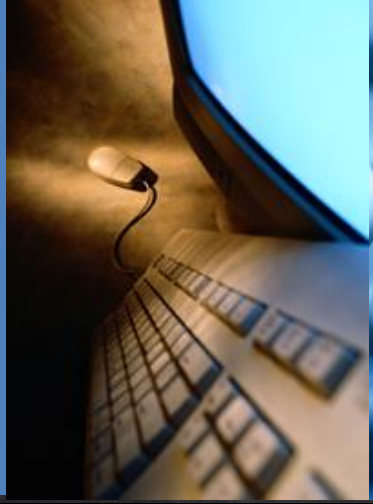



分布式系统
Distributed Systems

陈鹏飞
数据科学与计算机学院
chchenpf7@mail.sysu.edu.cn
办公室：超算5楼529d
主页：http://sdcs.sysu.edu.cn/node/3747



第六讲 — 同步

引言

过程之间的协同，完成一个整体过程

大纲

1. 时钟同步
2. 逻辑时钟
3. 互斥
4. 选举算法
3. 小结

1 时钟同步

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；

编译器示例

- 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；

- 分布式环境中的编译

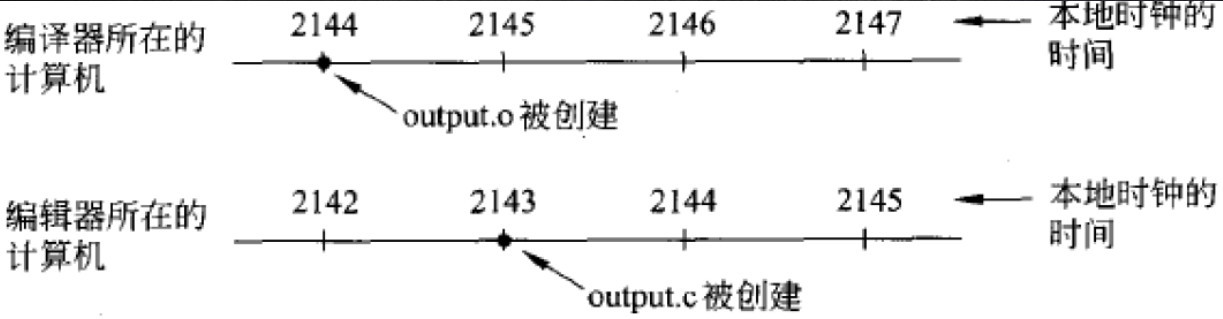


图 6.1 当每台机器都有它自己的时钟时，一个发生于另一个事件之后的事件可能会被标记上较早的时间

物理时钟

- **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）,用于补偿国际时钟与太阳秒2之间的误差；
- **注意**
  - UTC时间通过无线电和卫星通过短波广播（WWV）。为性传播可能导致±0.5ms的误差；

时钟同步

➢ 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 $C_p(t)$ 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：

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

时钟漂移

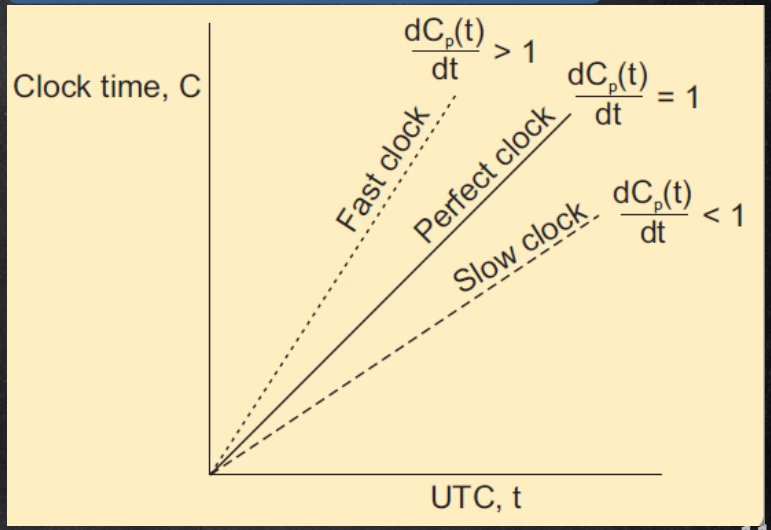
➢ 观察发现

- 利用硬件时钟中断，我们将软件时钟与硬件时钟耦合时会产生时钟漂移：

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

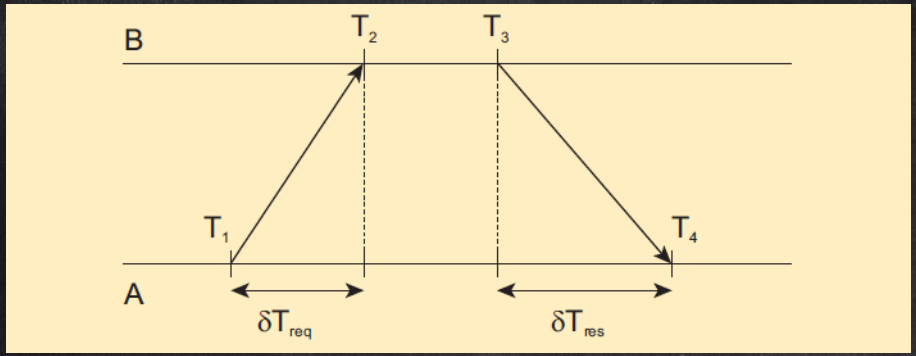
$$\Rightarrow \forall t : 1-\rho \le \frac{dC_p(t)}{dt} \le 1+\rho$$

