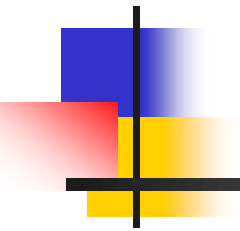
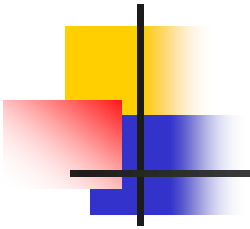


# 第六章 时间和全局状态

# 第6章 时间和全局状态

- 简介
- 时钟、事件和进程状态
- 同步物理时钟
- 逻辑时间和逻辑时钟
- 全局状态
- 分布式调试
- 小结

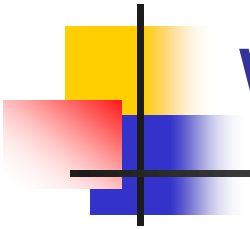


# 简介

- 为什么需求全局时间？
- 如何计时？
- 如何同步时钟？
- 没有物理时钟能否确定事件的顺序？

# 简介

- 时间的重要性
  - 需要精确度量——审计电子商务
  - 某些算法依赖于时钟同步——数据一致性维护
  - 计算全局状态——事件排序
- 时间的复杂性
  - 节点具有独立的物理时钟
  - 精确同步物理时钟非常困难
- 全局状态的捕获
  - 依赖于逻辑时钟
  - 逻辑时钟与物理时钟无必然联系

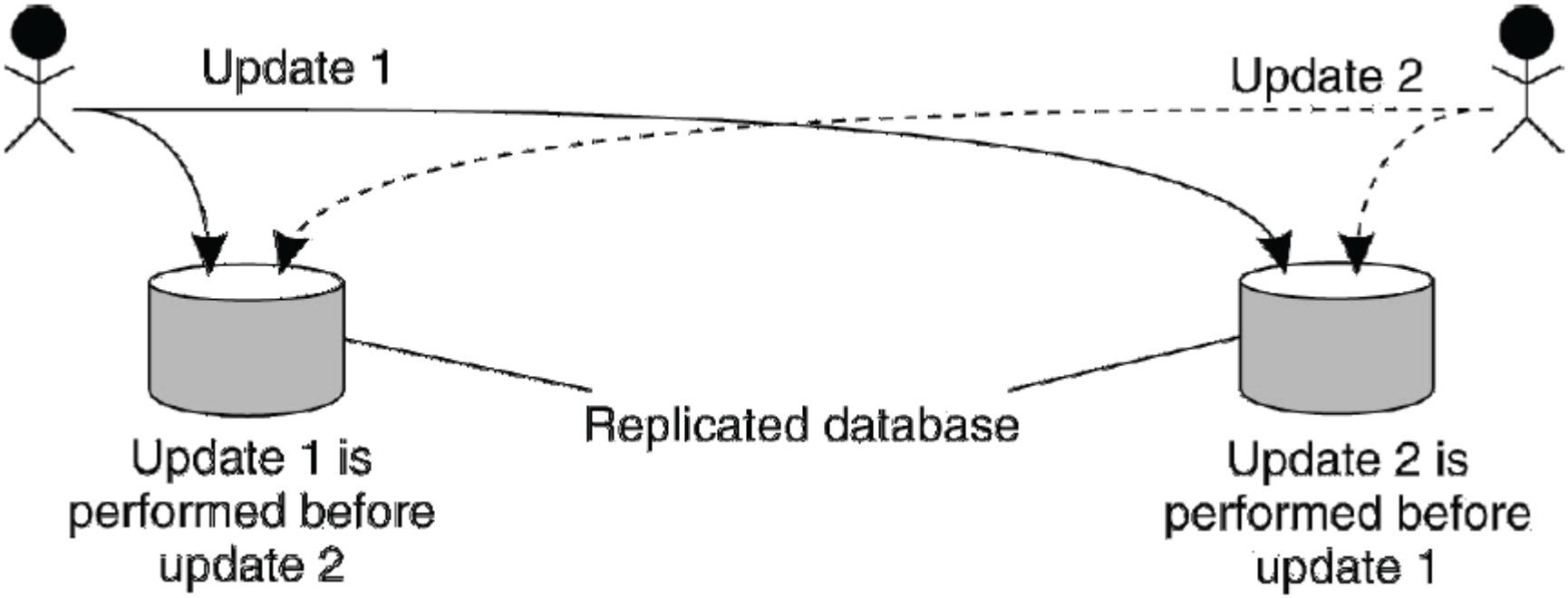


# Why Global Timing?

- Suppose there were a globally consistent time standard
- Would be handy
  - Who got last seat on airplane?
  - Who submitted final auction bid before deadline?
  - Did defense move before snap?

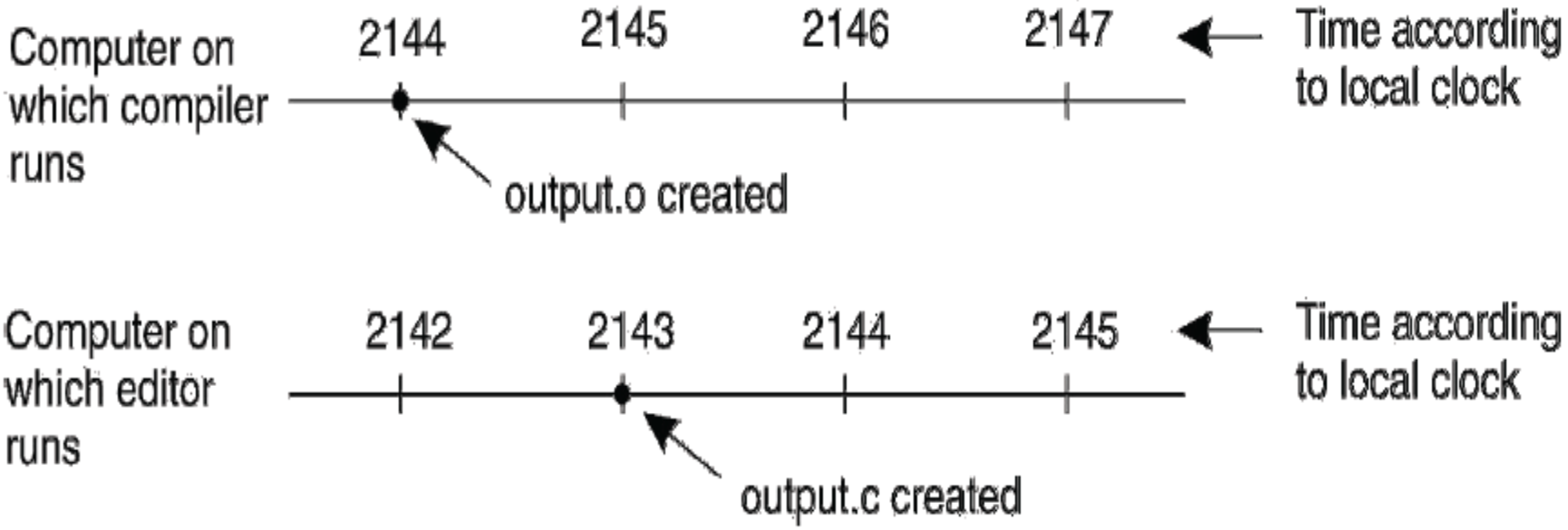
# Replicated Database Update

- Updating a replicated database and leaving it in an inconsistent state



# Impact of Clock Synchronization

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



# 第6章 时间和全局状态

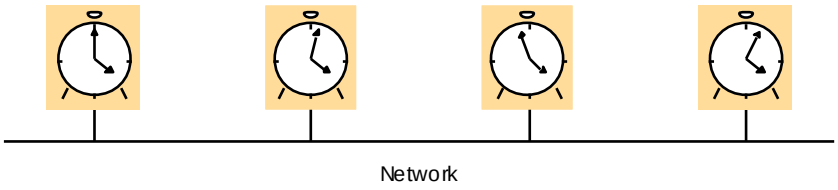
- 简介
- 时钟、事件和进程状态
- 同步物理时钟
- 逻辑时间和逻辑时钟
- 全局状态
- 分布式调试
- 小结

# 时钟、事件和进程状态

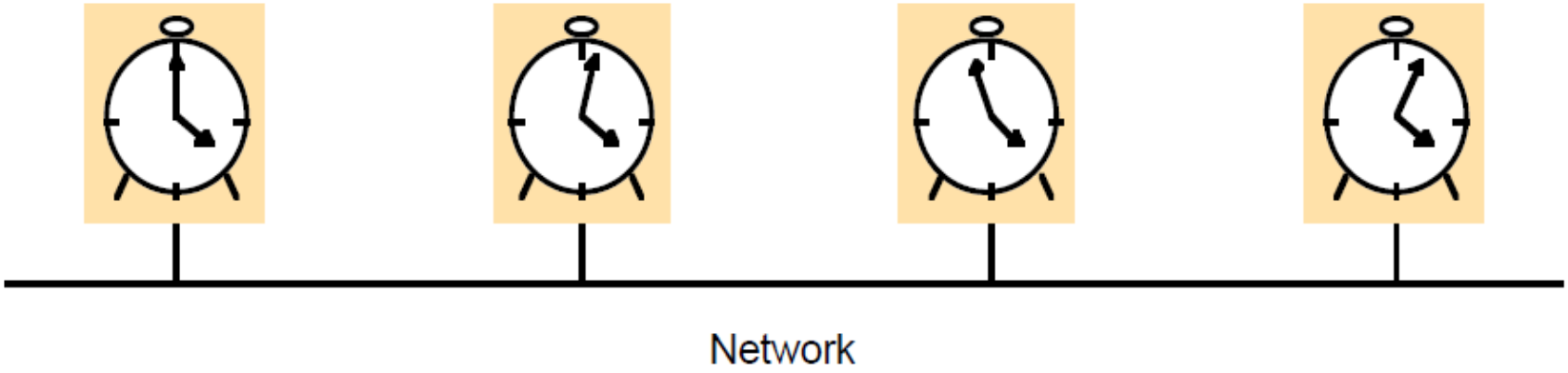
- 假设
  - 每个进程在单处理器上执行
  - 处理器之间不共享内存
  - 进程之间通过消息进行通信
- 进程状态
  - 所有变量的值
  - 相关的本地操作系统环境中的对象的值
- 事件
  - 定义：一个通信动作或进程状态转换动作
  - 进程历史：$history(p_i)=<e_i^0, e_i^1, e_i^2, \ldots>$

# 时钟、事件和进程状态

- 计算机时钟
  - 晶体具有固定震荡频率
  - 硬件时钟：$h_i(t)$
  - 软件时钟：$C_i(t)=\alpha h_i(t)+\beta$
- 时钟漂移
  - 频率不同
  - 时钟频率随温度变化而有所差别
- 时钟偏移不可避免



