Consensus and Distributed Transactions

马震

2020-02-07

什么是Consensus（共识）

- Consensus（共识）是分布式计算中**最重要**和**最基本**的问题之一
- 分布式系统可能遇到各种问题
  - 数据包丢失，乱序，重复和延迟
  - 时钟不同步
  - 节点可能在任意时刻暂停或crash
  - 节点故意发送虚假消息，试图欺骗其他节点
- 需要一种抽象的协议，简化分布式系统的开发
- 这个最重要的抽象就是consensus
  - 让分布式系统中的一组节点就某事达成一致

什么是Consensus（共识）

- 设计一个正确的consensus算法非常困难
  - 没有全局变量，没有共享内存，不能在节点间共享状态，唯一可以做的就是从一个节点向另一个节点发送消息
  - Google chubby服务的创建者Mike Burrows说过：all working protocols for asynchronous consensus we have so far encountered have Paxos at their core.
- 应用场景
  - 分布式事务的原子提交
  - Leader election
  - 分布式锁
  - Membership/coordination service
- 几乎所有的分布式系统都基于consensus构建

Consistency(一致性)和Consensus(共识)

- Consistency（**一致性**）在不同的上下文，代表着不同的东西
  - 在事务的上下文中，ACID里的C，对数据的一组特定陈述必须始终成立
  - 在分布式系统的上下文中，例如线性一致性：使多个副本的数据看起来好像只有一个副本（系统保证从任何副本读取到的值都是最新的），且所有操作都以原子的方式生效（一旦某个新值被任一客户端读取到，后续任意读取不会再返回旧值）
- 可以使用Consensus算法来实现分布式一致性(Consistency)
- Consensus有时也翻译成一致性，但翻译成“共识”更准确

Consensus算法正确性要求

- **Agreement** – 所有节点决定相同的值
- **Validity** – 决定的值必须是由其中的某个节点提出
- **Termination** – 所有节点最终都完成决定

Agreement 定义了consensus的核心要求

Validity 用于排除trivial解决方案：不管实际状态如何，都指示每个节点确定默认值

Termination 关注系统容错。即使某些节点发生故障，其他节点也必须做出决定。例如2PC不满足这个要求：如果coordinator失败，则其他参与者无法决定是提交还是中止

分布式算法的两个属性

- **安全性(safety)** - 坏的事情不会发生
- **活性(liveness)** - 某些好的事情最终一定会发生

分布式算法通常要求在所有可能情况下始终保持safety

对于liveness则有限制条件：例如只有在大多数节点没有崩溃，且网络最终从中断中恢复时

2PC满足safety，不会有坏的数据被写入到数据库。但是它的liveness并不好：如果事务管理器在某个错误的点上挂掉，那么协议会被阻塞

System Model

- 算法的实现不能过于依赖于运行它的硬件和软件细节
- Consensus算法要容忍分布式系统的各种故障，所以需要以某种方式将系统中发生的各种故障进行形式化
- 我们定义一个**系统模型**，它是对算法在某方面假设条件的抽象

System Model – 时间假设

- **Asynchronous model** -进程间完全隔离，只能用消息传递通信。发送的消息在有限时间内到达对方，但这个“有限的时间”无法给出上界，同时进程以未知的任意的速率运行
- **Synchronous model** - 消息传输和进程执行都有时间上界
- **Partially synchronous model** -大多数情况下行为类似于同步模型，但有时会有超出上界的网络延迟，进程暂停

System Model – 节点故障

- **Crash-stop -**节点可能随时崩溃，此后该节点永远消失不再恢复
- **Crash-recovery** - 节点可能随时崩溃，并且可能在未知的时间后从故障中恢复并继续执行
- **Byzantine faults** - 节点可以做任何事情，可能以完全任意的方式背离协议，包括试图欺骗其他节点
  - 拜占庭将军问题：一组将军分别指挥一部分军队，每一个将军都不知道其它将军是否是可靠的，也不知道其他将军传递的信息是否可靠，但是他们需要通过投票选择是否要进攻或者撤退

