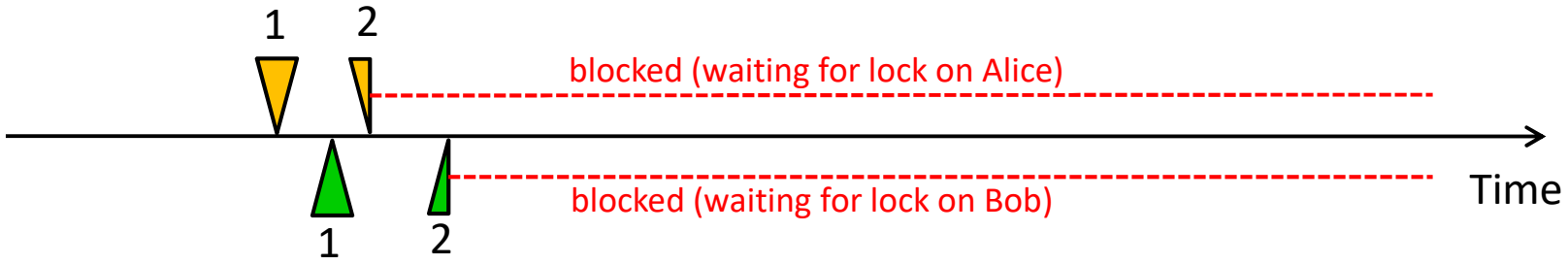
Alice's balance:	$200	$200	$100	$600	$600
Bob's balance:	$800	$900	$900	$900	$400

Problem: Deadlock




```
1) LOCK(Bob)
2) LOCK(Alice)
3) B=Balance(Bob)
4) A=Balance(Alice)
5) SetBalance(Bob,B+100)
6) SetBalance(Alice,A-100)
7) UNLOCK(Alice)
8) UNLOCK(Bob)
```

```
1) LOCK(Alice)
2) LOCK(Bob)
3) A=Balance(Alice)
4) B=Balance(Bob)
5) SetBalance(Alice,A+500)
6) SetBalance(Bob,B-500)
7) UNLOCK(Bob)
8) UNLOCK(Alice)
```



- Neither processor can make progress!

事务(Transaction)

- 事务(Transaction)是由一系列对系统中数据进行访问与更新的操作序列所组成的一个程序执行逻辑单元（Unit)。狭义的事务往往特指数据库事务
  - 当多个应用程序并发访问数据库，事务可以在这些应用程序之间提供一种隔离方法，以防止彼此的操作互相干扰
  - 事务为数据库操作序列提供了一个从失败中恢复到正常状态的方法

事务的ACID特性

- Atomicity（原子性）：一个事务中所有操作都必须全部完成，要么全部不完成
- Consistency（一致性）： 在事务开始和结束时，数据库应该在一致状态
- Isolation（隔离性）：如果有多个事务同时执行，彼此互不影响
- Durability（持久性）：事务运行成功后，对系统状态的更新是永久的，不能撤销

BASE模式

- BASE模型是反ACID模型，完全不同于ACID模型，牺牲高一致性，获得可用性或可靠性：
  - Basically Available（基本可用）：指一个分布式系统的一部分发生问题变得不可用时，其他部分仍然可以正常使用，也就是允许分区失败的情形出现
    - Fast response even if some replicas are slow or crashed
  - Soft state（软状态）：“软状态（soft-state）”是与“硬状态（hard-state）”相对应的一种提法。数据库保存的数据是“硬状态”时，可以保证数据一致性，即保证数据一直是正确的。“软状态”是指状态可以有一段时间不同步，具有一定的滞后性
    - No durable memory
  - Eventually consistent（最终一致）：最终数据是一致的就可以了，而不是时时高一致。即在给定时间窗口内数据会达到一致状态
    - If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

分布式事务

- 分布式事务是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于分布式系统的不同节点。通常一个分布式事务中会涉及对多个数据源或业务系统的操作
  - 跨银行转账操作
    - 一个是本地银行提供的取款服务
    - 另一个是目标银行提供的存款服务

分布式事务

- 每个参与事务的节点运行一个事务管理器（transaction manager)
  - 负责该节点上子事务的管理
  - 与其他事务管理器交互
  - 执行子事务的prepare, commit, abort 等操作
- 每个子事务在总事务结束前必须就某操作达成共识(Consensus)

Commits Among SubTransactions = Consensus：共识问题

- 共识问题(Consensus)：分布式系统中的进程经常需要就一些值达成协定，即共识问题
  - 假定有N个进程，每个进程p_i最初处于未决(undecided)状态，并且提议集合D中的一个值v_i
  - 进程之间互相通信，交换值
  - 每个进程设置一个决定变量d_i，并进入决定(decided)状态
  - 最终就某个决定变量达成一致共识

共识问题(Consensus)

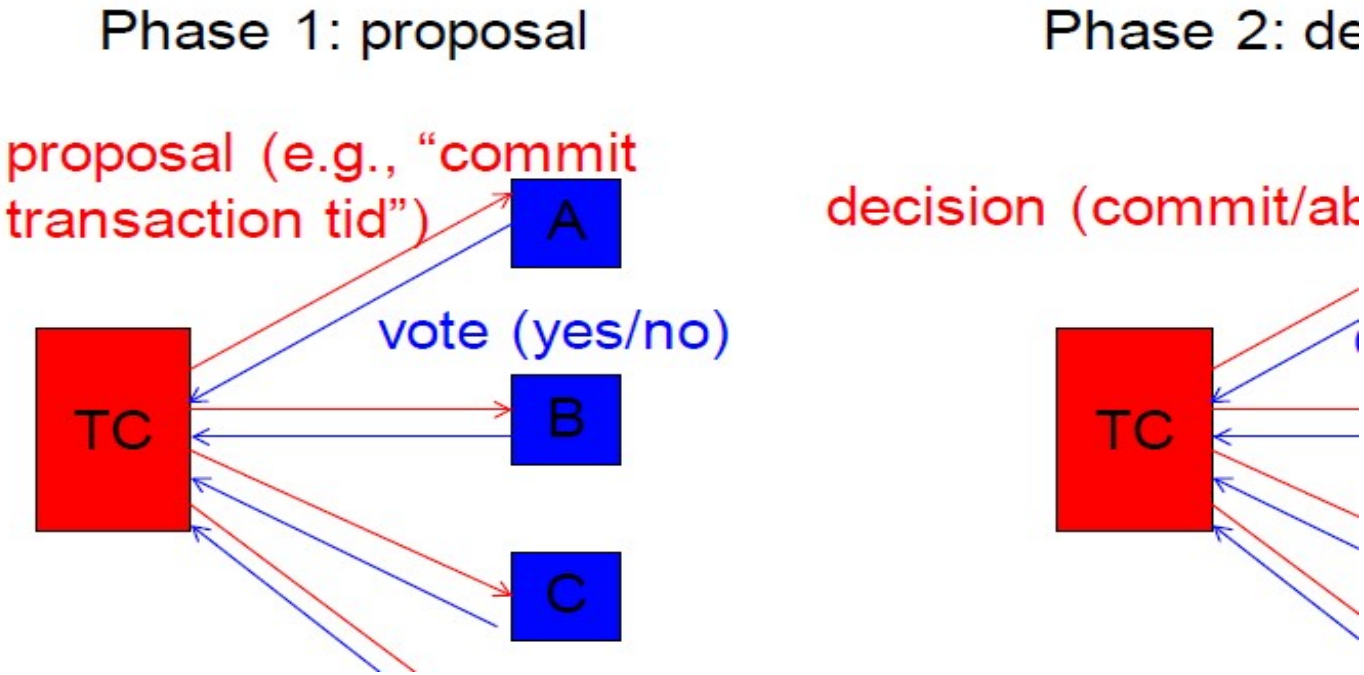
- Leader选举：就当选进程达成协议
  - 选择一个唯一的进程来扮演特定角色，并将结果通知其他进程
- 分布式互斥：就哪个进程可以进入临界区达成协议
  - 分布式进程常常需要协调它们的动作。如果一组进程共享一个或一组资源，那么访问这些资源时，常需要互斥来防止干扰并保证一致性

2PC(Two-Phase Commit Protocol)

- 2PC即两段式提交，是为了使基于分布式系统架构下的所有节点在进行事务处理过程中能够保持原子性和一致性而设计的一种算法
  - 选取一个节点作为协调者—*coordinator*；其他节点作为参与者—*participants*
  - 事务中的所有节点同意则执行，否则放弃
  - 所有节点不会永久损坏，损坏后仍可以恢复
  - 所有节点采用预写式日志，且日志被写入后即被保持在可靠的存储设备上

Two-phase commit (2PC)

- Phase 1: 投票（Voting）
  - Each participant prepares to commit, and votes on whether or not it can commit
- Phase 2: 执行(Committing)
  - Each participant actually commits or aborts

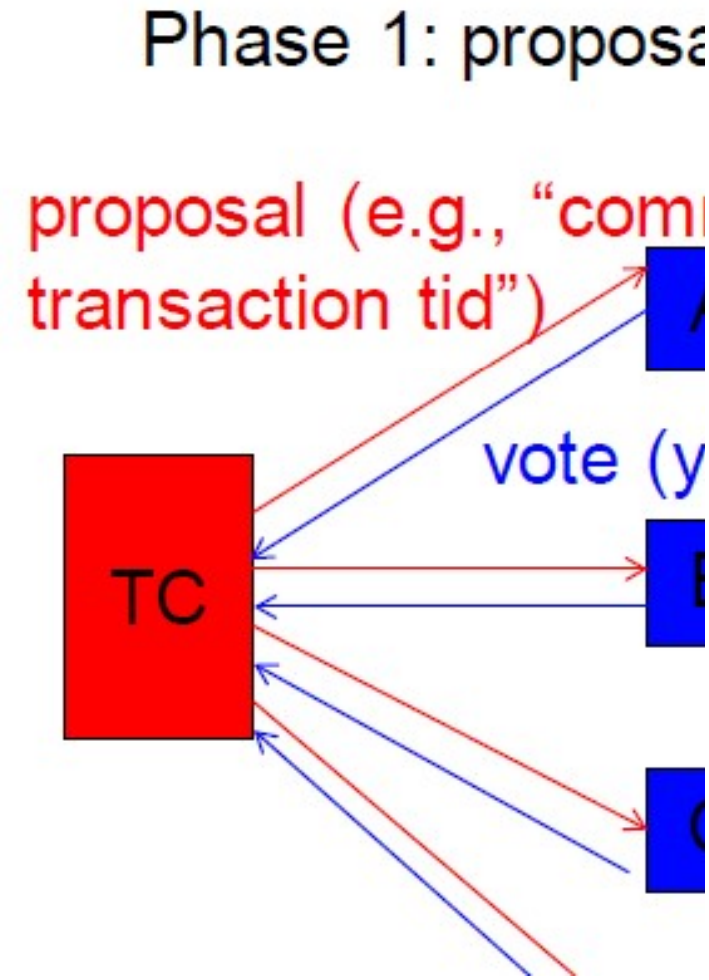


Example

- 周六与4位好友相约外出
  - 目标: 所有好友同意外出，则成行，否则取消
- 如何规划实施?
  - 给每一位好友打电话询问周六是否一起外出 (voting phase)
  - 如果每一位好友都同意，再打一遍电话来确认 ACK (commit)
  - 如果有一位好友不能成行，则给其他三位打电话放弃(abort)

