共识

赵来平
天津大学软件学院

容错的全序广播

- 全序广播对于状态机复制非常有用。
- 一种简单的实现方式是：把所有消息都通过**单个主节点（leader）**来发送。

容错的全序广播

- **全序广播对于状态机复制非常有用。**
- **一种简单的实现方式是：把所有消息都通过**单个主节点（leader）**来发送。**
- **问题是：如果这个 leader 宕机或不可用了怎么办？**
  - ▶️ **人工故障切换（manual failover）**：
    由人工运维人员选出一个新的 leader，并把每个节点都重新配置为使用新的 leader。
    很多数据库都在用这种方式！在**计划内维护**场景下还不错。
    但一旦是**突发故障**，人很慢，系统可能要等很久才能恢复……
  - ▶️ 那么，我们能不能**自动选出一个新的 leader 呢？**

共识与全序广播

- **若干节点需要就某个单一的值达成一致。**
  - 在 全序广播 的语境下，这个“值”就是：下一条要投递的消息。
  - 一旦有一个节点对某个消息顺序作出了决定，所有节点最终都必须对同样的消息顺序作出相同的决定。
  - 从形式化角度看，共识和全序广播是等价的。

原子提交 vs 共识

原子提交	共识
每个节点**投票**决定是 *提交* 还是 *中止*（abort）。	一个或多个节点会**提出一个值**（proposal）。

原子提交 vs 共识

原子提交	共识
每个节点**投票**决定是 *提交* 还是 *中止*（abort）。	一个或多个节点会**提出一个值**（proposal）。
如果**所有节点**都投票提交，则必须提交；如果有 **≥1 个节点投票中止**，则必须中止。	在所有被提出的值中，**任意一个**都可以被选为最终决定值。

原子提交 vs 共识

原子提交	共识
每个节点**投票**决定是 *提交* 还是 *中止*（abort）。	一个或多个节点会**提出一个值**（proposal）。
如果**所有节点**都投票提交，则必须提交；如果有 **≥1 个节点投票中止**，则必须中止。	在所有被提出的值中，**任意一个**都可以被选为最终决定值。
如果某个参与节点发生崩溃，则**必须中止。**	即使有节点崩溃，只要仍有一个**法定多数（quorum）节点正常工作**，系统就**仍然可以达成决定。**

全序广播行为约束

- **全序广播需要满足的要求可以分为两类：**
- ▶️ **安全性（Safety）：“坏事不会发生”**
  1. 设有两个节点 (N_1) 和 (N_2)，它们都交付了两条消息 (m_1) 和 (m_2)。若 (N_1) 在交付 (m_2) 之前交付了 (m_1)，那么 (N_2) 也必须在交付 (m_2) 之前交付 (m_1)。
  2. 如果某个节点交付了一条消息 (m)，那么这条消息 (m) 之前一定是由某个节点广播过的。
  3. 任一节点对同一条消息不会交付超过一次。

全序广播行为约束

- **全序广播（total order broadcast）的性质可以分为两大类：**
- **安全性（Safety）：“坏事不会发生”**

1. 设有两个节点 (N_1) 和 (N_2)，它们都交付了两条消息 (m_1) 和 (m_2)。若 (N_1) 在交付 (m_2) 之前交付了 (m_1)，那么 (N_2) 也必须在交付 (m_2) 之前交付 (m_1)。
2. 如果某个节点交付了一条消息 (m)，那么这条消息 (m) 之前一定是由某个节点广播过的。
3. 任一节点对同一条消息不会交付超过一次。

- **最终活性（Eventual Liveness）：“好事最终会发生”**

4. 如果某个节点广播了一条消息 (m)，并且没有崩溃，那么该节点最终一定会交付这条消息 (m)。
5. 如果某个节点交付了一条消息 (m)，那么任何没有崩溃的其他节点也最终都会交付这条消息 (m)。

选举问题

- 给定一组处理节点，选择一个*Leader* 去完成一些特定任务:
  - 组中的所有节点都知道该leader。
- 如果leader节点宕机：
  - 其他节点会检测到leader死亡（使用故障检测算法！）
  - 然后…？
- 本节重点：*选举算法。*目标：
  - 1. 从未发生故障的剩余节点中选择新leader。
  - 2. 所有的未发生故障节点最终就新leader达成一致。

系统模型

- *N* 个处理节点.
- 每个节点都有一个单独的 *id*.
- 消息最终会被交付到目的地，即使花费较长时间。*eventually delivered*.
- 选举协议在执行期间也可能会发生新的节点故障。

启动选举

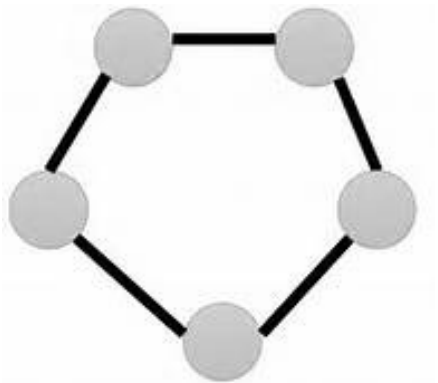
- 任何一个节点都可启动选举协议。
- 任何时候，每个节点都只能启动最多一次选举协议。
- 多个节点可能同时各自启动选举协议：
  - 但是，新leader必须只能有一个。
- 选举结果与协议发起节点无关。

选举问题

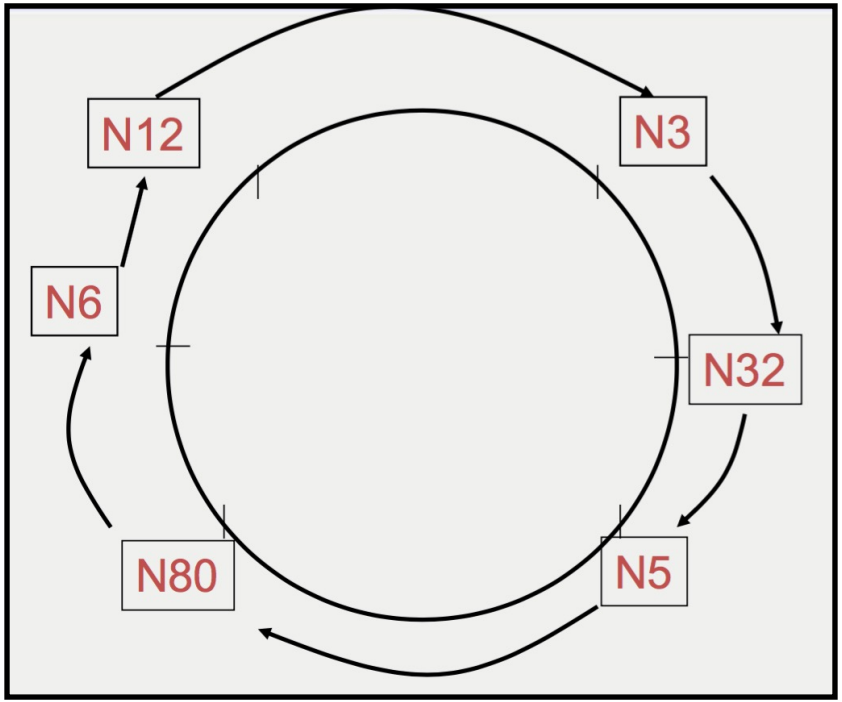
- **选举协议结束，属性表现优秀的正常节点被选为leader**
  - 常用属性：leader的id最大。
  - 其它可用属性：leader的IP地址最大，或者CPU最快，或者存储空间最大，或者文件存储量最多，等等。

环选举

- **N 个节点组织成环状结构：**
  - 第i节点*pi* 可以向节点 *p(i+1) mod N*发送消息。
  - 所有的消息按照顺时针顺序发送。



环



环选举协议

- 协议开始：任意节点pi如果发现旧的leader已经死掉，就发起选举协议，并向其下一节点发送 “*Election*”消息，消息中含有pi的id:attr。

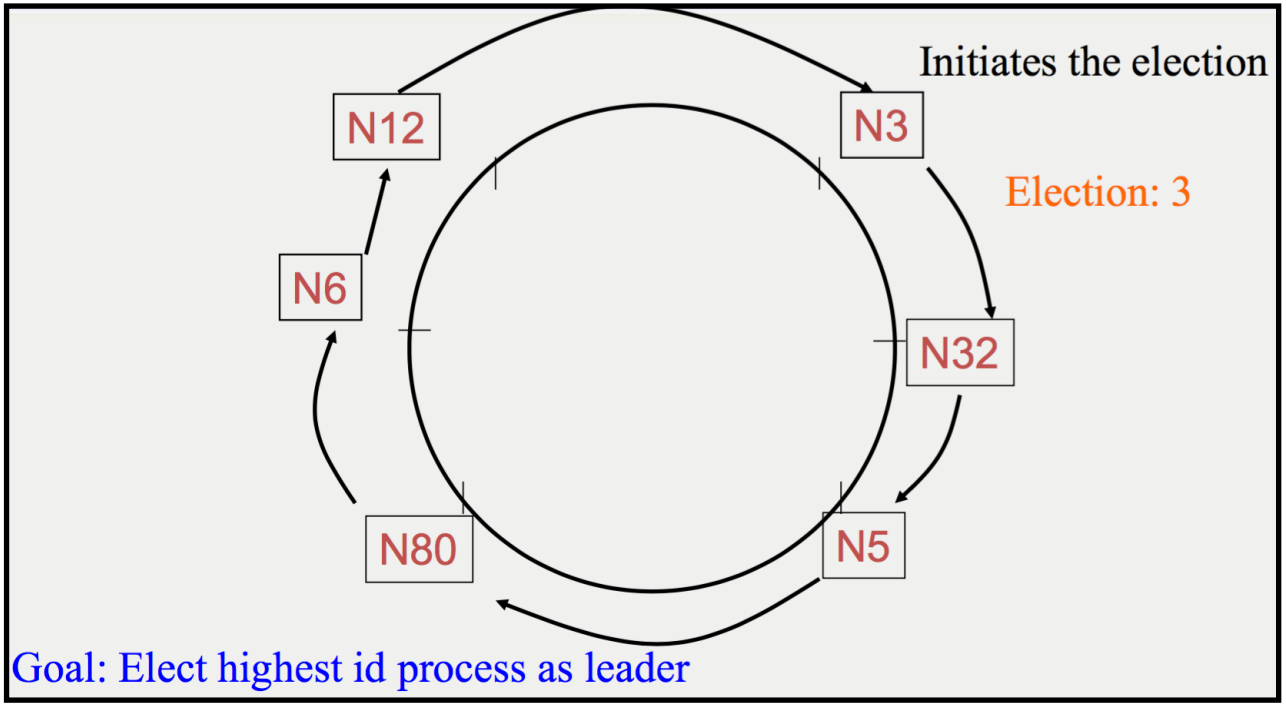
环选举协议

- **协议开始：任意节点pi如果发现旧的leader已经死掉，就发起选举协议，并向其下一节点发送 “*Election*”消息，消息中含有pi的id:attr。**
- **当节点pi收到一条 “*Election*”消息时，将消息中的id:attr与自己的id:attr比较：**
  - 如果消息中的attr更大，pi只转发该消息到下一节点。
  - 如果消息中的attr小，并且pi在之前并没有转发过该消息，那么它修改消息体中的attr值为自己的attr，并转发。
  - 如果消息中的id:attr正是自己本身的id:attr，那么pi的attr一定是最大的（为什么？），pi成为新的leader，并向下发出带有自己的id的“*Elected*”消息，宣布选举结果。

