第6讲 协同（同步）

§6.1 时钟同步

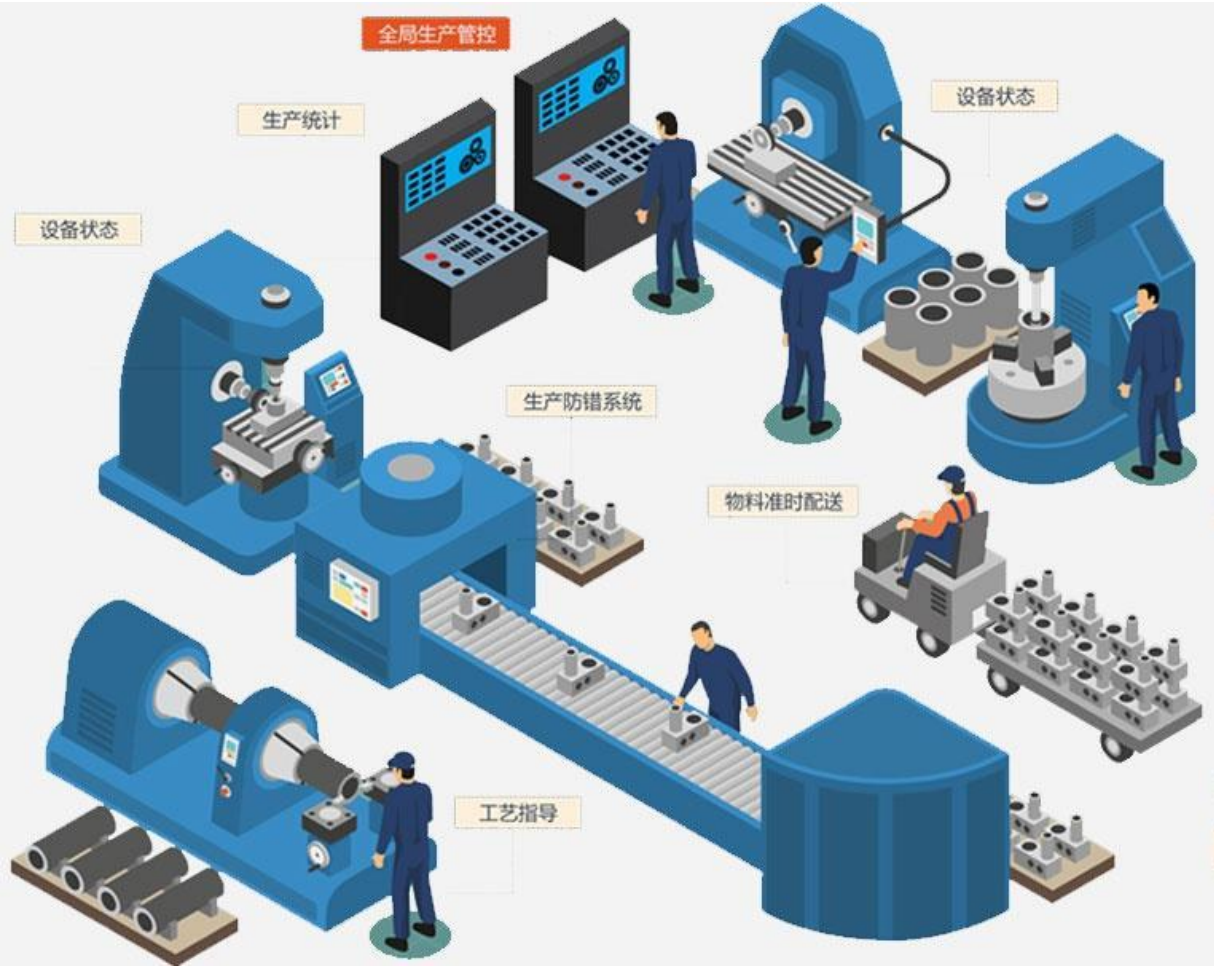
§6.2 逻辑时钟

§6.3 互斥算法

§6.4 选举算法

§6.5 定位系统

引言



过程之间的协同，完成一个整体过程

背景

- 同步
  - 事件之间进行协调，在时间上达成一致；
  - 进程同步：一个进程等待其他进程执行完；
  - 数据同步：保证两个数据集合相同；
- 协作
  - 管理系统中的行为之间的交互和依赖关系；
- 分布式系统的协作
  - 分布式系统中的协作要比单节点多处理器系统复杂得多。

§6.1 时钟同步

- 一个例子：
  - Make编译程序
    - 如果源文件修改时间晚于目标文件的修改时间，重新编译
  - 分布式环境中编译
    - 不同机器时钟不同步会导致新版源文件被遗漏

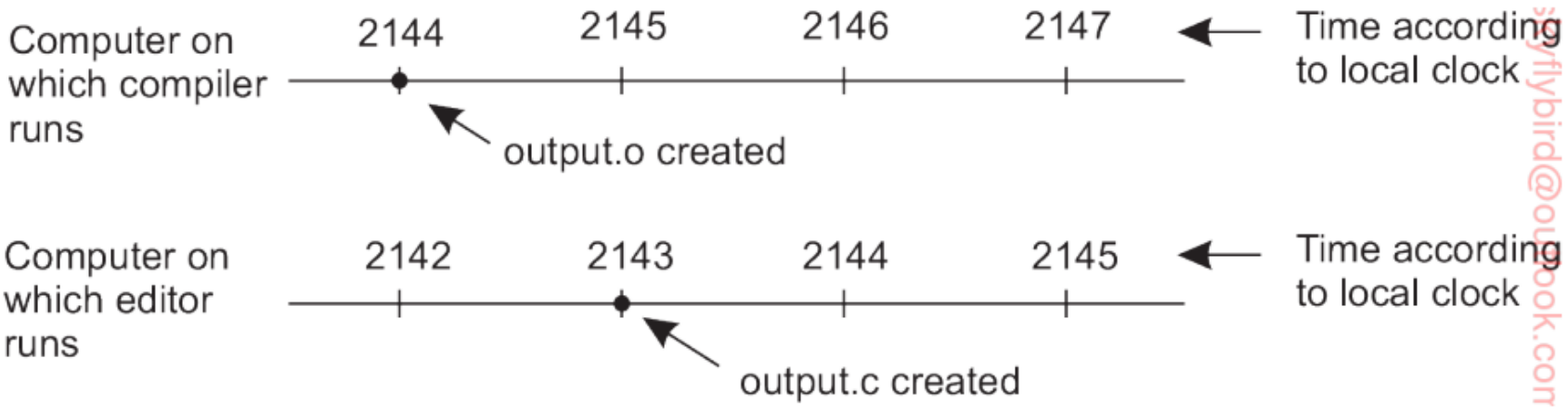


Figure 6.1: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

物理时钟、GPS

- 问题
  - 有时，我们需要精确的时间，而不仅仅是顺序
- 解决方案： 统一协调时间 （UTC）
  - 基于铯133原子的每秒钟跃迁的次数进行计数；
  - 当前时钟使用的是世界上50个铯原子时钟的平均值；
  - 引入闰秒（leap second），补偿国际时钟与太阳秒之间的误差；

  1s=1/86 400 of the mean solar day
- 注意
  - UTC时间通过广播站（WWV）和一些卫星（GEOS）广播；分别有±10ms、±0.5ms的误差

时钟同步算法

- 精度
  - 目的是保证任意两个机器的时钟之间的偏差在一个特定的范围内，即精度 π： $\forall t, \forall p, q: |C_p(t) - C_q(t)| \leq \pi$

    其中$C_p(t)$表示机器 p 在 UTC 为 t 时的时钟时间
- 准确度
  - 对于准确度，我们的目的是保证时钟边界在 α 之内：

    $$\forall t, \forall p: |C_p(t) - t| \leq \alpha$$
- 同步
  - 内部同步：保证时钟的精度
  - 外部同步：保证时钟的准确度

时钟漂移

- 时钟规约
  - 每一个时钟都有一个最大时钟漂移率 ρ;
  - 令 F(t) 表示 t 时刻硬件时钟的晶振频率;
  - F 为时钟的理想（固有）频率 -> 应该满足的规约为:

$$\forall t: (1-\rho) \leq \frac{F(t)}{F} \leq (1+\rho)$$

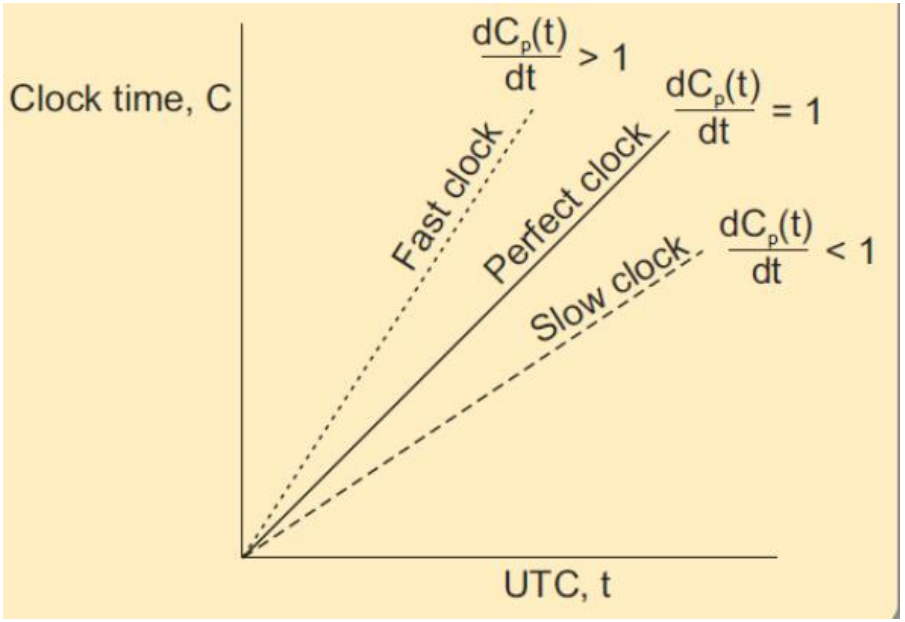
时钟漂移

- 观察发现
  - 利用硬件时钟中断，将软件时钟与硬件时钟耦合时会产生漂移：

$$C_p(t) = \frac{1}{F}\int_0^t F(t)dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

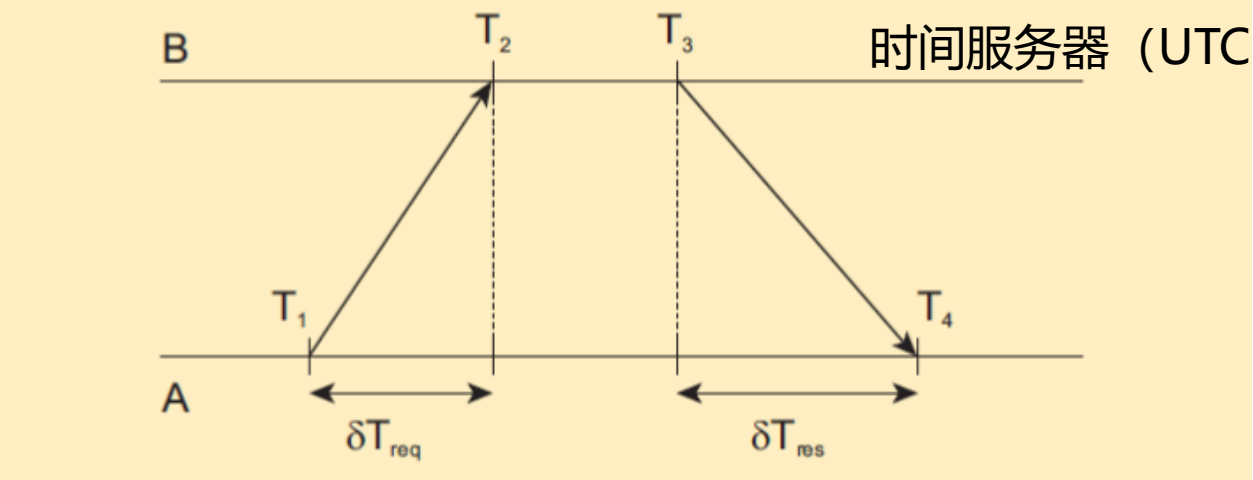
$$\Rightarrow \forall t : 1-\rho \le \frac{dC_p(t)}{dt} \le 1+\rho$$

较快、准确、较慢的时钟



检测和调整不正确的时间

- 从一个时间服务器上获得当前的时间



如果快了作何处理?
如果慢了呢?