# Clocks in a Distributed System

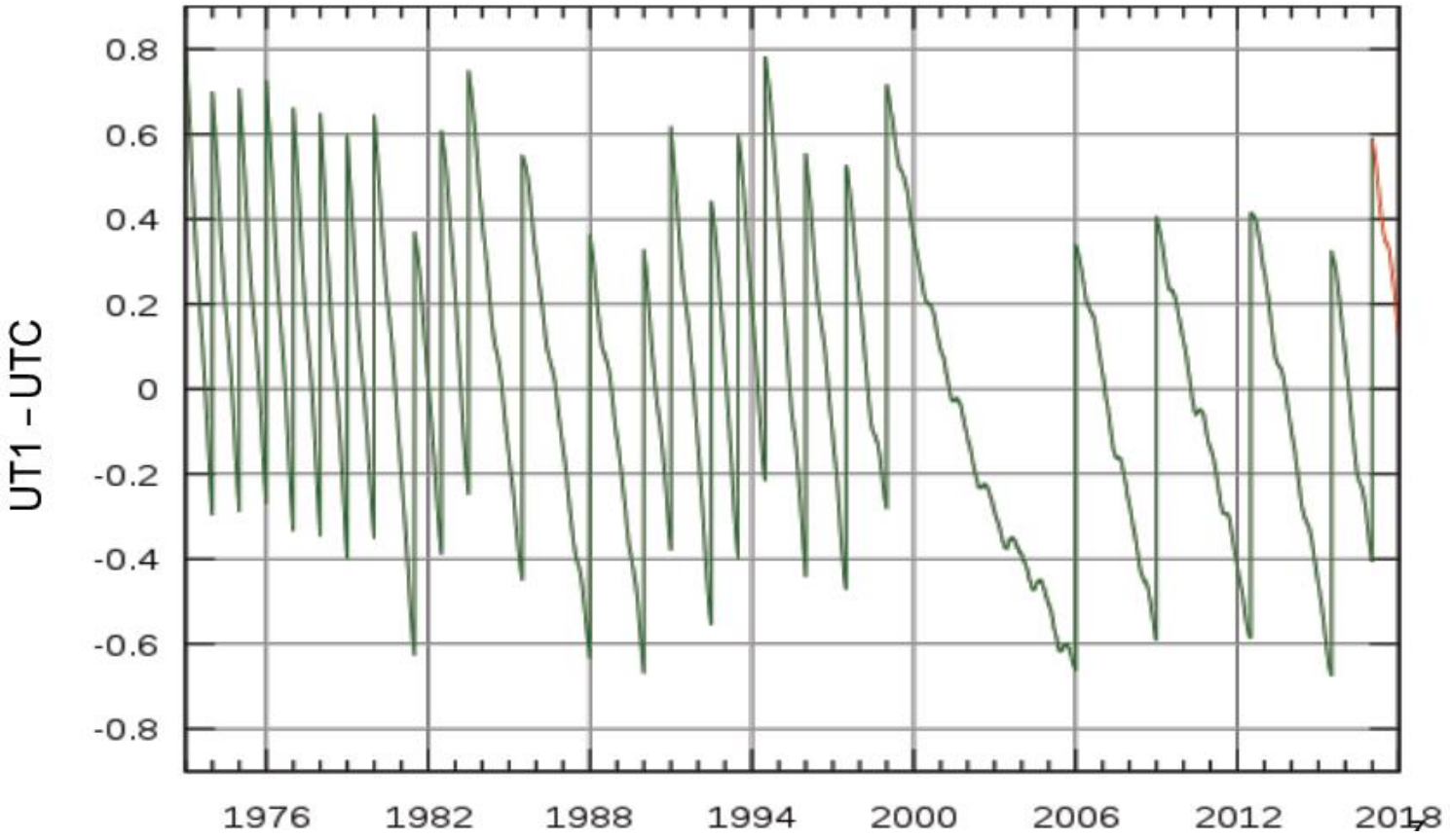


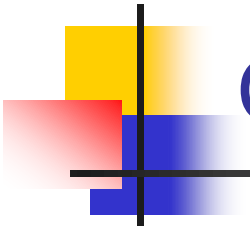

- Computer clocks are not generally in perfect agreement
  - **Skew**: the difference between the times on two clocks (at any instant)
- Computer clocks are subject to clock drift (they count time at different rates)
  - **Clock drift rate**: the difference per unit of time from some ideal reference clock
  - Ordinary quartz clocks drift by about 1 sec in 11-12 days. ($10^{-6}$ secs/sec).
  - High precision quartz clocks drift rate is about $10^{-7}$ or $10^{-8}$ secs/sec

# Time Standards

- **天文学时间（UT1）**
  - 太阳日：两次连续的太阳中天之间的时间间隔
  - 太阳秒：1/86400个太阳日，“Greenwich Mean Time”
- **国际原子时间(TAI)**
  - 基于铯原子跳跃周期， Started Jan 1, 1958
  - 秒：9 192 631 770次跳跃周期
  - Has diverged from UT1 due to slowing of earth’s rotation
- **通用协调时间(UTC)**
  - 基于原子时间，TAI + leap seconds to be within 0.9s of UT1
  - 采用闰秒，与天文时间保持一致
  - Currently 35，Most recent: June 30, 2012

# Comparing Time Standards

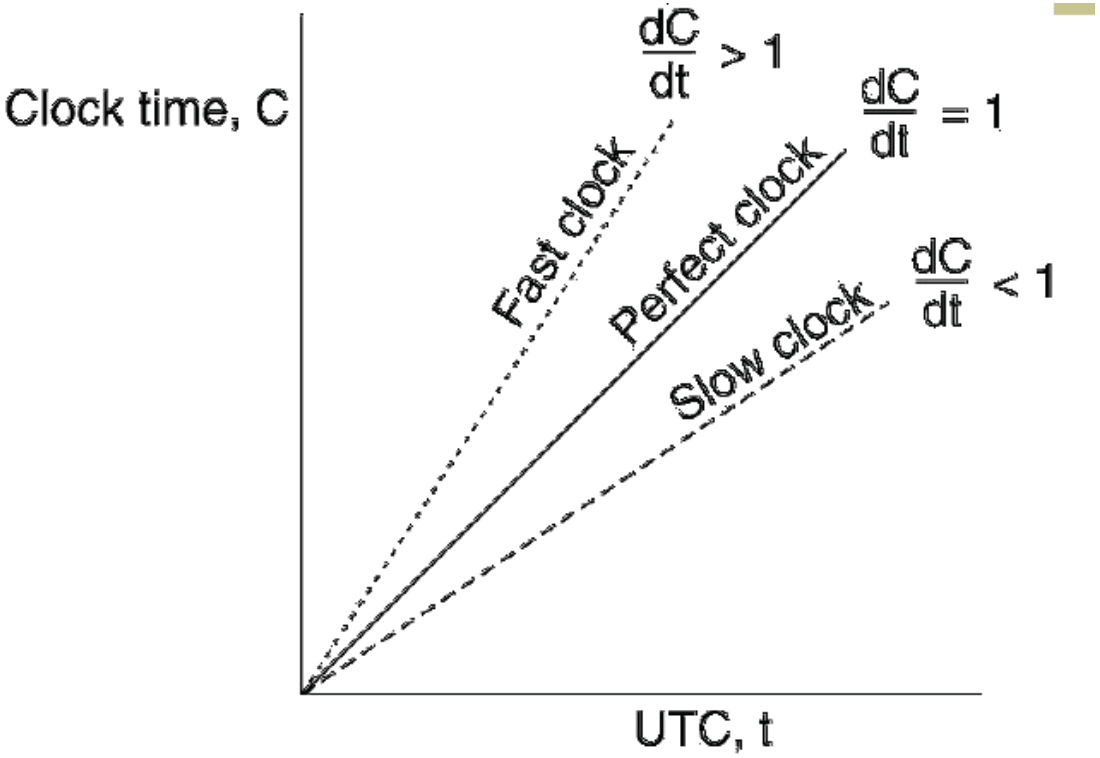




# Coordinated Universal Time (UTC)

- Is broadcast from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1-10 millisecond
- Signals from GPS are accurate to about 1 microsecond
  - Why can't we put GPS receivers on all our computers?

# Clock Synchronization Algorithms




- The relation between clock time and UTC when clocks tick at different rates.

# 第6章 时间和全局状态

- 简介
- 时钟、事件和进程状态
- 同步物理时钟
- 逻辑时间和逻辑时钟
- 全局状态
- 分布式调试
- 小结

# 同步物理时钟

- 外部同步
  - 采用权威的外部时间源
  - 时钟$C_i$在范围D内是准确的

$$|S(t)-C_i(t)|<D, i=1,2,...,N$$

- 内部同步
  - 无外部权威时间源，系统内时钟同步
  - 时钟$C_i$在范围D内是准确的

$$|C_i(t)-C_j(t)|<D, i, j=1,2,...,N$$

- 关系

若P在范围D内外部同步，则P在范围2D内内部同步

# 同步物理时钟

- 时钟正确性
  - 基于漂移率

$$(1-\rho)(t'-t) \le H(t') - H(t) \le (1+\rho)(t'-t)$$

  - 基于单调性

$$t' > t \Rightarrow \le C(t') > C(t)$$

- 基于混合条件

  单调性 + 漂移率有界 + 同步点跳跃前进

- 时钟故障
  - 崩溃故障：时钟完全停止滴答
  - 随机故障：其它故障，如“千年虫”问题

# 同步物理时钟

- 同步系统中的同步
  - 假设条件

    -已知时钟漂移率范围

    -存在最大的消息传输延迟

    -进程每一步的执行时间已知
  - 方法

    若一个进程将时间t发送至另一个进程，且消息传输时间的不确定性为u=max－min，则

    接收进程：t+min，则时钟偏移可能为u

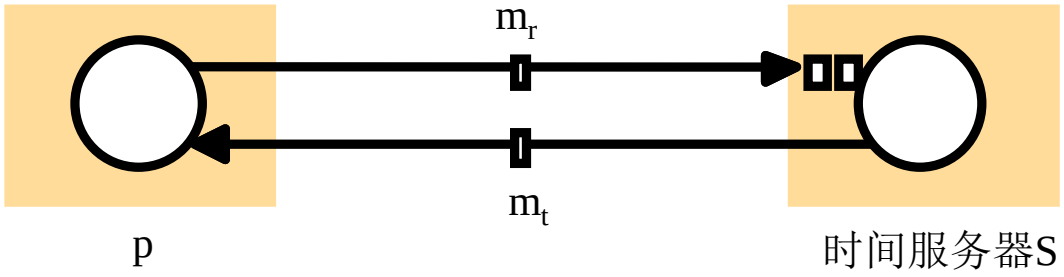
    t+max，则时钟偏移至多为u

    t+(max+min)/2，则时钟偏移至多为u/2

见论文《局部分布式虚拟现实系统物理时钟同步研究》

# 同步物理时钟

- **Cristian方法**
  - **应用条件**
    - 存在时间服务器，可与外部时间源同步
      A time server S receives signals from a UTC source
    - 消息往返时间与系统所要求的精度相比足够短
  - **协议**
    - 进程p根据消息$m_r$, $m_t$计算消息往返时间$T_{round}$
    - 根据服务器在$m_t$中放置的时间t设置时钟为：$t+T_{round}/2$

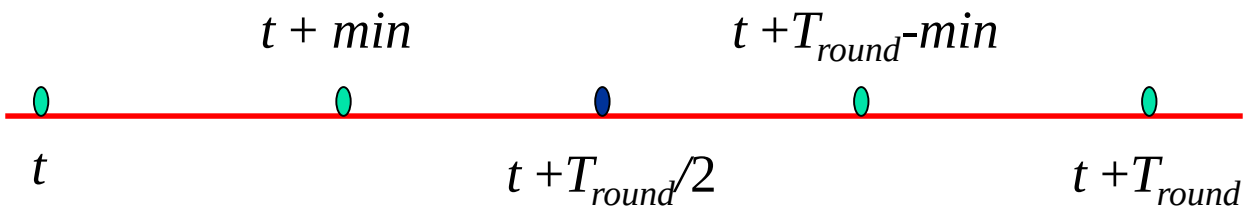


# 同步物理时钟--Cristian方法

- **精度分析**

  若消息的单向最小传输时间为*min*，则精度为：

  $\pm(T_{round}/2 - min)$



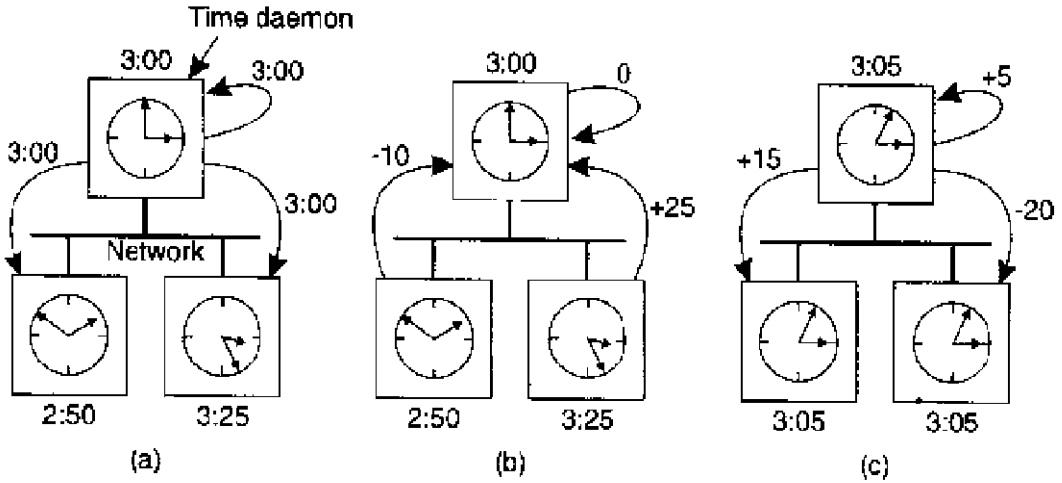

Cristian's algorithm -

- a single time server might fail, so they suggest the use of a group of synchronized servers
- it does not deal with faulty servers