Byzantine faults

- 确定性算法
  - 一旦达成对某个结果的共识就不可逆转，即共识是最终结果
- 概率类算法
  - 共识结果则是临时的，随着时间推移或某种强化，共识结果被推翻的概率越来越小，成为事实上的最终结果

Byzantine faults – 确定性算法

- The Byzantine Generals Problem (1982)
  - 在Synchronous网络环境中，拜占庭将军问题存在解。当超过 1/2 的节点为恶意节点时，会有一些限制条件。协议解代价很高，需要大量的多轮消息传递。最早应用于航空、航天。
- Consensus in the Presence of Partial Synchrony
  - 在Partially Synchronous网络环境中，协议可以容忍最多 1/3 的拜占庭故障
- 在Asynchronous网络环境中，具确定性质的协议无法容忍任何错误

Byzantine faults – 概率类算法

- Pricing via Processing or Combatting Junk Mail (1992)
  - 工作量证明（PoW，Proof-of-Work）是一个用于阻止拒绝服务攻击、垃圾邮件等服务错误问题的协议
  - 用消耗对手能源的手段来阻止一些恶意的攻击，或是像垃圾邮件这样的对服务的滥用
  - 应用在区块链中，例如比特币，以太坊都使用了PoW

System Model

- Partially Synchronous 更接近真实系统
- 如果IDC中所有节点可控，没有Byzantine fault问题
- 容忍轻度“撒谎”还是有价值的
  - 硬件故障引起的错误数据，配置错误等
  - 应用层协议的checksums
  - NTP客户端可以配置有多个服务器地址
- 本文主要讨论容忍**非拜占庭故障**的共识算法

FLP result

- Fischer，Lynch和Patterson于1985年4月发表论文 Impossibility of distributed consensus with one faulty process
  - 这篇只有6页不到的论文，证明了一个分布式系统领域最重要的结论。这个著名的结论被称为FLP result或者FLP impossibility
- **FLP result**：在异步环境中，只要一个节点有崩溃的风险，那么就不存在一个协议，能保证有限时间内使所有节点达成共识(consensus)
  - 问题的核心在于，你无法区分一个进程到底是终止了，还是正在以极低的速度执行，这使得在异步系统中的错误处理几乎是不可能的
- FLP result确定了在异步环境中，分布式系统可以实现的目标的上限

Consensus算法简史

- JimGray在“Notes on Database Operating Systems”(1979)中描述了两阶段提交(2PC)
  - 如果事务管理器在某个错误的时间点失败的话，2PC就会被阻塞
- Dale Skeen在“NonBlocking Commit Protocols”(1981)中指出：对于一个分布式系统，需要3阶段的提交算法来避免2PC中的阻塞问题
  - 3PC的故障模型crash-stop，在某些网络条件下，crash-recovery会导致3PC错误
- Lamport在“The Part-Time Parliament” (submitted in 1990, published 1998)中提出了Paxos算法
  - Paxos是一个异步算法，没有显式的超时设置。然而只有当系统表现出同步的方式时，它才能达到consensus
  - 根据FLP result，Paxos在某些情况下无法达到consensus

Consensus算法简史

- Butler Lampson 发表 How to Build a Highly Availability System using Consensus (1996)
  - Lamport的Paxos论文在1990年就提交ACM，由于采用希腊民主议会的比喻，当时没有人理解他的算法
  - 1996年微软的Butler Lampson在论文中重新提到Paxos
- Lamport又发表了Paxos Made Simple (2001)
  - 用简单易懂的语言重述了Paxos
- Lamport和Jim Gray 发表 Consensus on Transaction Commit (2005)
  - 论文证明2PC是Paxos的退化版本
  - 使用Paxos来对2PC中的事务管理器进行高效的备份
  - 没有使用Paxos算法来直接解决事务提交问题，而是用来让系统容错

Consensus算法简史

- Google The Chubby lock service for loosely-coupled distributed systems (2006)
  - Google的分布式锁服务Chubby底层以Paxos算法作为基础
  - 填补了Paxos基本算法和工程实践之间的空白
- ZooKeeper (Zab) and etcd (Raft)
- Lamport因发明Paxos算法获得了2013年度图灵奖

Transaction

- Transaction是应用程序将多个读取和写入”打包”到一个逻辑单元
- 事务是一层抽象，简化了应用程序模型，应用可以忽略部分失败和并发问题
- ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Transaction - Atomicity