➢ 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



- 服务器之间的时间偏差和延迟为：

Assumption: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

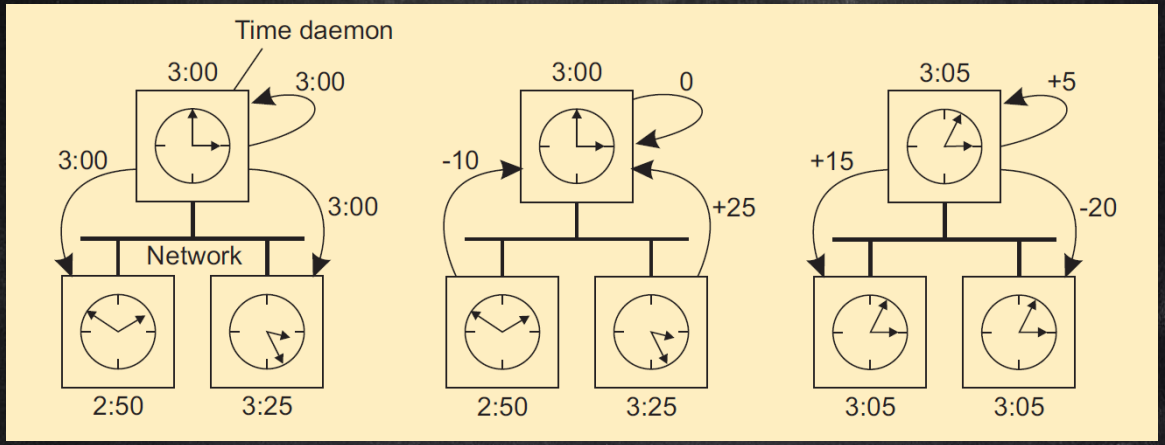
$$\theta = T_3 + ((T_2 - T_1) + (T_4 - T_3))/2 - T_4 = ((T_2 - T_1) + (T_3 - T_4))/2$$

$$\delta = ((T_4 - T_1) - (T_3 - T_2))/2$$

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**
  - 原理：时间服务器周期性地扫描所有的服务器，计算时间均值，并告诉所有其他机器如何根据当前时间进行调整；
- **利用时间服务器**



无线网络中的时钟同步

- 参考广播同步化（RBS）
  - 一个节点广播参考信息 m， => 每个接收节点 p 记录收到消息m 的$T_{p,m}$， $T_{p,m}$ 为本地时间；

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

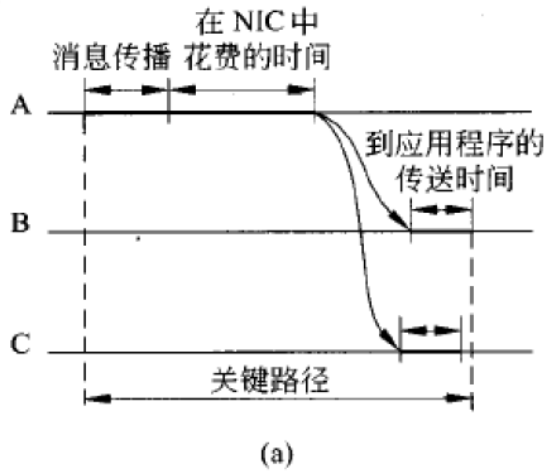
- 问题：时钟是会变化的，计算平均值无法包含时钟漂移 => 使用线性回归 （LR）；

$$\text{NO:} \quad Offset[p,q](t) = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$$

$$\text{YES:} \quad Offset[p,q](t) = \alpha t + \beta$$

无线网络中的时钟同步

- **RBS最小化关键路径**



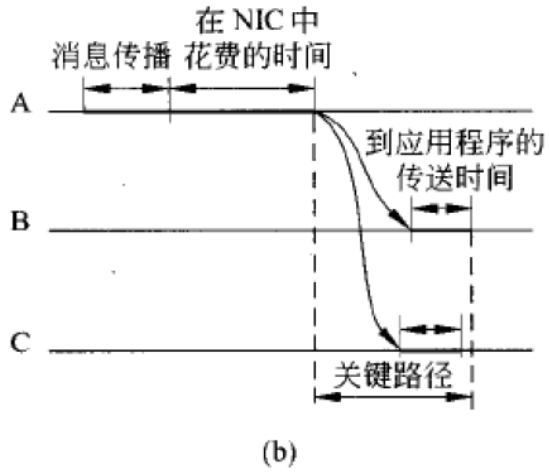


图 6.8

(a) 通常情况下确定网络延时的关键路径；(b) 在 RBS 情况下的关键路径

2 逻辑时钟

Happened-before关系

➢ 问题

- 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”；

➢ Happen-before 的定义

- 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；
- 如果 a 是消息的发送者，b是消息的接收者，则 a->b；
- 如果 a->b 并且 b->c 则 a->c；