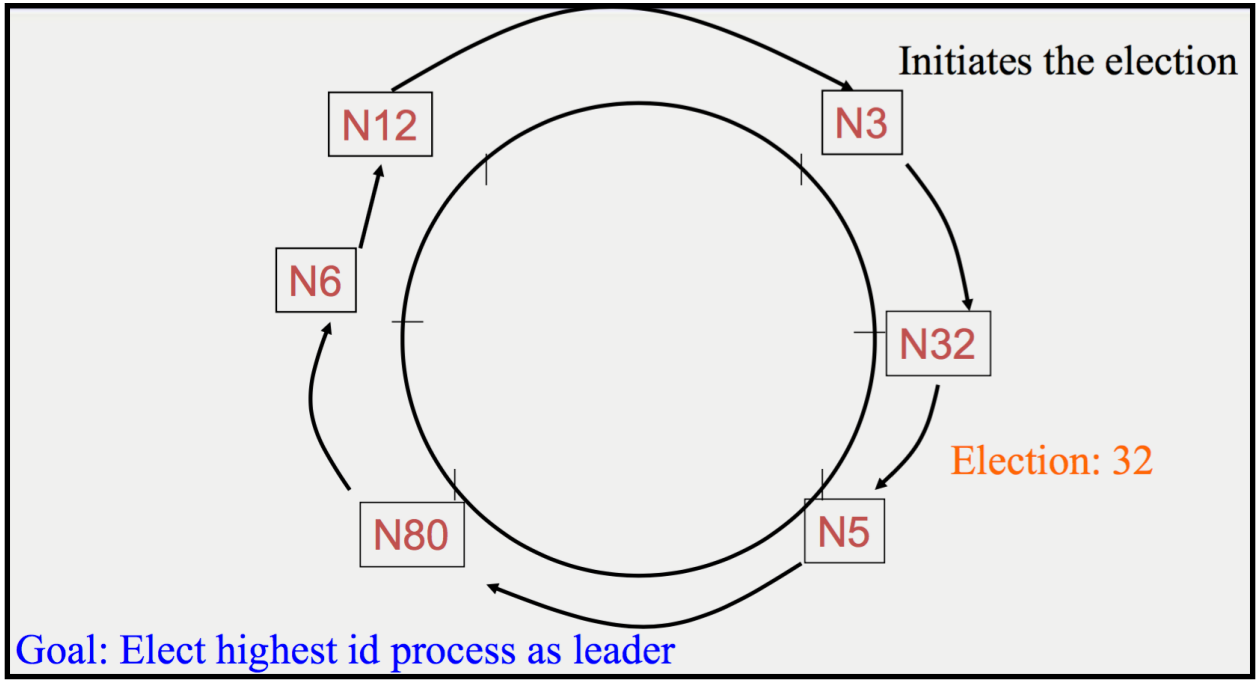
环选举协议

- **当节点pi收到一条“*Elected*” 消息时：**
  - 设置本地私有变量$elected_i$ <— 消息中所带id。
  - 转发该消息（如果pi不是新leader时）。

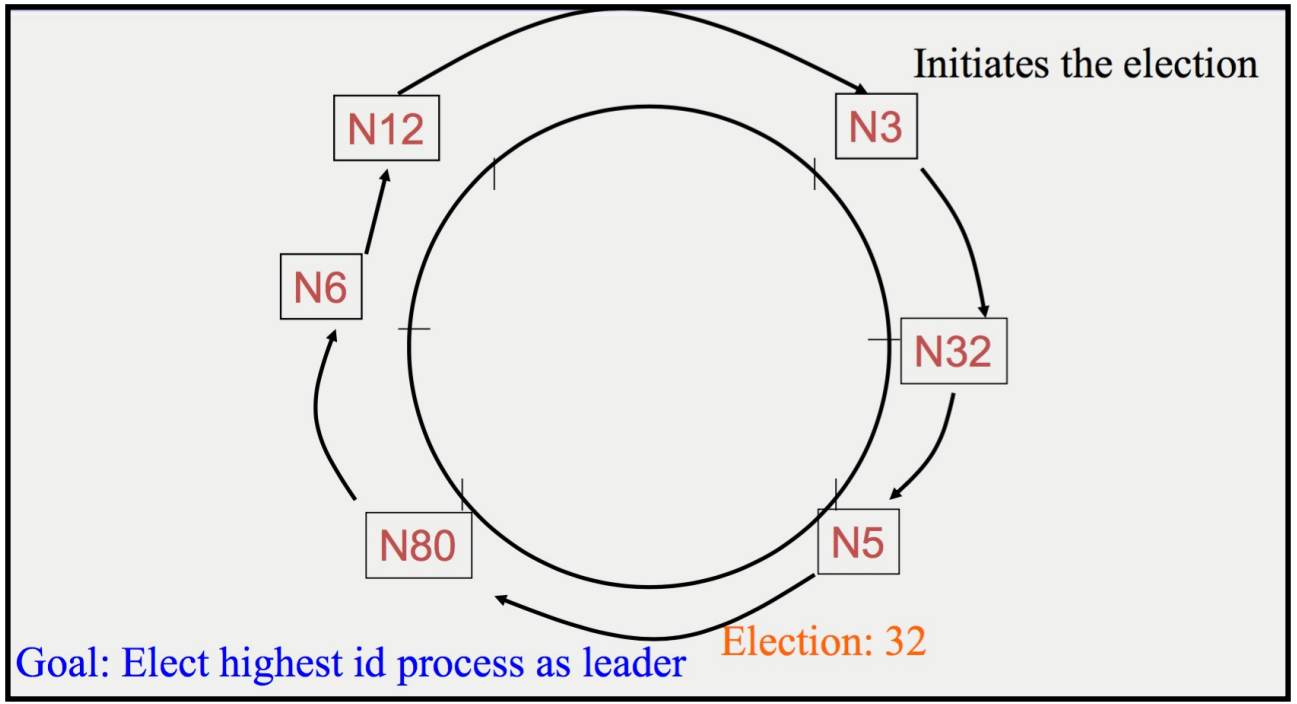
举例


Initiates the election
N12
N3
Election: 3
N6
N32
N80
N5
Goal: Elect highest id process as leader

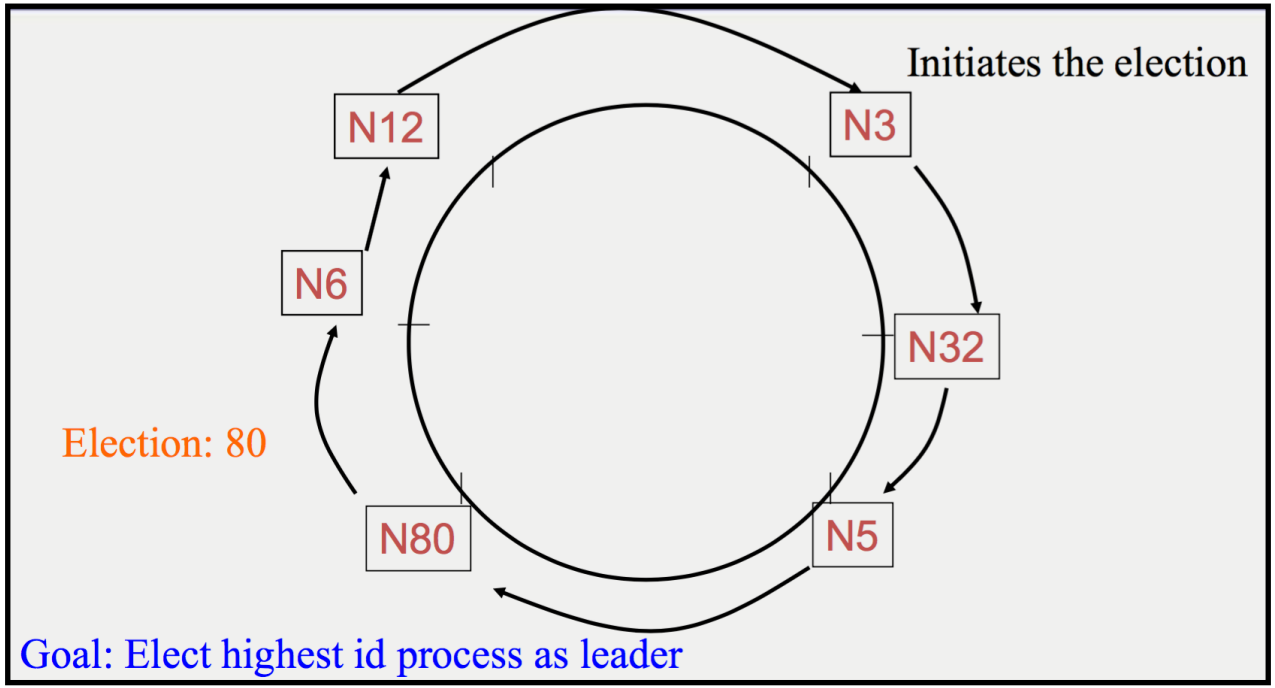
举例


Initiates the election
N12
N3
N6
N32
Election: 32
N80
N5
Goal: Elect highest id process as leader