- 服务器之间的时间偏差和延迟为:

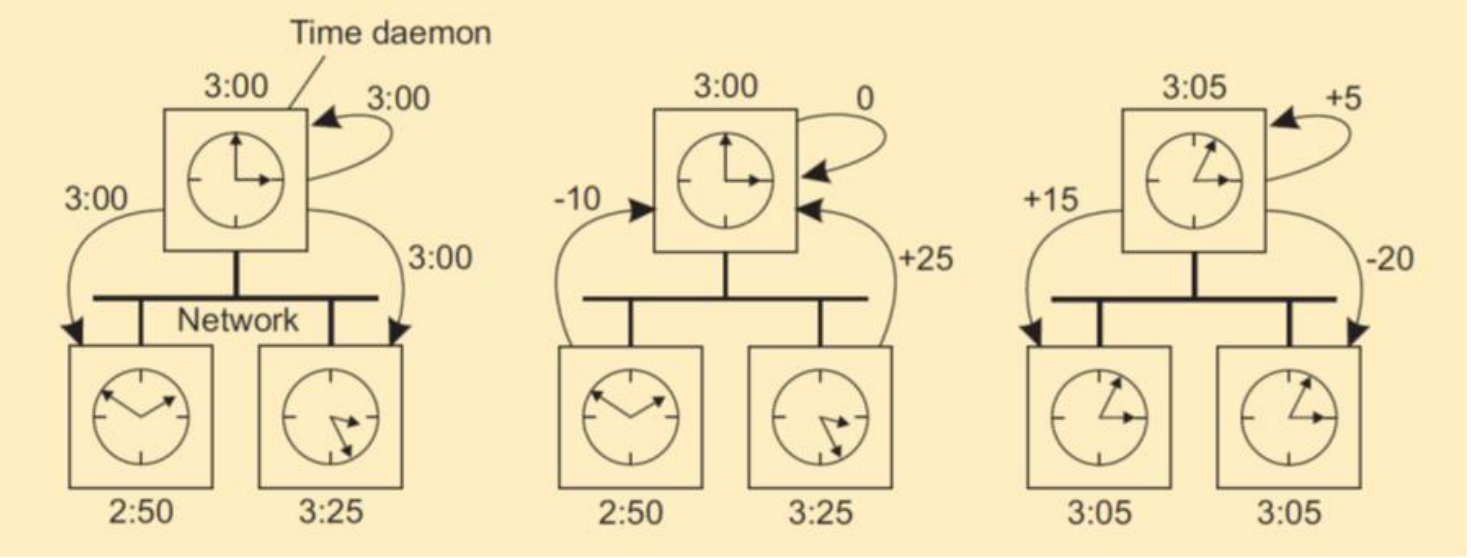
$$\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res} \qquad \delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

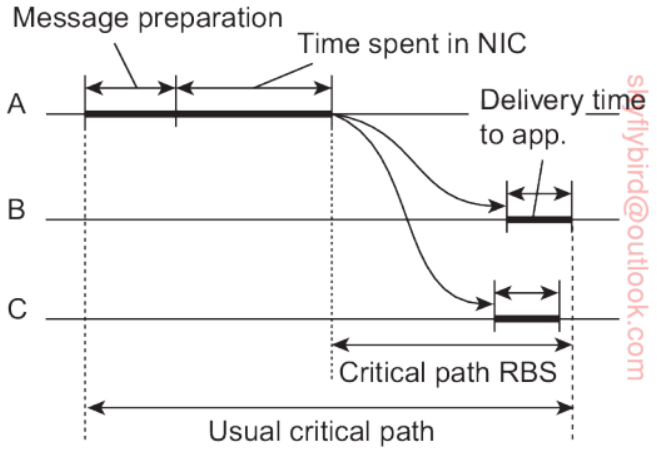
NTP（Network time protocol）：收集多个（θ, δ），选择最小 δ 对应的θ

没有UTC的情况下保障时间的准确性

- Berkeley 算法
  - 时间服务器周期性地查询所有服务器，计算时间均值
  - 时间服务器通知其他机器调整时间：跳至新时间或者减速等待对齐时间
  - 内部时钟、没有专门时间服务器的场景



无线网络中的时钟同步



- 参考广播同步化（RBS）
  - 一个节点广播参考消息 m,
  - 每个接收节点 p 记录收到消息m 的$T_{p,m}$ （本地时间）
  - 时间偏差：

$$Offset[p,q] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$$

- 时钟漂移：时钟是会变化的，计算平均值无法包含时钟漂移，因此使用线性回归（LR）

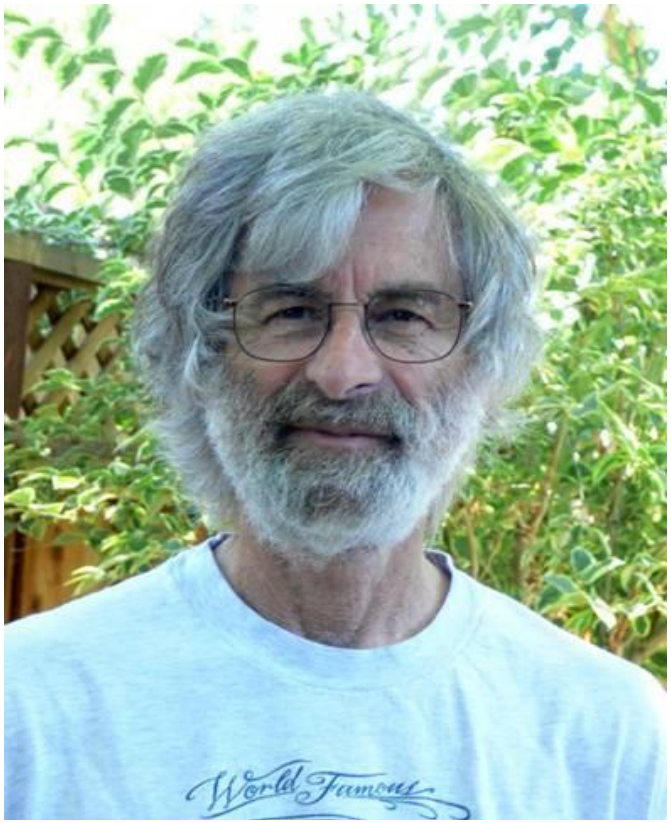
$$Offset[p,q](t) = \alpha t + \beta$$

The constants α and β are computed from the pairs $(T_{p,k}, T_{q,k})$.

发送者自己不进行同步！只对齐接收者的时间。

§6.2 逻辑时钟

- Lamport逻辑时钟



Leslie B. Lamport (Feb.7-1941~)
American computer scientist

Best known for:
- seminal work in distributed systems and
- initial developer of LaTeX

Winner of the 2013 Turing Award for:
- imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems

He devised important algorithms and developed formal modeling and verification protocols that improved correctness, performance, and reliability of computer systems.

Happens-before关系

- Happens-before 关系 a->b
- 也称为因果关系（Causality）
  - P1：如果 a 和 b是同一个进程中的两个事件，并且 a 在 b之前到达，则有：a->b
  - P2：如果 a 是消息的发送者，b是消息的接收者，则 a->b
  - 传递性：如果 a->b 并且 b->c 则 a->c
- 本质
  - 所有的进程并不一定在时间上达成一致，而只需要在时间发生顺序上达成一致，也就是需要“排序”
- 并发（concurrent）关系
  - a->b和b->a均不成立

逻辑时钟

- 如何维护分布式系统行为的全局视图，使其与Happens-before关系保持一致?
- 即逻辑时钟:

  为每一个事件e分配时间戳 C(e) 使其满足:

  如果 a->b，那么：C(a) < C(b)

关键点:
如何在没有全局时钟的情况下为事件分配时间戳?

Lamport逻辑时钟

Each process P_i maintains a local counter C_i and adjusts this counter

1. For each new event that takes place within P_i, C_i is incremented by 1.
2. Each time a message m is sent by process P_i, the message receives a timestamp $ts(m) = C_i$.
3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

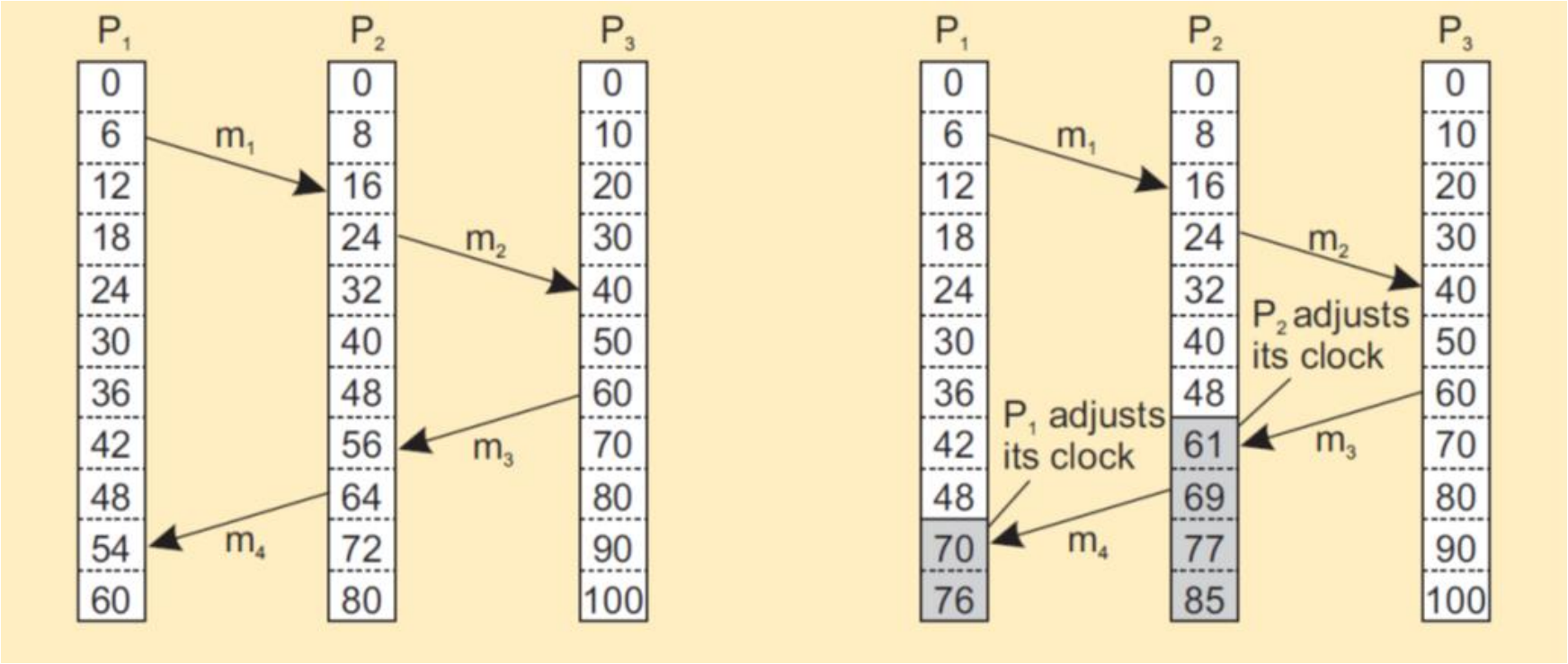
Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