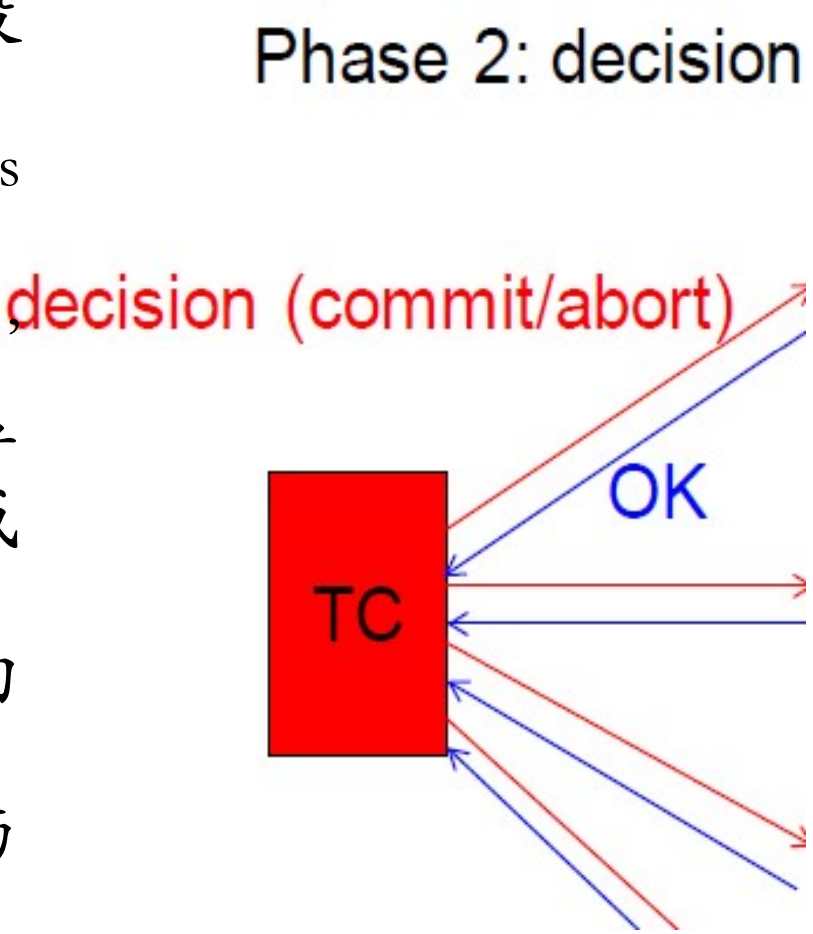
2PC阶段1：Proposal

- 协调者Coordinator 询问每一个参与者participant: 是否可以执行canCommit?；并等待响应
- 各参与者participant响应协调者Coordinator发起的询问，并返回“同意”或“终止”
  - 进入阻塞状态
  - 结果不允许改变



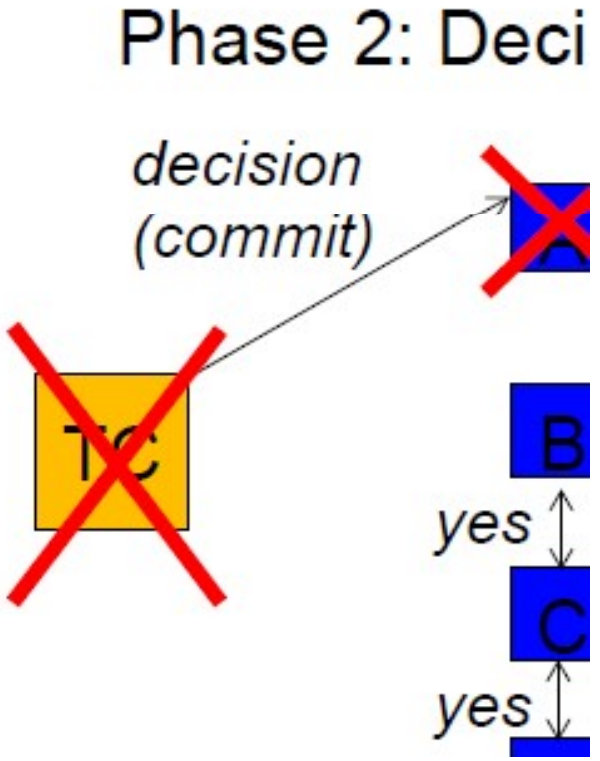
2PC阶段2：Commit

- 协调者coordinator收集所有的投票
  - If unanimous “yes”, causes commit
  - If any participant voted “no”, causes abort
- 协调者coordinator向所有参与者participant广播 *doCommit* 或 *doAbort* 请求
- 参与者participant收到信息，向协调者节点发送“OK”消息
- 协调者coordinator收到所有参与者节点“OK”消息，完成事务



Problem of 2PC

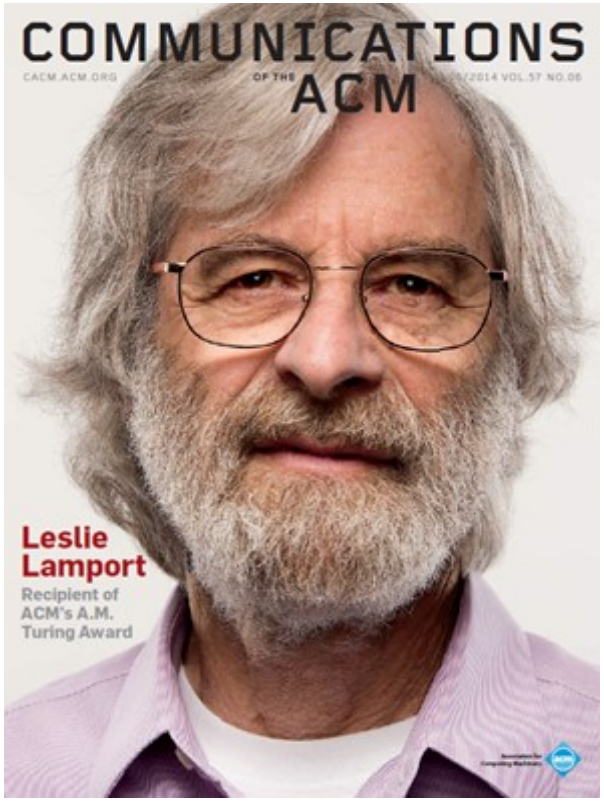
- 同步阻塞
  - 在2PC第二阶段提交的执行过程中，所有参与该事务操作的逻辑都处于阻塞状态
- 数据不一致
  - 在2PC第二阶段提交的执行过程中，发生了局部网络异常或协调者在尚未发送完Commit请求之前自身发生崩溃，会出现整个分布式系统数据不一致
- 单点问题
  - 一旦协调者出现问题，整个2PC流程将无法运行
- 太过保守
  - 若参与者出现故障，协调者只能依靠自身超时机制来判断是否需中断事务



提纲

- 分布式系统概述
- Paxos
- Gossip

Leslie Lamport



2013年”图灵奖”得主

互联网建立在分布式系统技术的基础知识之上，而后者又建立在莱斯利所发明的理论基础之上。所以，如果你喜欢使用互联网，那么你就该感谢莱斯利

Leslie Lamport

- 让分布式计算系统看起来混乱的行为变得清晰、定义明确，而且具有连贯性。
- 制定了“重要的算法”，开发“形式建模和验证协议，完善了实时分布式系统的质量。这些贡献提高了计算机系统的正确率、性能以及可靠性。”
  - 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）
  - 面包店算法(用于解决多线程同步)
  - 安全性和活性问题（并发系统的正确性验证）
  - 拜占庭将军问题（分布式系统在任意故障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）

拜占庭位于如今的土耳其的伊斯坦布尔，是东罗马帝国的首都。由于当时拜占庭罗马帝国国土辽阔，为了防御目的，因此每个军队都分隔很远，将军与将军之间只能靠信差传消息。 在战争的时候，拜占庭军队内所有将军和副官必需达成一致的共识，决定是否有赢的机会才去攻打敌人的阵营。但是，在军队内有可能存有叛徒和敌军的间谍，左右将军们的决定又扰乱整体军队的秩序。在进行共识时，结果并不代表大多数人的意见。这时候，在已知有成员谋反的情况下，其余忠诚的将军在不受叛徒的影响下如何达成一致的协议

Leslie Lamport

- Paxos协议是少数在工程实践中证实的强一致性、高可用的去中心化分布式协议
  - 1989年，投稿到TOCS: ACM Transaction on Computer Science
  - 1989年被Reject.
    - Too much archaeological stories of Greek parliament
    - Too few scientific formula
- 1998年（十年后），论文发表于TOCS 1998
  - This submission was recently discovered behind a filing cabinet in the TOCS editorial office.
  - The author is currently doing field work in the Greek isles and can not be reached…

Importance of Paxos

- Paxos算法是基于 消息传递 且具有 高度容错特性 的一致性算法 ,是目前公认的解决 分布式一致性 问题 最有效 的算法之一
- 在常见的分布式系统中,总会发生诸如 机器宕机 或 网络异常 (包括消息的延迟、丢失、重复、乱序,还有网络分区)等情况。Paxos算法需要解决的问题就是如何在一个可能发生上述异常的分布式系统中,快速且正确地在集群内部对 某个数据的值 达成 一致 ,并且保证不论发生以上任何异常,都不会破坏整个系统的一致性

Consensus is Difficult

- 消息传递异步无序(asynchronous): 消息延时、丢失，节点间消息传递做不到同步有序(synchronous)
- 节点宕机(fail-stop): 节点持续宕机，不会恢复
- 节点宕机恢复(fail-recover): 节点宕机一段时间后恢复，在分布式系统中最常见
- 网络分化(network partition): 网络链路出现问题，将N个节点隔离成多个部分
- 拜占庭将军问题(byzantine failure): 节点或宕机或逻辑失败，甚至不按套路出牌抛出干扰决议的信息

Importance of Paxos

- There is only one consensus protocol, and that ‘s Paxos - all other approaches are just broken versions of Paxos
- The only known completely-safe and largely-live agreement protocol
  - Safety (Correctness)
    - Bad things never happen.
    - Any process in the group should not decide a different value than others.
    - The value should be meaningful
  - Liveness
    - Good things eventually happen
    - Eventually, the process will all agree on a single value.
- All working protocols for asynchronous consensus we have so far encountered have Paxos at their core

Paxos

- Paxos Paper on TOCS 1998

The part-time parliament
L Lamport - ACM Transactions on Computer Systems (TOCS), 1998
Page 1. The Part-Time Parliament LESLIE LAMPORT Digital Equipm
Recent archaeological discoveries on the island of Paxos reveal that t
functioned despite the peripatetic propensity of its part-time legislator

- Paxos made simple / ACM Sigact News, 2001

[PDF] Paxos made simple
L Lamport - ACM Sigact News, 2001 - cs.utexas.edu
... It can be shown that phase 2 of the Paxos consensus algorithm has the minir
of any algorithm for reaching agreement in the presence of faults [2]. Hence, the
rithm is essentially optimal. ... [5] Leslie Lamport. The part-time parliament. ...