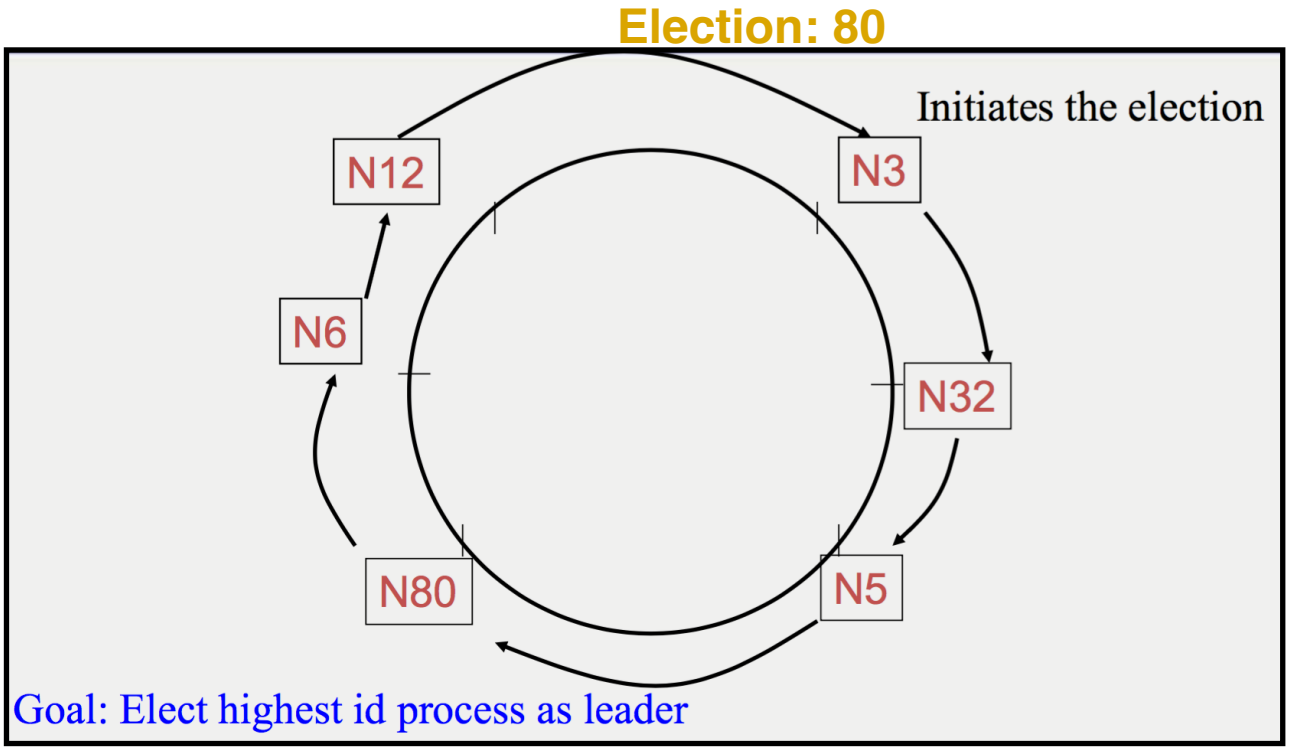
举例


Initiates the election
N12
N3
N6
N32
N80
N5
Election: 32
Goal: Elect highest id process as leader

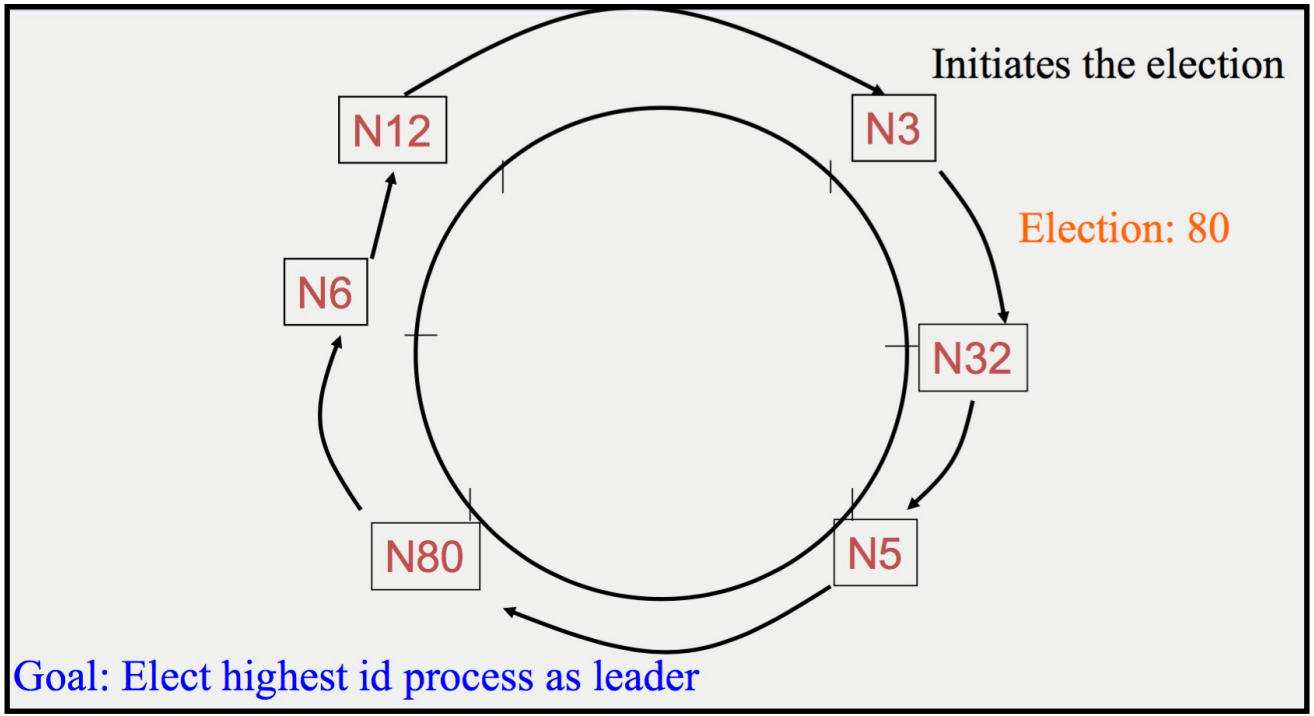
举例


Initiates the election
N12
N3
N6
N32
Election: 80
N80
N5
Goal: Elect highest id process as leader

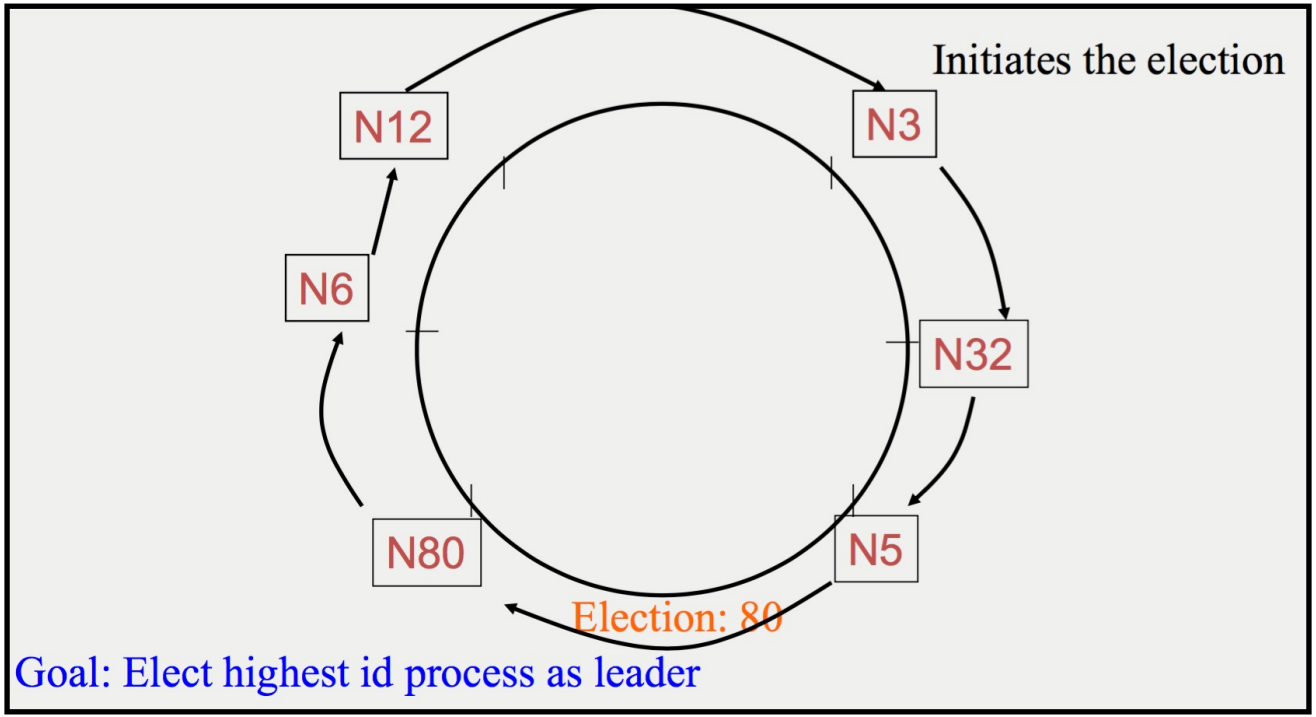
举例


Election: 80
Initiates the election
N12
N3
N6
N32
N80
N5
Goal: Elect highest id process as leader

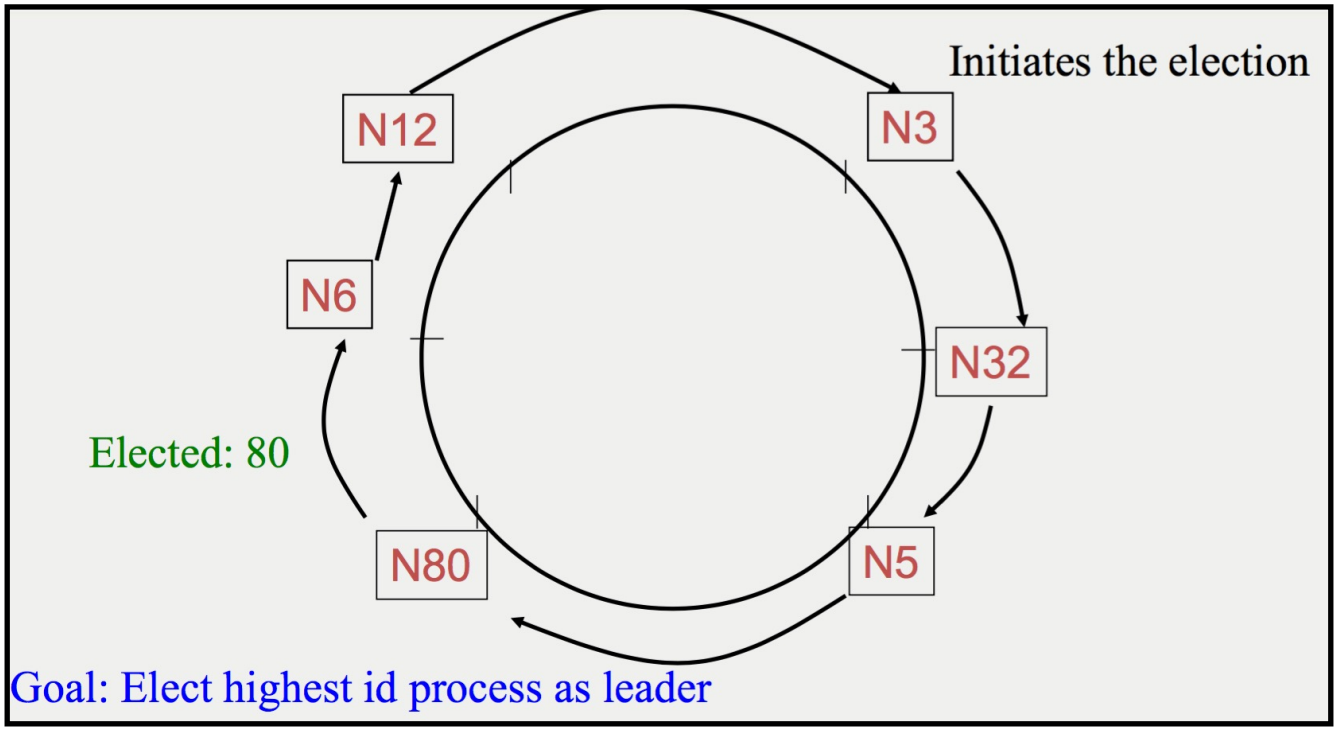
举例


Initiates the election
N12
N3
Election: 80
N6
N32
N80
N5
Goal: Elect highest id process as leader