P1：如果 a 和 b是同一个进程中的两个事件，并且 a 在 b之前到达，则有：a->b
P2：如果 a 是消息的发送者，b是消息的接收者，则 a->b

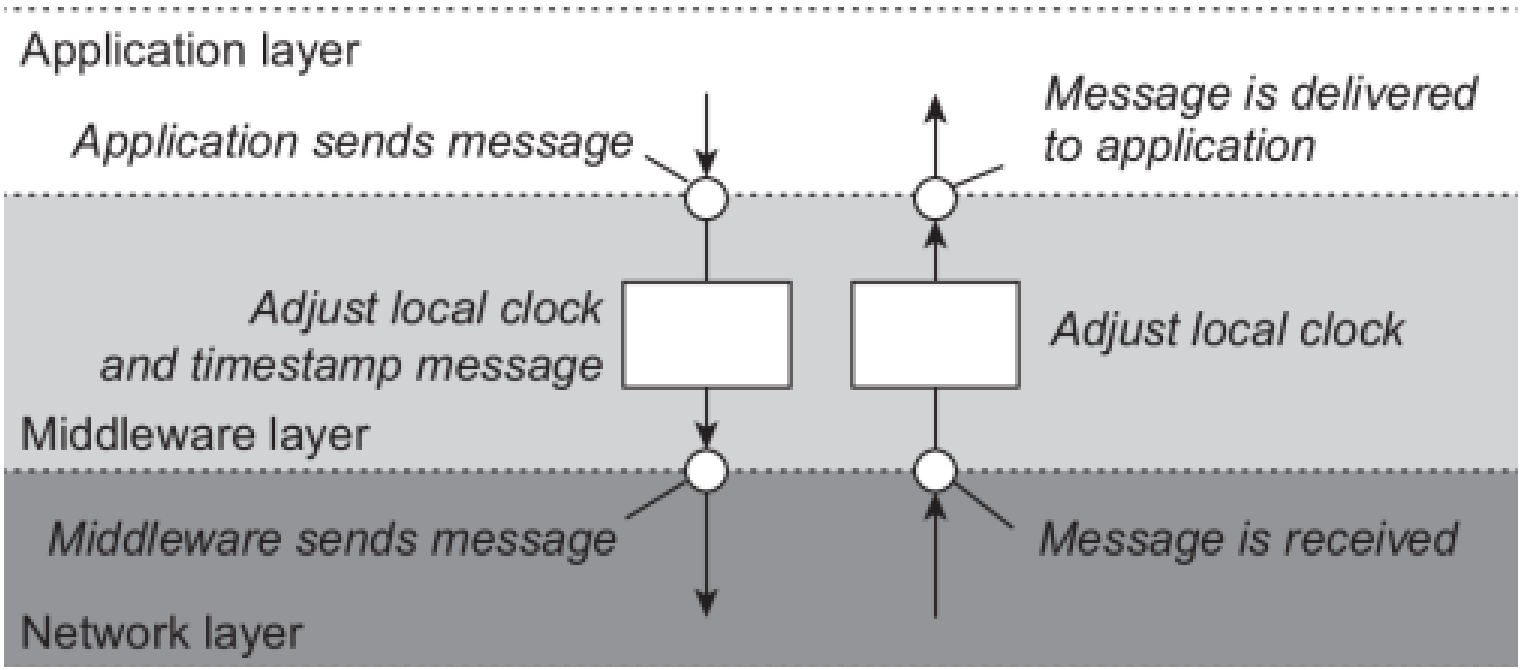
逻辑时钟：例子

- Lamport算法校正三个进程的不同时钟



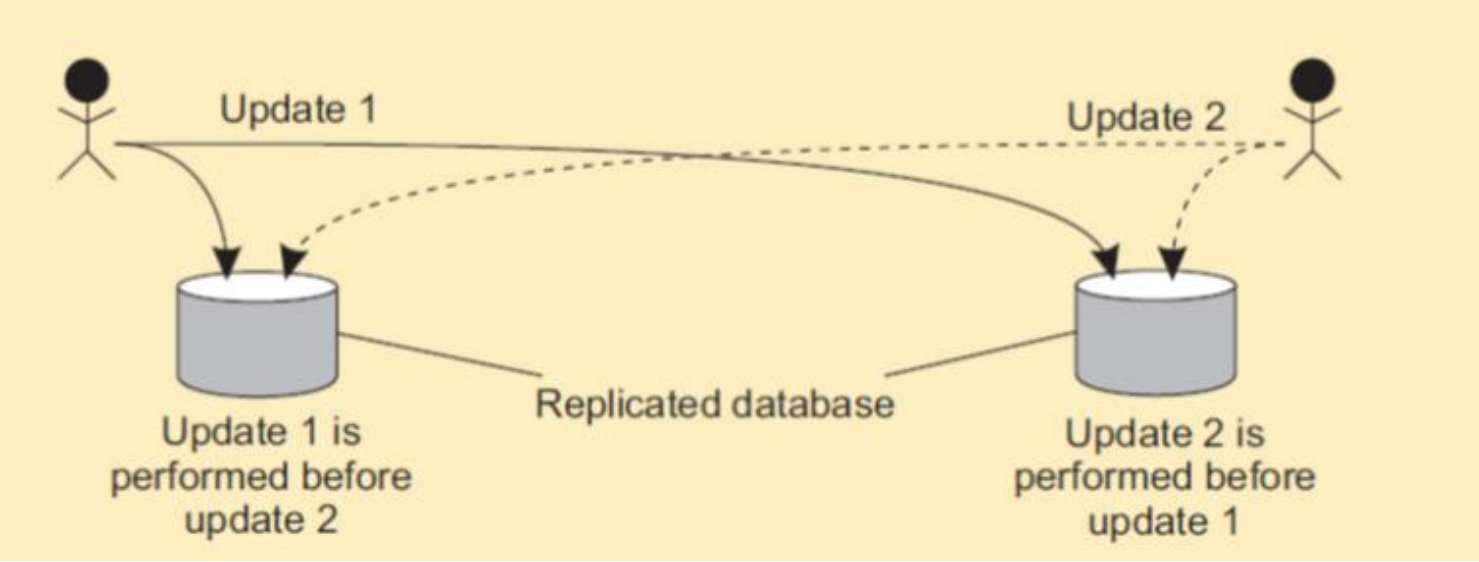
逻辑时钟：如何实现

- 在中间件中实现



应用示例：副本数据库更新

- 在一个副本数据库上的并发更新数据
  - 一个银行账户，初始为1000$，保存两个副本；
  - P1为该账户转入100$；
  - P2为该账户计算利息1%；
- 如果自发更新，可能导致两个副本不一致：



副本1 变成1111$，副本2变成1110$

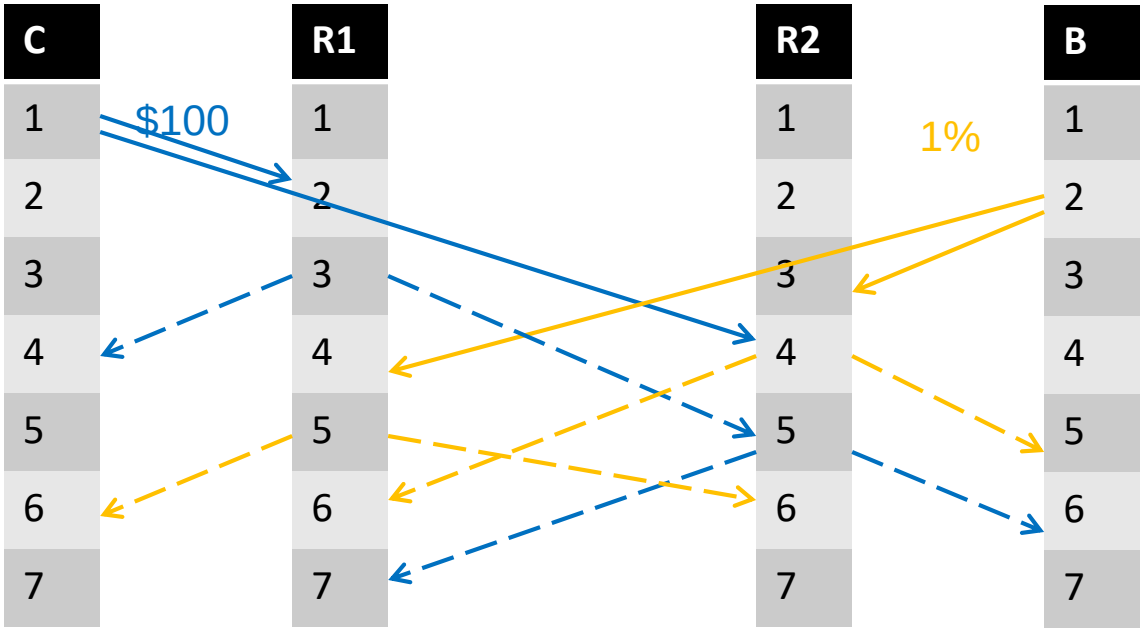
全序多播（Total-ordered Multicast）

- 核心思路
  - 进程P_1将加上时间戳的消息m_i发送到所有的进程， 消息本身保存在本地的队列queue中；
  - 任何新到达P_j的消息放在队列$queue_j$中，根据它的时间戳进行排序，并向所有进程广播确认消息；
- P_j在满足如下条件时会将消息m_i交付给应用程序：
  - m_i是队列$queue_j$的队列头；
  - 对每个进程P_k，$queue_j$ 中有一个m_i 的Ack确认消息。

  （可以替换为：在队列$queue_j$存在一个消息m_k具有更大的时间戳）

注意：假定通信是可靠的，并且是FIFO排序的。

应用示例：副本数据库更新



$	1
%	2
A%	4
A$	5

1	$
2	%
3	$A
5	%A

Why logical time is necessary?