# 同步物理时钟--Berkeley算法

- Berkeley算法
  - 主机*周期轮询*从属机时间（内部同步）
  - 主机通过消息往返时间估算从属机的时间

    与Cristian方法类似
  - 主机计算*容错平均值*
  - 主机发送每个从属机的*调整量*



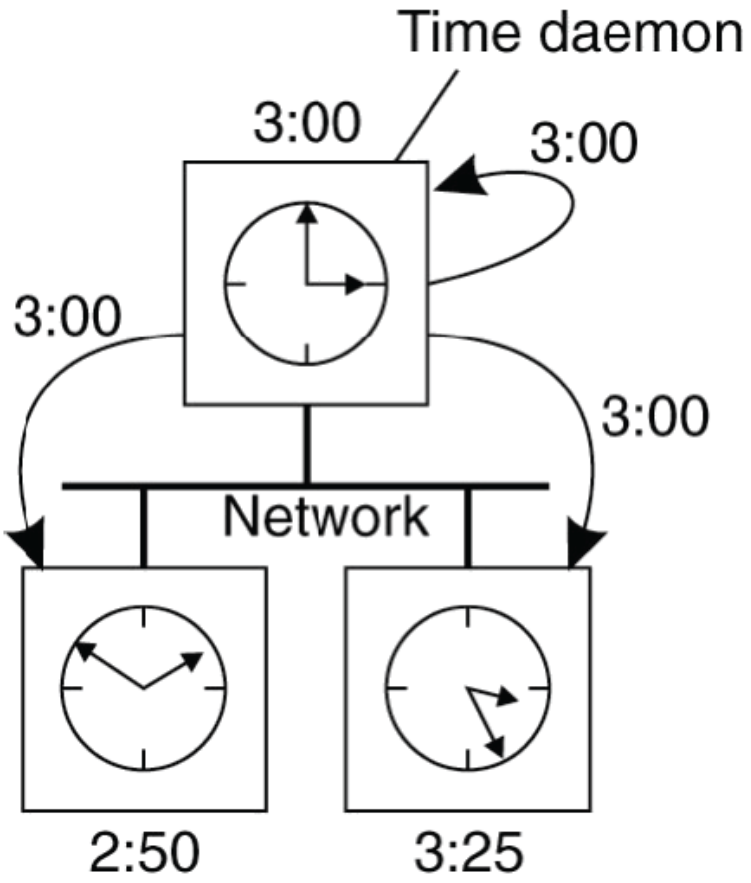
# 同步物理时钟--Berkeley算法

- An algorithm for internal synchronization of a group of computers
- A master polls to collect clock values from the others (slaves)主机*周期轮询*从属机时间
- 主机通过消息往返时间估算从属机的时间

  与Cristian方法类似
- 主机计算*容错平均值*
- 主机发送每个从属机的*调整量*

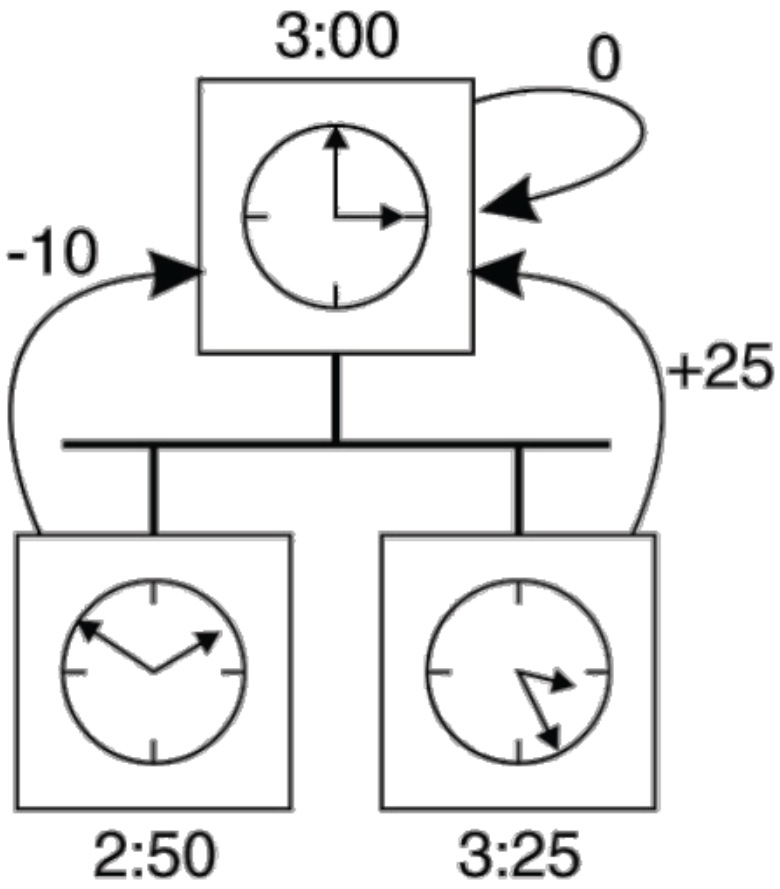
# The Berkeley Algorithm (1)

- The time daemon asks all the other machines for their clock values.



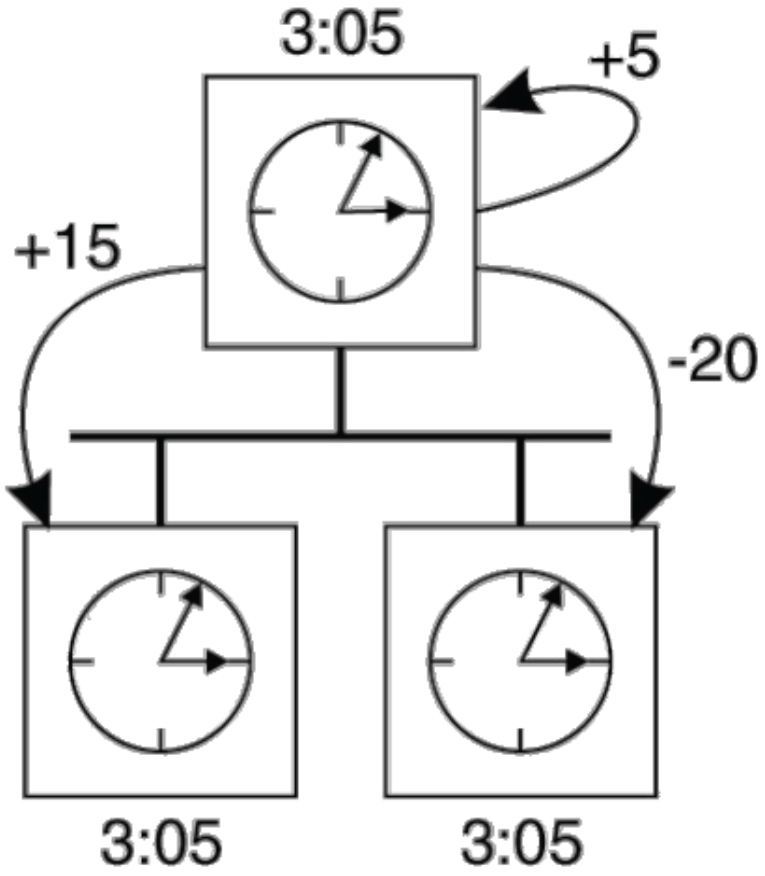
# The Berkeley Algorithm (2)

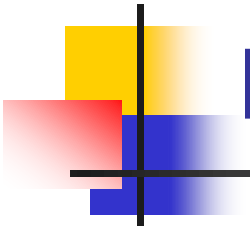
- The machines answer



# The Berkeley Algorithm (3)

- The time daemon tells everyone how to adjust their clock.





# Berkeley algorithm (4)

- It takes an average (eliminating any above average round trip time or with faulty clocks)
- It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
- Measurements
  - 15 computers, clock synchronization 20-25 millisecs drift rate < $2x10^{-5}$
  - If master fails, can elect a new master to take over (not in bounded time)

# 同步物理时钟-- NTP

## 网络时间协议(NTP)

- 设计目标

-可外部同步

使得跨Internet的用户能精确地与UTC(通用协调时间)同步

-高可靠性

可处理连接丢失，采用冗余服务器、路径等

-扩展性好

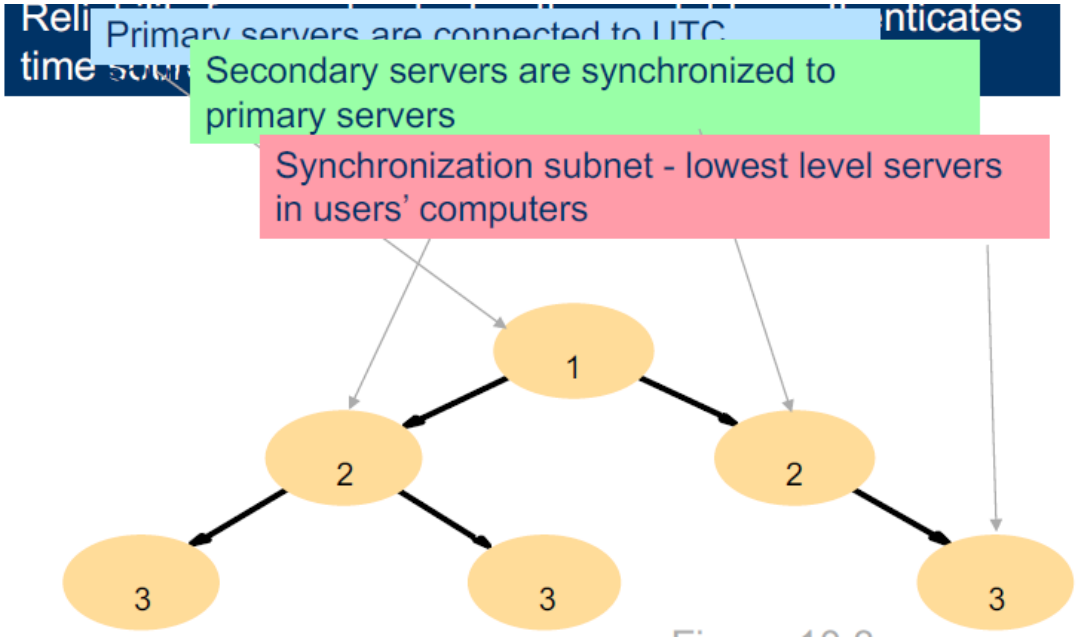
大量用户可经常同步，以抵消漂移率的影响

-安全性强

防止恶意或偶然的干扰

# 同步物理时钟

- 协议结构
  - -层次结构
  - -主服务器直接与外部UTC同步
  - -同步子网可重新配置

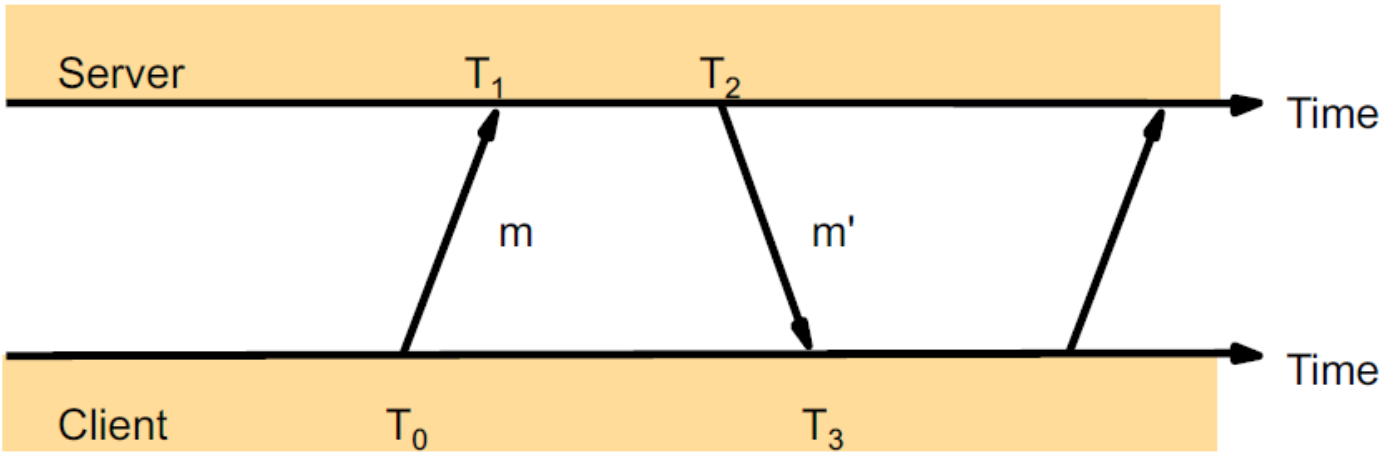


# The Network Time Protocol (NTP)

- **Uses a hierarchy of time servers**
  - **Class 1 servers have highly-accurate clocks**
    - **Connected directly to atomic clocks, etc.**
  - **Class 2 servers get time from only Class 1 and Class 2 servers**
  - **Class 3 servers get time from any server**
- **Synchronization similar to Cristian's alg.**
  - **Modified to use multiple one-way messages instead of immediate round-trip**
- **Accuracy: Local ~1ms, Global ~10ms**

# NTP Protocol

- All modes use UDP
- Each message bears timestamps of recent events:
  - Local times of Send and Receive of previous message
  - Local times of Send of current message
- Recipient notes the time of receipt $T_3$ (we have $T_0$, $T_1$, $T_2$, $T_3$)



# Accuracy of NTP

Timestamps

- $t_0$ is the client's timestamp of the request packet transmission,
- $t_1$ is the server's timestamp of the request packet reception,
- $t_2$ is the server's timestamp of the response packet transmission and
- $t_3$ is the client's timestamp of the response packet reception.

$$\begin{aligned} \text{RTT} &= \text{wait\_time\_client} - \text{server\_proc\_time} \\ &= (t_3 - t_0) - (t_2 - t_1) \\ \text{Offset} &= ((t_1 - t_0) + (t_2 - t_3))/2 \\ &= ((\text{offset} + \text{delay}) + (\text{offset} - \text{delay}))/2 \end{aligned}$$

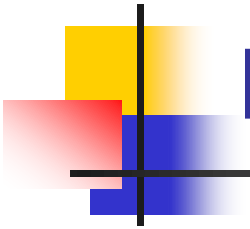
- **NTP servers filter pairs $<rtt_i, offset_i>$, estimating reliability from variation, allowing them to select peers**
- **Accuracy of 10s of millisecs over Internet paths (1 on LANs)**