- 事务作为一个整体被执行，包含在其中的所有操作要么全部被执行，要么都不执行
- Atomicity简化了应用处理部分失败的情况：如果事务中止，这个事务的所有写入都可以被丢弃，应用程序可以确保没有做任何更改，因此可以安全地重试
- ACID中的Atomicity和多线程编程中的Atomicity含义不一样，和并发无关，不是用来描述多个并发进程访问同一个数据的场景。并发访问的场景由Isolation来描述

Transaction - Consistency

- 确保数据库的状态从一个一致状态转变为另一个一致状态
- ACID中的Consistency是一个应用程序相关的概念，所以需要由应用程序自己来保证，例如账户不能为负数。
- Consistency会保证事务前后，数据的状态都是“好”的。
- 应用程序依靠数据库的Atomicity和Isolation来达到Consistency ，并不是由数据库单独完成的

Transaction - Isolation

- 多个事务并发执行时，一个事务的执行不应影响其他事务的执行
- 经典的数据库书把隔离性称为“可串行化”，每个事务都假装他们是唯一正在数据库上运行的事务
- 隔离级别
  - read uncommitted
  - read committed
  - repeatable read
  - *Serializable snapshot isolation (oracle， PostgreSQL)*
  - serializable

Transaction - Durability

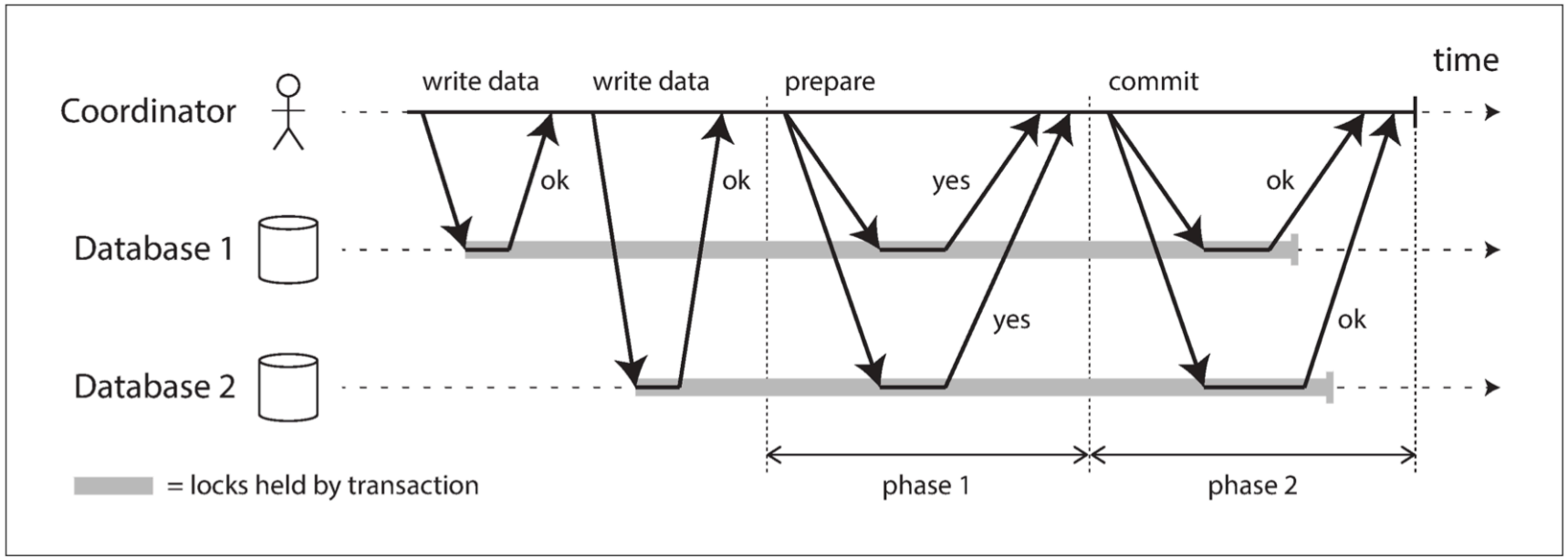
- 已被提交的事务对数据库的修改应该永久保存在数据库中。在事务结束时，此操作将不可逆转
- 持久性是一种承诺，即一旦事务成功提交，即使存在硬件故障或数据库崩溃，也不会丢失已写入的任何数据