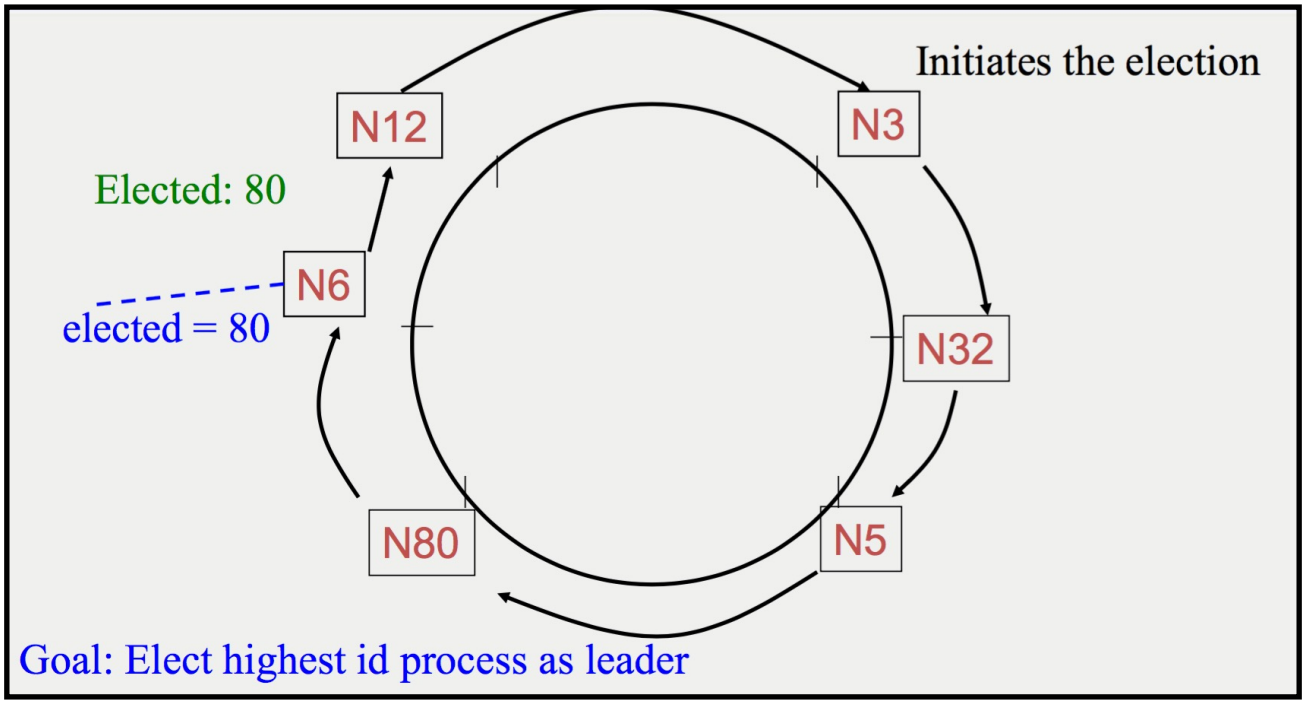
举例


Initiates the election
N12
N3
N6
N32
N80
N5
Election: 80
Goal: Elect highest id process as leader

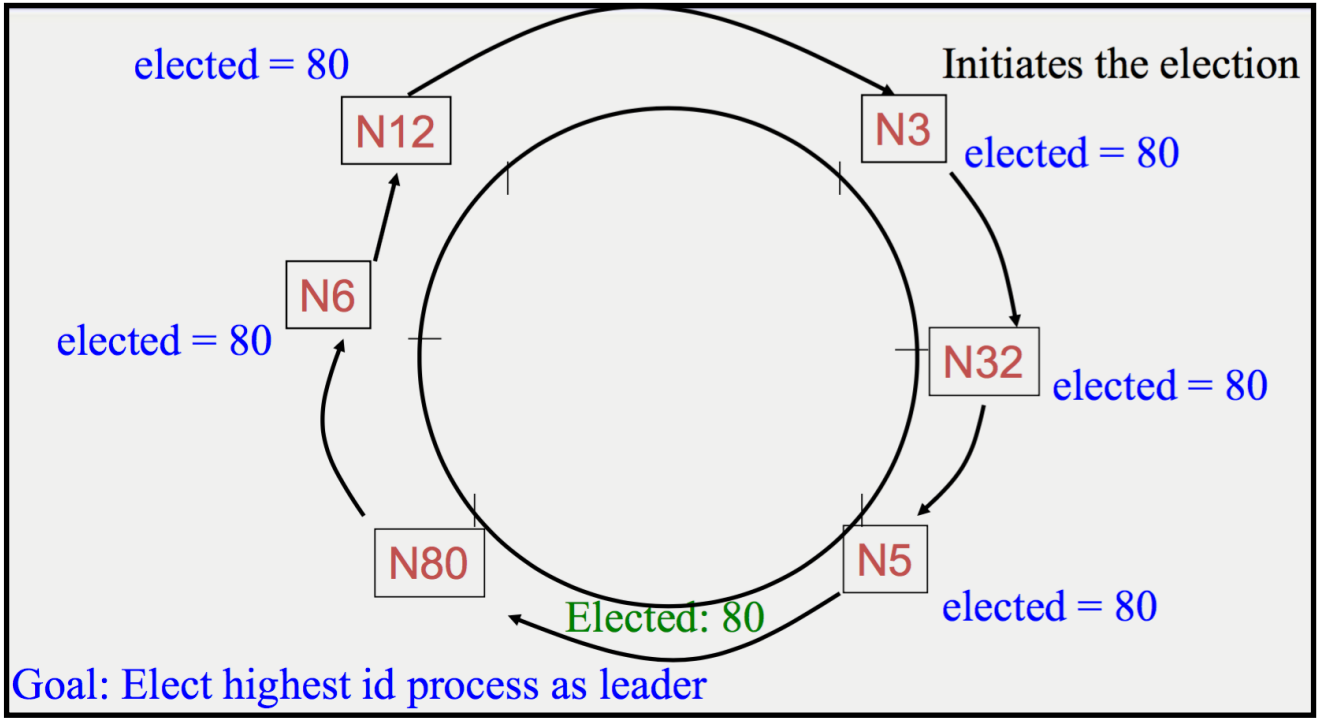
举例


Initiates the election
N12
N3
N6
N32
Elected: 80
N80
N5
Goal: Elect highest id process as leader

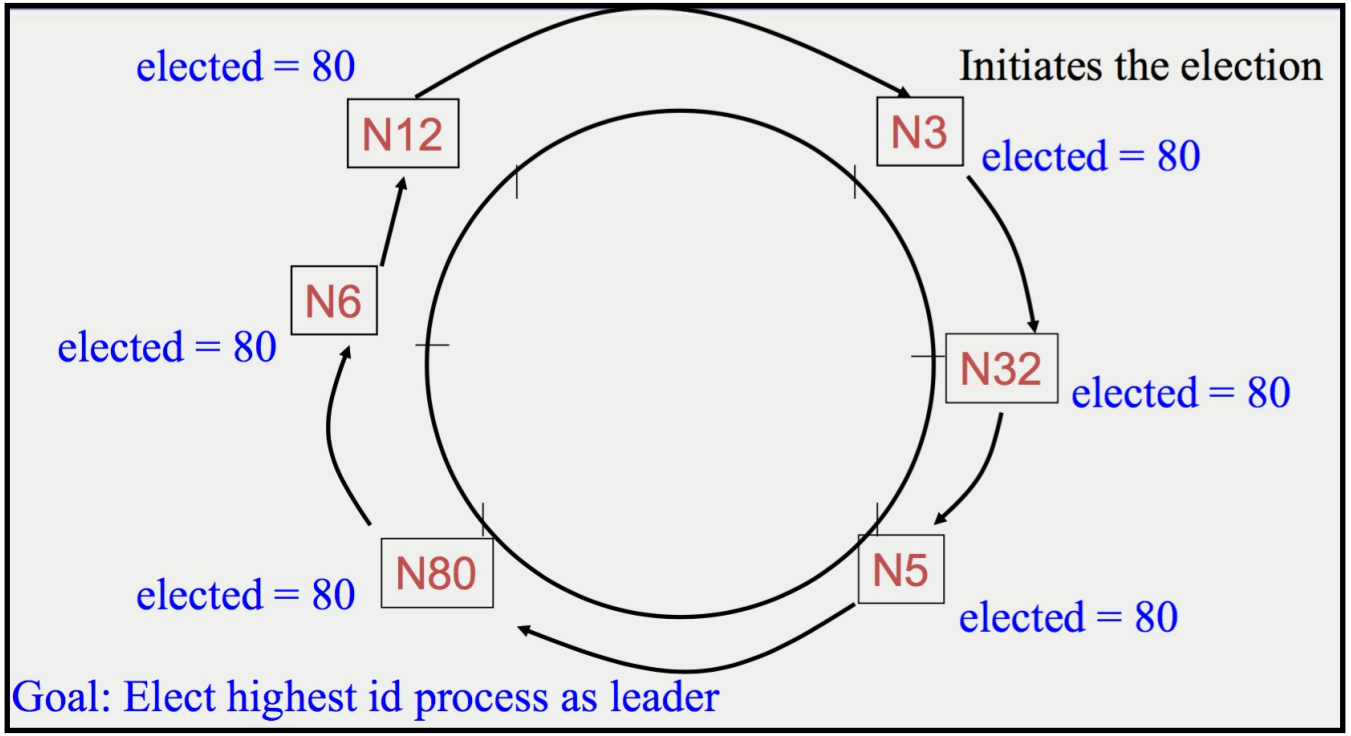
举例


Initiates the election
N12
N3
Elected: 80
N6
elected = 80
N32
N80
N5
Goal: Elect highest id process as leader

举例


elected = 80
Initiates the election
N12
N3
elected = 80
N6
elected = 80
N32
elected = 80
N80
N5
Elected: 80
elected = 80
Goal: Elect highest id process as leader

举例


elected = 80
Initiates the election
N12
N3
elected = 80
N6
elected = 80
N32
elected = 80
N80
N5
elected = 80
elected = 80
Goal: Elect highest id process as leader