# 同步物理时钟

- NTP采用过滤离中趋势算法，保留8个最近的<$rtt_i$, $offset_i$>，用以估算偏移offset
- NTP采用对等方选择算法，可改变用于同步的对等方

  -优先选择层次较低的对等方

  -优先选择过滤离中趋势数值较低的对等方



# How To Change Time

- Can’t just change time
  - Why not?
- Change the update rate for the clock
  - Changes time in a more gradual fashion
  - Prevents inconsistent local timestamps

# 第6章 时间和全局状态

- 简介
- 时钟、事件和进程状态
- 物理时钟同步
- 逻辑时间和逻辑时钟
- 全局状态
- 分布式调试
- 小结

# 逻辑时间和逻辑时钟

- 逻辑时间的引入
  - 分布式系统中的物理时钟无法完美同步
    - -消息传输延时的不确定性
  - 事件排序是众多分布式算法的基石
    - -互斥算法
    - -死锁检测算法
  - 缺乏全局时钟
    - -后发生的事件有可能赋予较早的时间标记

# 逻辑时间和逻辑时钟

- 逻辑时钟
  - 众多应用只要求所有节点具有相同时间，该时间*不一定与实际时间相同*
  - Lamport(1978)指出：不进行交互的两个进程之间不需要时钟同步

    对于不需要交互的两个进程而言，即使没有时钟同步，也无法察觉，更不会产生问题。
  - 所有的进程需要在时间的*发生顺序*上*达成一致*

    如make程序

# 逻辑时间和逻辑时钟

- 事件排序
  - “系统i中的事件a发生在系统j中的事件b之前”是不准确的
    - 事件发生和观察之间存在时延
    - 不同系统中的时钟存在偏差
  - 时间戳(Lamport 1978)
    - 用于分布式系统中的事件排序
    - 与物理时钟无关
    - 实用高效，应用广泛

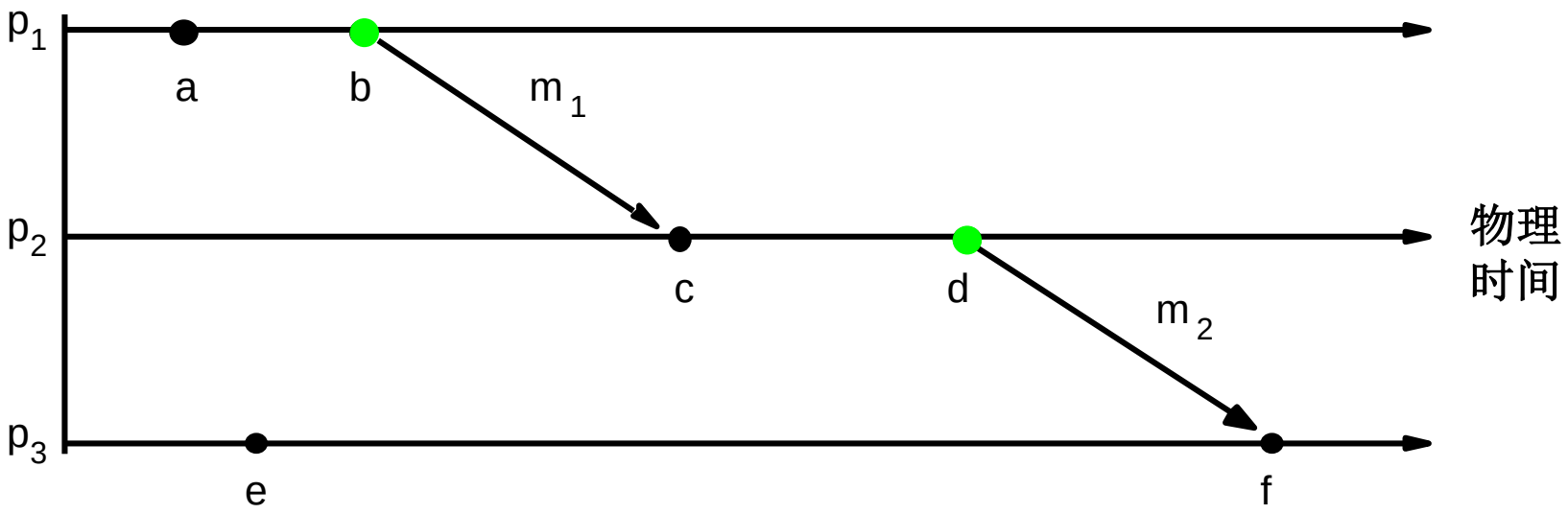
# 逻辑时间和逻辑时钟

- 两个基本事实
  - -同一进程中的两个事件存在关系“$\rightarrow_i$”
  - -任一消息的发送事件发生在该消息的接收事件之前
- “发生在先(happens-before)”关系定义
  - -若存在进程$p_i$满足$e \rightarrow e_i'$，则$e \rightarrow e'$
  - -对于任一消息m，存在send(m) →recv(m)
  - -若事件满足$e \rightarrow e'$和$e' \rightarrow e''$，则$e \rightarrow e''$
- 并发关系定义

  $X \rightarrow Y$ 与 $Y \rightarrow X$均不成立，则称事件X、Y是并发的，表示为X ||Y

# 逻辑时间和逻辑时钟

- 事件排序示例



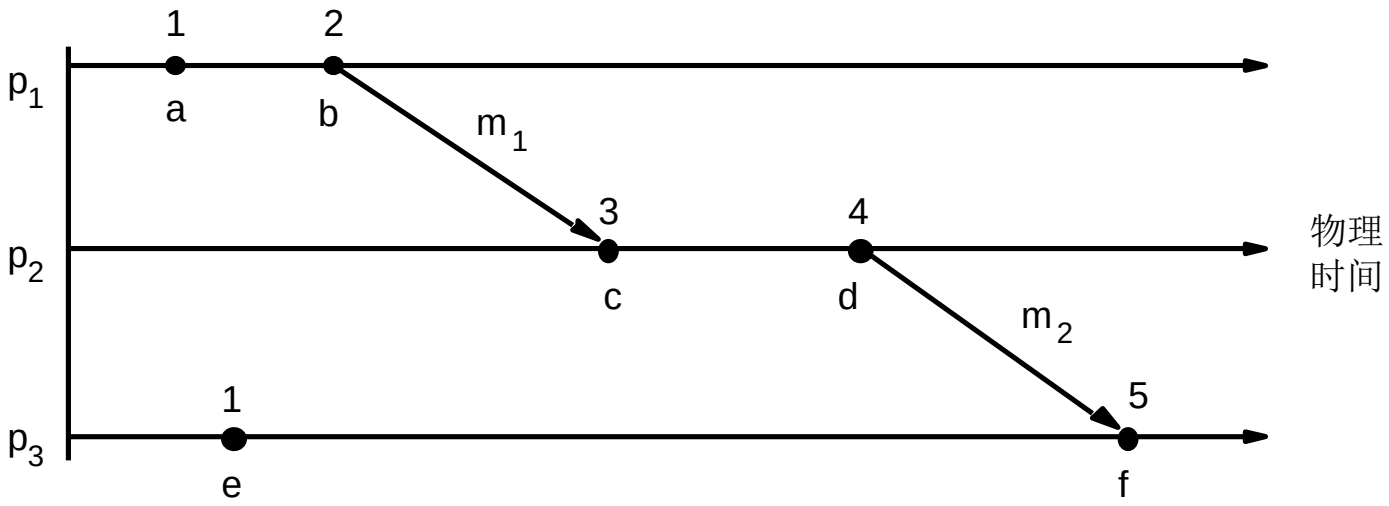

- b→c，c→d和d →f成立
- b→f与e→f均成立
- 事件b和e无法比较，即b||e

# 逻辑时间和逻辑时钟

- **Lamport时钟**
  - **机制**
    - 进程维护一个单调递增的软件计数器，充当逻辑时钟
    - 用逻辑时钟为事件添加时间戳
    - 按事件的时间戳大小为事件排序
  - **逻辑时钟修改规则**
    - 进程$p_i$发出事件前，逻辑时钟$L_i:=L_i+1$
    - 进程$p_i$发送消息m时，在m中添加时间戳$t=L_i$
    - 进程$p_j$在接收(m,t)时，更新$L_j:=max(L_j,t)+1$，给s事件recv(m)添加时间戳后发送给应用程序

# 逻辑时间和逻辑时钟

- **Lamport时钟示例(一)**

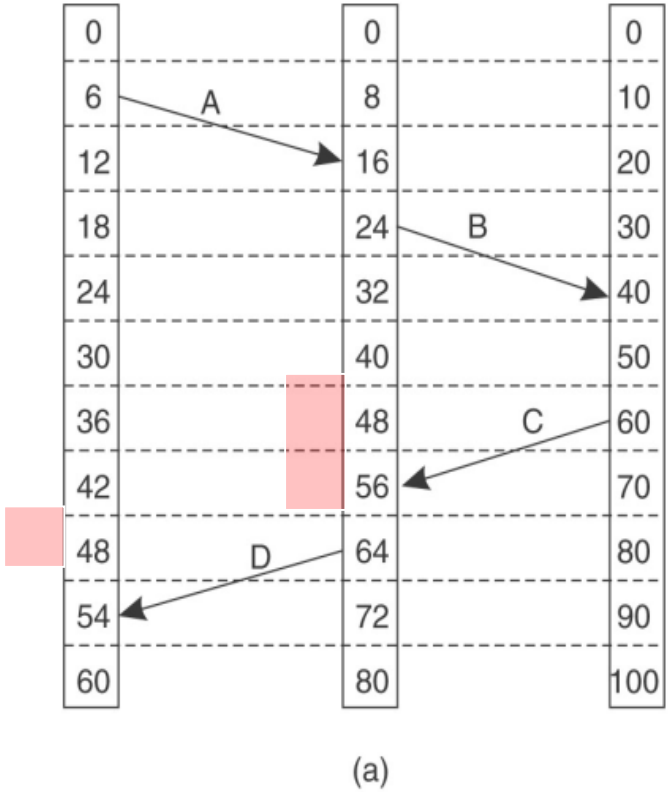



$a \rightarrow b \Rightarrow L(a) < L(b)$

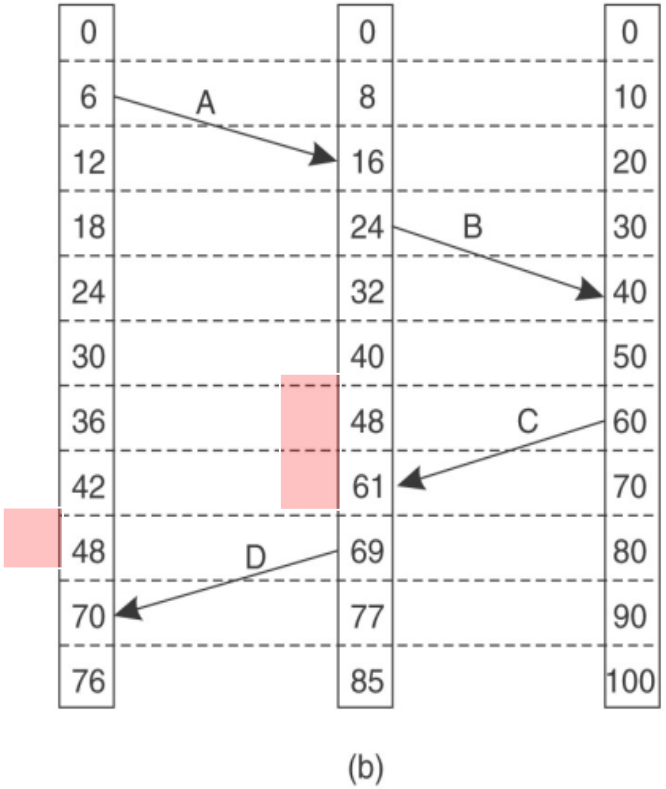
$L(a) < L(b) \not\Rightarrow a \rightarrow b$