Transaction – 实现

- 对于单节点的数据库，通常由存储引擎实现
- 一般通过write-ahead log和事务日志完成
  - 如果数据库在事务过程中崩溃，则当节点重新启动时，事务从日志中恢复
  - 如果在崩溃之前成功将commit记录写入磁盘，则事务被视为已提交
  - 否则，则回滚该事务的所有写操作
- 事务的提交与否仅依赖单个设备（某个特定磁盘的controller）

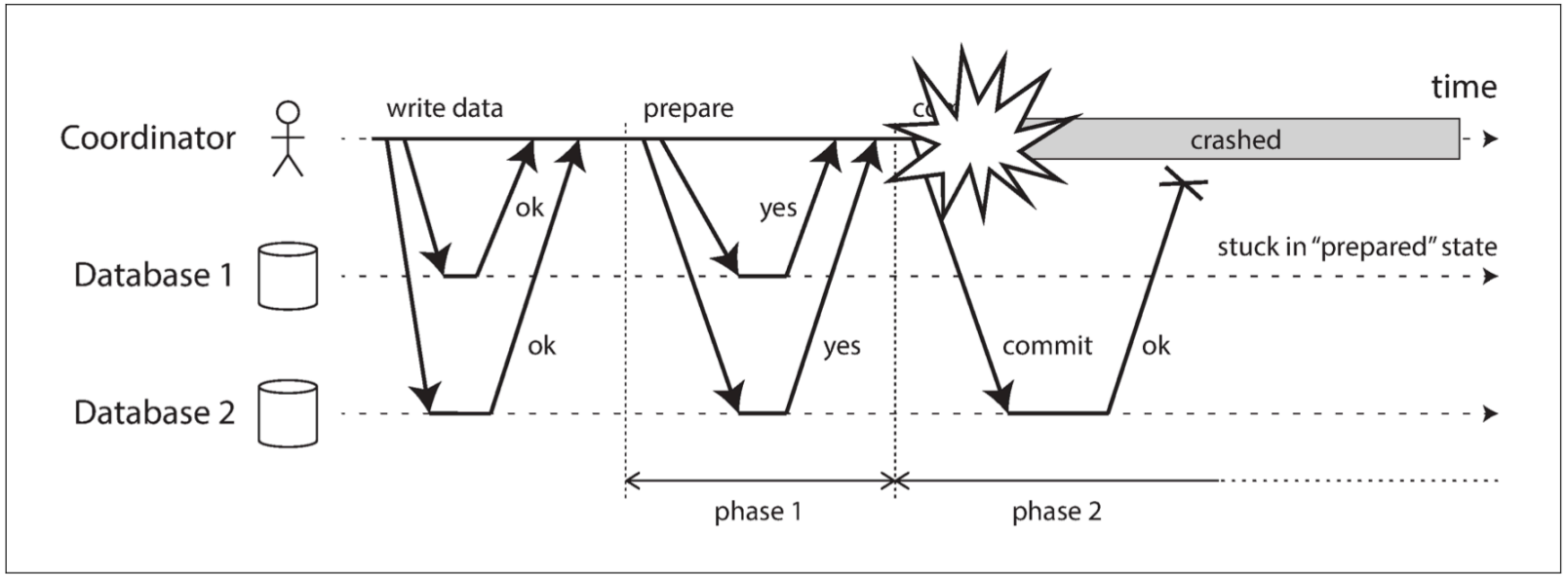
Two-Phase Commit (2PC)

- 2PC是一种实现跨多个节点的atomic commit的算法



- 2PC使用了通常不在单节点事务中出现的新组件： coordinator （也称为transaction manager ）
- 在没有故障的情况下，2PC确实是一种满足要求的Consensus算法
  - Agreement, Validity, Termination

Two-Phase Commit (2PC)