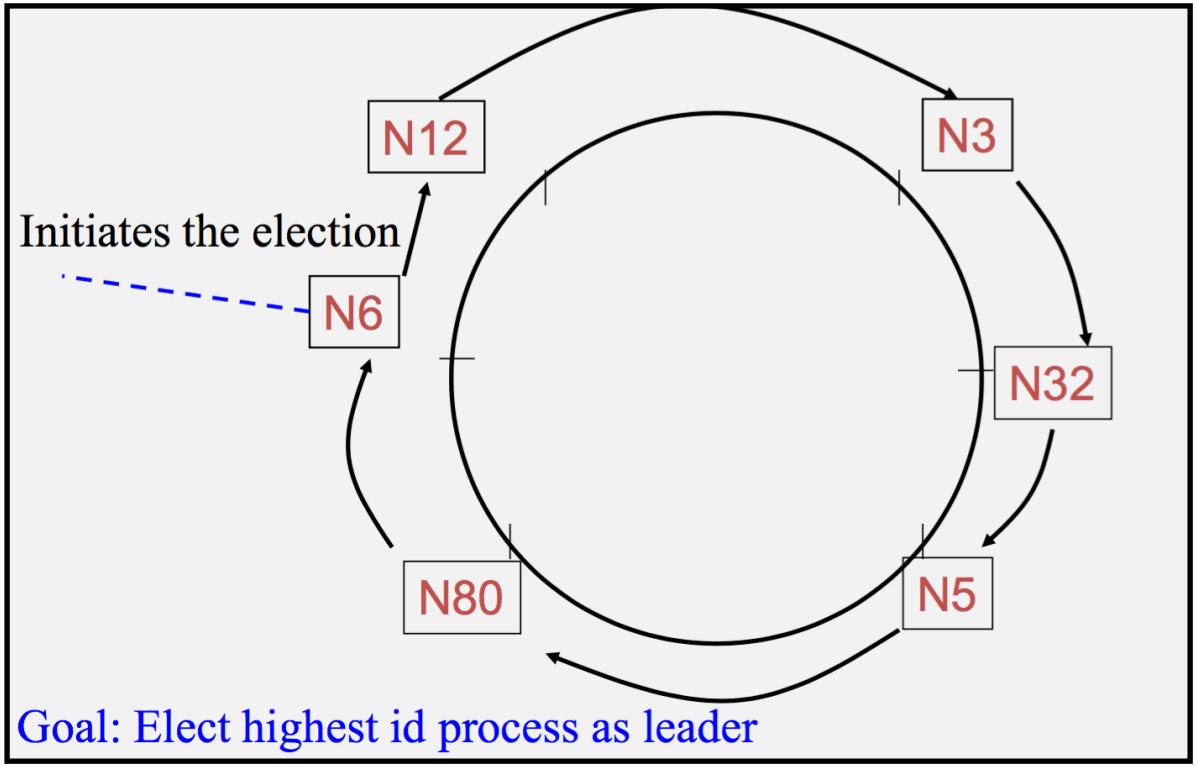
分析

- 假设协议执行期间没有发生任何故障。
- 给定N个节点，问一共产生多少条消息？

分析

- 假设协议执行期间没有发生任何故障。
- 给定N个节点，问一共产生多少条消息？
- 最坏情况分析：
  - 当协议发起者是新leader在环中的下一节点时。

最坏情况分析


N12
N3
Initiates the election
N6
N32
N80
N5
Goal: Elect highest id process as leader

最坏情况分析

- Election消息从初始节点 (N6)到id最大节点(N80)：(N-1) 条
- Election消息再从id最大节点N80回到N80（消息内容无变化）：N条
- Elected消息的传输绕环一周：N条。
- 消息复杂度：(3N-1)
- 完成时间：(3N-1) 条消息的传输时间。
- 因此，如果不发生故障，选举协议结束后（活性），每个节点都获悉属性最好节点作为leader（安全性）。

最好情况分析

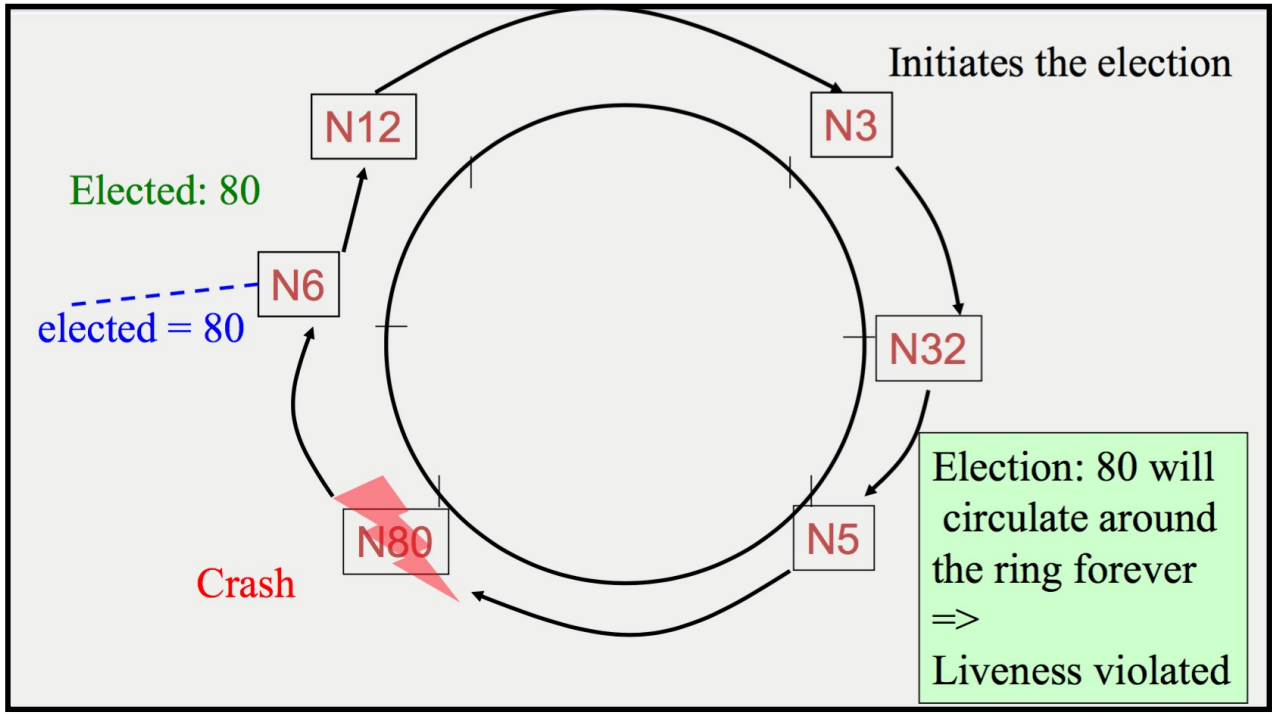
最好情况分析

- 协议发起者即为新leader：N80为协议发起者。
- 消息复杂度：2N.
- 完成时间：2N条消息的传输时间.

多个协议发起者问题

- **每个处理节点缓存自己收到的每条Election/Elected消息的发起者。**
- **（任意时刻）每个处理节点会忽略来自于更小id发起者的Election/Elected消息**
  - 不作消息转发。
- **当收到来自于更大的id的发起者的Election/Elected消息时，更新缓存。**
- **结果只有最大id协议发起者的选举最终执行完成。**

发生故障情况


Initiates the election
N12
N3
Elected: 80
N6
elected = 80
N32
Election: 80 will
circulate around
the ring forever
=>
Liveness violated
N80
N5
Crash

发生故障情况

- **一种方案：在新leader节点（N80）的上游节点（或下游节点）设置故障检测，并在故障发生后重新发起选举协议：**
  - 可能需要重新发起选举：
    - 收到Election消息，但是等待Elected消息超时。
    - 或者，收到Elected:80消息后，长时间未再收到来自于N80节点的消息。
  - 但是，如果该上游节点也发生故障了怎么办?
  - … …

发生故障情况

- **另一种方案：使用故障检测器.**
- **所有节点，在收到Election:80消息后，都利用本地故障检测器不断检测N80是否正常工作。**
  - 如果N80故障，则发起新一轮选举协议。
- **然而，故障检测器可能是不完善、不准确的：**
  - 不完善 => 未检测到N80的故障 => 违反安全性。
  - 不准确 => N80被错误的检测为已故障
    - => 新的选举协议开始，并无限循环下去。
    - => 违反活性。

为什么解决选举问题这么困难？

- **因为它是 consensus problem (共识问题)!**
- **如果选举问题能完美解决，那么共识问题也能完美解决！**
  - 比如，使用leader的id的最后一位作为一致决定。
- **然而，在异步系统下，共识问题是不可能解决的，所以选举问题不可解。**

共识问题

- **过程：每个节点都贡献一个决定值。**
- **目标：让所有节点最终做出相同的决定：**
  - 一旦决定做出之后，不可再修改。
- **其他要求：**
  - 正确性 = 如果每个人都提出相同的值，则最终决定应该为该值。
  - 完整性 = 最终决定值必须是由某个节点所提出的值。
  - 至少有一个初始系统状态产生全0的最终输出，也至少有一个初始系统状态产生全1的最终输出。

重要性

- **分布式系统中的许多问题都等同或难于共识问题。**
  - 完美故障检测
  - 选举问题
  - Agreement问题
- **解决一致性问题，也就解决了以上问题。**

分布式系统

- **异步系统：即，时间不受限，可以为任意长时间**
  - 消息在网络上的传输延迟不受限
  - 任务处理时间延迟不受限
  - 节点之间时钟频率差不受限。
- **同步系统：时间受限**
  - 消息在有限时间内被接收。
  - 节点之间时钟频率差受限。
  - 节点处理任务的时间受限：lb < time < ub。

共识问题的可解性

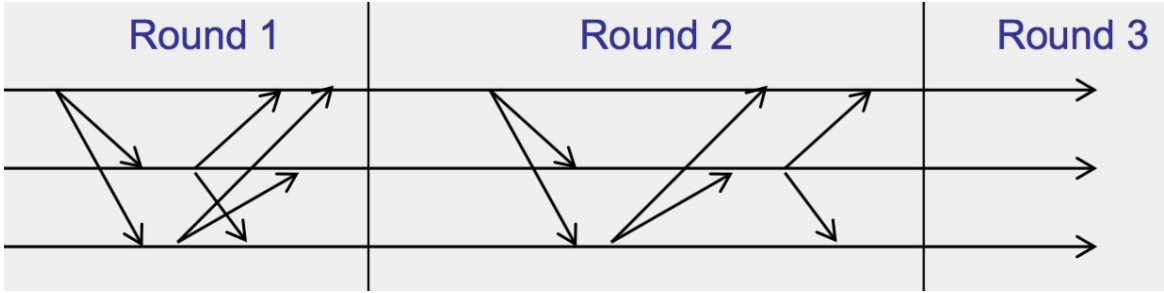
- **同步系统下，**
  - 共识问题可解
- **异步系统下，**
  - 共识问题不可解
  - 不论如何设计协议/算法，总能找到一种最坏情况（比如出现故障或消息延迟）导致系统不能达成最终一致。
  - 折衷性解决方案。

共识问题求解

- **先考虑同步系统：**
  - 时间受限：
    - 消息在有限时间内被接收。
    - 节点之间时钟频率差受限。
    - 节点处理任务的时间受限：lb < time < ub。
- **节点可能会因宕机故障导致停机（故障发生后，机器不再有任何活动）。**

同步系统中的共识问题

- **假设最多有f个节点发生故障：**
  - 所有的节点被同步，并以“轮”组织协议中的操作。
  - 协议会运行f+1轮，使用可靠多播通信。
  - 变量$Values^r_i$: 表示在第r轮初始时刻，pi所接收到的所有的提案。



同步系统中的共识问题

Possible to achieve!