应用实例：逻辑时钟解决互斥访问

- 临界区（Critical Section）
  - 在同一个时刻至多允许一个进程执行的代码片段。
  - 多个进程需要在访问顺序上达成一致。

```
struct RTL_CRITICAL_SECTION
{
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;
};
```

Lamport逻辑时钟解决互斥访问

- 利用全序多播
  - 每个进程建立请求队列，以相同的顺序交付消息；
  - 在进程进入临界区的顺序上达成一致。

```
class Process:
  def __init__(self, chan):
    self.queue      = []                    # The request queue
    self.clock      = 0                     # The current logical clock

  def requestToEnter(self):
    self.clock = self.clock + 1                         # Increment clock value
    self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
    self.cleanupQ()                                     # Sort the queue
    self.chan.sendTo(self.otherProcs, (self.clock,self.procID,ENTER)) # Send request

  def allowToEnter(self, requester):
    self.clock = self.clock + 1                         # Increment clock value
    self.chan.sendTo([requester], (self.clock,self.procID,ALLOW)) # Permit other

  def release(self):
    tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ALLOWs
    self.queue = tmp                                    # and copy to new queue
    self.clock = self.clock + 1                         # Increment clock value
    self.chan.sendTo(self.otherProcs, (self.clock,self.procID,RELEASE)) # Release

  def allowedToEnter(self):
    commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
    return (self.queue[0][1]==self.procID and len(self.otherProcs)==len(commProcs))
```

Figure 6.11: (a) Using Lamport's logical clocks for mutual exclusion.

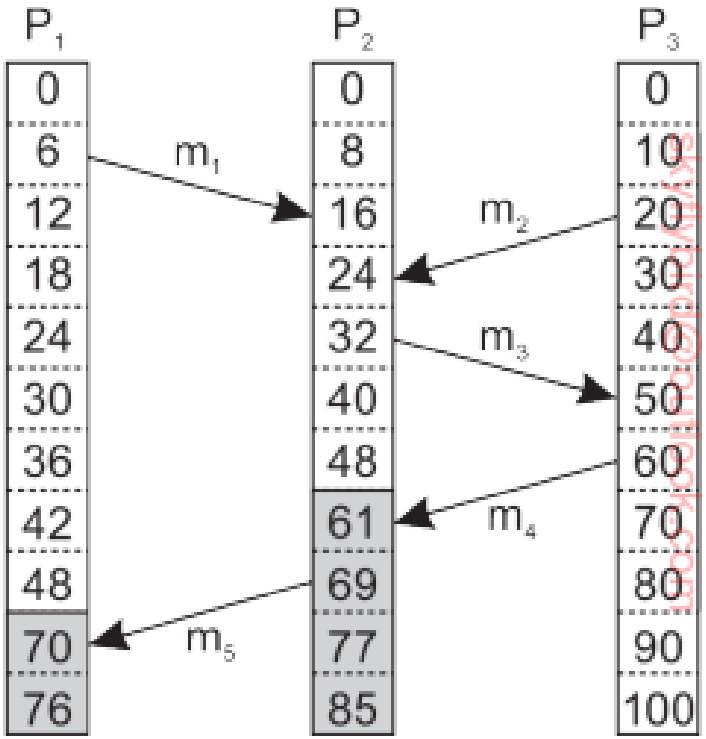
```
def receive(self):
  msg = self.chan.recvFrom(self.otherProcs)[1]      # Pick up any message
  self.clock = max(self.clock, msg[0])              # Adjust clock value...
  self.clock = self.clock + 1                       # ...and increment
  if msg[2] == ENTER:
    self.queue.append(msg)                          # Append an ENTER request
    self.allowToEnter(msg[1])                       # and unconditionally allow
  elif msg[2] == ALLOW:
    self.queue.append(msg)                          # Append an ALLOW
  elif msg[2] == RELEASE:
    del(self.queue[0])                              # Just remove first message
  self.cleanupQ()                                   # And sort and cleanup
```

Figure 6.11: (b) Using Lamport's logical clocks for mutual exclusion: handling incoming requests.

- 每个进程收到ENTER即发回ALLOW；
- 当自己的ENTER在队头并且其他进程都发了ALLOW，则进入CS。

向量时钟 Vector Clock

- Lamport逻辑时钟保证：a→b，则C(a)<C(b)，
- 但是C(a)<C(b)不一定a→b（无法捕获因果关系）



消息m1与m2并没有happens-before的关系

向量时钟

- 每一个进程 P_i维护一个向量VC_i

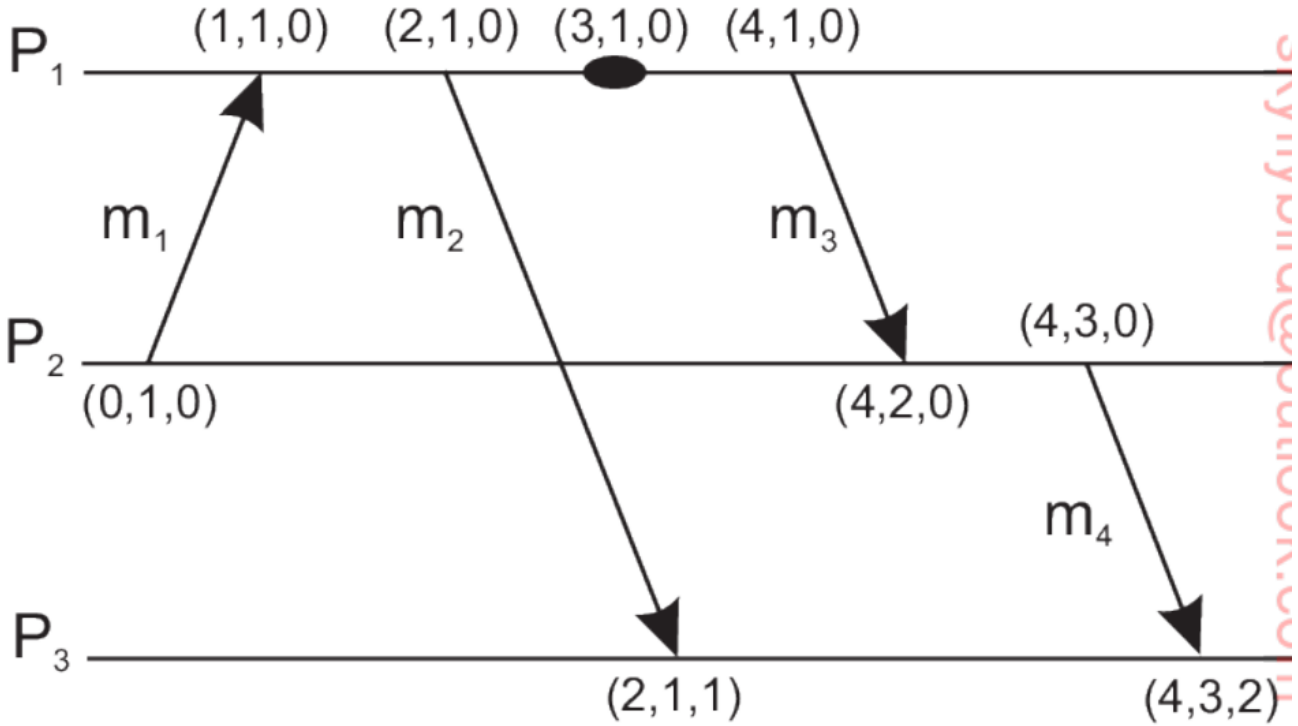
- $VC_i[i]$ is the local logical clock at process P_i.
- If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j.

- 时钟维护：

1. Before executing an event P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(m)$ equal to VC_i after having executed step 1.
3. Upon the receipt of a message m, process P_j sets $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k, after which it executes step 1 and then delivers the message to the application.

向量时钟

- Timestamp ts(m) tells the receiver：
  - how many events in other processes have preceded the sending of m, and on which m may causally depend.

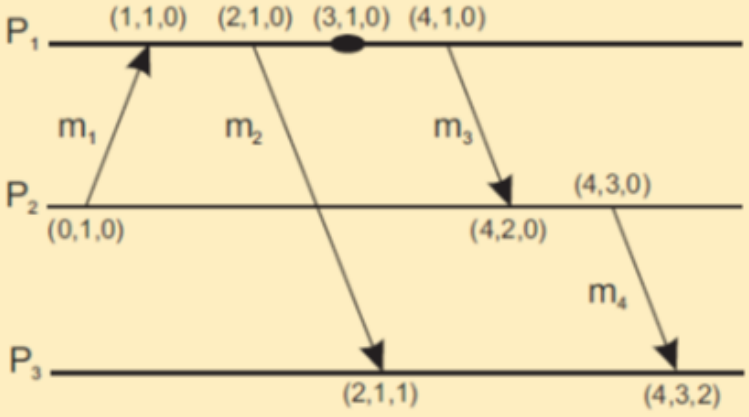


向量时钟与因果关系

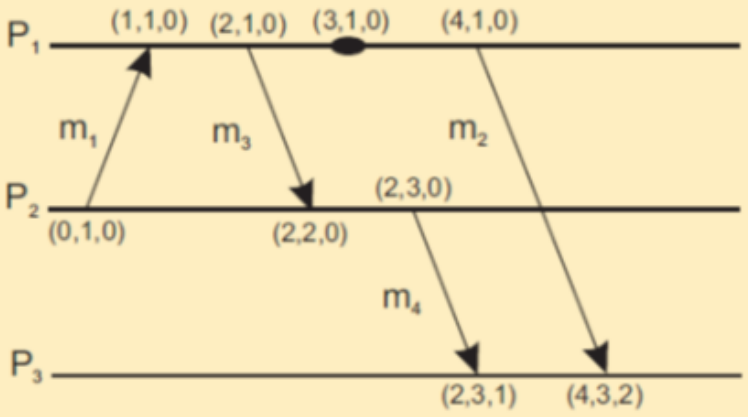
- 时间戳大小关系：ts(a) < ts(b)
- We say $ts(a) < ts(b)$ if and only if:
  - for all k, $ts(a)[k] \leq ts(b)[k]$, and
  - there is at least one index k' : $ts(a)[k'] < ts(b)[k']$
- 向量时钟可以“捕捉”因果依赖性
  - a→b等价于ts(a) < ts(b)
  - ts(a) 与ts(b)不可比，则得出：a与b并发

向量时钟与因果关系

Capturing potential causality when exchanging messages



(a)



(b)