- 如果coordinator在收到参与者的Yes后挂掉，此时参与者只能等待，不能再单方面中止
- 如上图， coordinator实际的决定是commit，它在只发送了commit请求给DB2，还没发commit给DB1时挂掉。DB1不知道该提交还是终止，即使设置了超时也没用
- 此时协议被blocked，只能等待coordinator恢复。 2PC满足安全性(safety) ，不会有坏的数据被写入到数据库，但是它的活性(liveness)并不好

Two-Phase Commit (2PC)

- Coordinator需要在持久性存储中记录协议状态
  - 本地文件
  - 强一致性的分布式存储（基于Paxos/raft实现）
- 如何恢复coordinator
  - 手工恢复
  - 由其他节点自动take over
    - 如何判断节点是否挂了，又是一个consensus问题
    - 简单的心跳检测可能会有脑裂问题
    - 引入ZooKeeper之类的仲裁者

Two-Phase Commit (2PC)

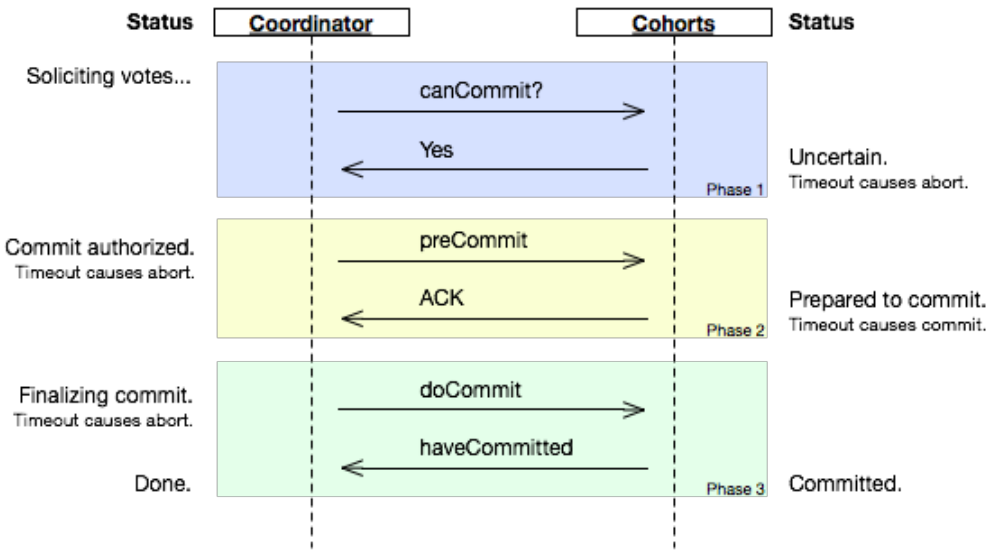
- 事务处于未决状态，数据库怎么办？
  - 因为事务没有结束，数据库不能释放参与事务的数据行锁
  - 正确的2PC实现必须保留未决事务的锁，即使数据库重启
  - 由数据库管理员人工处理 - heuristic decisions (参与者单方面决定提交或回滚未决事务)
    - javax.transaction.HeuristicCommitException & javax.transaction.HeuristicRollbackException 的由来
    - RM自作主张提交/回滚，等TM恢复后，又给RM发送和之前决策相反的指令

Two-Phase Commit (2PC)

- 本质问题分析
  - 2PC为了达到consensus，实际上是退化到由单节点来实现atomic commit
  - 在phase 2阶段， coordinator向所有参与者发送commit/abort前，这时只有它知道当前事务的结果
  - 如果另一个参与者在coordinator恢复前挂掉，这时即使coordinator回归，协议也不能恢复。更糟的情景是，coordinator自己也是参与者
- 2PC被称为blocking atomic commit，3PC解决2PC的阻塞问题，但会增加消息延迟

Three-Phase commit (3PC)

- 将2PC的phase 2分成两个sub-phases
  - ‘prepare to commit’ - 该阶段的目的是将投票结果传达给每个参与者，以便无论哪个节点挂了，都可以恢复协议的状态
  - ‘commit or abort’ - 和2PC的phase 2几乎完全相同
- 如果coordinator挂掉，其他节点可以接管事务，并从剩余参与者中查询事务状态