- **假设最多有f个节点发生故障：**
  - 所有的节点被同步，并以“轮”组织协议中的操作。
  - 协议会运行f+1轮，使用可靠多播通信。
  - 变量$Values^r_i$: 表示在第r轮初始，pi所接收到的所有的提案。

- Initially $Values^0_i = \{\}$; $Values^1_i = \{v_i\}$
  for round = 1 to $f+1$ do
    multicast ($Values^r_i$ – $Values^{r-1}_i$) // iterate through processes, send each a message
    $Values^{r+1}_i \leftarrow Values^r_i$
    for each V_j received
      $Values^{r+1}_i = Values^{r+1}_i \cup V_j$
    end
  end
  d_i = minimum($Values^{f+1}_i$)

单轮协议运行情况

同步系统中的共识问题

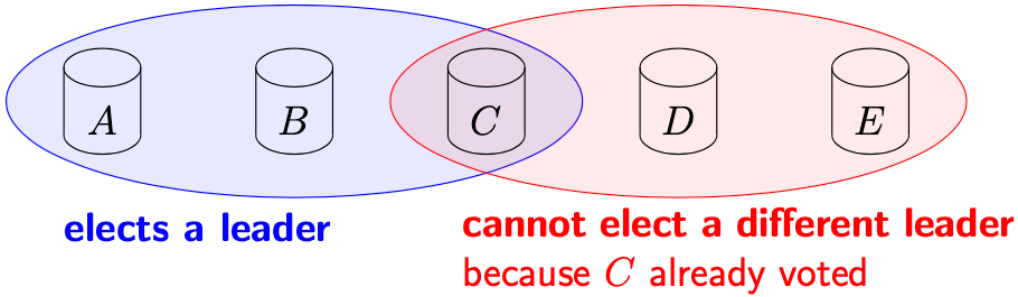
- 第f+1轮结束，所有正常节点都会收到相同的数值集合。
  - 反证法。
- **假设存在两个节点，pi 和 pj，在第f+1轮之后接收到的数值集合不同。假设pi的集合内含有数值v，而pj不包含v。**
  - 可知，pi一定是在最后一轮才刚接收到数值v，
    - 否则的话，pi早已经将v成功发送给pj了。
  - 因此，在第f轮中：存在第三节点pk将数值v发送给了pi，并在将v发送给pj之前发生了故障。
  - 类似的，在第f-1轮中：存在第四节点pu将数值v发送给pk之后发生了故障。否则的话，pi和pj早就收到v了。
  - 依次类推，我们可断定每一轮都至少有一个节点发生故障。
  - 也就是说，共发生f+1个故障，与假设矛盾。

异步系统下的共识算法

- 常见的共识算法包括：
  - Paxos：单值共识（single-value consensus）
    Multi-Paxos：对 Paxos 的推广，用来实现全序广播
  - Raft、Viewstamped Replication、Zab：
    默认就实现了 FIFO 全序广播

Leader election（领导者选举）

- **Multi-Paxos、Raft 等协议使用一个 leader 来为消息排序。**
  - 使用**故障检测器**（基于超时 timeout）来判断 leader 是否可能崩溃或不可用。
  - 一旦怀疑 leader 崩溃，就发起新一轮选举，选出一个新的 leader。
  - 必须避免**同时出现两个 leader**（即所谓的“脑裂 / split-brain”现象）！
- **为了确保每个任期（term）中最多只有 1 个 leader：**
  - 每开启一次领导者选举，**term 就加 1**；
  - 每个节点在同一个 term 中**只能投票一次**；
  - 在某个 term 中，必须有**法定多数（quorum）节点**投票，才能正式选出一个 leader。

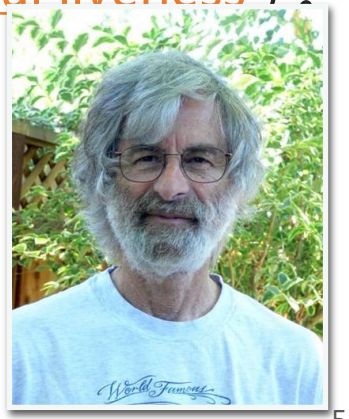


共识系统模型

- Paxos、Raft 等算法假设的是一个“部分同步（partially synchronous）、崩溃-恢复（crash-recovery）”的系统模型。
- **为什么不直接假设“完全异步（asynchronous）”呢？**
  - ▶️ FLP 定理（Fischer, Lynch, Paterson）：在一个异步的、崩溃-停止（crash-stop）系统模型中，
    不存在一个“确定性的共识算法”，能够保证一定会终止。
  - ▶️ Paxos、Raft 等算法中：时钟只用于超时（timeout）和故障检测（failure detector），用来保证活性（liveness）；它们的安全性（safety）不依赖于时间，也就是说，即使网络很慢、时钟很不准，只要消息最终能送达，就不会产生“错决”。
- 此外，也存在面向“部分同步的拜占庭系统模型”的共识算法（例如区块链中使用的那些），用来在存在拜占庭错误（节点可能作恶或乱来）的情况下仍然达成共识。

Paxos协议

- **Paxos协议**
  - 最常用的“解决一致性问题”的算法。
  - 并没有真正解决异步系统中的一致性问题。
  - 但是能保证安全性（safety） 和 最终活性（eventual liveness）。
  - 许多系统使用Paxos协议：
    - Zookeeper (Yahoo!), Google Chubby等
  - Paxos 由Leslie Lamport提出。



Paxos岛

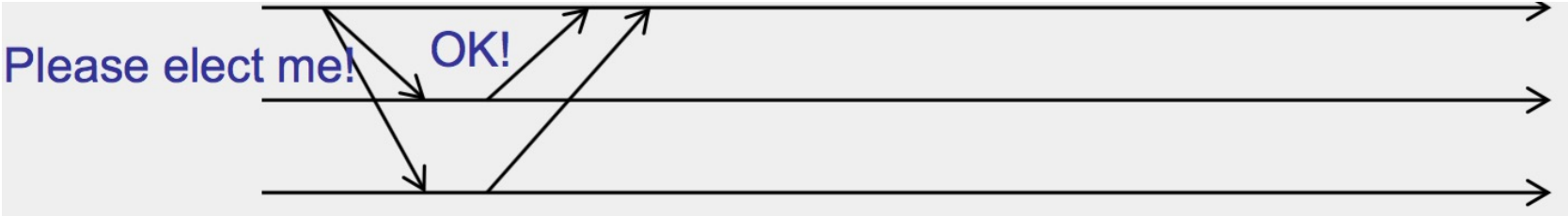
Paxos协议

- **Paxos协议也被组织为多“轮”操作**
  - 每一轮都有其计数id。
- **异步系统：**
  - 不要求时间严格同步。
  - 如果你正处在第j轮，但是却收到第j+1轮的消息，那么放弃一切操作，直接进入第j+1轮。
  - 使用超时机制。
- **每一轮又被分为三个“阶段”：（同样是异步）**
  - 阶段1: 选取一个leader (Election Phase).
  - 阶段2: Leader多播一个提案数值，其他节点收到消息后确认。 (Bill Phase).
  - 阶段3: Leader多播最终决定。 (Law Phase).

阶段1: 选举 (Election)

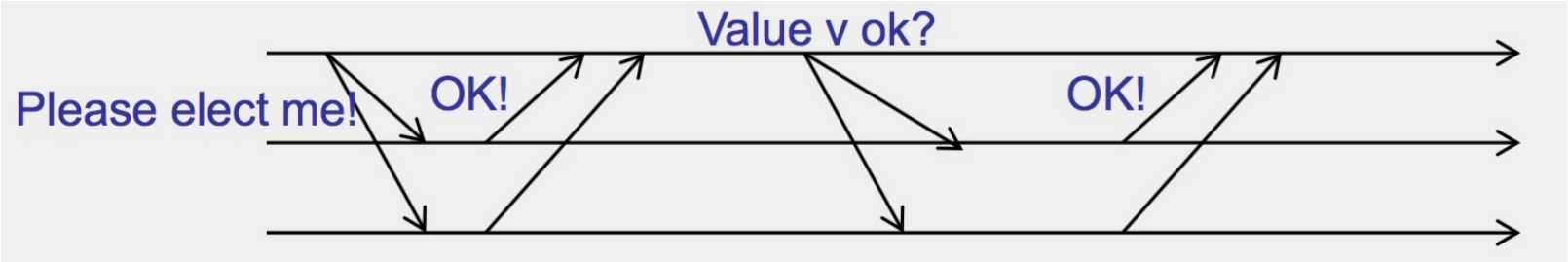
- **潜在的leader选择一个比之前遇到的都大的值作为投票id。**
- **将id发送给其他节点。**
- **节点等待，向发出最大投票id的节点回复：**
  - 如果潜在leader看到一个更大的投票id，他主动放弃成为leader。
  - Paxos允许有多个leader同时出现，但是我们只讨论只有1个leader 的情况。
  - 节点将受到的投票id写入本地日志。
- **如果一个节点在之前的轮中，已经做出过决定，且决定值为v’ ，那么节点会在回复消息中加入v’ 。**
- **如果获得超过一半节点回复OK，则正式成为leader。**
  - 如果没有节点获得超过一半的OK回复，则重启协议进入新一轮。
- **（如果一切顺利）一轮中不可能出现两个leader（为什么？）。**

阶段1: 选举 (Election)



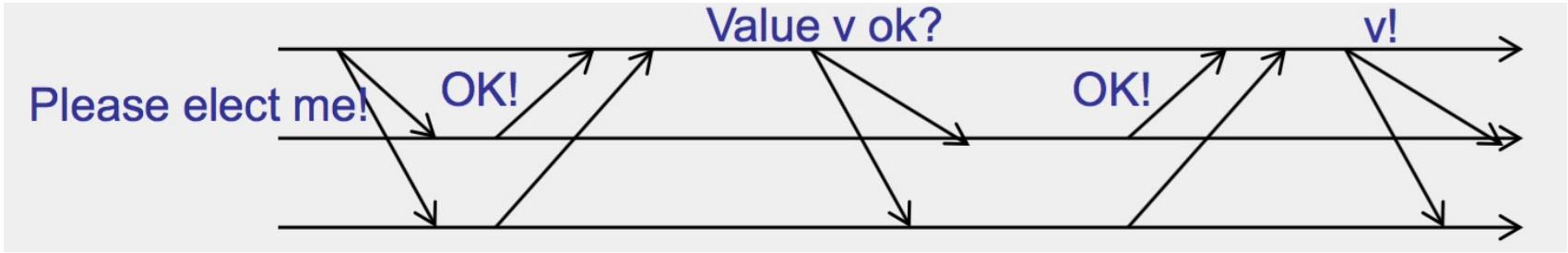
阶段2: 提案 (Bill)