逻辑时钟

- **问题**

  如何维护系统行为的全局视图，使其与Happened-before关系保持一致？

- **为每一个事件 e 分配一个时间戳 C(e) 使其满足以下属性：**
  - P1 如果 a 和 b 是同一个进程中的事件，并且 a->b，那么有：C(a) < C(b);
  - P2 如果 a 是信息 m 的发送方，并且 b 是信息的接收者，那么 C(a) < C(b);

问题： 如何在没有全局时钟的情况下为事件分配时间戳？

逻辑时钟

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

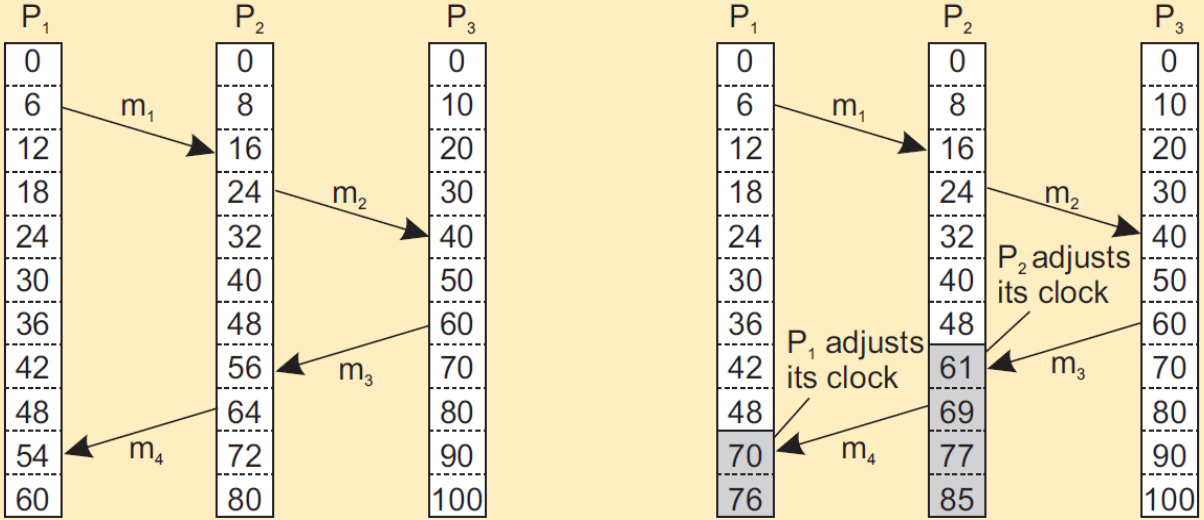
1. For each new event that takes place within P_i, C_i is incremented by 1.
2. Each time a message m is sent by process P_i, the message receives a timestamp $ts(m) = C_i$.
3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step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