Three-Phase commit (3PC)

- 3PC的问题
  - Coordinator在发出‘prepare to commit’后，还没收到参与者的响应就遇到故障，这时另一个节点接管，它询问完所有参与者，然后指示它们提交事务。在这时候，发生故障的coordinator从中断的位置恢复运行，它注意到还没有收到参与者对‘prepare to commit’的响应，并且已经超时，所以它会发送回滚事务的消息，和接管节点的commit指示相冲突
- 问题的本质
  - 在asynchronous model（消息传输和程序执行时间没有上限）下，没法判断节点是完全挂了，还是load高无法及时响应，等它缓过来又会扰乱系统状态
  - 3PC适合的System mode是**synchronous + crash-stop**

Paxos

- Paxos是第一个被证明在异步网络环境下正确的Consensus算法
- 特性
  - 基于消息传递，允许消息传输的丢失，重复，乱序，但是不允许消息被攥改（非拜占庭故障）
  - 在节点数少于半数失效的情况下，仍然能正常的工作
  - 节点失效可以在任何时候发生，不影响算法正常执行

Paxos

- 可是， FLP result不是说，异步环境下，即使只有一个节点失败，也没有一种确定性的Consensus算法
- Paxos可以容忍异步性，在网络恢复正常时，或者说表现出同步性时，才能最终达到Consensus
- 在一些场景下，Paxos会一直得不到结果，著名的“live lock”问题
- Paxos满足了safety，不满足liveness

Paxos – 三种角色

- Proposer
  - Proposer 是proposal（提议）的发起者。Paxos的目标就是在一群proposer发出的proposal（提议）中确定一个唯一的proposal，然后在各个acceptor中达成共识
- Acceptor
  - acceptor是proposal的接受者，acceptor参与投票，但不参与决策
- Learner
  - leaner被动接受已经达成共识的proposal。不参与决策和投票

Paxos中一个节点可以同时担任这三个角色

Paxos - Sequence numbers

- 每个proposal都关联一个sequence number，该sequence number全局唯一，单调递增
- sequence number用于对proposal进行排序，让acceptor能判断出proposal的先后
- sequence number由proposer生成，实践中可以用(Timestamp，address，seq. number)构建

Paxos - 阶段一

- 阶段1a - Prepare（预定Proposal序列号）
  - proposer 拿到某个Client的请求Value后，生成一个proposal的sequence number，然后将sequence number发送给所有Acceptor
- 阶段1b---Respond with Promise
  - 每个Acceptor收到Proposal序列号n后，先检查之前是否响应过序列号更高的Proposal
    - 若没有，那么就给出Response，这个Response带有自己已经Accept的序列号最高的Proposal（若还没Accept任何Proposal，回复null）
    - 否则，发送拒绝的Respond
  - Acceptor承诺不会Accept任何proposal id <= n的 prepare请求
  - Acceptor承诺不会Accept任何proposal id < n的 accept请求

Paxos - 阶段二

- 阶段2a---发起Proposal，请求Accept
  - Prepare如果得到了来自超过半数的Acceptor的Response，那么就有资格向Acceptor发起Proposal<proposal_ID,value>
    - proposal_ID是阶段1a中发送的序列号
    - value是收到的Response中序列号最大的Proposal的Value。若收到的Response全部是null，那么Value自定义，可直接选Client请求的Value
- 阶段2b--Accept Proposal
  - Acceptor检查收到的Proposal的序列号是否违反阶段1b的Promise，若不违反，则Accept收到的Proposal。否则拒绝该次请求。