- **Leader提议数值v，并多播v给其他节点：**
  - 如果leader在上一阶段从某节点的ok消息中收到v’，则直接设置v=v’。
- **节点收到消息，将消息写入日志，并回复OK。**



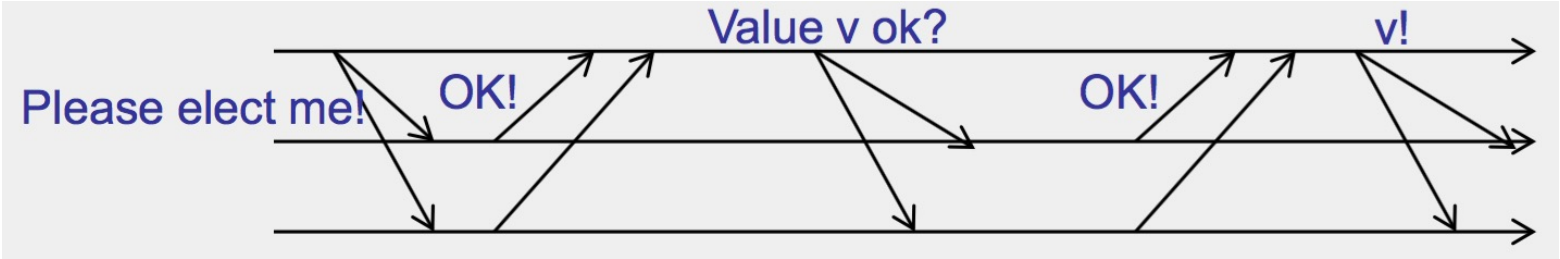
阶段3: 决定 (Law)

- 如果leader收到超过一半的节点回复OK，最终决定值达成，多播消息给所有节点。
- 节点收到决定值，写入日志。



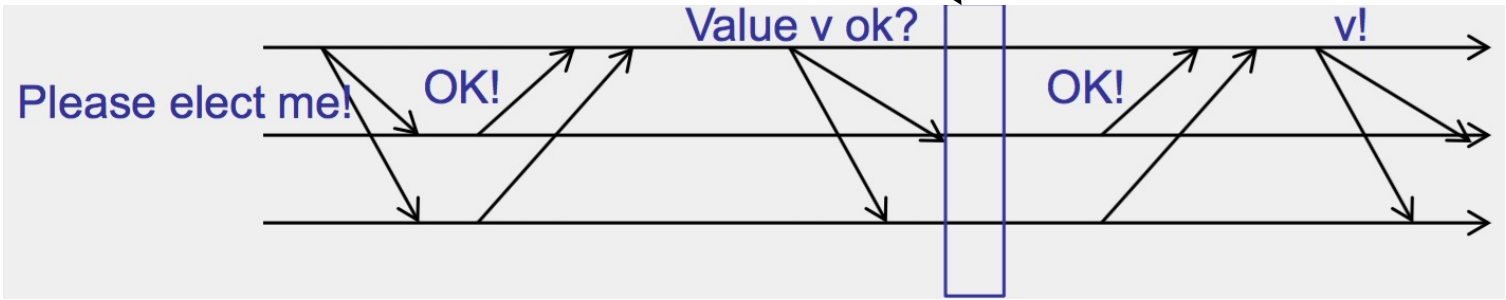
共识达成

- 什么时间点可认为已经达成一致性，系统不可能再更改决定值？



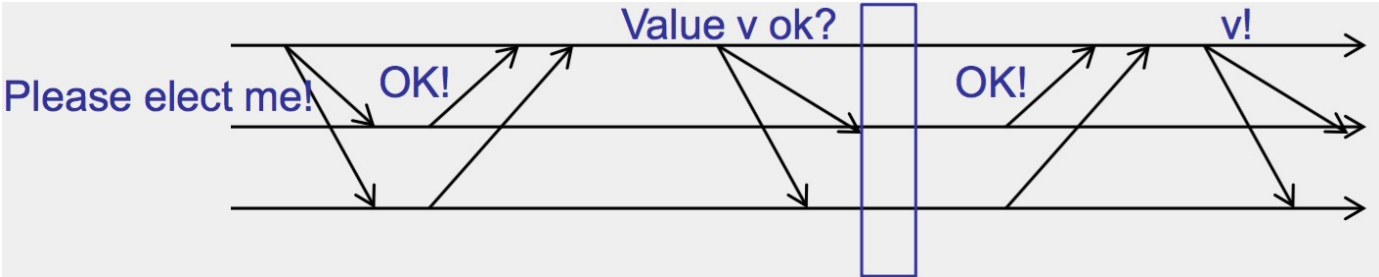
共识达成

- **当超过一半的节点收到阶段2中的消息时，可认为已经达成一致决定值。**
- **此刻，节点自己可能并不知道，但是最终决定值实际上已经确定了：**
  - 即使leader也可能在此刻还不知道。
- **如果leader在此时发生故障了怎么办?**
  – 进入下一轮，重复，一直到在某一轮中三阶段都顺利结束。



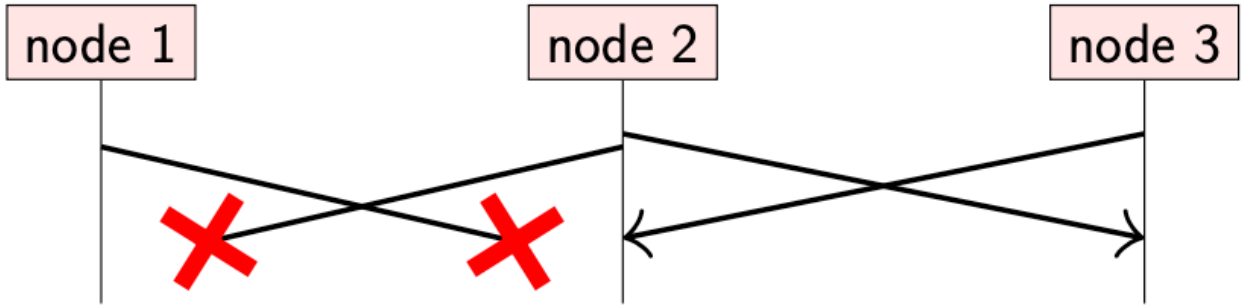
安全性

- **假设在某一轮中，超过一半的节点已经收到并接受消息v’ ，在后面的每轮中：1）要么选择v’ 作为决定值；2）要么该轮失败，继续进入下一轮。**
- **证明:**
  - 潜在的leader在阶段1等待超过半数节点回复OK。
  - 至少1个OK消息中包含数值v’ (因为两个超过半数，则至少有一个处在两者交集中)。
  - 在阶段2，leader会将消息v’ 多播给所有节点。
- **回复消息一定要超过半数，两个超过半数的集合会有至少一个节点的交集。**



能否保证系统中始终只有一个 leader？

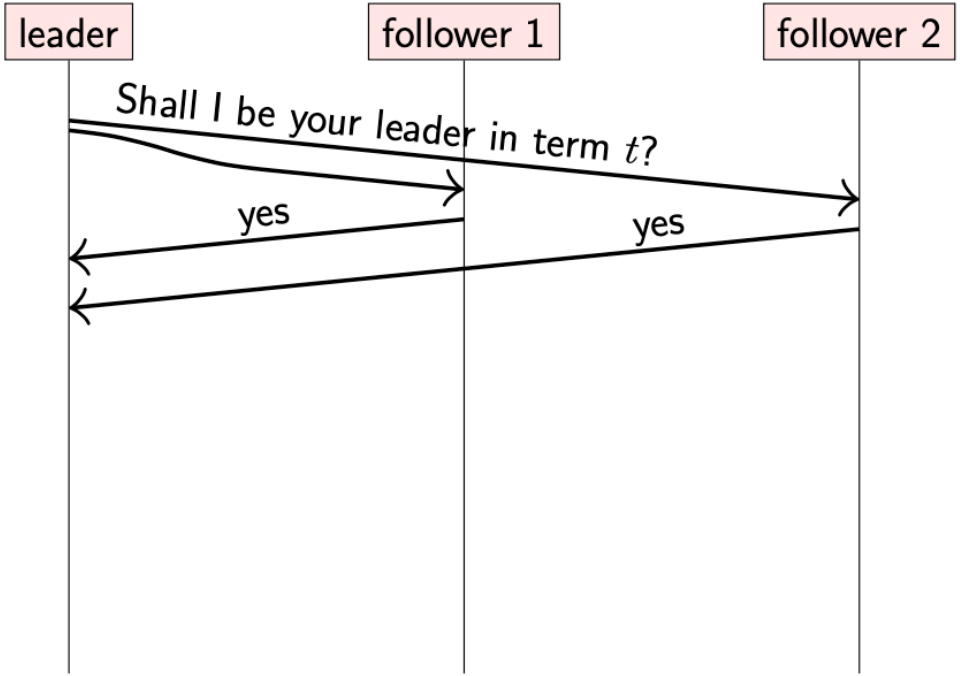
- 我们能保证的是：每轮内的 leader 是唯一的。
- 但我们无法完全避免：在不同轮里，可能同时存在多个 leader。
- 例如：节点 1 在轮 t 中是 leader，但由于网络分区，它无法再和节点 2、3 通信：（此时另一侧有可能在新的任期里又选出一个新的 leader）