# 逻辑时间和逻辑时钟

- **Lamport时钟示例(二)**




**(a)** 三个不同速率的时钟

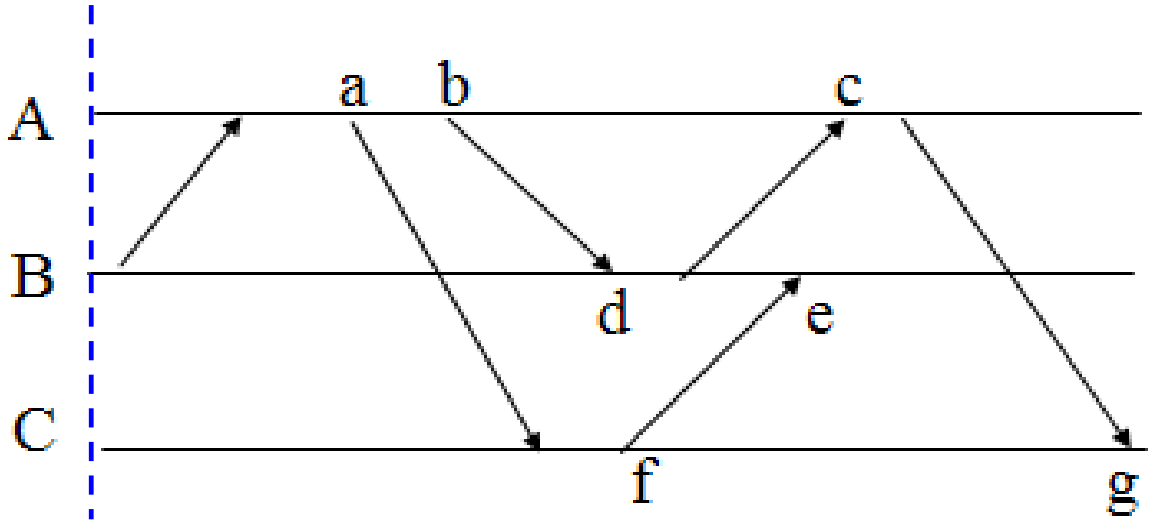



**(b) Lamport**算法校正时钟

# 逻辑时间和逻辑时钟

- **Lamport时钟练习**

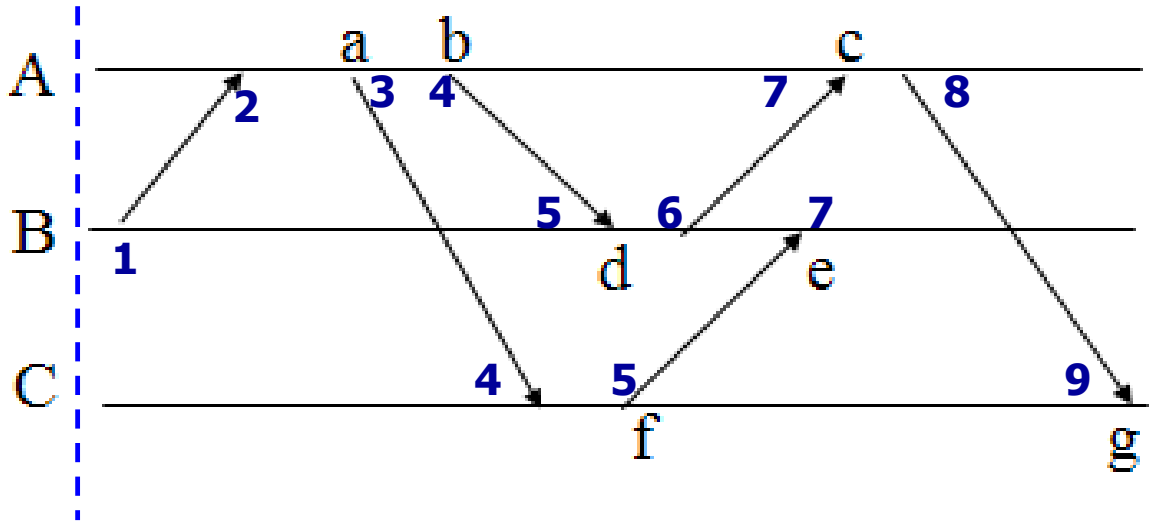
假设系统中只存在消息发送和接收事件，如下图所示，请给出事件a-g的逻辑时钟。




逻辑时钟 0

# 逻辑时间和逻辑时钟

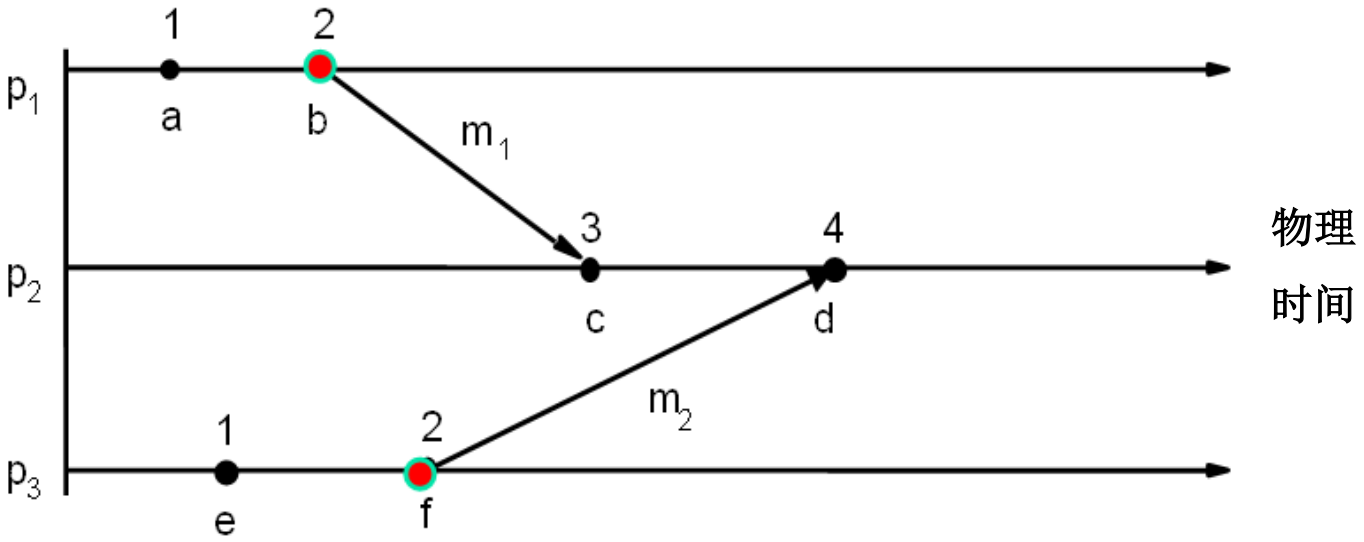
- Lamport时钟练习答案




逻辑时钟：0

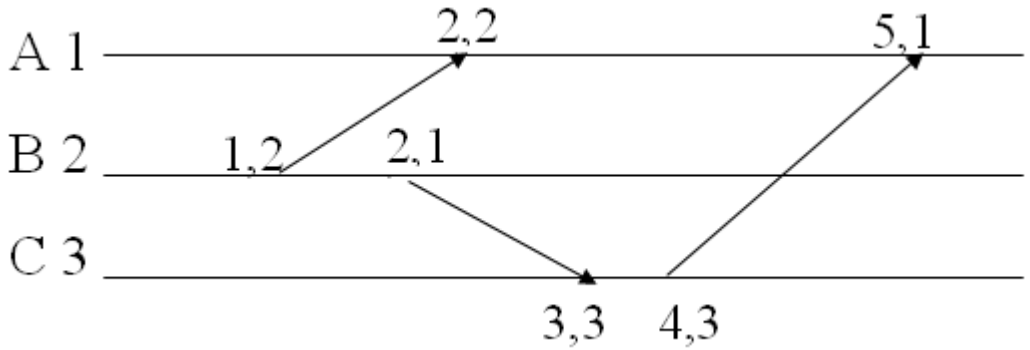
# 逻辑时间和逻辑时钟

- *不同*进程产生的消息可能具有*相同数值*的Lamport时间戳



# 逻辑时间和逻辑时钟

- 基于Lamport时间戳的事件排序---总结
  - 算法不依赖于事件发生的真实时间
  - 与真实物理时间中事件的发生顺序可能不一致

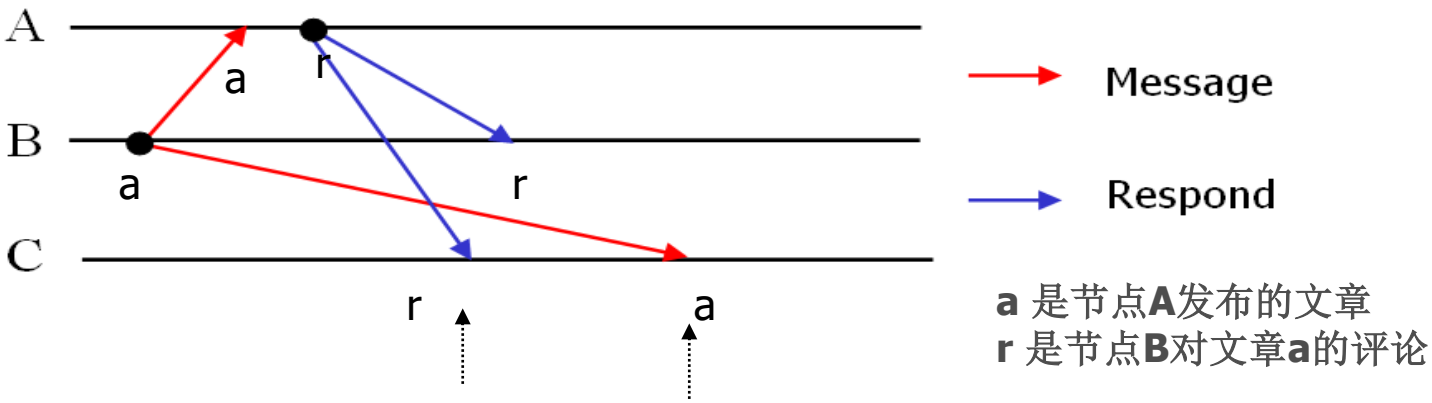



基于Lamport时间戳的排序中，在时刻(2,1)发生的事件发生比在时刻（2，2）发生的事件要早，然而在真实物理时间中可能恰好相反。(有错吗？)

# 逻辑时间和逻辑时钟

- **Lamport时钟不具备性质：若L(A) < L(B)则A→B**
- **没有捕获事件的因果关系**

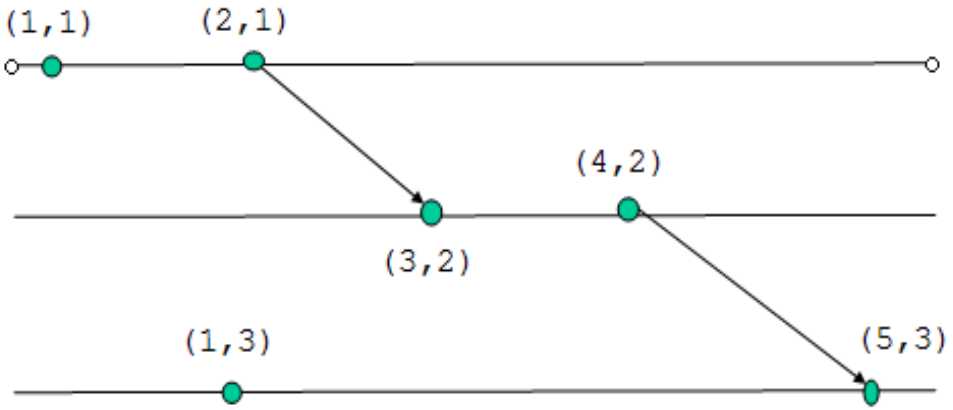
节点B发布一篇文章并传送给节点A和C。节点A就此发表评论并传送给节点B和C。




**我们无法准确确定r的先后关系：** $C(a) < C(r) \not\Rightarrow a \rightarrow r$

# 全序逻辑时钟

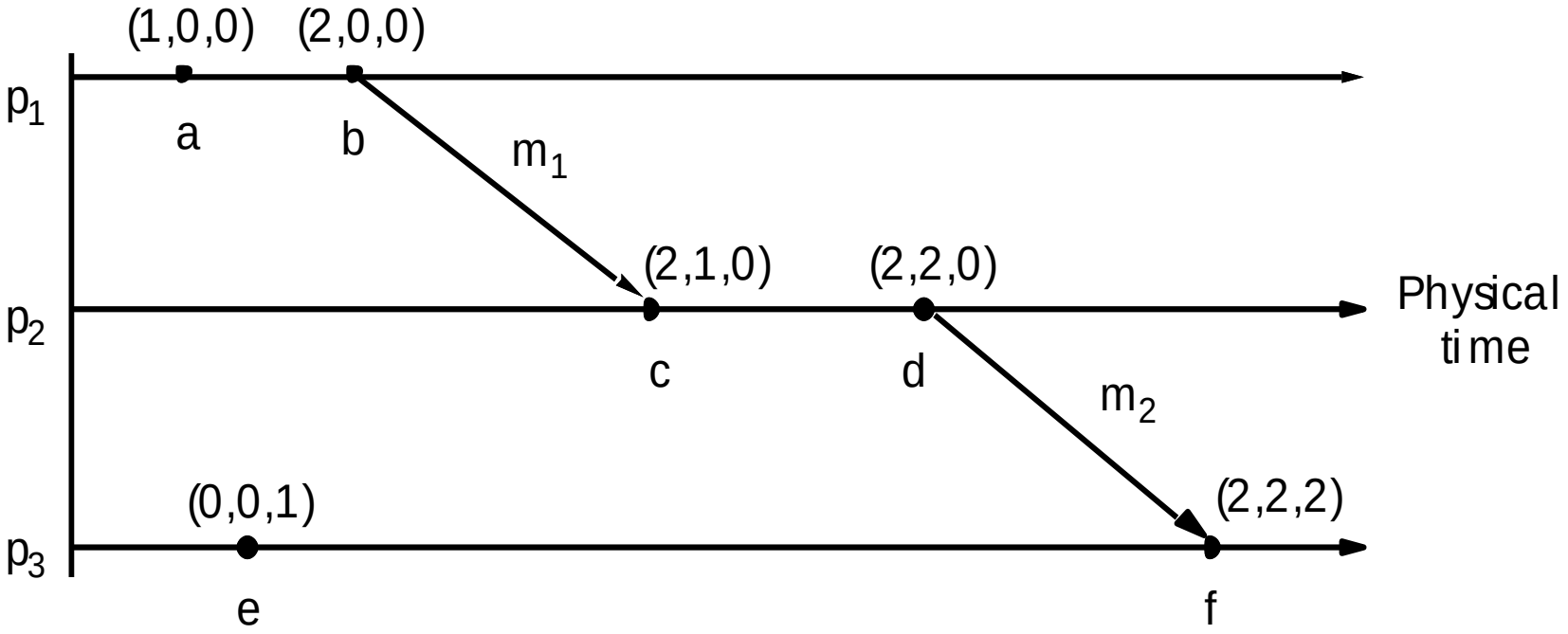
- 引入进程标示符创建事件的全序关系
- 若e、e'分别为进程$p_i$、$p_j$中发生的事件，则其全局逻辑时间戳分别为$(T_i,i)$、$(T_j,j)$。
- $e \rightarrow e' \Leftrightarrow T_i<T_j \parallel T_i=T_j \;\&\&\; i<j$
- 系统中各个事件Lamport时间戳均不相同