Paxos

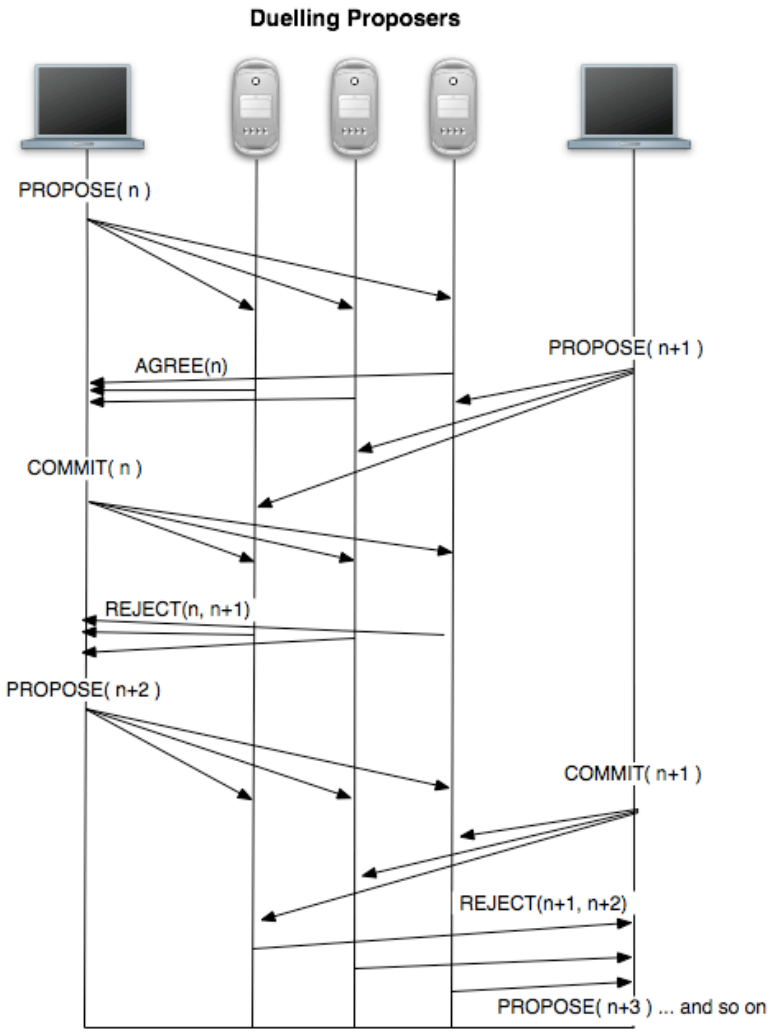
- **Paxos的核心是，给定固定数目的节点，任何一个多数者集合与其他的多数者集合必然至少存在一个公共元素**
- 如果proposer失败，安排另一个proposer接管，然后发布新的proposal。在将来某一时刻，自从某个Proposal被超过半数的Acceptor Accept后，之后Accept的Proposal Value一定和之前被Accept的Proposal Value相同

Paxos – 一个简化的例子

- A,B两个proposer，C,D,E三个acceptor
- A发起的proposal id=3，value=n 被D，E接受，但A这时候挂了
- B携带proposal id=4，发起prepare，然后收到C返回(proposal id=2, value=m) ，D返回(proposal id=3, value=n)
- B从多个Acceptor的返回中选择proposal id最大的value，提交自己的proposal(proposal id=3, value=n)
- value = n 之前已经被超过半数的acceptor接收，后续的proposal如果成功，那么确定的value一定是n