Analysis

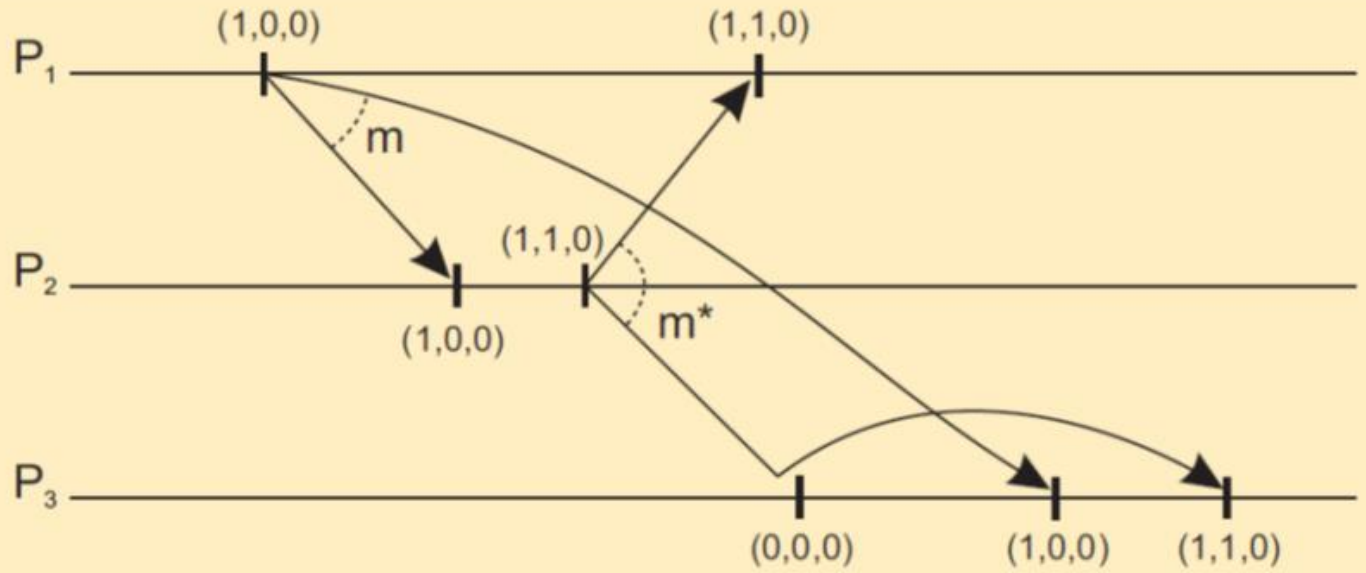
Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	**Conclusion**
(a)	(2,1,0)	(4,3,0)	Yes	No	m_2 may causally precede m_4
(b)	(4,1,0)	(2,3,0)	No	No	m_2 and m_4 may conflict

应用示例：因果有序多播

- Causally Ordered Multicasting
  - 因果有序的多播比“全序多播”更弱：
  - 如果两个消息没有因果关系，则不需要强制统一顺序。
- 向量时钟可以确保：
  - 所有因果在先的消息先于当前消息被接受(delivered)。
- 操作
  - 时钟仅当发送或“接受”消息时才调整VC_j；
  - P_j 推迟“接受”消息 m直到：
    - $ts(m)[i] = VC_j[i] + 1$ //m是P_j应该从P_i接收的消息
    - $ts(m)[k] \leq VC_j[k]$，$k \neq i$ // P_j已经接受所有P_i在发送m时已经接受的消息

因果有序多播

- 强制因果有序的通信



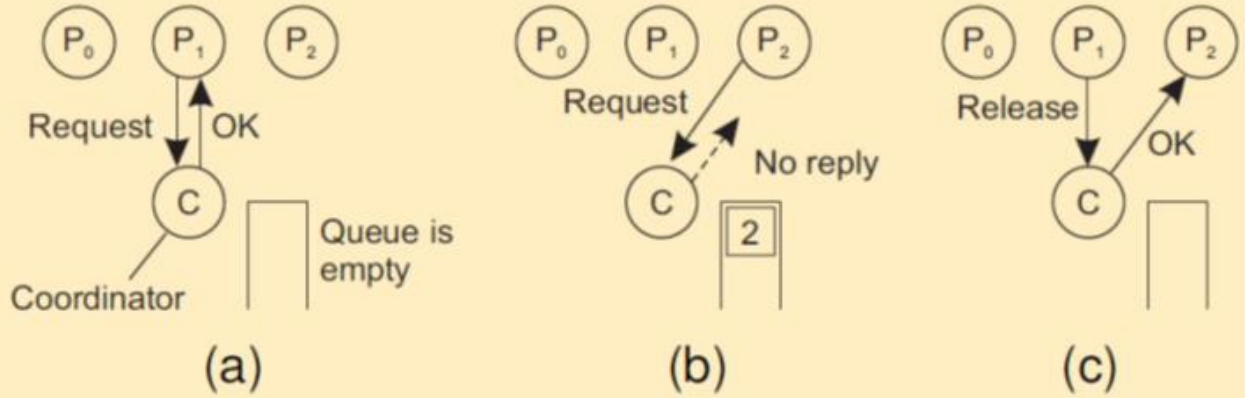
  - 当P_3收到 m*来自P_2，需要推迟“接受”，直到收到来自P_1的m

§6.3 互斥

- 系统中的多个进程需要互斥地访问某些资源
- 基于许可的方法：
  - 如果进程需要访问临界区或者访问资源，需要从其他进程获得许可。
- 基于令牌的方法：
  - 令牌（特殊的消息）在进程之间传递，收到令牌的进程可以访问临界区。
- 正确性要求：
  - 公平性（starvation free）、活性（deadlock free）

集中式算法

- 单一协作者算法



(a) Process P_1 asks the coordinator for permission to access a shared resource. Permission is granted.

(b) Process P_2 then asks permission to access the same resource. The coordinator does not reply.

(c) When P_1 releases the resource, it tells the coordinator, which then replies to P_2.

非集中式算法

- 要点
  - 每种资源有 N 个副本，每个副本有自己的协作者，用于控制访问。
  - 进程只要获得 m > N/2 个协作者的大多数投票就可以访问资源。一个协作者总是立即响应请求。
- 前提
  - 当一个协作者宕机后，它会迅速恢复，但会忘记已经发出去的许可。
- 目的
  - 减少单个协作者由于失效造成的影响，同时提高性能。

非集中式算法

- 健壮性分析

How robust is this system?

- Let $p = \Delta t / T$ be the probability that a coordinator resets during a time interval Δt, while having a lifetime of T.
- The probability $\mathbb{P}[k]$ that k out of m coordinators reset during the same interval is

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- f coordinators reset $\Rightarrow$ correctness is violated when there is only a minority of nonfaulty coordinators: when $m - f \leq N/2$, or, $f \geq m - N/2$.
- The probability of a violation is $\sum_{k=m-N/2}^{N} \mathbb{P}[k]$.

非集中式算法

- 健壮性分析

$$\sum_{k=m-N/2}^{N} \mathbb{P}[k].$$

Violation probabilities for various parameter values

N	m	p	Violation
8	5	3 sec/hour	$< 10^{-15}$
8	6	3 sec/hour	$< 10^{-18}$
16	9	3 sec/hour	$< 10^{-27}$
16	12	3 sec/hour	$< 10^{-36}$
32	17	3 sec/hour	$< 10^{-52}$
32	24	3 sec/hour	$< 10^{-73}$

N	m	p	Violation
8	5	30 sec/hour	$< 10^{-10}$
8	6	30 sec/hour	$< 10^{-11}$
16	9	30 sec/hour	$< 10^{-18}$
16	12	30 sec/hour	$< 10^{-24}$
32	17	30 sec/hour	$< 10^{-35}$
32	24	30 sec/hour	$< 10^{-49}$

分布式算法Ricart & Agrawala 算法

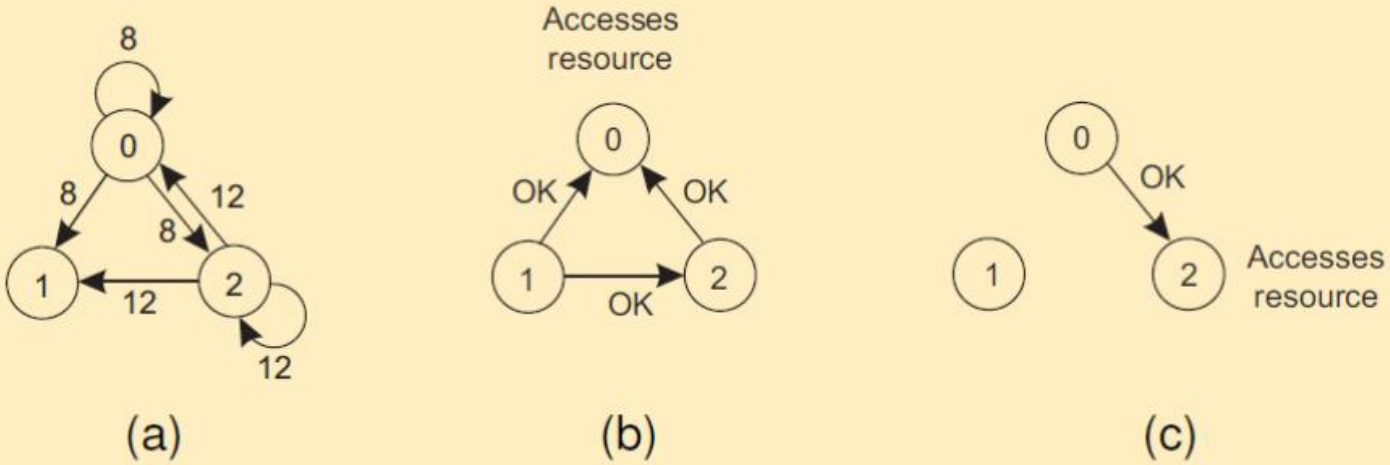
- Ricart & Agrawala 算法
- 要求系统中的所有事件都是完全排序的。
- 对于每对事件，比方说消息，哪个事件先发生都必须非常明确。
- 进程要访问共享资源时，构造一个消息，包括资源名、它的进程号和当前逻辑时间。
- 然后发送给所有的其他进程。

Ricart & Agrawala互斥算法

- 接收进程接收到消息的决策动作分为三种情况：
- 自身没有在临界区且没有访问需求：向发送者返回OK消息；
- 已获得对资源的访问：不应答，将请求放入队列；
- 自身有访问请求在等待：比较访问请求的时间戳，
  - 时间戳早的那个进程获胜，如果接收到的消息的时间戳比较早，那么返回一个OK消息。
  - 如果它自己的消息的时间戳比较早，那么接收者将收到的消息放入队列中，并且不发送任何消息。

Ricart & Agrawala互斥算法

- 三个进程例子



(a) Two processes want to access a shared resource at the same moment.
(b) P_0 has the lowest timestamp, so it wins.
(c) When process P_0 is done, it sends an *OK* also, so P_2 can now go ahead.