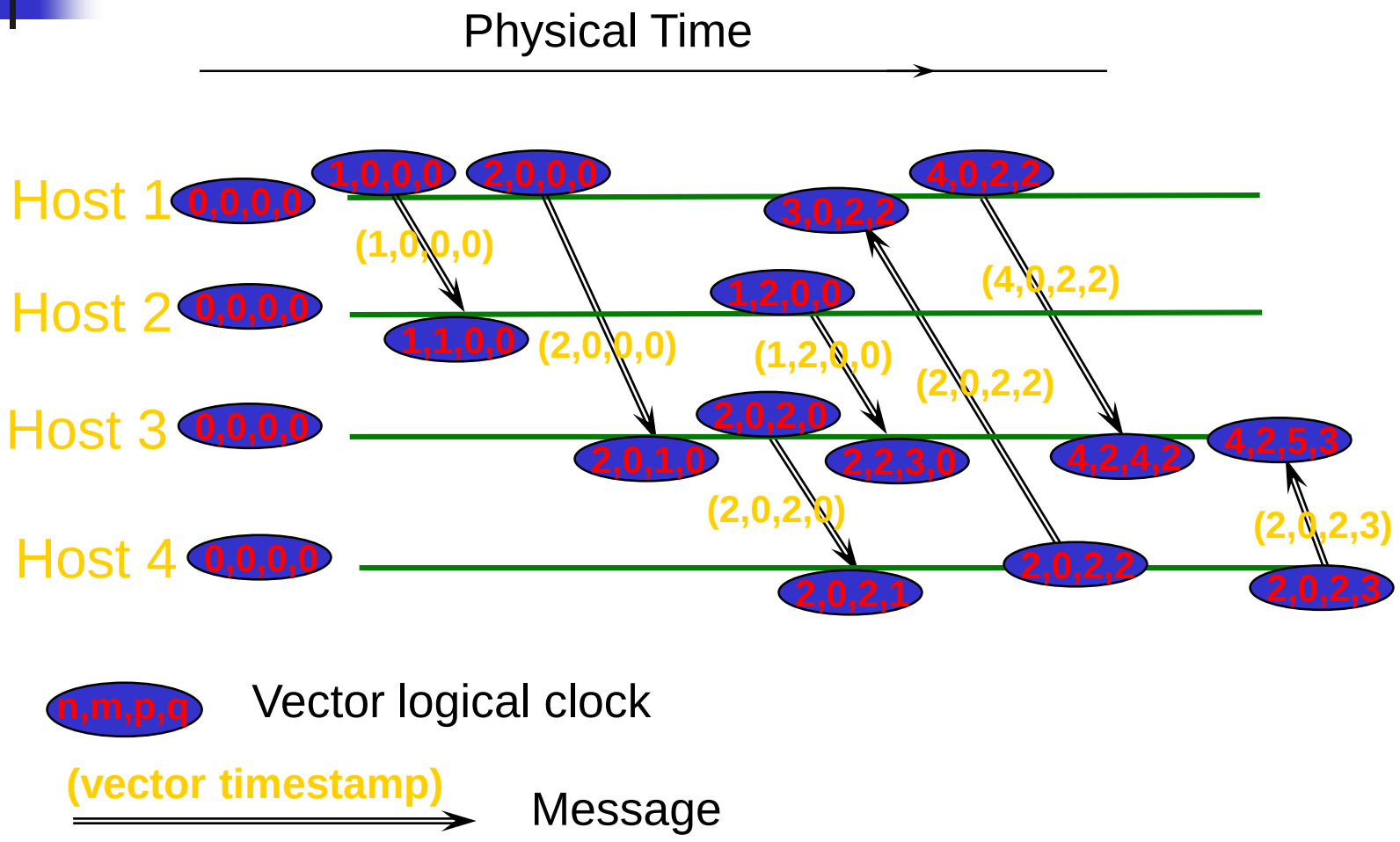
# 向量时钟

- 克服Lamport时钟的缺点：若L(e) < L(e')不能推出则e→e'。
- 每个进程维护它自己的向量时钟Vi
- VC1:初始情况下，$V_i[j]=0,i,j=1,2,...N.$
- VC2:在$p_i$给事件加时间戳之前，设置$V_i[i]= V_i[i]+1$。
- VC3:$p_i$在它发送的每个消息中包括$t = V_i$。
- VC4：当$p_i$接收到消息中的时间戳t时，设置$V_i[j]=max(V_i[j],t[j]),j=1,2,...,N$。取两个向量时间戳的最大值称为合并操作。

# 向量时钟



# 向量时钟


Physical Time
Host 1
Host 2
Host 3
Host 4
0,0,0,0
1,0,0,0
2,0,0,0
4,0,2,2
3,0,2,2
(1,0,0,0)
(4,0,2,2)
1,2,0,0
1,1,0,0
(2,0,0,0)
(1,2,0,0)
(2,0,2,2)
2,0,2,0
2,0,1,0
2,2,3,0
4,2,4,2
4,2,5,3
(2,0,2,0)
(2,0,2,3)
2,0,2,2
2,0,2,1
2,0,2,3
n,m,p,q
Vector logical clock
(vector timestamp)
Message

# 向量时钟

- V1 = V2,

  *iff* V1[i] = V2[i], i = 1, … , n
- V1 ⩽ V2,

  *iff* V1[i] ⩽ V2[i], i = 1, … , n
- V1 < V2,

  *iff* V1 ⩽ V2 &

  ∃ j (1 ⩽ j ⩽ n & V1[j] < V2[j])
- V1 is concurrent with V2

  *iff* not (V1 ⩽ V2 OR V2 ⩽ V1)

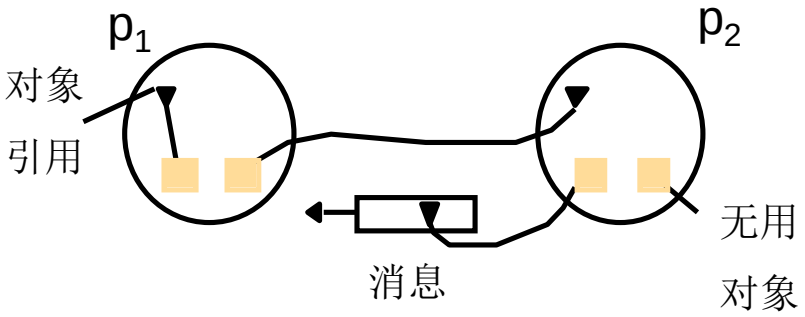
# 第6章 时间和全局状态

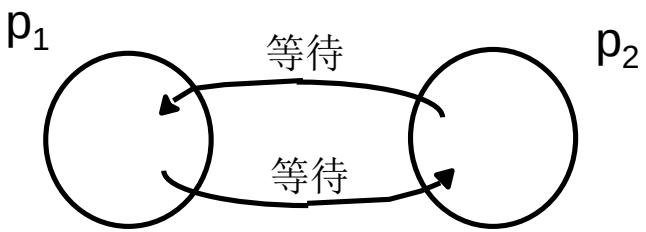
- 简介
- 时钟、事件和进程状态
- 同步物理时钟
- 逻辑时间和逻辑时钟
- 全局状态
- 分布式调试
- 小结

# 全局状态

- 观察全局状态的必要性
  - 分布式无用单元的收集
    - -基于对象的引用计数
    - -必须考虑信道和进程的状态
  - 分布式死锁检测

观察系统中的“等待”关系图中是否存在循环

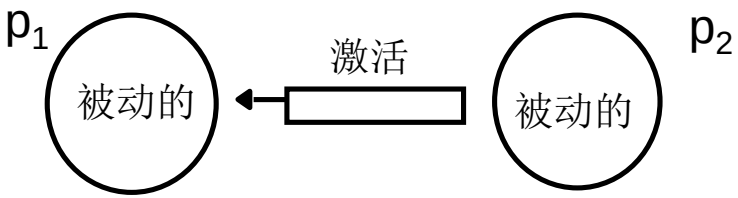


# 全局状态

- 分布式终止检测

  与进程的状态有关——“主动”或“被动”




- 分布式调试

  需要收集同一时刻系统中分布式变量的数值

# 全局状态

- 全局状态和一致割集
  - 观察进程集的状态——全局状态非常困难

    *根源：缺乏全局时间*
  - 进程的历史

    $h_i = \langle e_i^0, e_i^1, e_i^2 \ldots \rangle$
  - 进程历史的**有限前缀**

    $h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$
  - 全局历史——**单个进程历史的并集**

    $H = h_1 \cup h_2 \cup \ldots \cup h_N$

# 全局状态

- 进程状态

  $s_i^k$：进程$p_i$在第k个事件发生之前的状态

- 全局状态——单个进程状态的集合

  $S = (s_1, s_2, \ldots s_N)$

- 割集——系统全局历史的子集

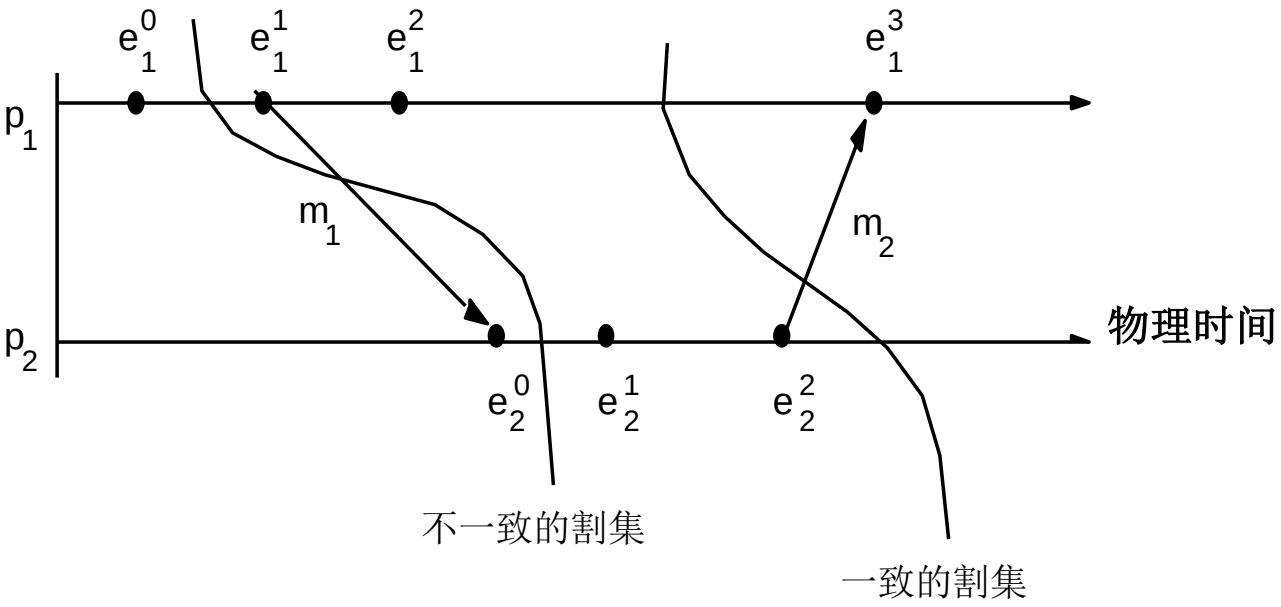
  $C = \langle h_1^{c1}, h_2^{c2} \ldots h_3^{c3} \rangle$