Paxos - live lock

- 最终Paxos将在网络稳定后正确执行
- 一个Proposer只需back off足够长的时间，以使其他Proposer能够完成它的proposal

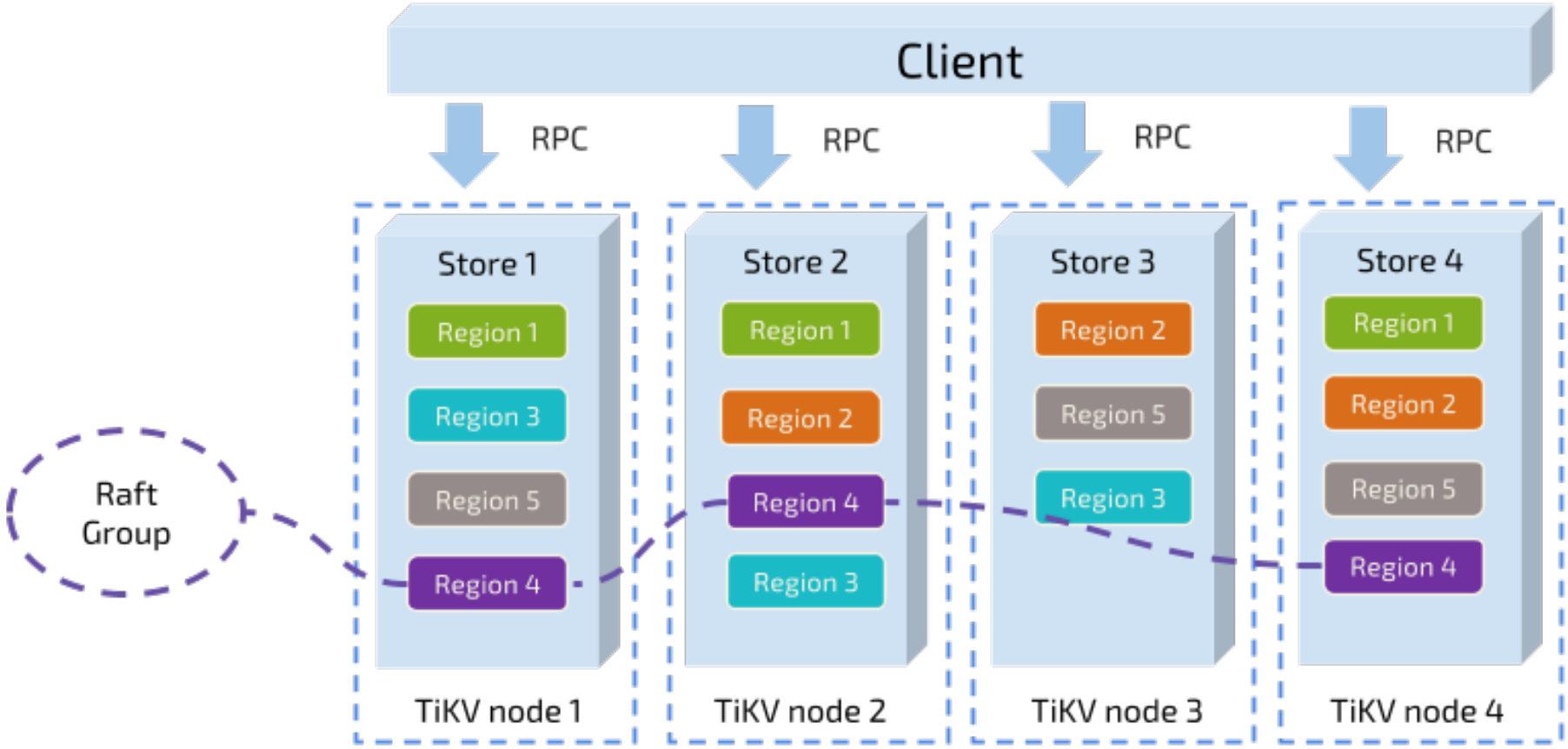


Multi-Paxos

- Paxos算法最终目标是确定一个值，称为一个Paxos instance
- 而我们常用的场景都是确定多个值，比如数据库的WAL
  - 引入Multi-Paxos。Multi-Paxos就是多个Paxos instance
  - 不同 Logid 标识的日志都是一个个相互独立的 Paxos Instance，每条日志独立执行完整的 Paxos 两阶段协议

Multi-Paxos

- TiKV : A distributed transactional key-value database



利用Paxos改进2PC

- Consensus on Transaction Commit (2005)
  - 对于事务中涉及的每个RM使用一个Paxos实例来决定该RM是否提交该事务
  - 对于没有错误发生的情况下，Paxos提交可以通过两个阶段完成
- Percolater和MVCC
  - Percolater是一个经过优化的二阶段提交
  - 将2PC的状态存储在Paxos（或Raft）提供的高可用数据副本中，避免原有高可用的问题
- 事务日志的高可用
  - 将事务日志保存在外部存储中，强一致性的分布式存储或数据库
  - 使用ZooKeeper或etcd选主

总结

- Consensus算法正确性要求：Agreement, Validity, Termination
- 分布式算法的两个属性：安全性(safety)，活性(liveness)
- System Model
  - Asynchronous model ,Synchronous model
  - Crash-stop, Crash-recovery, Byzantine faults
- FLP result
- Transaction ACID 特性
- 2PC协议及其问题
- 3PC协议及其问题
- Paxos