- 节点 2 和 3 可能会在轮 t+1 中选出一个新的 leader。
- 此时节点 1 甚至可能完全不知道已经选出了一个新 leader！

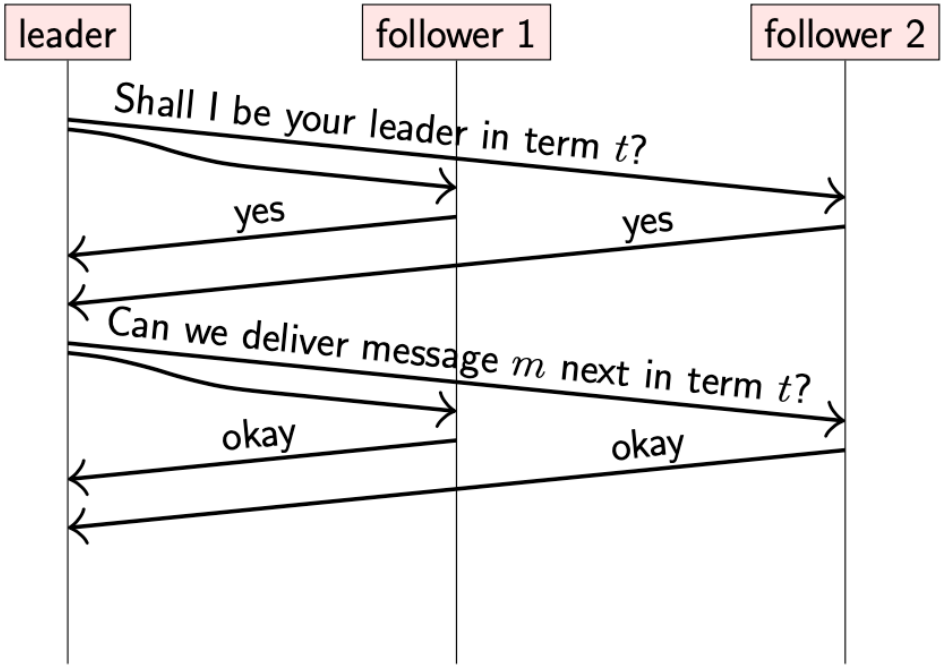
检查一个 leader 是否已经被“投票下台”：

- 对于每一个决议（要投递的一条消息），leader 都必须先从一个仲裁多数（quorum）节点获得确认（acknowledgement）。



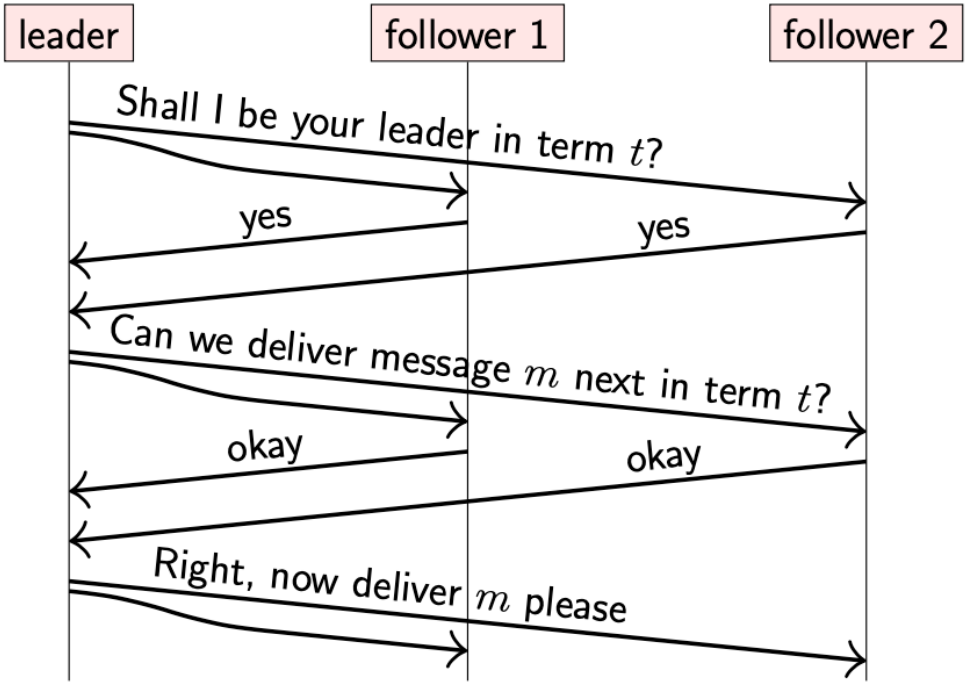
检查一个 leader 是否已经被“投票下台”：

- 对于每一个决议（要投递的一条消息），leader 都必须先从一个仲裁多数（quorum）节点获得确认（acknowledgement）。

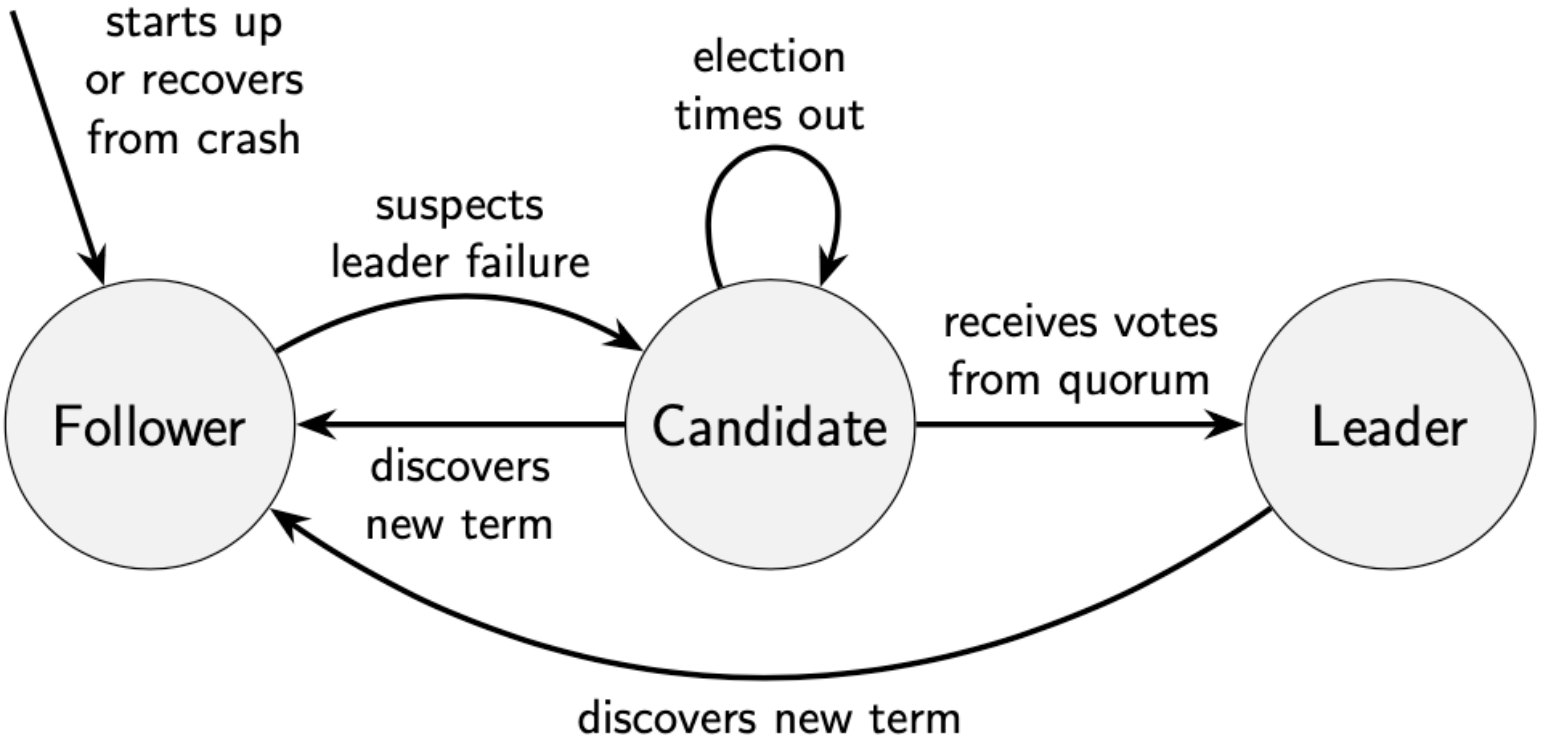


检查一个 leader 是否已经被“投票下台”：

- 对于每一个决议（要投递的一条消息），leader 都必须先从一个仲裁多数（quorum）节点获得确认（acknowledgement）。



Raft协议


starts up
or recovers
from crash
election
times out
suspects
leader failure
receives votes
from quorum
Follower
Candidate
Leader
discovers
new term
discovers new term

Raft协议

- 让一组机器对“操作序列”（日志）达成一致：谁先记哪条、记了就不能乱改。只要大多数机器还活着，就能继续工作。
- 角色：三种身份
  - Follower（跟随者）：平时只听指挥、被动接收日志。
  - Leader（领导）：唯一“记账员”。所有客户端请求都先发给它。
  - Candidate（候选人）：要竞选领导的临时身份。
- Raft 的核心设计：任何时刻尽量只有一个 Leader，这样大家不吵架。

Raft协议

- 关键概念：多数派 + 任期
  - 多数派（Quorum）：> N/2 的节点。同意的人里至少有一个和下一次多数派重叠，这是安全的根。
  - Term（任期）：把时间切成一段段的“届”。每次选举都会进入新任期，任期号只增不减。任期越大说明越“新”。

1) Leader 怎么选出来（选举）

1. **Leader 会定期发 心跳 给所有人：“我还活着！”**
2. **如果某个 Follower 一段随机时间没收到心跳，就以为 Leader 挂了：**
    1. 自己变 **Candidate**
    2. 任期 term+1
    3. 给所有人发“投我一票”（RequestVote）
3. **谁拿到 多数票，谁当 Leader。**
4. **为了避免选出“日志落后”的 Leader，投票时会要求：候选人的日志不能比我旧（至少一样新）。**

- **随机超时很重要：大家不太容易同时起身竞选，减少“平票反复”。**

2) 日志怎么复制（写请求如何达成一致）

- 把每个客户端写请求想成“一条账目”。

1. 客户端把请求发给 Leader
2. Leader 先把这条记到自己日志末尾（还没算“正式生效”）
3. Leader 把这条账目发给所有 Follower（AppendEntries）
4. 只要有 多数派回复“我也记好了”，Leader 就宣布这条 提交（commit）
5. 提交后，Leader 和 Followers 才会把这条操作真正“应用到状态机”（更新本地数据）

- 直觉：
  “多数派都写进小本本了，这条账就算板上钉钉；就算有人之后掉线，剩下的人也不会把它弄丢。”

3) 崩溃/网络乱序时怎么保证不出错

Leader 挂了

- Followers 超时 → 重新选举 → 新 Leader 出来继续记账。
- 旧 Leader 只要发现别人任期更大，会立刻退回 Follower（承认自己过期了）。

日志冲突怎么办（最关键的“纠错机制”）

如果某个 Follower 的日志跟 Leader 不一致（比如之前跟错了人）：

- Leader 会强制让它“对齐”：从冲突点开始删掉旧的，按 Leader 的版本重新追加。
- 但已经 commit 的账目不会被推翻（这是 Raft 的安全性底线）。

结果就是：未提交的东西可能被覆盖，已提交的东西永远保留。

Raft协议总结

- 先选一个唯一 Leader（多数票 + 任期）
- 所有写都走 Leader（Leader 统一排序）
- 多数派复制才算提交（commit）
- 冲突就以 Leader 为准对齐（未提交可回滚，已提交不动）

- PPT部分内容来自于剑桥大学和UIUC cst.cam.ac.uk/teaching/2526/ConcDisSys/