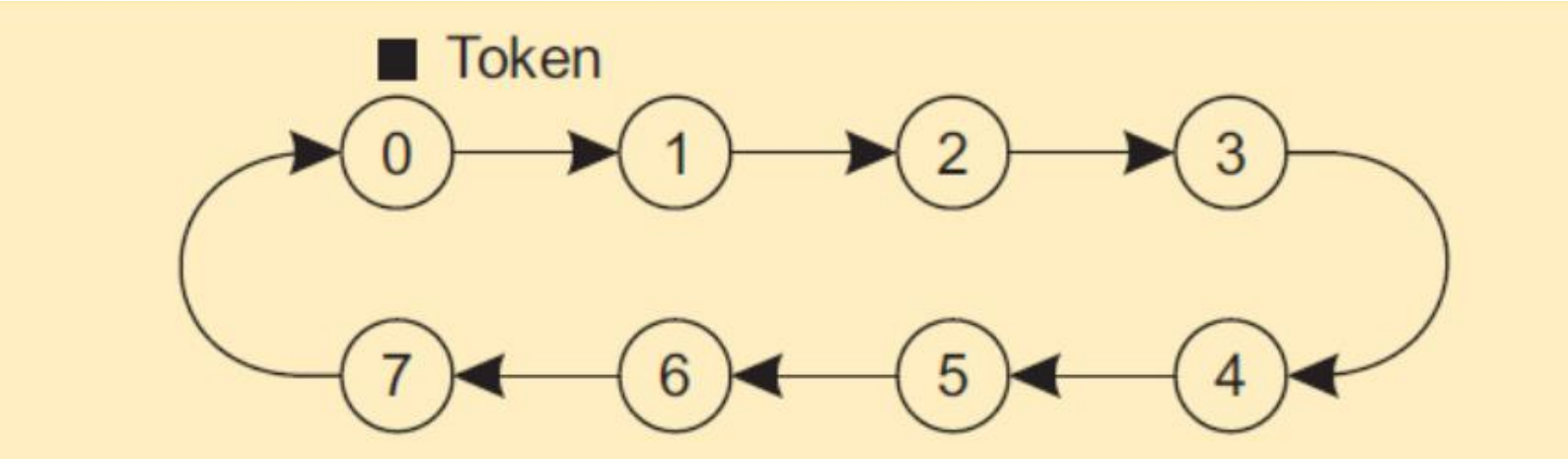
Ricart & Agrawala互斥算法

- 两个问题：
- 单个故障点被n个故障点所取代
  - 如果任何一个进程崩溃，就不能回答请求，造成阻塞。
    故障概率提高了 n 倍， 同时网络流量大大增加。
- 进程维护开销较大，不适用进程数目较多的情况。

如何解决?

令牌环算法

- 要点：
  - 将进程组织成逻辑环，令牌在这些进程之间传递。
  - 拥有令牌的进程允许进入临界区。
- 覆盖网络构成一个逻辑环



互斥算法比较

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed	$2 \cdot (N-1)$	$2 \cdot (N-1)$
Token ring	$1, \ldots, \infty$	$0, \ldots, N-1$
Decentralized	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

§6.3 选举算法

- 需求
  - 某些算法需要一些进程作为一个协作者
  - 问题是如何动态的选择这个特殊的进程
- 注意
  - 在很多系统中，协作者是手动选取的
  - 这会导致集中化的单点失效问题
- 问题
  - 选举算法有多大程度设计成分布式或者集中式的解决方案?
  - 完全分布式的方案是否总是比集中式的方案更加鲁棒?

选举算法基本假设

- 所有的进程都有唯一的ID；
- 所有的进程都知道系统中的其他进程的ID，但是不知道进程是运行还是停止；
- 选举过程就是要找到具有最大ID的运行的进程。

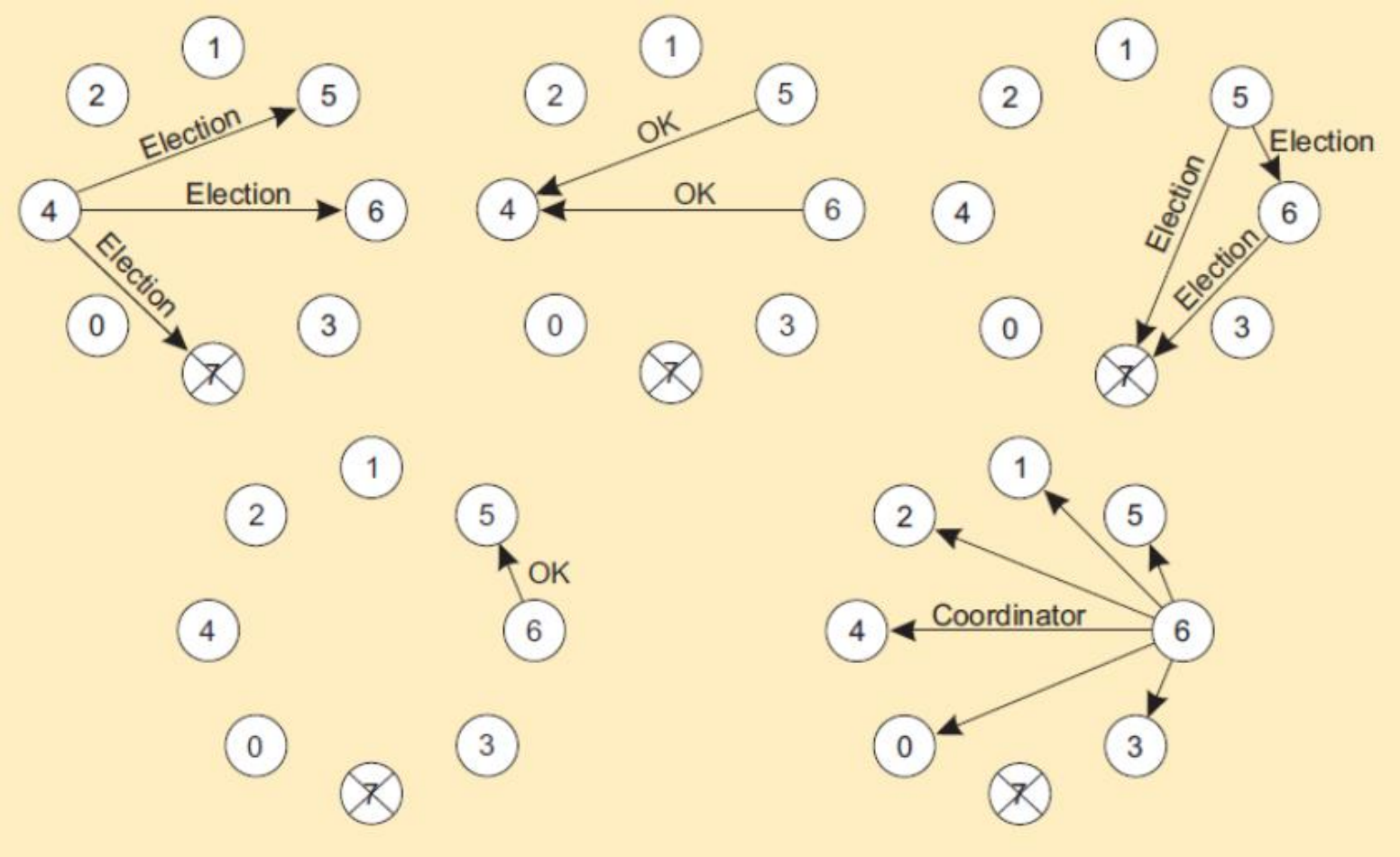
Bully算法

Principle

Consider N processes $\{P_0, \ldots, P_{N-1}\}$ and let $id(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:

1. P_k sends an *ELECTION* message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
2. If no one responds, P_k wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and P_k's job is done.

Bully算法

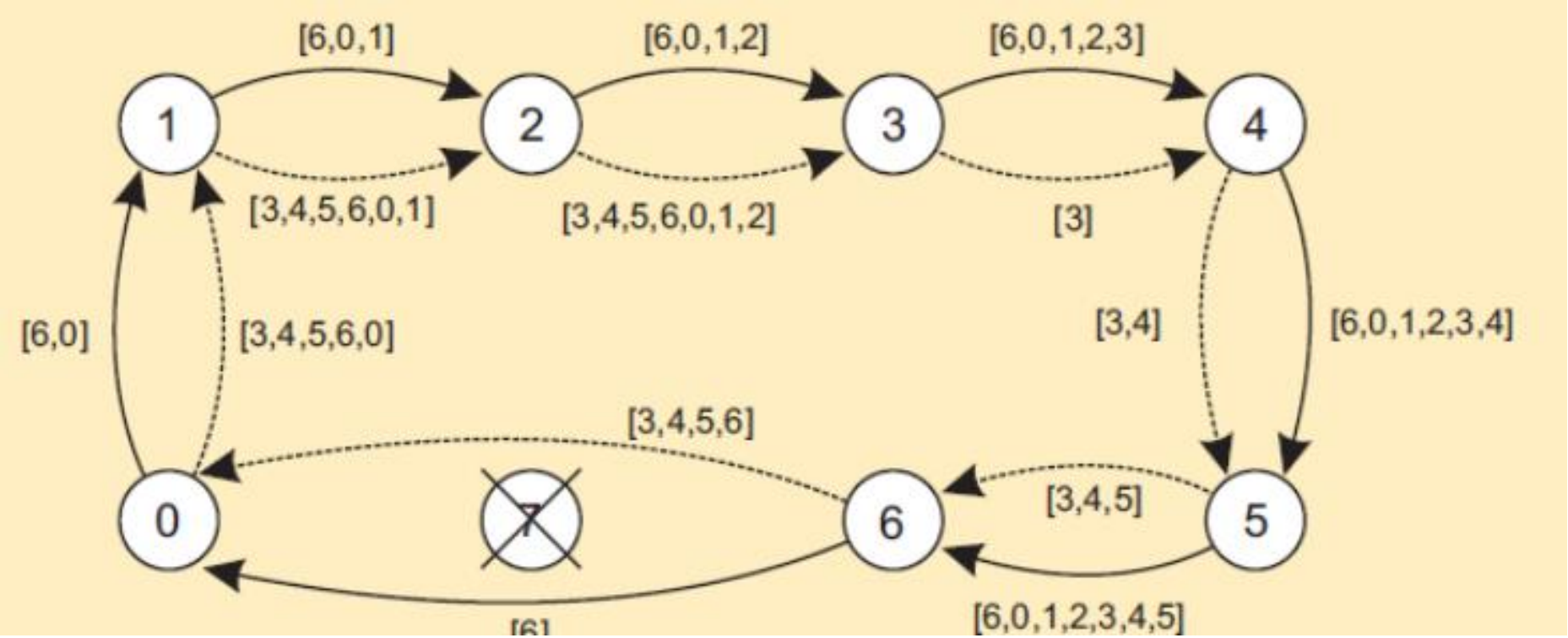


(a) Process 4 holds an election.
(b) (b) Processes 5 and 6 respond, telling 4 to stop.
(c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop.
(e) Process 6 wins and tells everyone.