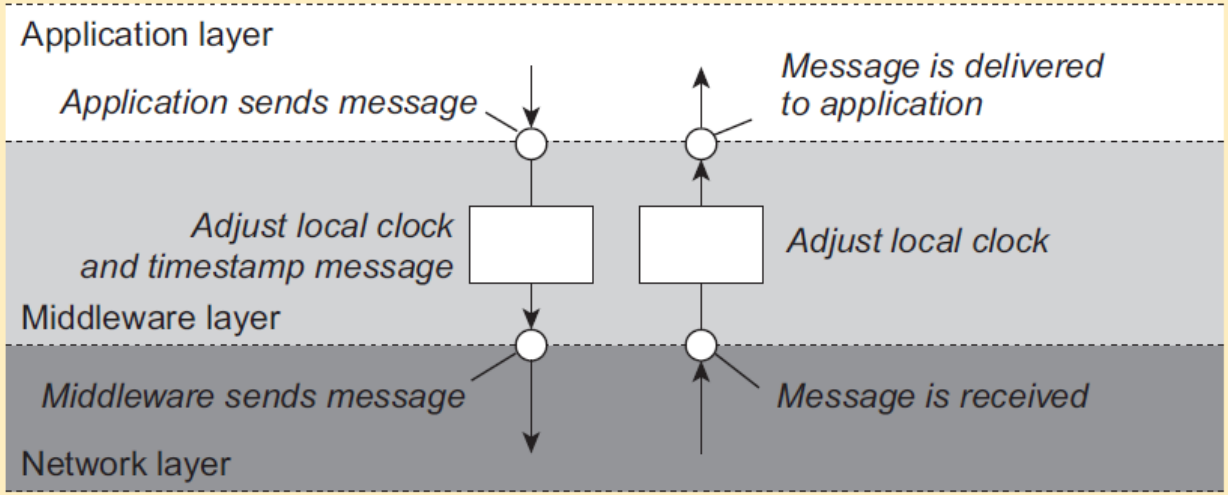
逻辑时钟：例子

Lamport算法校正三个进程的不同时钟



逻辑时钟：如何实现

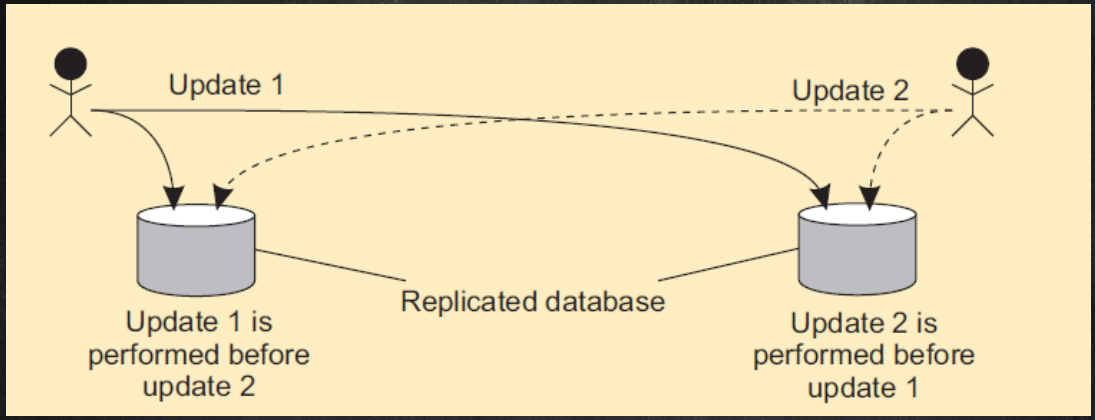
- 在中间件中实现



全序广播（Total-ordered Multicast）

- 在一个副本数据库上的并发更新在任何地方都应该是同样的顺序；

- P1 为一个银行账户（初始为1000 $）添加100$；
- P1 为该账户增加 1%；
- 存在两个副本；



副本1 变成1111$，副本2变成1110$

全序广播（Total-ordered Multicast）

- **解决方法**

  - 进程P1将加上时间戳的消息mi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Pj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传递到它的应用程序**

  - m_i是队列queue_j的队列头；
  - 对于每一个进程P_k，在队列queue_j存在一个消息m_k具有较大的时间戳 （确认消息）；

注意：我们假定通信是可靠的，并且是FIFO排序的；

Lamport逻辑时钟解决互斥访问

- **关键区（Critical Section）**

在同一个时刻至多允许一个进程执行的代码片段。与多播类似，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

```
1  class Process:
2    def __init__(self, chan):
3      self.queue      = []                    # The request queue
4      self.clock      = 0                     # The current logical clock
5
6    def requestToEnter(self):
7      self.clock = self.clock + 1                          # Increment clock value
8      self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
9      self.cleanupQ()                                      # Sort the queue
10     self.chan.sendTo(self.otherProcs, (self.clock,self.procID,ENTER)) # Send request
11
12   def allowToEnter(self, requester):
13     self.clock = self.clock + 1                          # Increment clock value
14     self.chan.sendTo([requester], (self.clock,self.procID,ALLOW)) # Permit other
15
16   def release(self):
17     tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ALLOWs
18     self.queue = tmp                                     # and copy to new queue
19     self.clock = self.clock + 1                          # Increment clock value
20     self.chan.sendTo(self.otherProcs, (self.clock,self.procID,RELEASE)) # Release
21
22   def allowedToEnter(self):
23     commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
24     return (self.queue[0][1]==self.procID and len(self.otherProcs)==len(commProcs))
```

Lamport逻辑时钟解决互斥访问

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

与全序多播类似

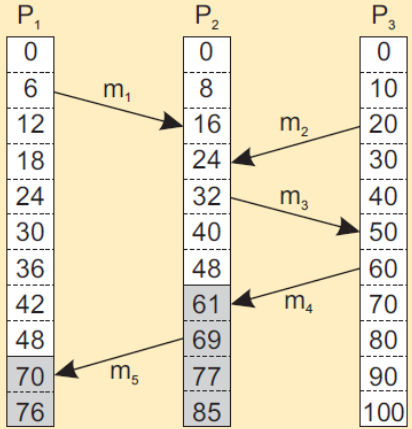
- 利用全序的多播方法，所有的进程建立单独的队列，以相同的顺序发送消息；
- 互斥访问在进程进入关键区的顺序上达成一致；

向量时钟

Observation

Lamport's clocks do not guarantee that if $C(a) < C(b)$ that a causally preceded b.

Concurrent message transmission using logical clocks



Observation

Event a: m_1 is received at $T = 16$;
Event b: m_2 is sent at $T = 20$.

Note

We cannot conclude that a causally precedes b.

因果依赖（causal dependency）

➢ 定义

We say that b may causally depend on a if $ts(a) < ts(b)$, with:

- for all k, $ts(a)[k] \leq ts(b)[k]$ and
- there exists at least one index k' for which $ts(a)[k'] < ts(b)[k']$

➢ 先于发生和依赖

- We say that a causally precedes b.
- b may causally depend on a, as there may be information from a that is propagated into b.