- 割集的一致性

割集C是一致的：对于所有事件$e \in C$，$f \rightarrow e \quad \Rightarrow f \in C$

# 全局状态

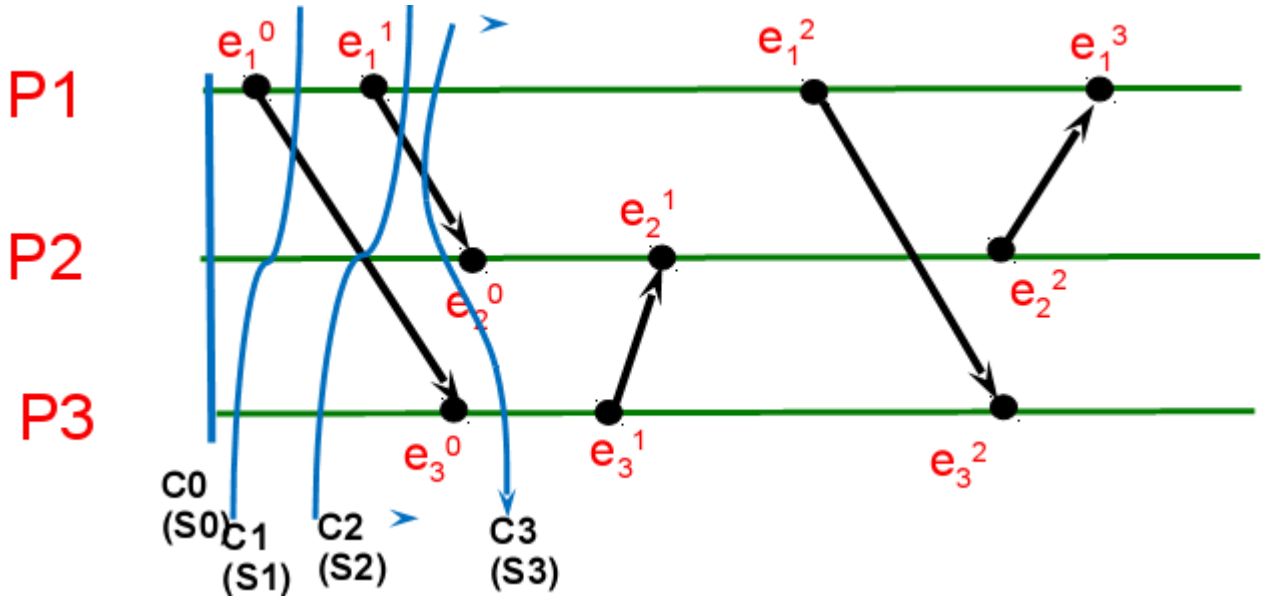
- 割集示例



# 全局状态

- 一致的全局状态——对应于一致割集的状态

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$



# 全局状态

- 走向(run)
  - 全局历史中所有事件的全序
  - 与每个本地历史顺序一致
  - *不是所有的走向都经历一致的全局状态*

– E.g., $<e_1^0, e_1^1, e_1^2, e_1^3, e_2^0, e_2^1, e_2^2, e_3^0 e_3^1, e_3^2>$

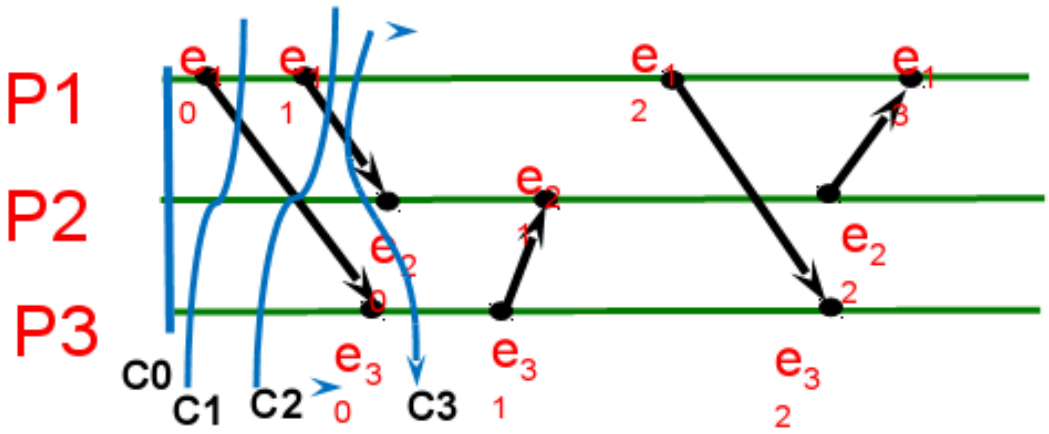
# 全局状态

- 线性化走向

  -所有的线性化走向只经历一致的全局状态

  -若存在一个经过S和S'的线性化走向，则状态S'是从S可达

  – E.g., $< e_1^0, e_1^1, e_3^0, e_2^0, \ldots >$, $< e_1^0, e_3^0, e_1^1, e_2^0, \ldots >$



# 全局状态

- Chandy和Lamport的“快照”算法
  - 目的

    捕获一致的全局状态
  - 假设
    - 进程和通道均不会出现故障
    - 单向通道，提供FIFO顺序的消息传递
    - 进程之间存在全连通关系
    - 任一进程可在任一时间开始全局拍照
    - 拍照时，进程可继续执行，并发送和接收消息

# 全局状态

- 算法基本思想
  - *接入通道+外出通道*
  - 进程状态+通道状态
  - 标记消息

    *标记接收规则*：强制进程在记录下自己的状态之后但在它们发送其他消息前发送一个标记。

    *标记发送规则*：强制没有记录状态的进程去记录状态

# 全局状态

- 算法伪码(一)

进程$p_i$的标记接收规则

$p_i$接收通道c上的标记消息：

if ($p_i$还没有记录它的状态)

$p_i$记录它的进程状态；

将c的状态记成空集；

开始记录从其他接入通道上到达的消息

else

$p_i$把c的状态记录到从保留它的状态以来它在c上接收到的消息集合中

end if

# 全局状态

- 算法伪码(二)

进程$p_i$的标记发送规则

在$p_i$记录了它的状态之后，对每个外出通道c:

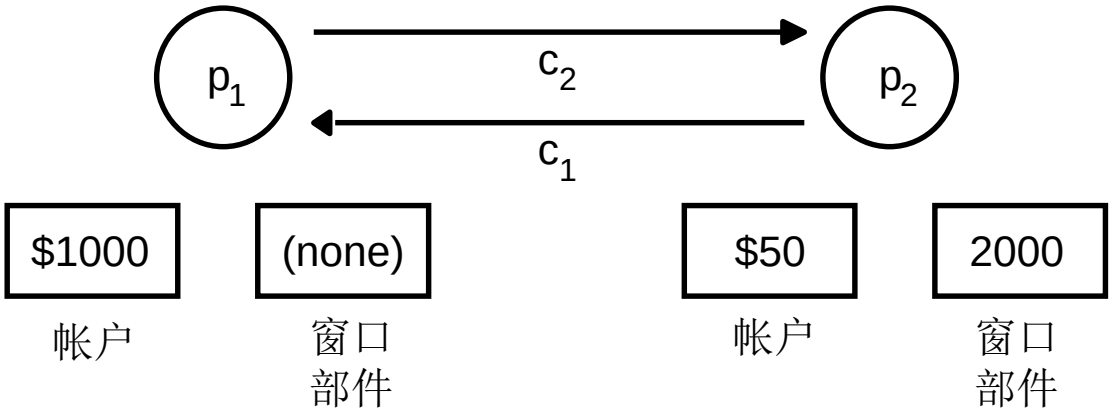
(在$p_i$从c上发送任何其他消息前)

$p_i$在c上发送一个消息标记

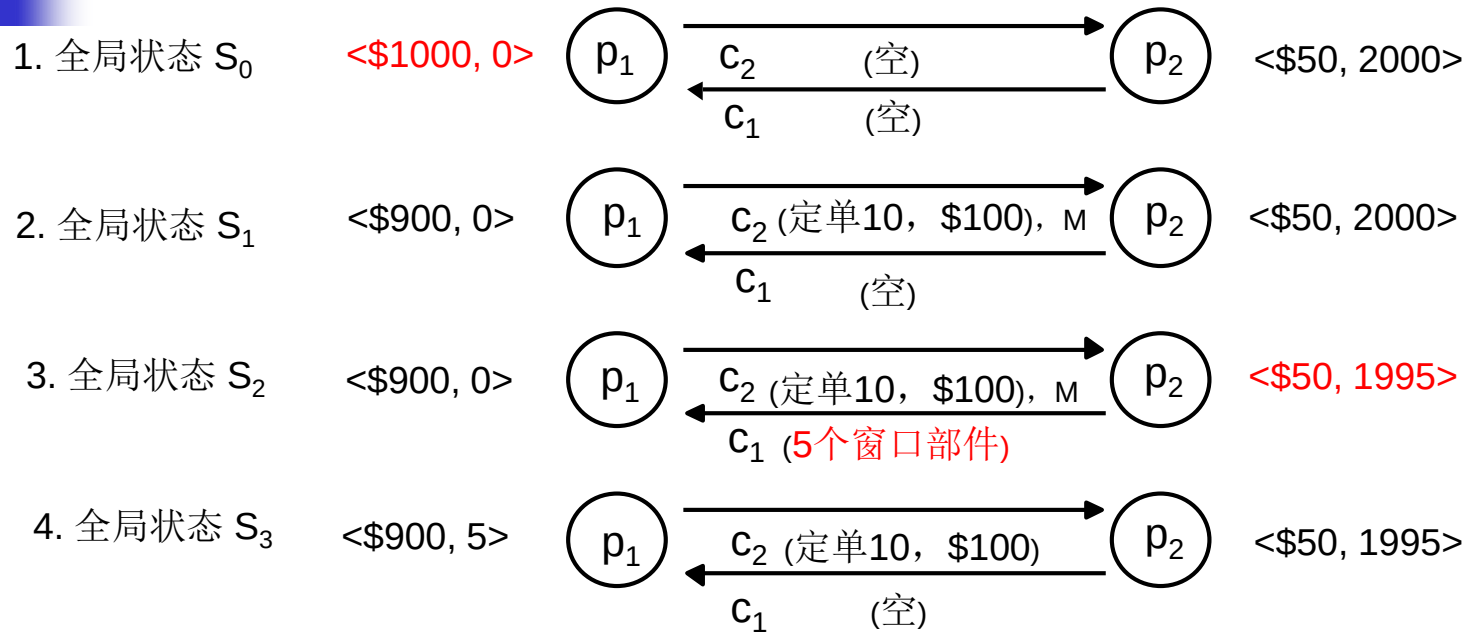
# 全局状态

- 算法示例
  - 两个进程$p_1$、$p_2$进行交易，每件10$
  - 初始状态

    进程$p_2$已经收到5件商品的定单，它将马上分发给$p_1$



# 全局状态




红色的为p1, p2, c1, c2<>的一个快照。

本质思路：发出消息的进程保存的是消息发出之前的状态，其他进程也应该是这个状态，或者是还没有处理的消息加上当时的状态

# 全局状态

- 算法终止

  - 假设：一个进程已经收到了一个标记信息，在有限的时间内记录了它的状态，并在有限的时间里通过每个外出通道发送了标记信息，

  - 若存在一条从进程$p_i$到进程$p_j$的信道和进程的路径，那么$p_i$记录它的状态之后的有限时间内$p_j$将记录它的状态，

  - 进程和通道图是强连接的，因此在一些进程记录它的状态之后的有限时间内，所有进程将记录它们的状态和进入通道的状态。

# 全局状态

- 算法一致性

$e_i$、$e_j$分别为进程$p_i$、$p_j$中的事件，且$e_i \rightarrow e_j$则我们有：若$e_j \in C \Rightarrow e_i \in C$，其中C为一个割集。即如果$e_j$在$p_j$记录它的状态之前发生，那么$e_i$必须在$p_i$记录它的状态之前发生. 证明思路如下：