Paxos

- 在古希腊有一个叫做Paxos的小岛，岛上采用议会的形式来通过法令，议会中的议员通过信使进行消息的传递。值得注意的是，议员和信使都是兼职的，他们随时有可能会离开议会厅，并且信使可能会重复的传递消息，也可能一去不复返。因此，议会协议要保证在这种情况下法令仍然能够正确的产生，并且不会出现冲突

Paxos

- Paxos算法解决的问题是一个分布式系统如何就某个值（决议）达成一致，进而达到分布式的一致性状态
- 这个过程本质上就是分布式系统中常见的quorum机制（原指为了处理事务做出决定而必须出席的成员数量（半数以上），即根据少数服从多数的选举原则产生一个决议）
- Paxos协议中只要有超过一半的节点正常，就可以工作，能很好对抗宕机、网络分化等异常情况

问题描述

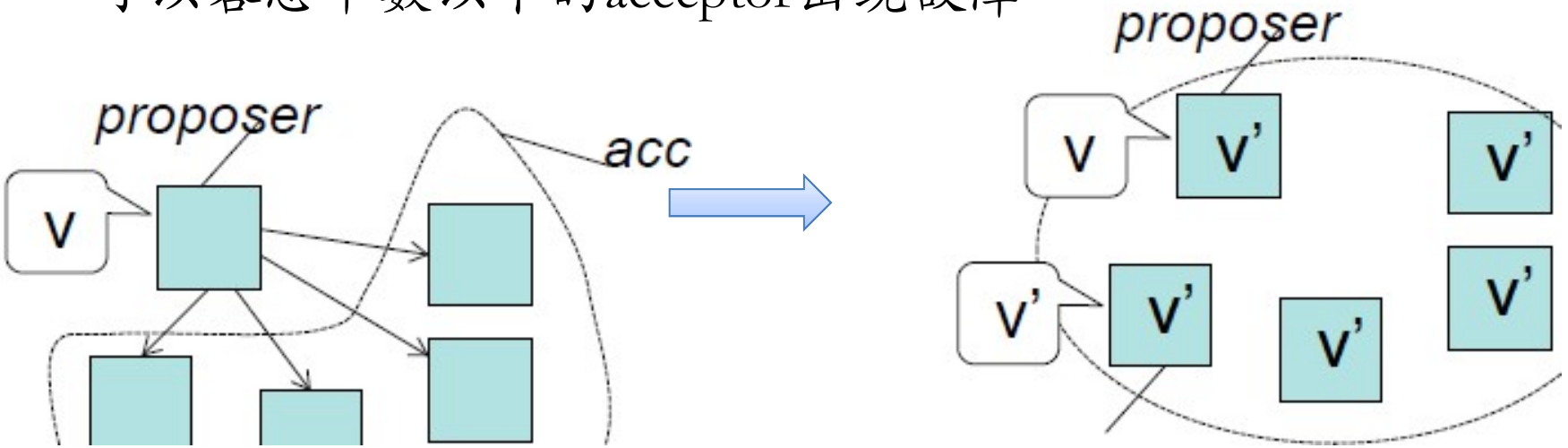
- 假设有一组可以提出提案的进程集合，那么对于一个一致性算法来说需要保证以下几点：
  - 在这些被提出的提案中，只有一个会被选定
  - 如果没有提案被提出，那么就不会有被选定的提案
  - 当一个提案被选定后，进程应该可以获取被选定的提案信息

Roles of Paxos

- processor可以担任三个角色“proposer”、“acceptor”和“learner”中的一个或多个角色
  - proposer可以propose（提出）proposal
  - acceptor可以accept（接受）proposal
  - learner可以learn （学习）value
- 每个参与者以任意的速度执行，可能会出错而停止，也可能会重启
- 各个processor之间信息的传递可以延迟、丢失，但在这个算法中假设传达到的信息都是正确的

Paxos

- Paxos的目标
  - 多个processor向多个acceptor提出提案，最后就某个决议达成一致
- 多acceptor下提案的选取
  - 当足够多Acceptor批准提案时，该提案被选定—“majority”
- Paxos对容错性的要求：
  - 可以容忍任意的proposer出现故障
  - 可以容忍半数以下的acceptor出现故障

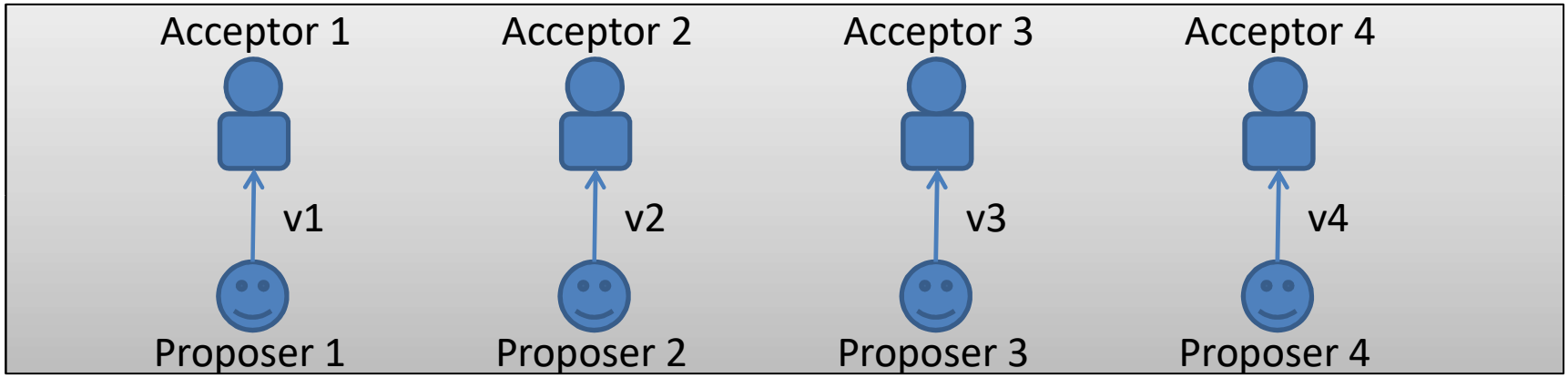


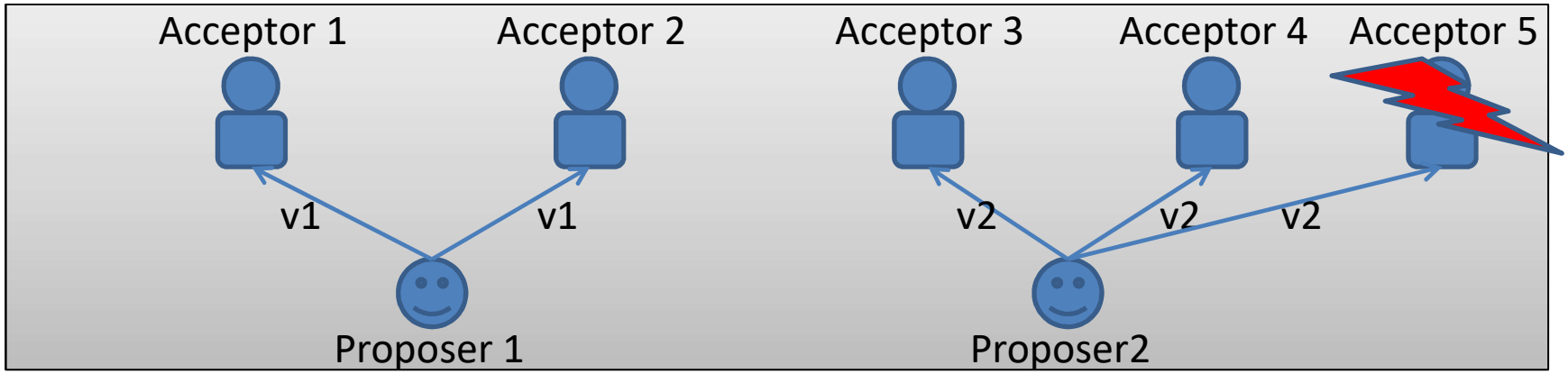
提案的选定：Choosing a value

- Two main requirements:
  - P1
  - P2
- With extensions:
  - $P2^a$
  - $P2^b$
  - $P2^c$
  - $P1^a$

P1

- 在没有失败和消息丢失的情况下，如果希望当只有一个提案被提出，仍然可以选出一个提案
- P1:一个Acceptor必须批准它收到的第一个提案



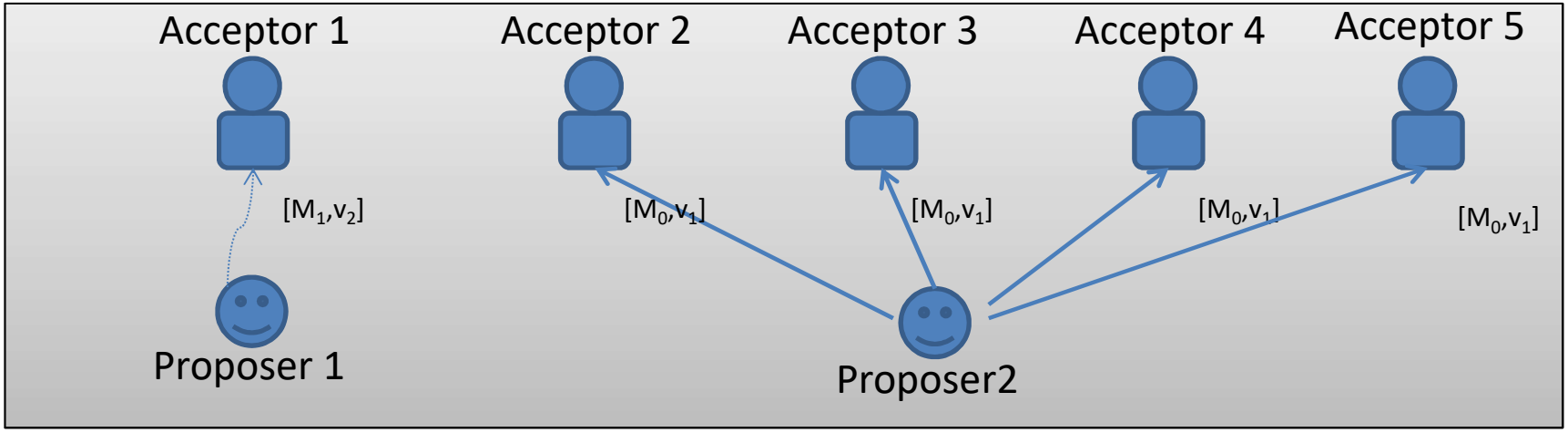


P2

- P1 再加上一个提案被选定需要由半数以上的Acceptor通过的需求暗示着一个Acceptor必须能够批准(accept)不止一个提案。我们为每个提案设定一个编号来记录一个Acceptor通过的那些提案。为了避免混淆，需要保证不同的提案具有不同的编号。当一个具有某value值的提案被半数以上的Acceptor通过后，我们就认为该value被选定了。此时我们也认为该提案被选定了
- Paxos允许多个提案被选定，但同时必须保证所有被选定的提案都具有相同的Value值
- P2：如果编号为M_0、具有value值 V_0 的提案(即[M_0 ，V_0])被选定(chosen)了，那么所有比编号M_0更高的、且被Acceptor选定的提案，其value值也必须是 V_0