获得因果关系（向量时钟）

- **解决方案：每一个进程 P_i维护一个向量VC_i**

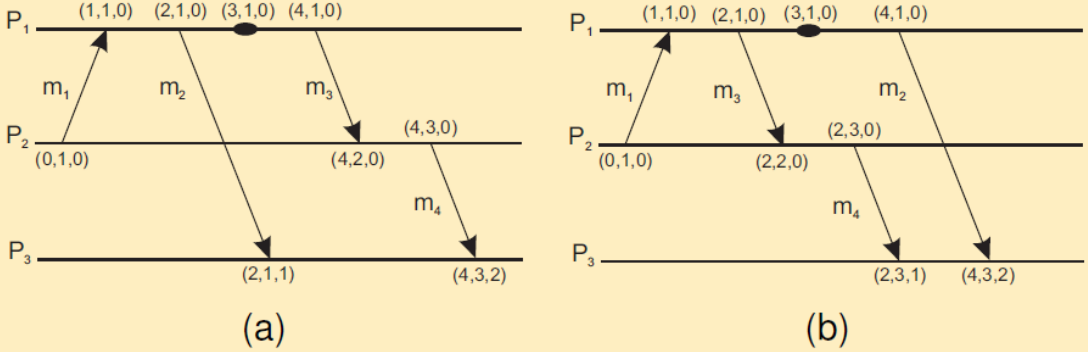
- $VC_i[i]$ is the local logical clock at process P_i.
- If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j.