- i=j时，显然成立
- i≠j时，反证法

# 全局状态

- 全局状态谓词、稳定性、安全性和活性
  - 全局状态谓词

    从系统P的进程全局状态集映射到{True,False}的函数
  - 稳定的谓词

    一旦系统进入谓词为True的状态，它将在所有可从该状态可达的状态中一直保持True。如系统死锁、系统终止等状态相关的谓词。
  - 安全性

    一个断言，即对所有可从$S_0$到达的所有状态。如a是可以成为死锁的性质，则对于所有可从$S_0$到达的所有状态S，a的值为False。
  - 活性

    对于任一从状态$S_0$开始的线性化走向L，对可从$S_0$到达的状态$S_L$，谓词为True。

# 第6章 时间和全局状态

- 简介
- 时钟、事件和进程状态
- 物理时钟同步
- 逻辑时间和逻辑时钟
- 全局状态
- 分布式调试
- 小结

# 分布式调试

- 目的

  对系统实际执行中的暂态作出判断

- 例子

  - 安全条件检查

    $x_i$为进程$p_i$的变量$(i=1,2,\ldots)$，安全条件为$|x_i-x_j|\leq \delta$

  - 控制系统

    所有阀门在某些时间是否全部处于开启状态

# 分布式调试

- 方法
  - 监控器进程

    收集进程状态信息
  - 全局状态谓词ϕ的判断

    -可能的ϕ：存在一个一致的全局状态S，H的一个线性化走向经历了这个全局状态S，而且该S使得ϕ(s)为True。

    -明确的ϕ：对于H的所有线性化走向L，存在L经历的一个一致的全局状态S，而且该S使得ϕ(s) 为True。

# 分布式调试

- 观察一致的全局状态
  - 进程的状态信息附有向量时钟值
  - 全局状态的一致性判断——CGS条件

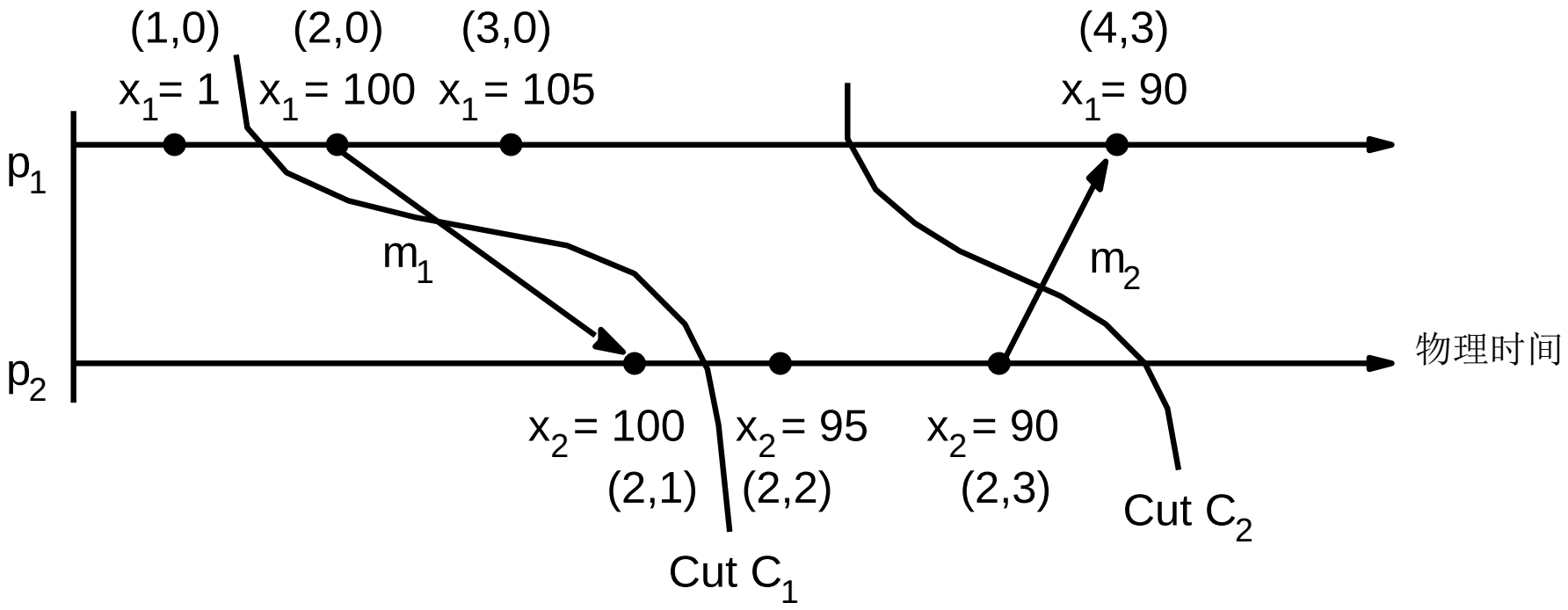
    设$S=(s_1,s_2,\ldots,s_N)$是从监控器进程接收到的状态信息中得出的全局状态，$V(s_i)$是从pi接收到的状态si的向量时间戳，则S是一致的全局状态当且仅当：

    $$V(s_i)[i] >= V(s_j)[i] \quad (i,j = 1,2,\ldots, N)$$

    即若一个进程的状态依赖于另一个进程的状态，则全局状态也包含了它所依赖的状态。

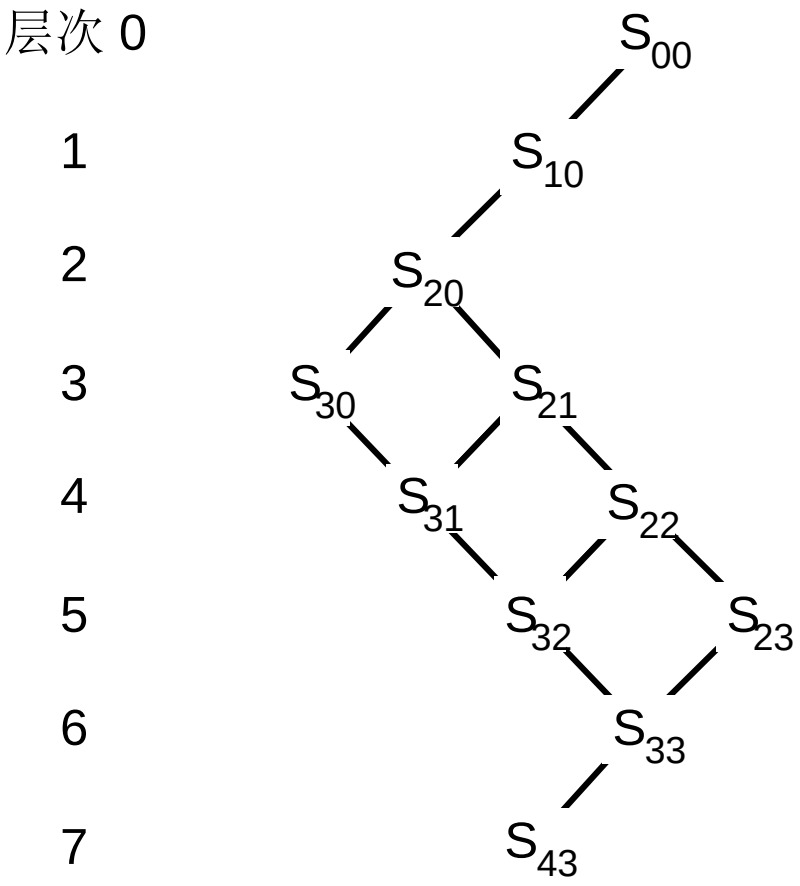
# 分布式调试

- 全局状态示例



# 分布式调试

- 一致全局状态网格




$S_{ij}$=在进程 1 发生事件i以及在进程 2 发生事件j之后的全局状态

# 分布式调试

- 判定可能的ϕ

从初始状态开始，遍历可达状态的网格。

L:=0;

States:={($s^0_1$, $s^0_2$, …, $s^0_N$)};

***while*** (对所有可能的S∈States，ϕ(s)=False)

L:=L+1;

Reachable:={S': H中从一些S∈States可到达的状态∧level(S')=L};

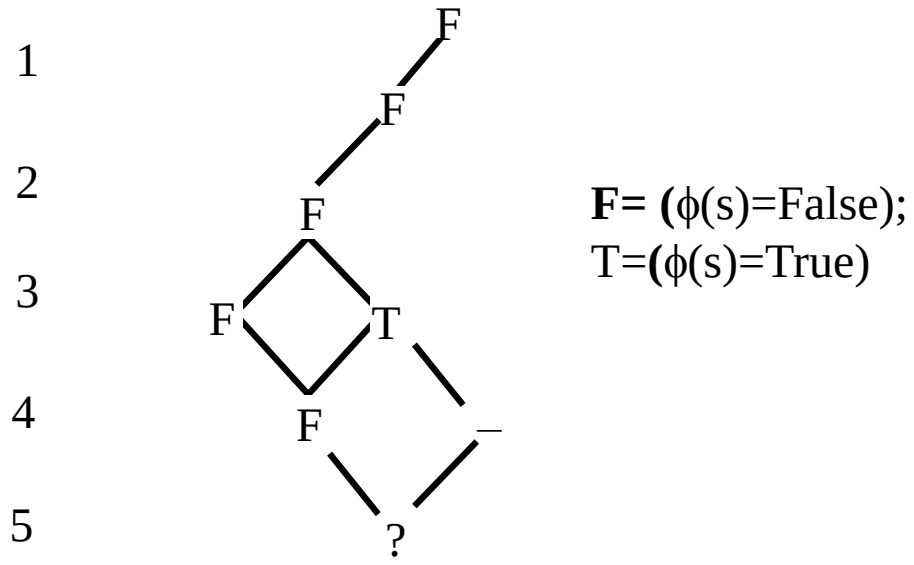
States:=Reachable;

***end while***

输出“可能的ϕ”；

# 分布式调试

## ϕ值判定示例




ϕ在第 4层的状态为True =》 明确的ϕ

ϕ在第 5 层的状态为False =》 可能的ϕ

# 分布式调试

- 异步系统

  开销很大，需要作$O(k^N)$次比较。

- 同步系统

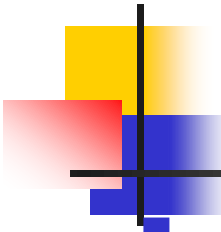
  物理时钟：$|C_i(t) - C_j(t)| < D$，即在范围D内同步。

- 同步系统中的算法改进
  - 消息中同时携带物理时间戳和向量时间戳
  - 测试条件

    $V(s_i)[i] \geqq V(s_j)[i]$，且$s_i$和$s_j$能在同一时间发生

# 第6章 时间和全局状态

- 简介
- 时钟、事件和进程状态
- 物理时钟同步
- 逻辑时间和逻辑时钟
- 全局状态
- 分布式调试
- 小结



# 小结

- 时钟偏移和时钟漂移
- 物理时钟同步
  - Cristian方法
  - Berkeley方法
  - 网络时间协议
- 逻辑时间
  - 发生在先关系
  - Lamport时间戳
  - 向量时钟

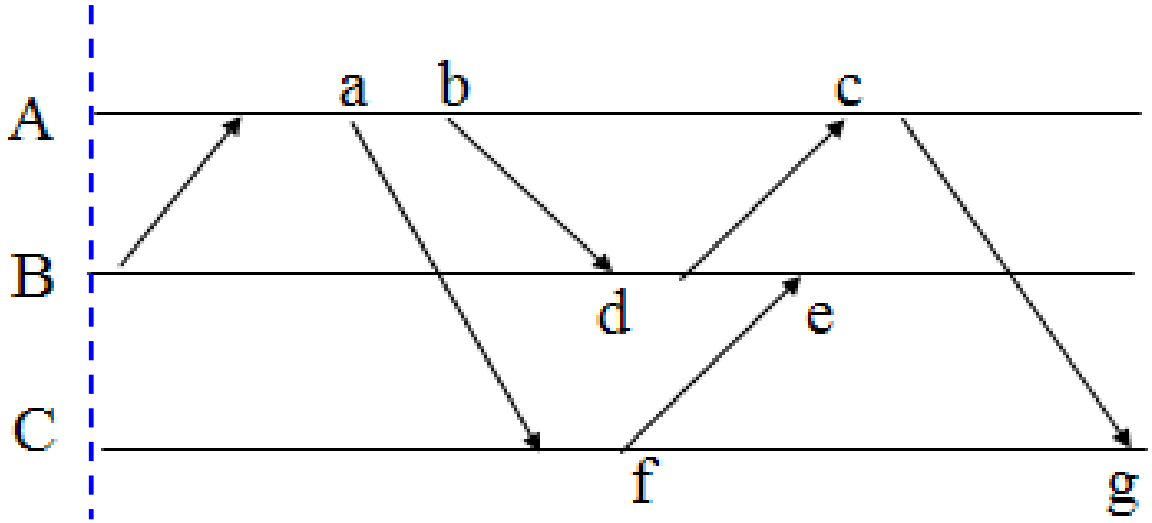
# 小结

- 全局状态
  - 一致割集，一致全局状态
  - “快照”算法
- 分布式调试
  - 状态收集
  - 判定可能的$\phi$和明确的$\phi$

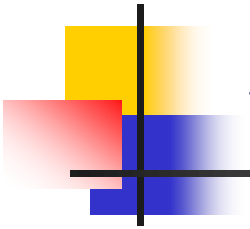
# 作业1

## Lamport时钟练习

假设系统中只存在消息发送和接收事件，如下图所示，请给出事件a-g的逻辑时钟并简单说明下理由。




逻辑时钟 0



# 作业2

Databases-R-Us runs a cluster of three servers A, B, and C, which communicate with one another.

the current clock skews between server pairs are as follows: A-B: 3 ms; B-C: 1 ms; C-A: -4 ms.

Further, correctness in the database requires that no two server clocks be more than 30ms apart.

If each of the servers has an absolute clock drift of 2 ms per minute, how many minimum (i.e., worst-case) minutes can the cluster go without running a synchronization algorithm among its servers?

# 作业3（选做）

a, b, and c are events and no two events belong to the same process.

Prove or disprove (give counter-example) the following:

(a) a is concurrent with b and b is before c implies that a is before c.

(b)a is concurrent with b and b is concurrent with c implies that a is concurrent with c.