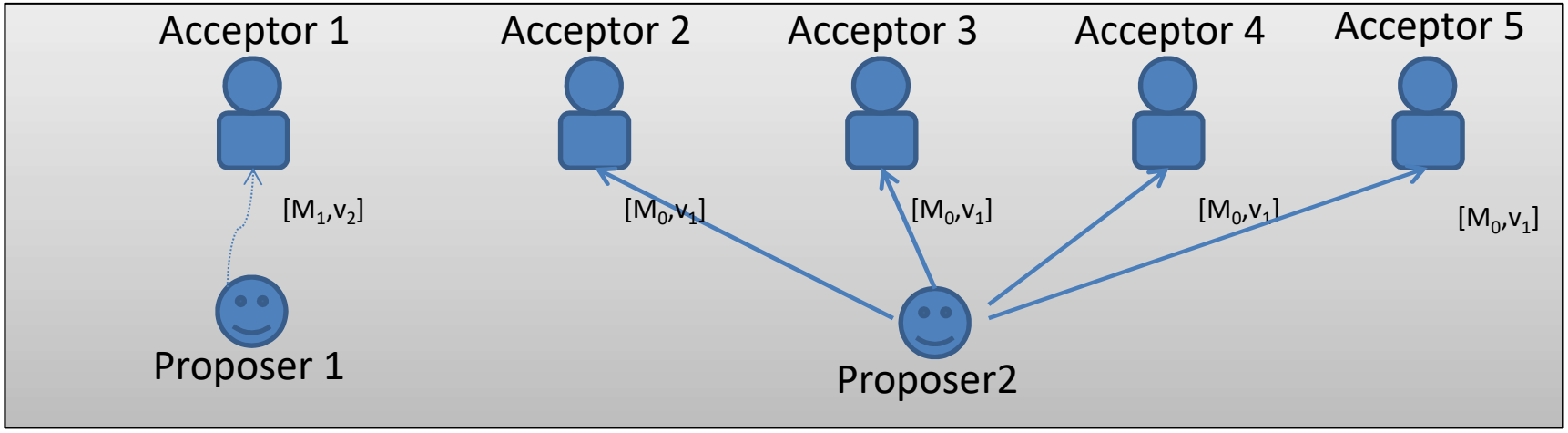
$P2^a$

- $P2^a$：如果编号为M_0、具有value值 V_0的提案(即[M_0，V_0])被选定(chosen)了，那么所有比编号M_0更高的被Acceptor批准的提案，其value值也必须是 V_0



P2^b

- 如果一个提案[M_0，V_0]被选定，那么所有比它编号更高的被Proposer提出的提案，其value值也必须是V_0

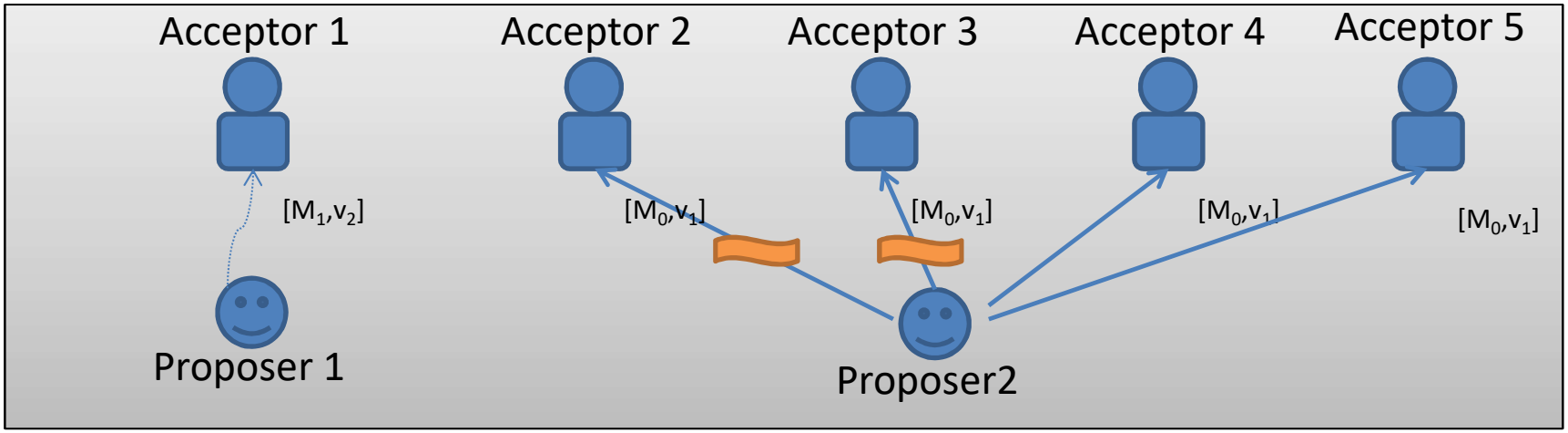


$P2^c$ —后者认同前者

- 对于任意的 Mn 和 Vn，如果编号为 Mn 和 value值为 Vn 的提案[Mn， Vn]被提出，那么肯定存在一个由半数以上的Acceptor组成的集合 S，可以满足条件 a) 或者 b) 中的一个:
  - S 中不存在任何的Acceptor批准过编号小于 Mn 的提案
  - 选取 S 中所有Acceptor批准的编号小于 Mn 的提案，其中编号最大的那个提案其value值是Vn

$P2^c$ —后者认同前者

- 为了维护 P2c 的不变性，一个Proposer在产生编号为 n 的提案时，必须要知道某一个将要或已经被半数以上Acceptor通过的编号小于 n 的最大编号的提案。获取那些已经被通过的提案很简单，但是预测未来会被通过的那些却很困难。在这里，为了避免让Proposer去预测未来，我们通过限定不会有那样的通过情况来控制它。换句话说，Proposer会请求Acceptors不要再通过任何编号小于 n 的提案。



Issuing proposals—提案生成算法(1)

- Proposer选择一个编号M_n并向acceptor集合的成员发送一个prepare请求（包含所选择的编号n)，要求acceptor返回promise，promise包含两个信息:
  - 向Proposer承诺，保证不再批准任何编号小于M_n的proposal
  - 如果该acceptor已经批准过任何提案，那么就向Proposer反馈当前该Acceptor所接受的小于M_n的最大编号的proposal（如果该acceptor还未接受过任何proposal则不用返回这个消息）

Issuing proposals—提案生成算法(2)

- 如果proposer接收了来自半数以上的acceptor所返回的promise，则该proposer可以提出编号为 M_n 、value值为 V_n 的proposal [M_n，V_n]：
  - 该proposal编号为M_n，value值 V_n为promise所带回的proposal的value的最大值
  - 如果promise没有带回有关的proposal则value可以为任意值

Acceptor批准提案

- Accept请求：在Proposer确定提案[M_n，V_n]之后，Proposer就会将该提案再次发送给某个Acceptor集合，并期望获得它们的批准
- P1^a：一个Acceptor可以接受一个编号为 M_n 的提案，只要它还未响应任何编号大于M_n的prepare请求

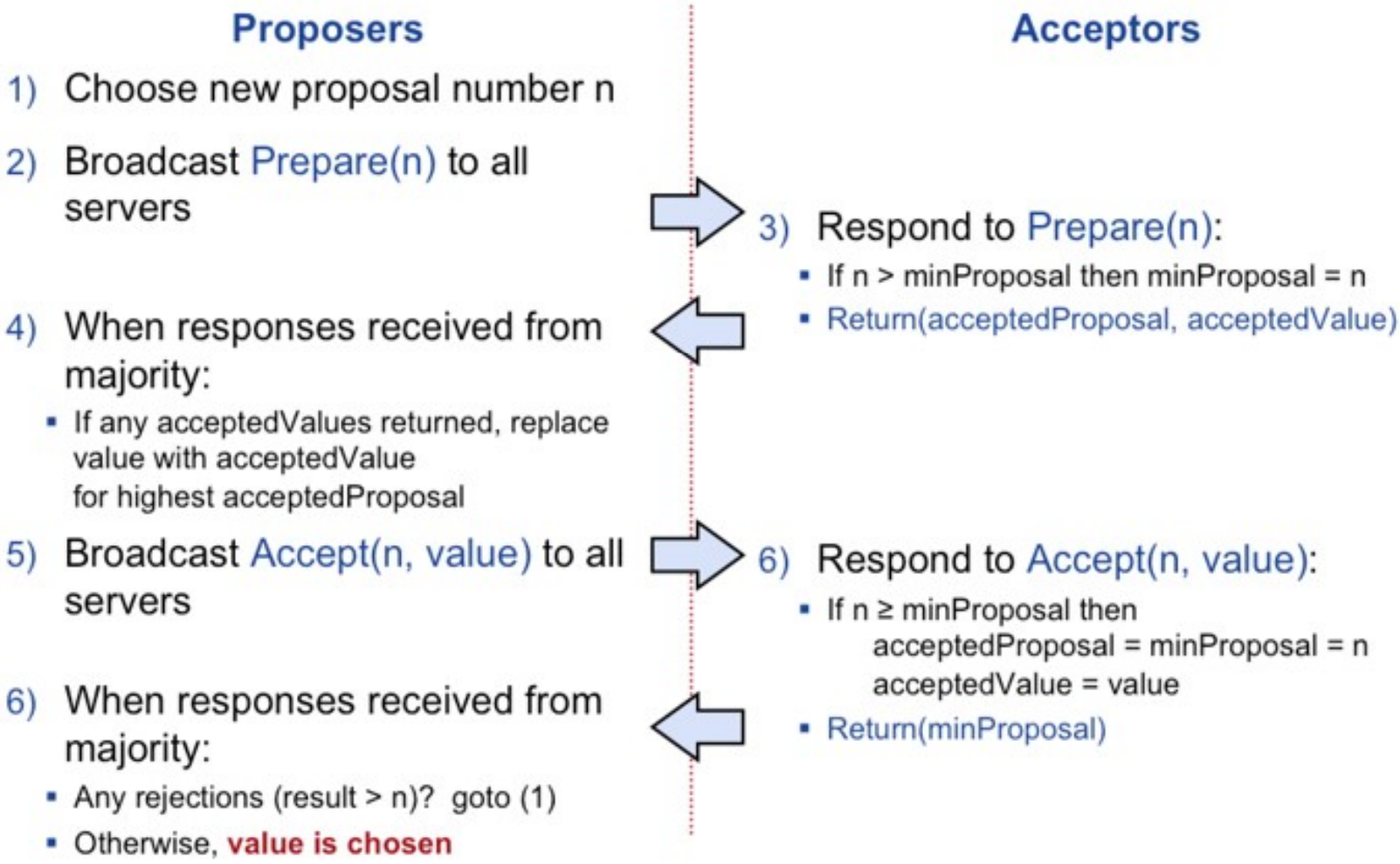
Paxos 算法详解：Phase 1

- Proposer选择一个提案编号M_n，并向acceptor的某个超过半数的子集发出一个编号为M_n的prepare请求
- 如果一个Acceptor收到一个编号为M_n的prepare请求，且编号M_n大于该Acceptor已经响应的所有Prepare请求的编号，那么它就会将它已经批准过的最大编号的提案proposal作为响应反馈给Proposer，同时该Acceptor会承诺不会再批准任何编号小于M_n的提案