- **维护向量时钟**

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：样例

Capturing potential causality when exchanging messages



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

因果有序的多播（Causally Ordered Multicasting）

- **与全序多播的关系**

  因果有序的多播比之前提到的全序多播更弱。尤其是如果两个消息互相没有任何关系，并不关心以那种顺序发送给应用程序。

- **观察**

  使用向量时钟，可以确保所有因果先于某个消息的所有消息接收后才传送这个消息。

- **调整**

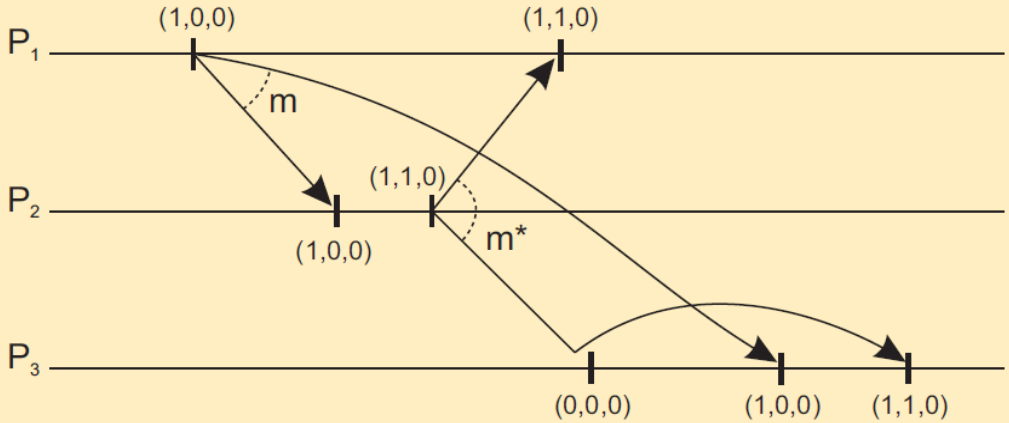
  仅当P_i发送消息时才增加VC[i]，当P_j接收到消息后才调整VC_j；

- **P_j 推迟发送消息 m直到：**

  1. ts(m)[i] = VC_j[i] + 1
  2. ts(m)[k] ≤ VC_j[k]，k ≠ i

因果有序的多播（Causally Ordered Multicasting）

- 强制因果有序的通信



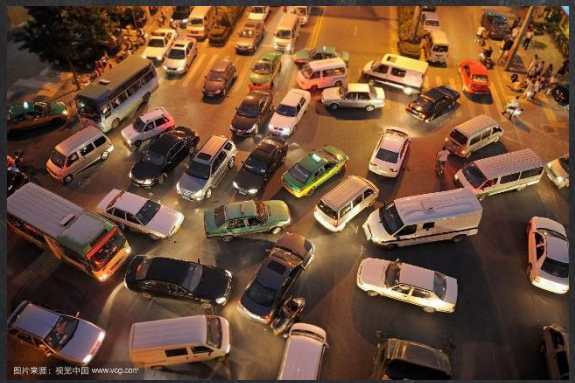
- 强制因果有序的通信

如果 VC_3=[0,2,2], ts(m) = [1,3,0] 来自于P_1，那么当P_3接收到消息 m 的时候如何做？



3

互斥



互斥

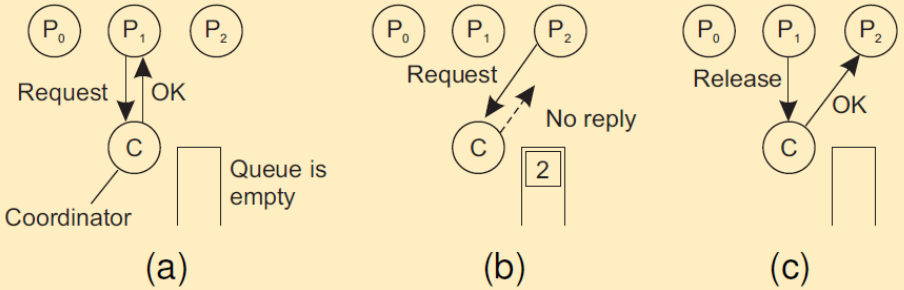
- **问题**
  - 分布式系统中的多个进程需要互斥地访问某些资源
- **基本的解决方法**
  - **基于许可的方法：**

    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：**

    仅有的一个令牌在进程之前传递。 拥有令牌的进程可以访问临界区或者将令牌传递给其他进程。

基于许可的集中式方法

➢ 简单利用协作者



存在的问题？

(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

- **原理**

假定每种资源有 N 个副本，每一个副本都有自己的协作者，用于控制访问。 进程只要活得 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。

- **假设**

当一个协作者宕机后，它会迅速恢复，但是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

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

DHT，存在问题？

非集中式算法

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	$< 10^{-49}$

可以得出什么结论？

分布式算法

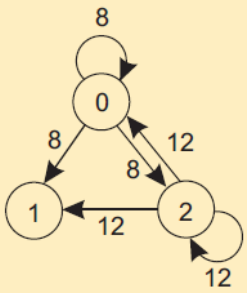
- **问题**
  - 拥有单个故障点往往是可接受，有必要采用分布式互斥算法。
- **Ricart & Agrawala 算法**
  - 要求系统中的所有事件都是完全排序的。对于每对事件，比方说消息，哪个事件先发生都必须非常明确。
  - 进程要访问共享资源时，构造一个消息，包括资源名、它的进程号和当前逻辑时间。然后发送给所有的其他进程。

Ricart & Agrawala互斥算法

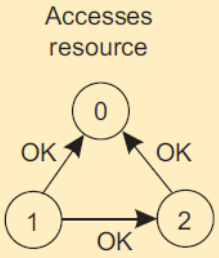
- **接收到消息的决策动作分为三种情况：**
  - 若接收进程没有访问资源，而且也不想访问资源，向发送者返回一个OK消息；
  - 若接收者已获得对资源的访问，那么它就不答应，而是将该请求放入队列中；
  - 如果接收者想访问资源但尚未访问时，它将收到消息的时间戳与它发送到其他进程的消息的时间戳进行比较。时间戳早的那个进程获胜，如果接收到的消息的时间戳比较早，那么返回一个OK消息。如果它自己的消息的时间戳比较早，那么接收者将收到的消息放入队列中，并且不发送任何消息；

Ricart & Agrawala互斥算法

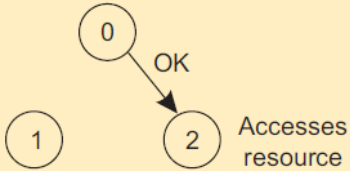
- 三个进程例子



(a)



(b)



(c)

(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.

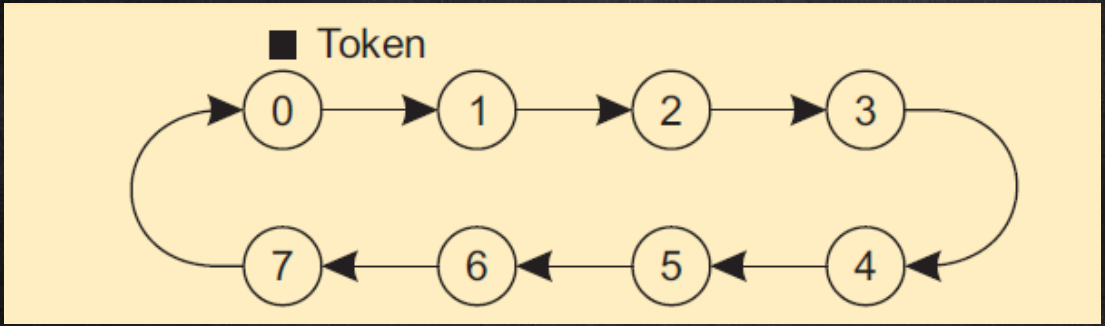
Ricart & Agrawala互斥算法

- 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

- **本质**
  - 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	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

4 选举算法

选举算法

➢ 原理

- 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。这回导致集中化的单点失效问题；

➢ 注意

- 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
  - 选举算法一位置找到具有最大ID的运行的进程；