Ring算法

- 进程按照物理或逻辑顺序排序，组织成环形结构。
- 任何进程都可以启动一次选举过程，该进程把选举信息传递给后继者。
  - 如果后继者崩溃，消息再依次传递下去。
- 在每一步传输过程中，发送者把自己的进程号加到该消息的列表中，使自己成为协作者的候选人。
- 消息返回到此次选举的进程，发起者接收到一个包含它自己进程号的消息时，消息类型变成Coordinator消息
- 再次绕环向所有进程通知谁是协作者、谁在系统中。

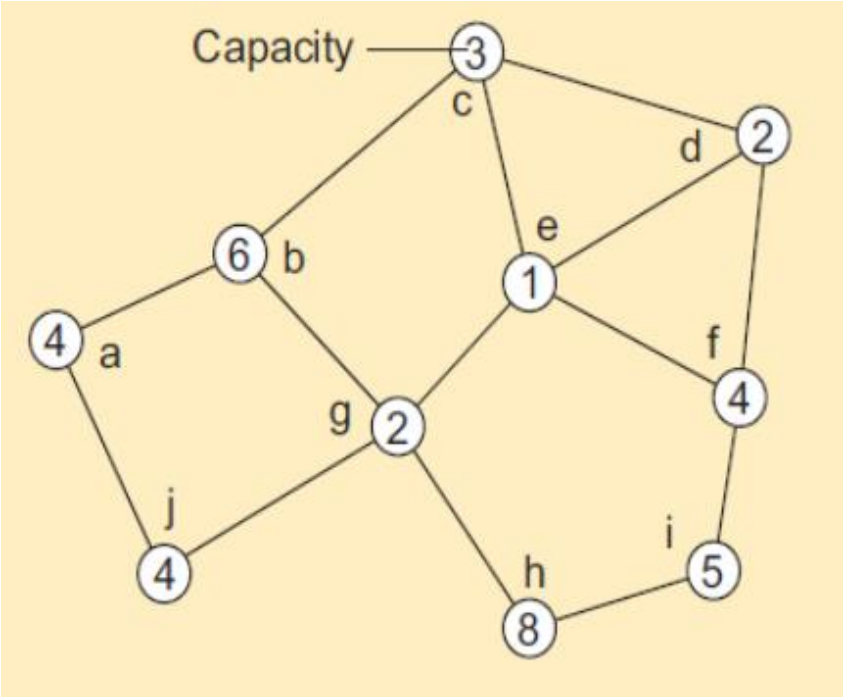
Ring算法



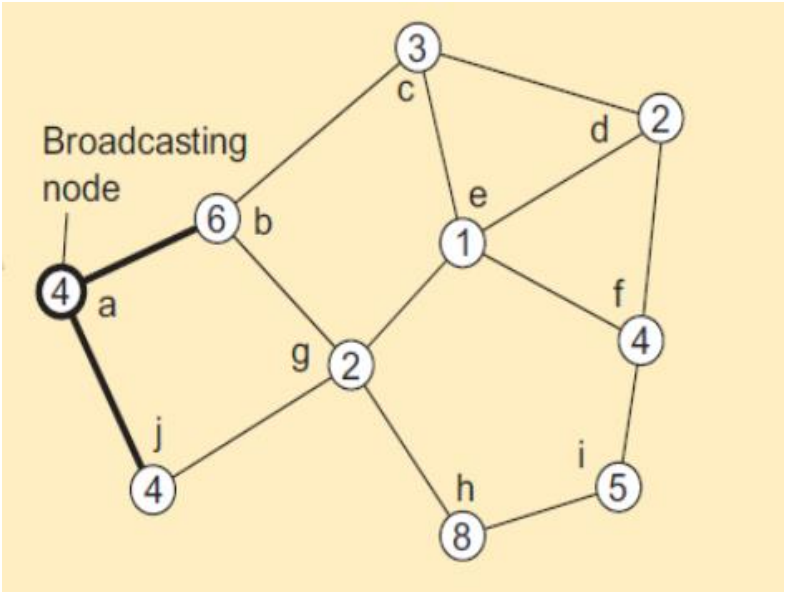
The solid line shows the election messages initiated by P6;
The dashed one those by P3.

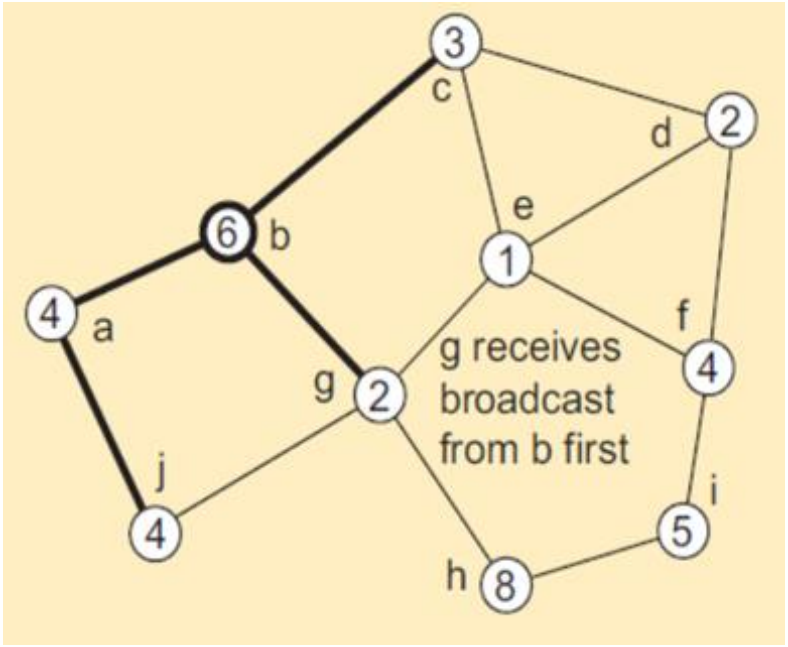
无线环境中的选举

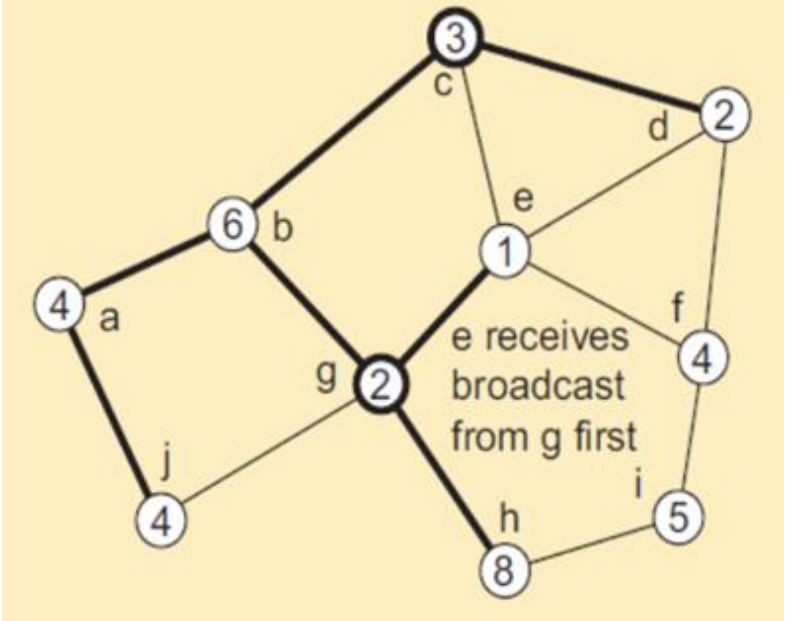
- 与传统系统中的选举不同
  - 传统算法假设消息传送是可靠的，网络拓扑是不会改变的；
  - 无线系统：消息以无线广播方式传输，网络拓扑动态的。