Paxos 算法详解：Phase 2

- 如果Proposer收到来自半数以上的Acceptor对于其发出的编号为M_n的Prepare请求的响应，那么它就会发送一个针对[M_n，V_n]提案的Accept请求给Acceptor（V_n的值就是收到的响应中编号最大的提案的值，如果响应中不包含任何提案，那么它就是任意值）
- 如果Acceptor收到这个针对[M_n，V_n]提案的Accept请求，只要该Acceptor尚未对编号大于M_n的Prepare请求做出过响应，它就可以批准这个提案

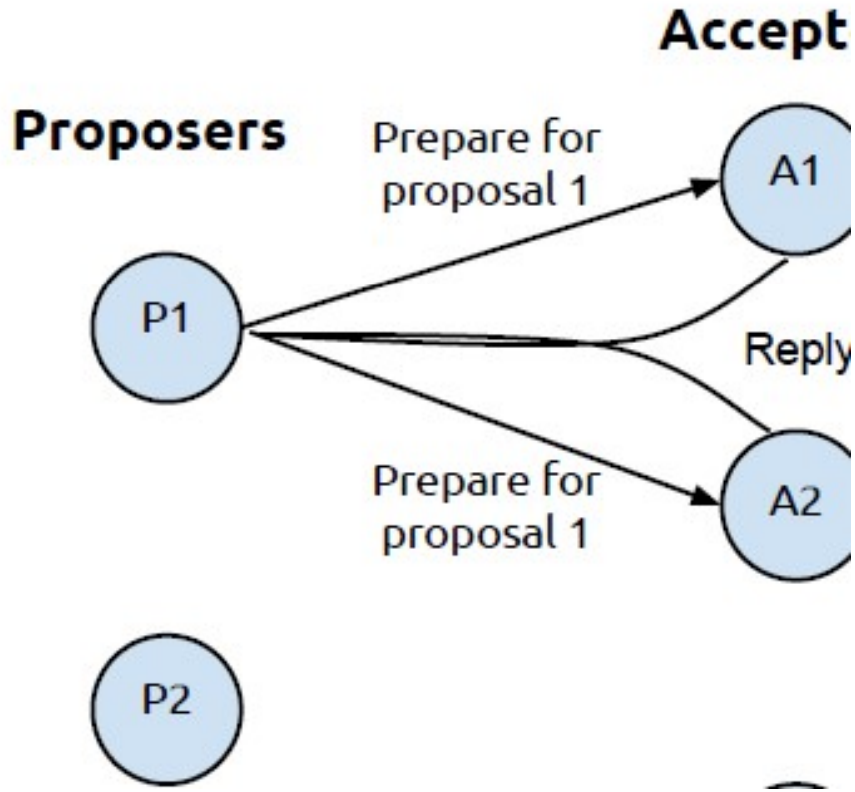
Paxos Protocol



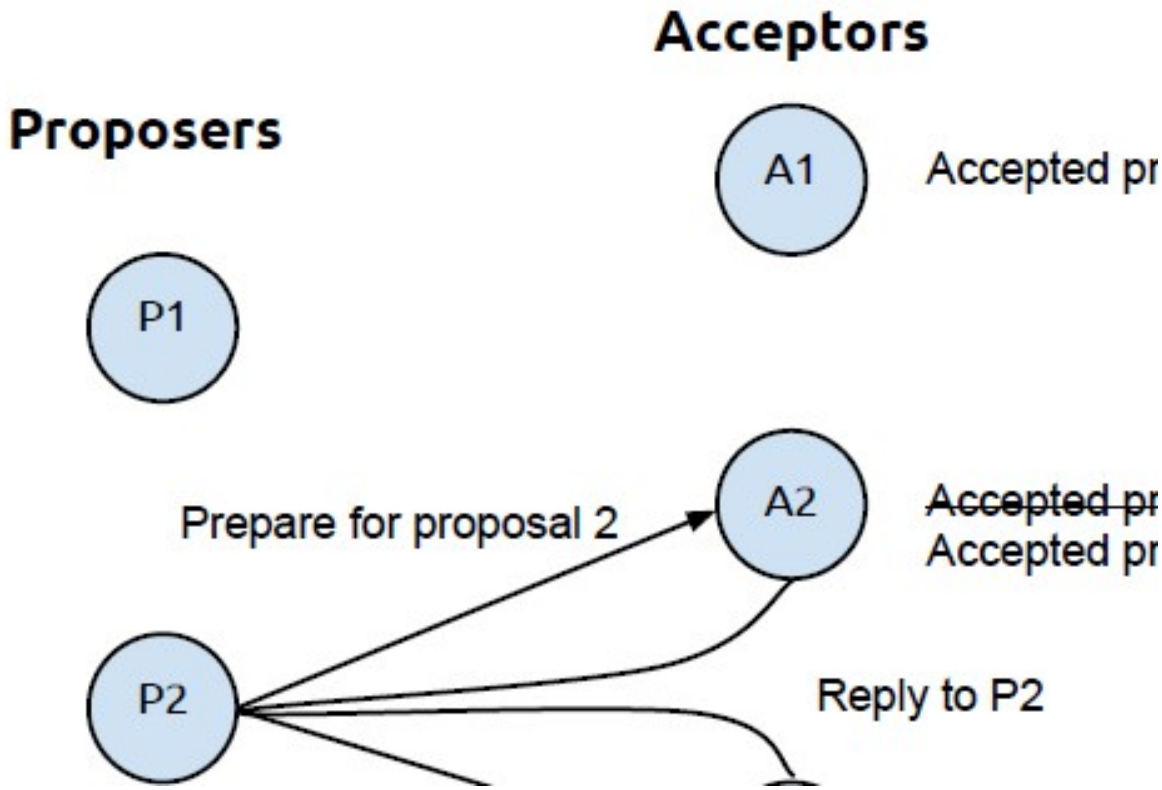
提案的获取

- 为了获取被选定的值，一个Learner必须确定一个提案已经被半数以上的Acceptor通过。
  - 让每个Acceptor，只要它通过了一个提案，就通知所有的Learners，将它通过的提案告知它们
  - 让所有的Acceptor将它们的通过信息发送给一个特定的Learner，当一个value被选定时，再由它通知其他的Learners
  - Acceptors可以将它们的通过信息发送给一个特定的Learners集合，它们中的每个都可以在一个value被选定后通知所有的Learners。这个集合中的Learners个数越多，可靠性就越好，但是通信复杂度就越高