Bully算法

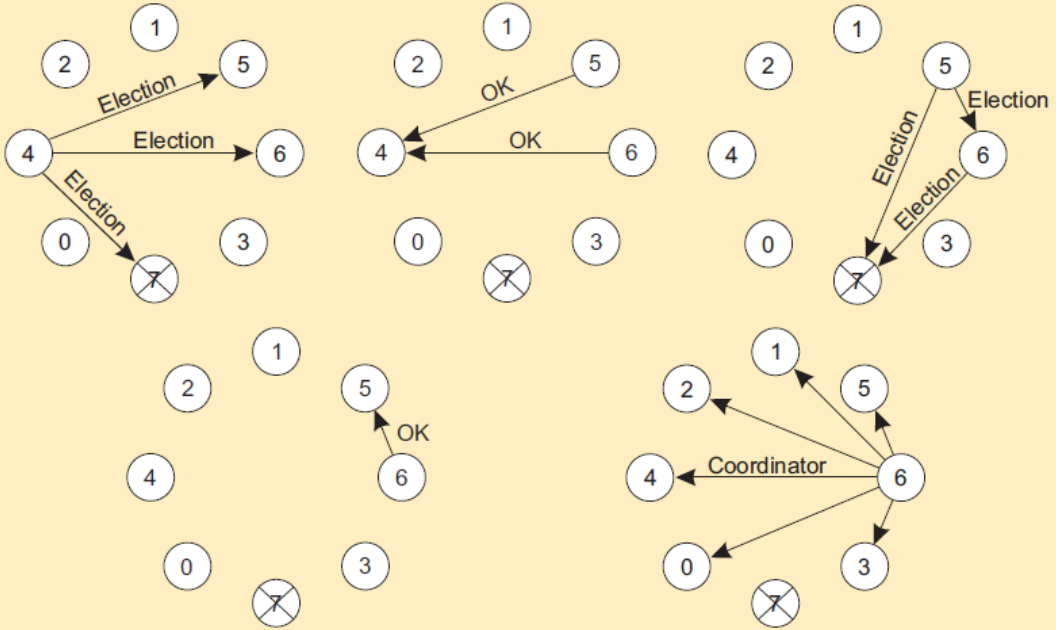
Principle

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

The bully election algorithm



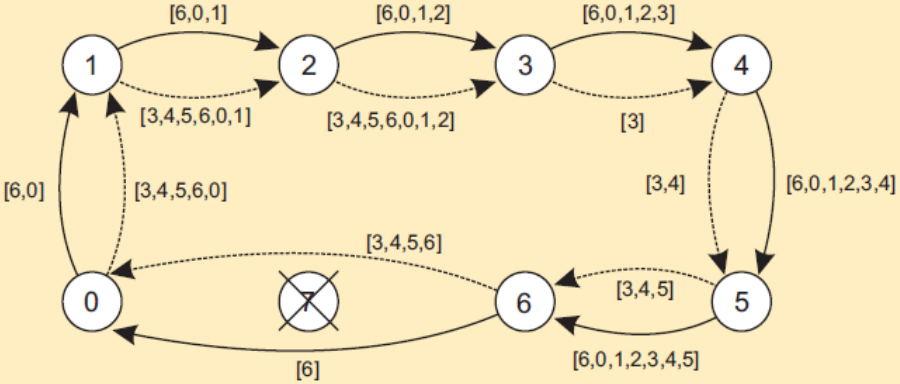
Ring算法

➢ 原理

- 进程按照物理或逻辑顺序进行排序，组织成环形结构。具有最大ID的进程被选为协作者。
- 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，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，并再次绕环向所有进程通知谁是协作者以及新环中的成员都有谁。

Ring算法

Election algorithm using a ring

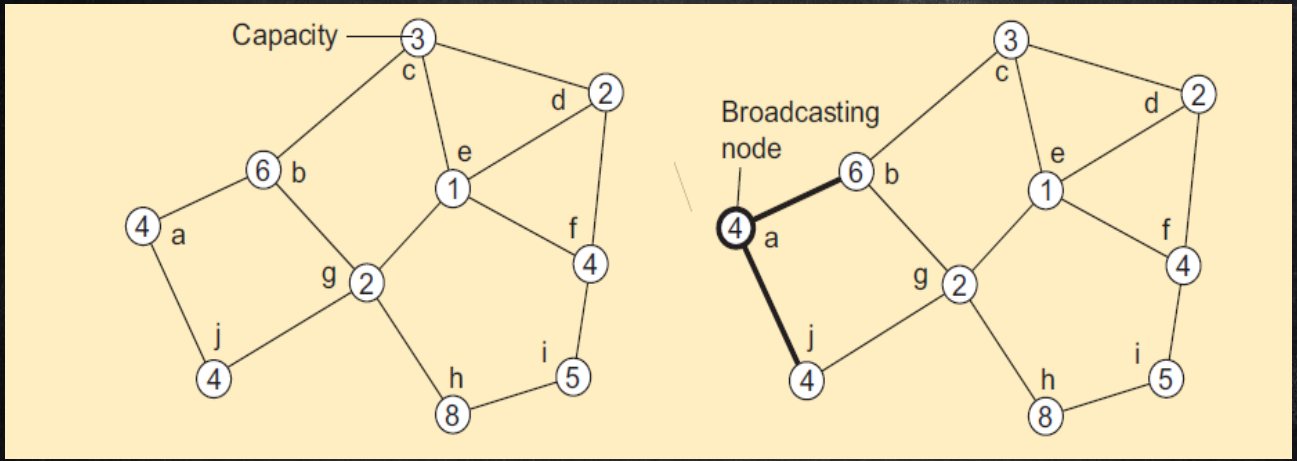


- The solid line shows the election messages initiated by P_6
- The dashed one the messages by P_3