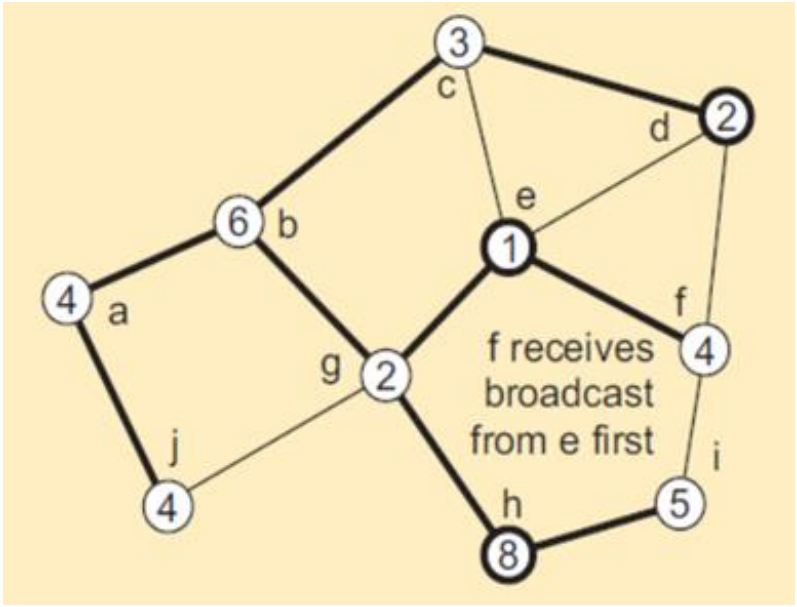


无线环境中的选举


Broadcasting
node
3
c
d
2
6
b
e
1
4
a
f
4
g
2
j
i
4
5
h
8

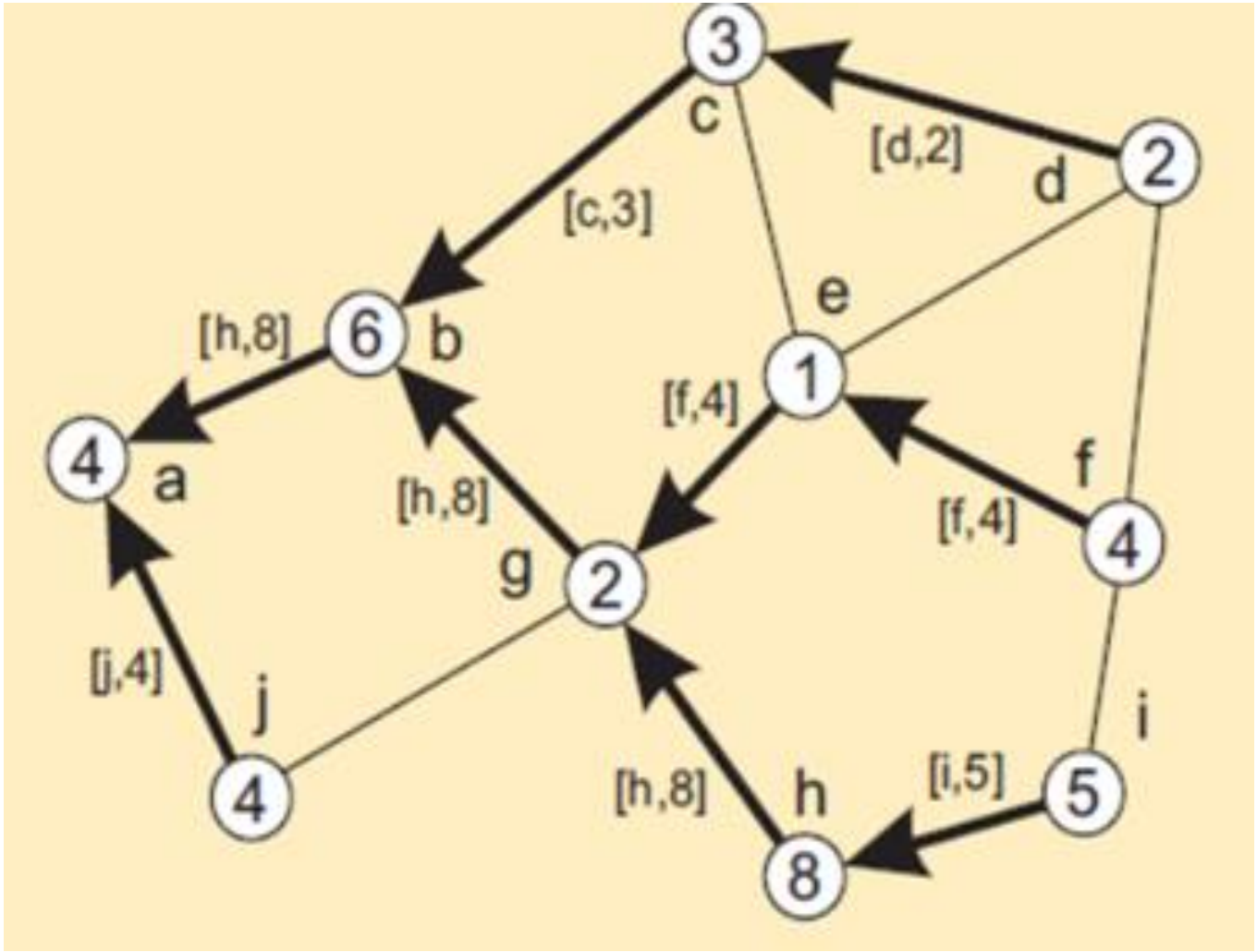

3
c
d
2
e
6
b
1
4
a
f
4
g
2
g receives
broadcast
from b first
j
i
4
5
h
8


3
c
d
2
e
6
b
1
4
a
f
4
g
2
e receives
broadcast
from g first
j
i
4
5
h
8


3
c
d
2
e
6
b
1
4
a
f
4
g
2
f receives
broadcast
from e first
j
i
4
5
h
8

无线环境中的选举

- 基于前面阶段构造的树结构，反向汇聚结果到发起节点



大规模系统中的选举

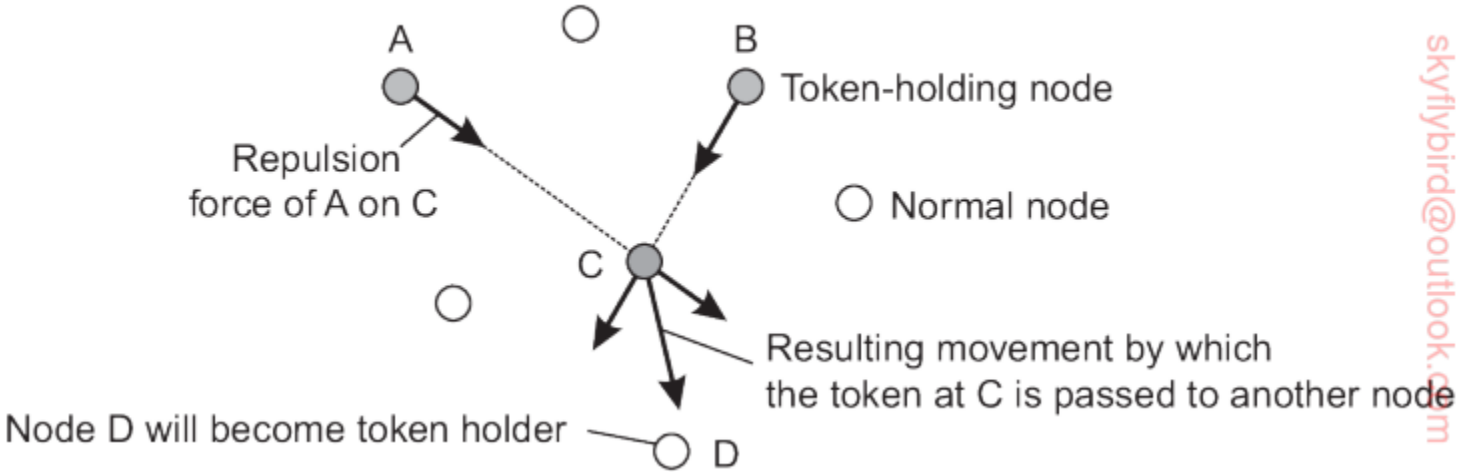
- 需求：
  - 选举多个leader/coordinator
  - 如选举p2p系统中的超级节点
- 假设：
  - Normal nodes should have low-latency access to super peers.
  - Super peers should be evenly distributed across the overlay network.
  - There should be a predefined portion of super peers relative to the total number of nodes in the overlay network.
  - Each super peer should not need to serve more than a fixed number of normal nodes.

DHT中的超级节点选举

- Each node receives a random and uniformly assigned *m*-bit identifier.
- Super peers can be elected (identified) by the first (i.e., leftmost) *k* bits.
- E.g.:
  - Chord system with $m = 8$ and $k = 3$
  - Identifying super peers：ID ^ 11100000
  - Looking up the key K: K ^11100000

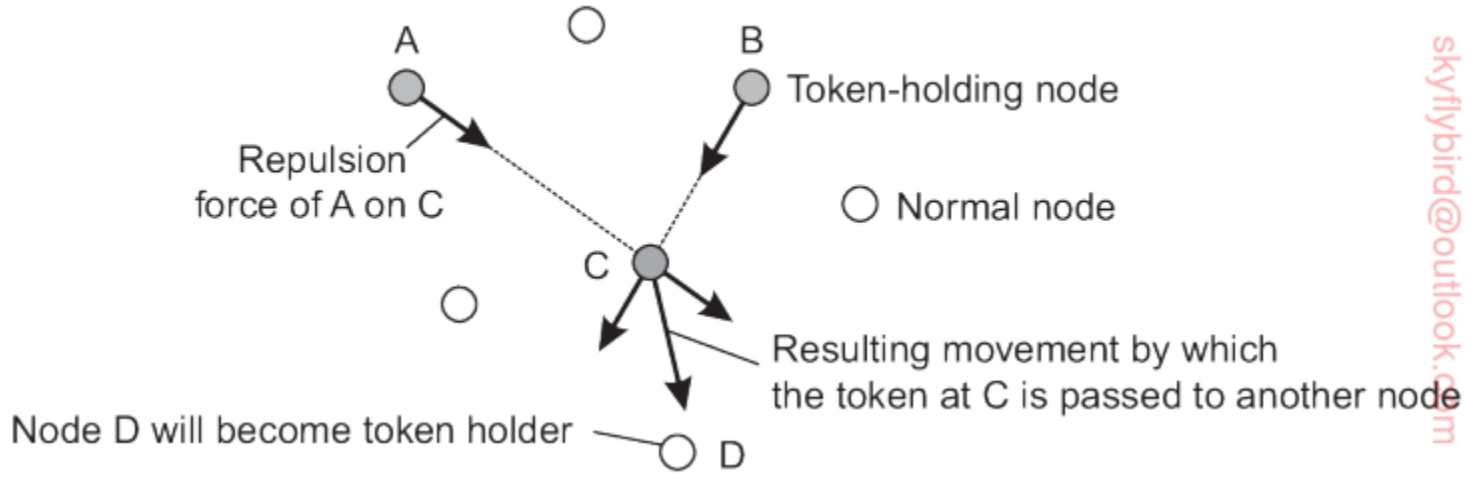
基于空间的超级节点选举

- 通过放置 *N*个token实现放置超级节点
- Each token represents a repelling force by which another token is inclined to move away.
- A gossiping protocol can be used to propagate a token’s force.



基于空间的超级节点选举

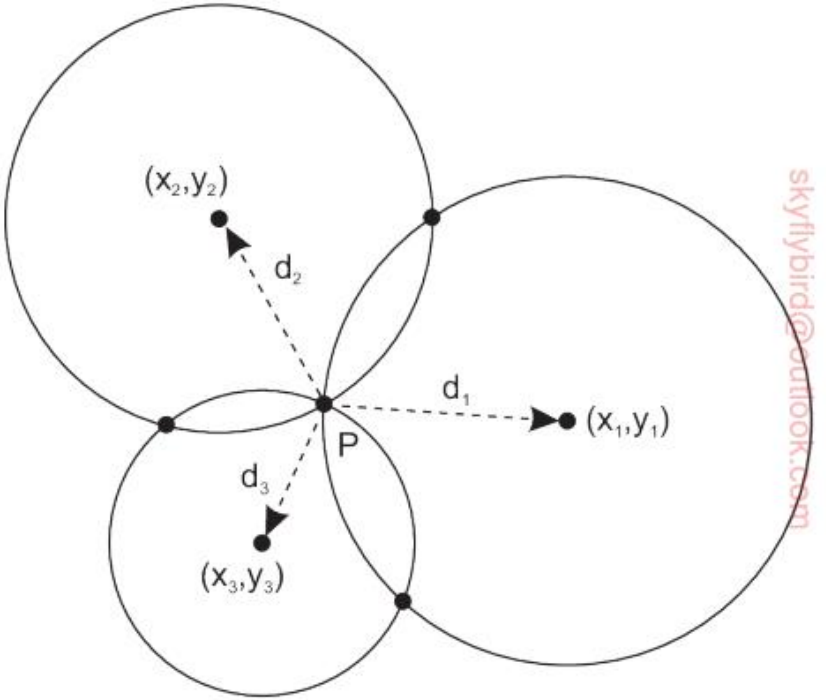
- If the total forces acting on a node exceed a threshold, the node will move the token in the direction of the combined forces.
- When a token is held by a node for a given amount of time, that node will promote itself to super peer.



§6.5 定位系统

- 有些大型系统（广域网范围）中节点位置及位置关系很重要
  - 譬如Overlay的构建
- GPS系统
  - 基于地球同步卫星计算位置
- 室内定位系统
  - WiFi…

A satellite continuously broadcasts its position, and time stamps each message with its local time.



Up to 72 satellites each circulating in an orbit at a height of approximately 20,000 km. Each satellite has up to four atomic clocks regularly calibrated.

逻辑定位系统

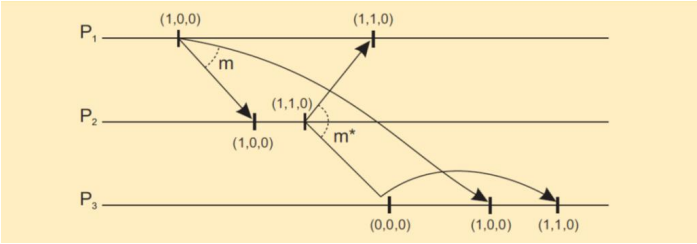
- Geometric overlay networks
  - a logical, proximity-based location
  - Network Coordinates System
- 应用场景
  - Location based client request distributing
  - Replica placement
  - position-based routing
- 基本方法
  - Landmark节点（类似GPS卫星）
  - 非集中的方法（基于相互距离）

Summary

- 同步是分布式系统的基本问题、核心问题
  - 时间 → 顺序
- 时钟同步：最根本的途径
  - 基于交换时钟值：UTC、GPS；NTP、Berkeley算法
- 顺序协同：更直接的途径
  - Happens-Before关系
  - Lamport逻辑时钟、向量时钟
- 进程同步：协作者控制、协调行为顺序
  - 互斥、选举
- 定位系统
  - 地理坐标、逻辑坐标

课后作业

1. 从分布式系统的角度，如果事件a和b具有happens-before的关系，a→b，我们说b对a有因果依赖。那么，从现实世界的角度，事件b的发生一定是与a有关系吗？为什么？
2. 在强制因果有序多播的例子中，如果发送、接收消息各作为一个事件增加时钟计数，如何修改算法中消息交付操作才能满足要求？



3. 基于环的选举算法中，如果两个Election消息同时在循环时，可以杀掉其中一个。设计一个机制实现这个功能。

谢谢！

wuweig@mail.sysu.edu.cn