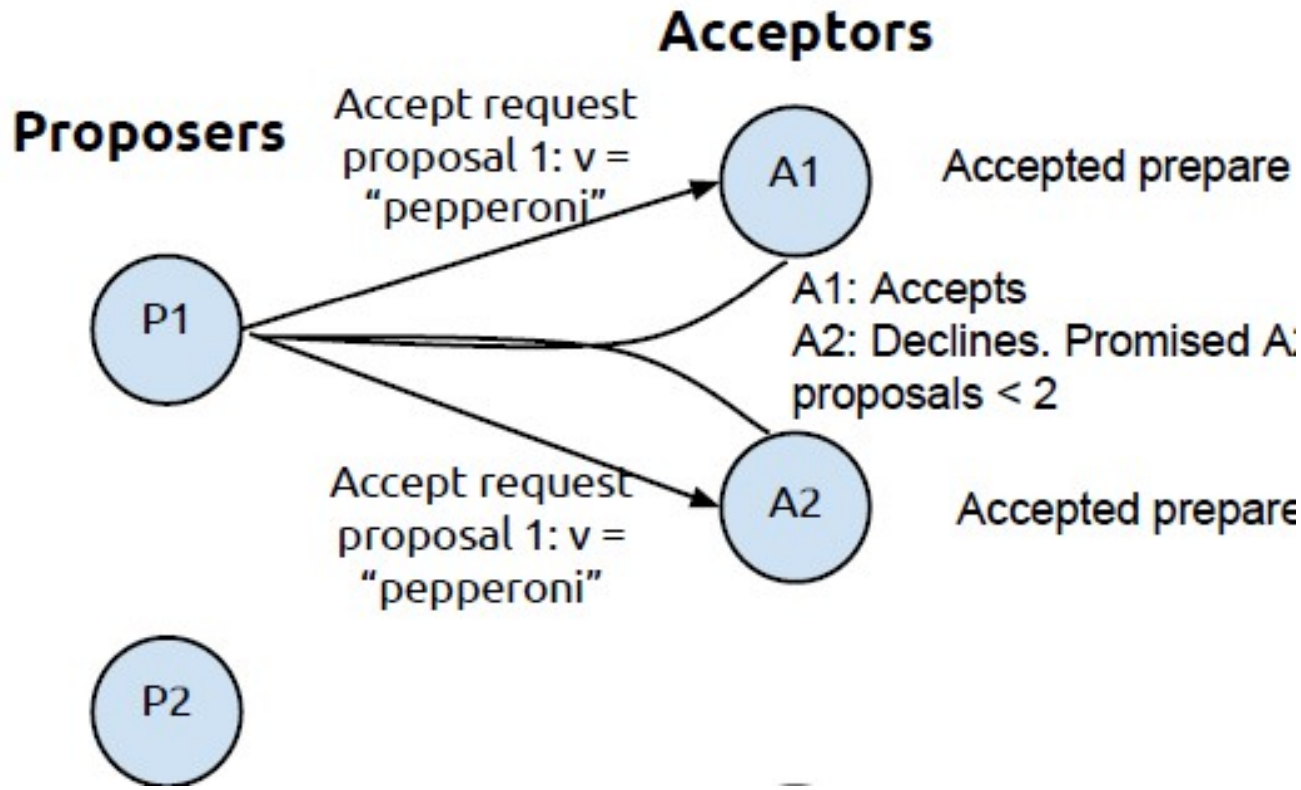
Example: Step 1



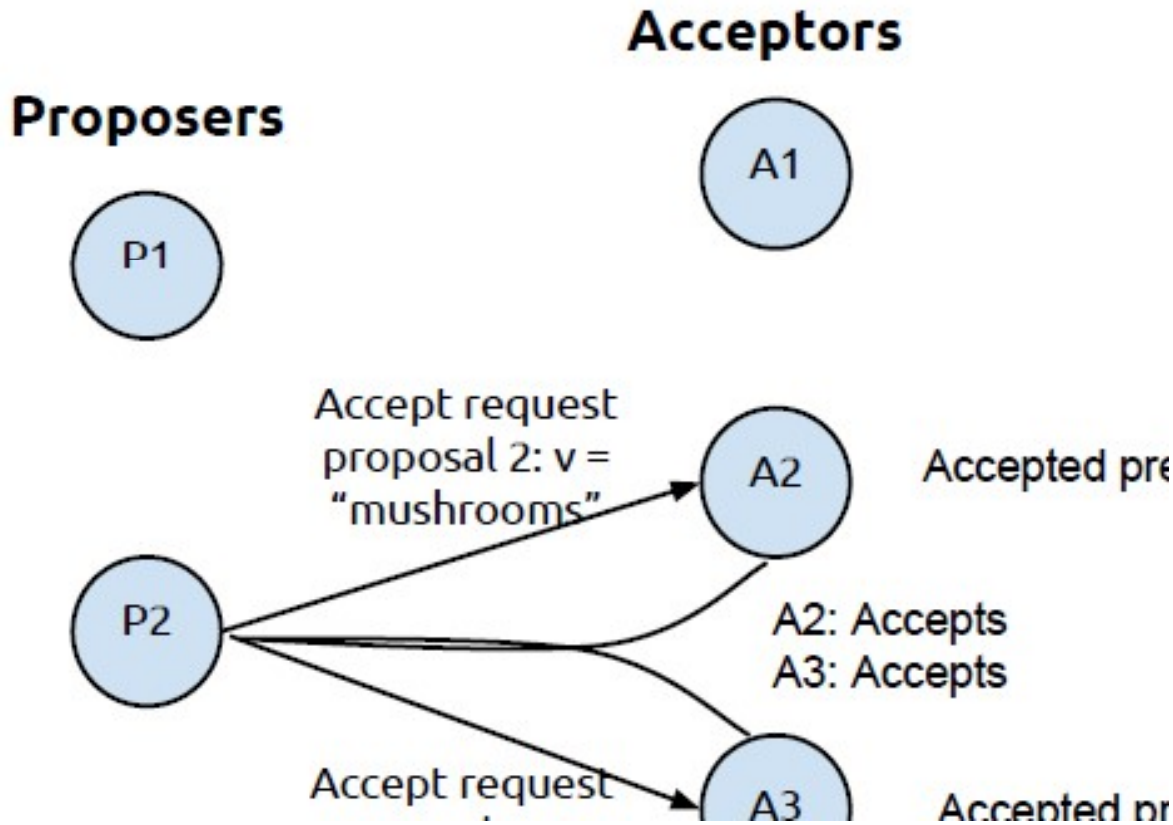
Example: Step 2



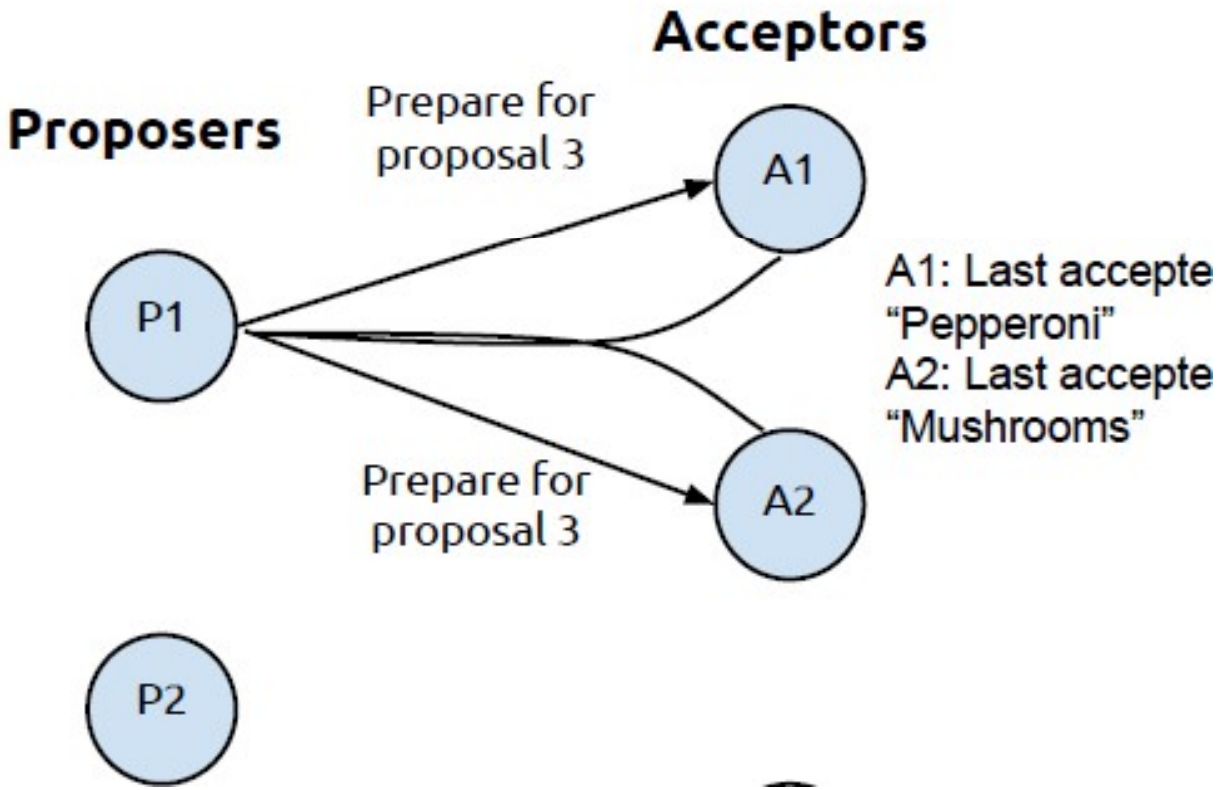
Example: Step 3



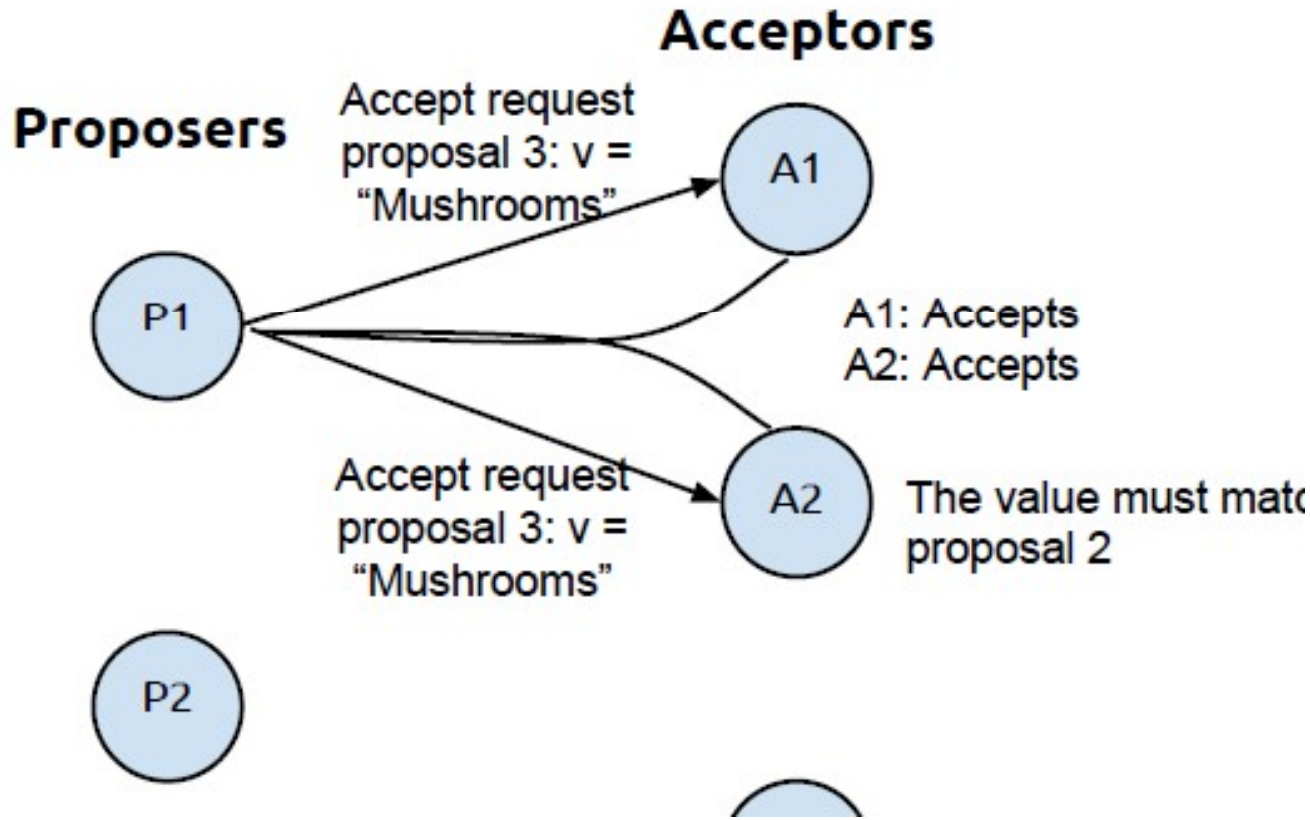
Example: Step 4 - The chosen value is “mushrooms”



Example: Step 5

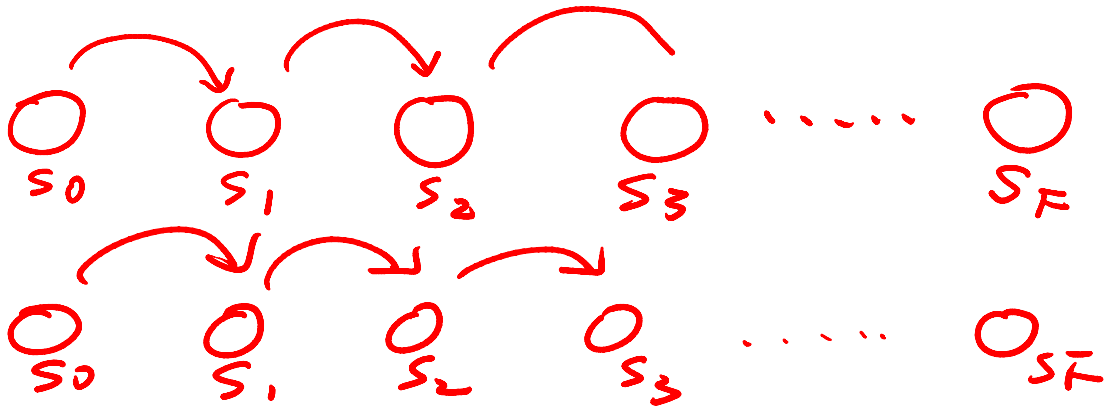


Example: Step 6



Paxos is useful

- Extremely useful, e.g.:
  - Master选举： nodes agree that Y is the primary
  - 副本执行顺序： nodes agree that Z should be the next operation to be executed
  - 分布式锁： nodes agree that client X gets a lock
  - nodes agree what time to meet