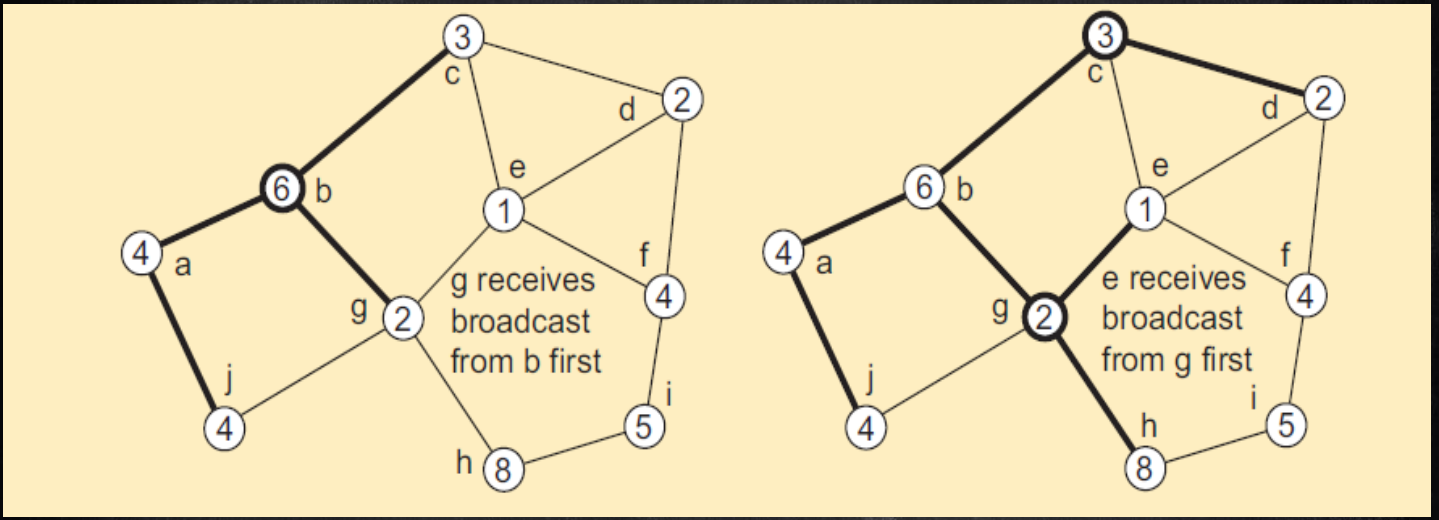
无线系统环境中的选举算法

➢ 传统选举算法的缺点

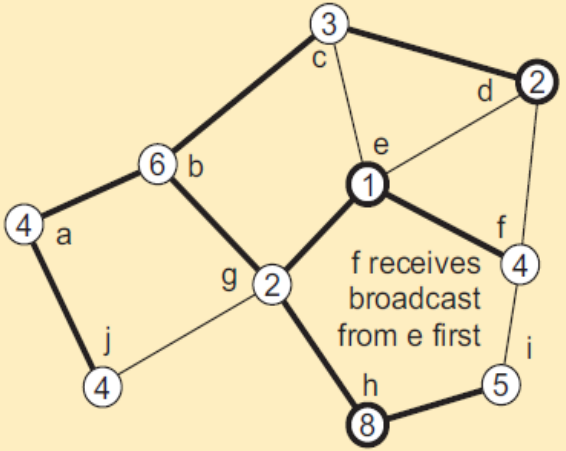
- 传统的选举算法假设消息传送是可靠的，网络的拓扑结构也是不会改变的；

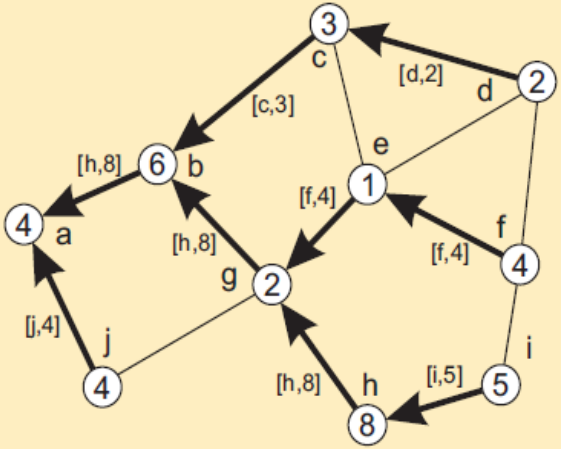


无线系统环境中的选举算法



无线系统环境中的选举算法





谢谢！