The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, Google

OSDI 2006

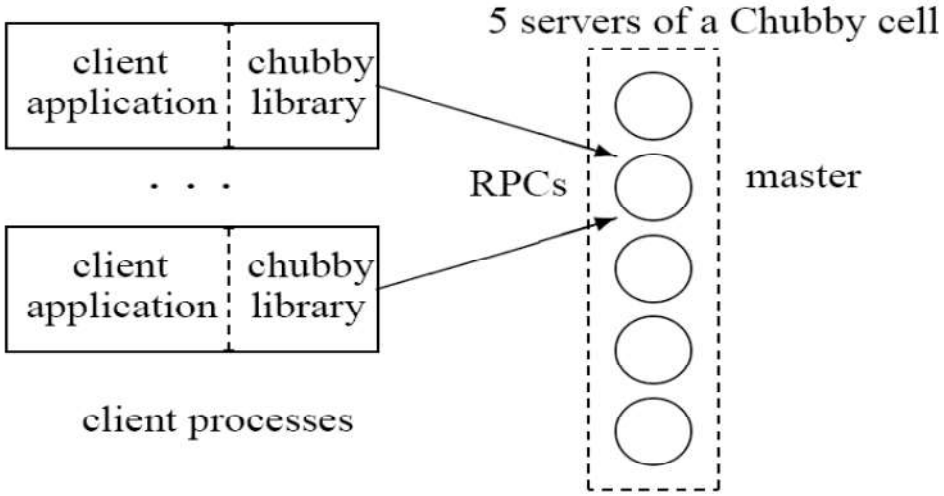
Chubby

- What is Chubby?
  - 在松耦合分布式系统中提供粗粒度锁服务，允许它的客户端进程同步彼此的操作，并对当前所处环境的基本状态信息达成一致
- Primary goals ?
  - 提高大规模集群的可靠性、可用性
- How is it used?
  - Used in Google: GFS, Bigtable, etc.
  - Purposes: Master选举; 元数据存储
  - Open-source version of Bigtable, Hbase, uses an opensource lock service, called Zookeeper

Chubby

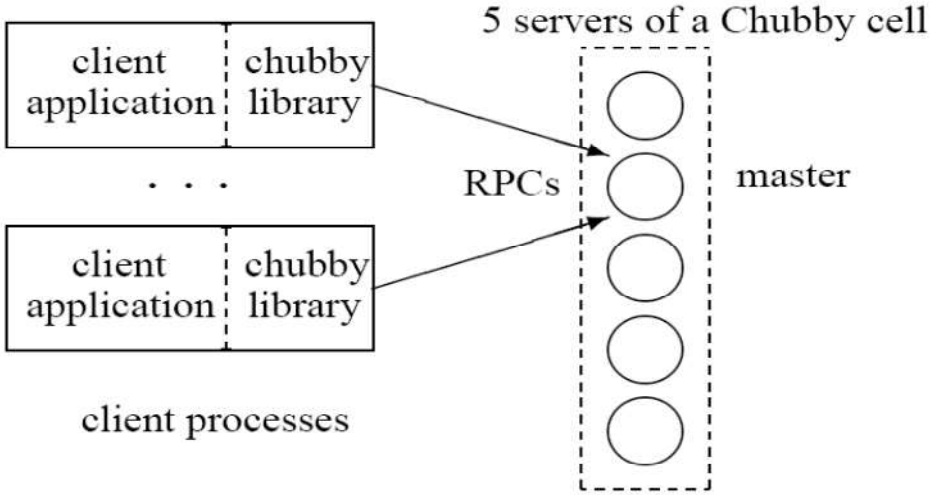
- Chubby本质上是一个分布式的、存储大量小文件的文件系统，它所有的操作都是在文件的基础上完成
- Chubby最常用的锁服务中，每一个文件代表一把锁，用户通过打开、关闭和读取文件，获取共享锁或者独占锁
- 选举主服务器过程中，符合条件的服务器都同时申请打开某个文件并请求锁住该文件
- 成功获得锁的服务器自动称为主服务器并将其地址写入这个文件夹，以便其他服务器和用户可以获知主服务器的地址信息

System Architecture



- Chubby整个系统结构主要由服务端和客户端两部分组成
- 一个典型的Chubby集群（chubby cell），通常由一组服务器组成(通常是5台)
- 各个副本服务器间使用Paxos来选Master，Master必须获取多数副本的选票，同时保证在某一段时期内副本不会再选举不同的Master（租期)
- 客户端通过向列在DNS中的副本服务器列表发送Master定位请求来获取Master信息

System Architecture



- 一旦客户端定位了Master，则客户端将所有请求直接发给Master，直到它停止响应或明确已不再是Master为止
- 对于写:
  - Master 采用Paxos将其广播给集群中所有的副本服务器
  - 过半服务器接受了写请求之后，再响应给客户端正确的应答
- 对于读:
  - Master服务器直接处理即可

提纲

- 分布式系统概述
- Paxos
- Gossip

分布式系统的挑战:容错

- 在分布式系统中,总会发生诸如机器宕机或网络异常等情况
- 如何在一个可能发生上述异常的分布式环境中,保证系统的正常运行?
- Q: how do you know if a machine has failed?
  - Failure detection
- Q: how do you program your system to operate continually even under failures?
  - Replication, gossiping

Gossip问题的由来(1972,Hajnal)

- 有n个妇女，每个人都知道一条特有的流言，她们通过电话互相联系；
- 任意两个妇女联系上以后，互相交流当前自己知道的所有流言；
- 最少需要多少次联系，使得 n个妇女每个人都知道所有流言
- “闲话算法”、“疫情传播算法”、“病毒感染算法”、“谣言传播算法”

数据复制：如何在分布式环境下进行数据副本的维护？

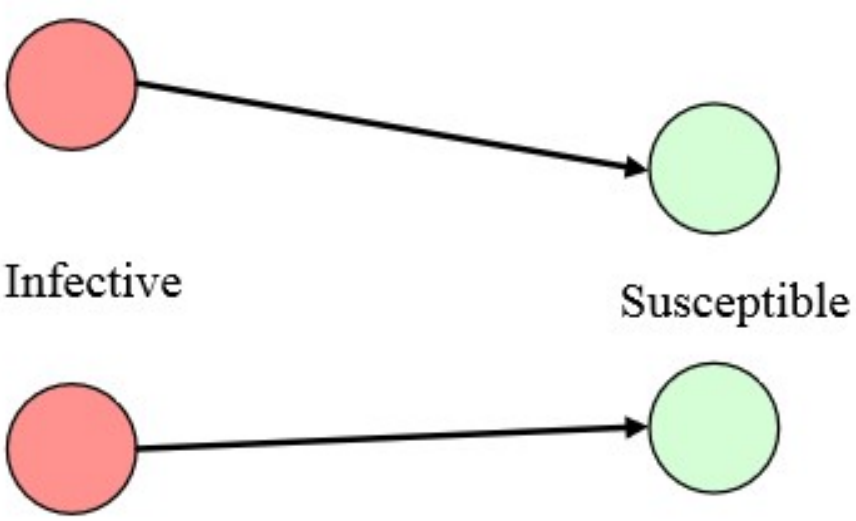
- 复制是一种增强服务的技术，它的作用体现在：
  - 增强性能：Web 缓存—》IP & DNS绑定；负载均衡
  - 容错: 组成员检测
  - 提高可用性：服务器故障；网络分区$(1-p^n)$

Gossip问题的定义(1988,Hedetniem)

- A是由网络中所有节点组成的集合，每个节点都有自己特有的信息，并需将其传播到网络中其它 所有节点;
- 用有序的节点对(i, j) 序列表示信息的传 播过程，每个节点对表示两者之间存在信息交换;
- 当序列的最 后一个节点对完成交互后，所有节点都知道所有信息，称为Gossip过程结束。
- 需要多少次信息交换Gossip过程能够结 束，需要多长时间?

节点状态

- Susceptible (S): the node does not know about the update
- Infected (I): the node knows the update and is actively spreading it
- Removed (R): the node knows the update, but it does not participate in the spreading any more

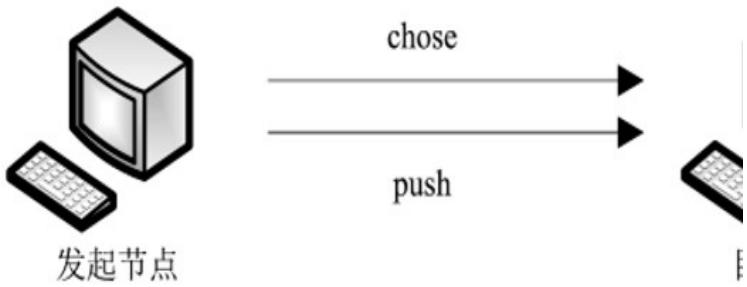


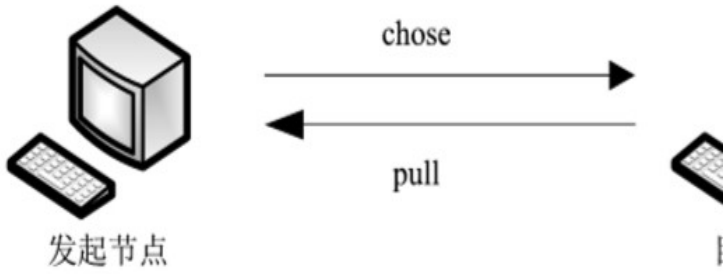
First gossip-based mechanism: Anti-entropy

- Each server lazily maintains the list of all other servers
- Periodically, each server
  - Selects a partner at random in the list of all servers
  - and perform a pair-wise sync of the entire DB with that server
- Period is a parameter — e.g., every 60 minutes (depends on the rate of updates vs. risk of inconsistency)
- All server eventually see all updates
- Can be modeled using theory of epidemics
  - Like a disease in the human population, the information spreads by pair-wise infections between individuals
    - Infected = the server with a new/updated version of a row
    - Susceptible = the server with an old/previous/inexistent version

信息交换方法

- in the **push** style, nodes periodically send (push) the current content of variable value to a node selected randomly
- in the **pull** style, nodes periodically ask (pull) new updates of the replicated variable from other, randomly selected nodes
- in the **push-pull** style, push and pull are combined together





Implementation of *update*()

% **Push version**

on timeout

$q \leftarrow \text{random}(P)$

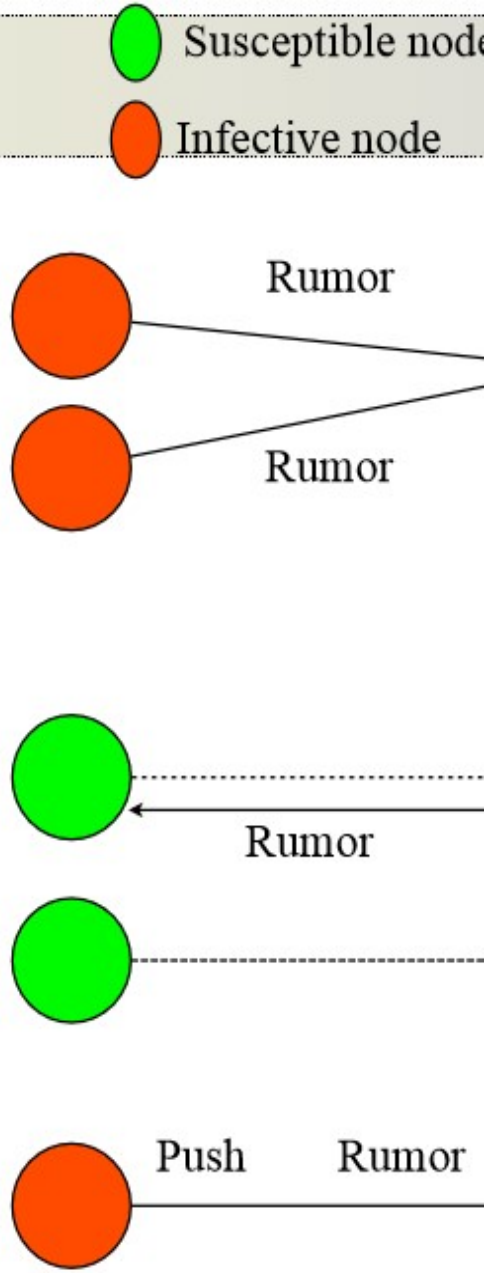
send $\langle \text{PUSH}, value \rangle$ **to** q

set timeout Δ

on receive $\langle \text{PUSH}, v \rangle$

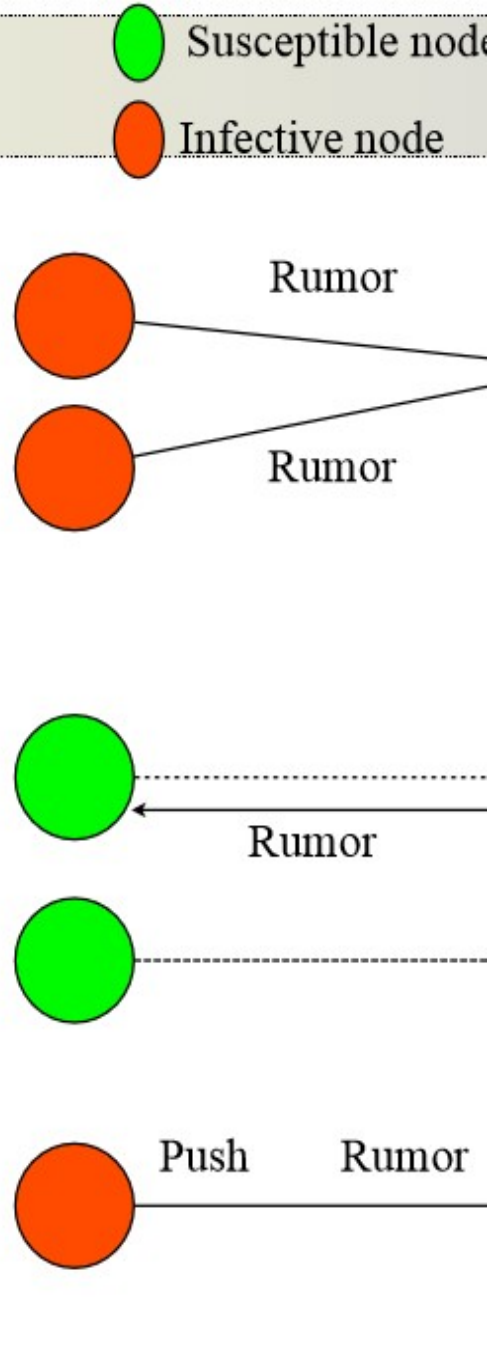
if *value.time* $<$ *v.time*

hen



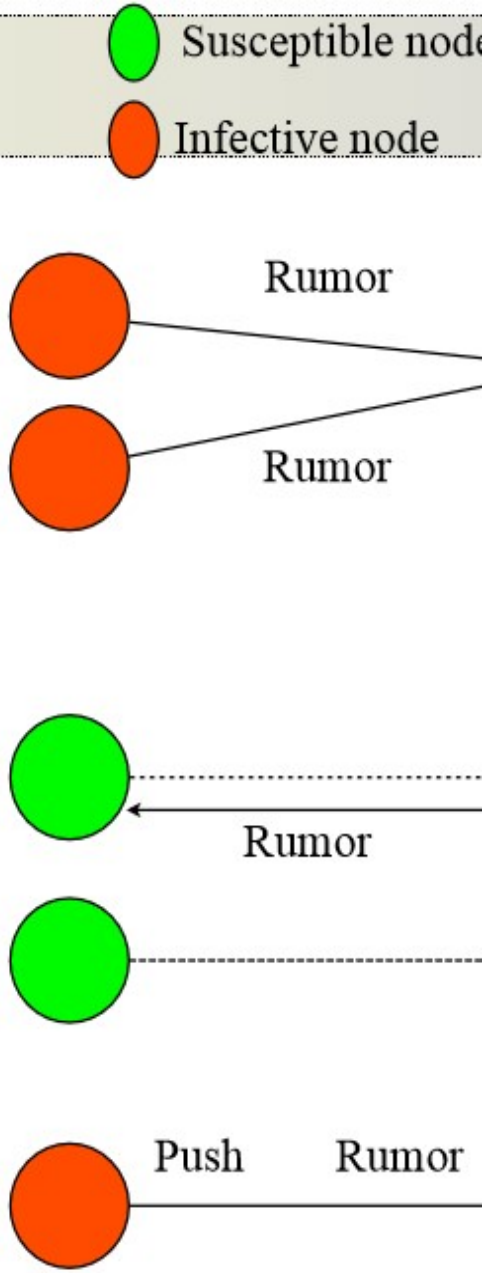
Implementation of *update*()

```
% Pull version

on timeout
    q ← random(P)
    send ⟨PULL, p, value.time
    set timeout Δ

on receive ⟨PULL, q, t⟩
    if value.time > t then
        send ⟨REPLY, value⟩ t

on receive ⟨REPLY, v⟩
    if value.time < v.time th
```



Implementation of *update()*

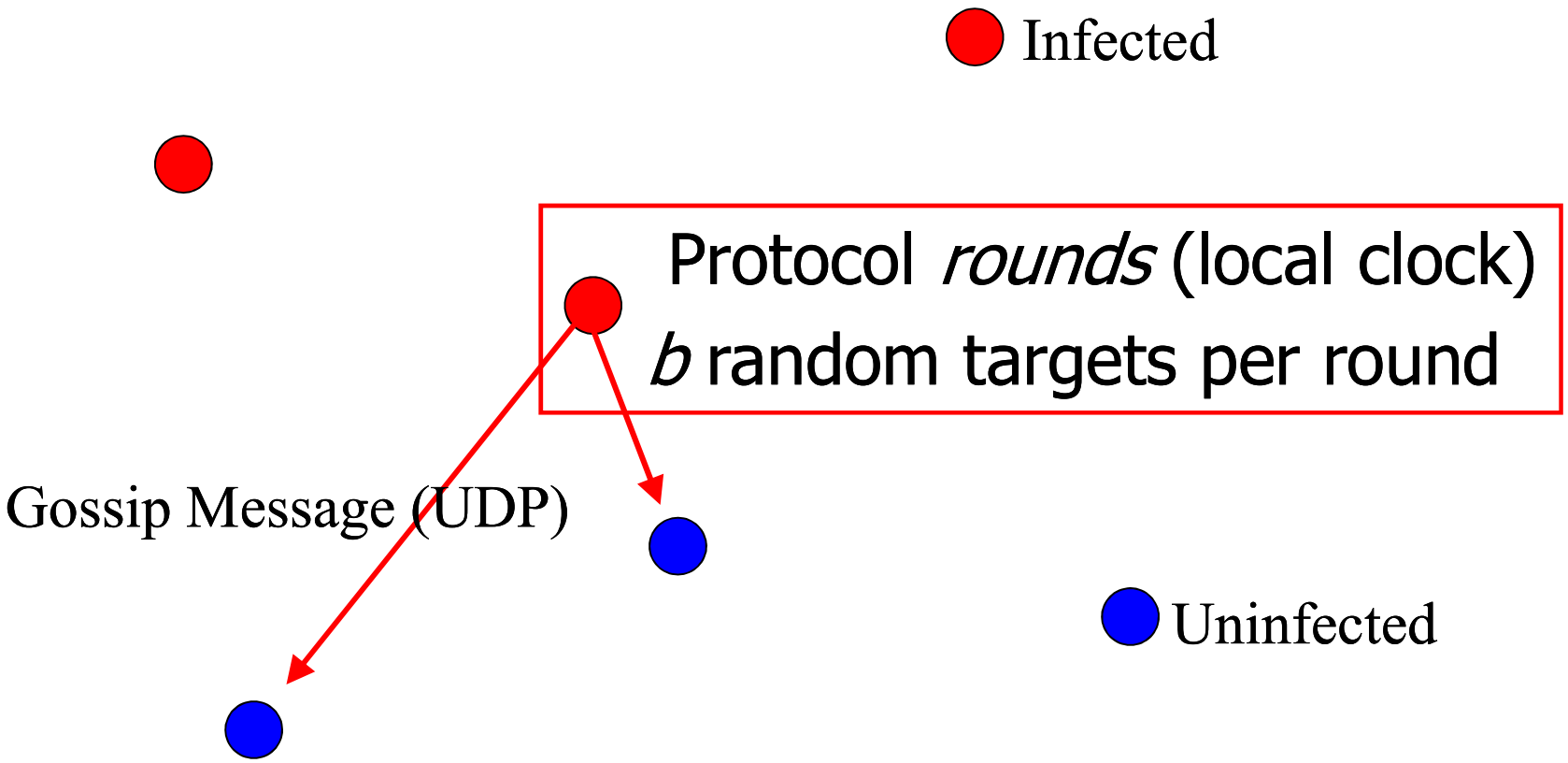
```
% Push-pull version

on timeout
    q ← random(P)
    send ⟨PUSHPULL, p, value⟩
    set timeout Δ

on receive ⟨PUSHPULL, q, v⟩
    if value.time < v.time the
        value ← v
    else if value.time > v.tim
        send ⟨REPLY, value⟩ to

on receive ⟨REPLY, v⟩
```



Second gossip-based mechanism: Rumor mongering

- This model includes a third state, called removed. For example, stop with probability 1/k
- As soon as an new update is available at a single infected node, the update is pushed towards other random nodes.
- Susceptible nodes receiving the update become infected, and start pushing the update as well.
- Eventually, the protocol terminates, when all susceptible and infected nodes have switched to the removed state.

“Gossip” (or “Epidemic”) Multicast



Gossip implementation

- BitTorrent
- PPLive
- Amazon Dynamo
- Facebook Cassandra
- UseNet NNTP

作业

- Paxos协议在分布式系统中实现了哪一级别的一致性？说明原因。
- 在Paxos协议中，什么情况下Proposer需要改变它提案中Value的内容?

END