

第二部分 分布式算法

第八次课

中国科学技术大学计算机系
国家高性能计算中心（合肥）

第4章

分布式系统中的计算模型

概述

■ 分布式系统计算模型的复杂性

- ❖ 系统由并发执行部件构成
- ❖ 系统中无全局时钟
- ❖ 必须捕捉系统部件可能的失效

■ 对策

- ❖ 因果关系 (Causality)
- ❖ 一致状态 (Consistent states)
- ❖ 全局状态

§ 4.1 基本知识

■ 协议 (Protocol)

■ 协议中的控制语句

1.Send (destination, action; parameters)

destination: 处理器抽象。实用中是通信实体的地址：
机器名，机器的端口号(即1个socket地址)

action: 控制msg，希望接收者采取的动作

parameters: 参数集合

假定:

msg发送是无阻塞、可靠的（语义类似于TCP套接字）；

有时假定较弱的msg传递层（等价于UDP）。

TCP与UDP的区别：①基于连接与无连接②对系统资源的要求（TCP较多，UDP少）
③UDP程序结构较简单④流模式与数据报模式

●TCP保证数据正确性，UDP可能丢包

●TCP保证数据顺序，UDP不保证

§ 4.1 基本知识

2.接收msg

接收msg可推广至接收事件，引起事件的原因是：

外部msg、超时设定、内部中断

事件在处理前，一般是在缓冲区(如事件队列)中，若一处理器想处理事件，它必须执行一个声明处理这些事件的线程。

例如，一个节点通过执行下述代码等待事件 A_1, A_2, \dots, A_n

```
waiting for  $A_1, A_2, \dots, A_n$  : //声明
```

```
 $A_1$  (Source; parameters)
```

```
Code to handle  $A_1$ 
```

```
....
```

```
 $A_n$  (Source; parameters)
```

```
Code to handle  $A_n$ 
```

当p执行send(q, A_1 ; parameters)且q执行上述代码时，q将最终处理由p发送的msg

§ 4.1 基本知识

3. 超时

当怀疑远程处理器失效时，可通过超时检测来判定：

①当T秒后仍未收到P的类型为event的msg时，执行指定的动作

```
waiting until P sends (event; parameters), timeout=T  
on timeout  
    timeout action
```

②仅当收到一个响应msg时才采取动作，超时不做任何动作

```
waiting until P sends (event; parameters), timeout=T  
on timeout;  
if no timeout occurred  
    { Successful response actions }
```

§ 4.1 基本知识

3. 超时

③ 处理器等待响应T秒

若处理器在等待开始后T秒内没响应，则等待结束，协议继续

waiting up to T seconds for (event; parameters) msgs

Event: < msg handling code >

§ 4.2 因果关系

分布式系统为何缺乏全局的系统状态？

1. 非即时通信

A和B同时向对方喊话

他们都认为是自己先喊话

C听到两人是同时喊话

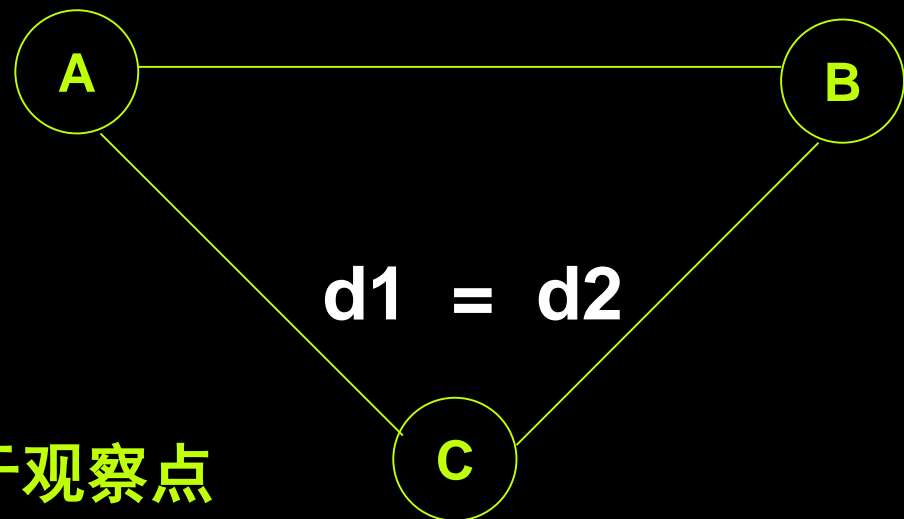
结论：系统的全局状态依赖于观察点

原因：

传播延迟

网络资源的竞争

丢失msg重发



§ 4.2 因果关系

2. 相对性影响

假设张三和李四决定使用同步时钟来观察全局状态：

他们约定下午5点在某餐馆会面，张三准时到达，但李四在一个接近**光速的日光系统**中游览。

张三在等待李四1小时后离开餐馆，而李四在自己的表到达5点时准时达到餐馆，但他认为张三未达到。

因为大多数计算机的实际时钟均存在漂移，故相对速度不同，时钟同步仍然是一个问题。

结论：使用时间来同步不是一个可靠机制。

§ 4.2 因果关系

3. 中断

假设张三和李四在同一起跑线上赛跑，信号为小旗，前两个问题可以忽略，但是...

即使可忽略其他影响，也不可能指望不同的机器会同时做出某些反应。因为现代计算机是一个很复杂的系统：CPU 竞争、中断、页错误等，执行时间无法预料。

结论：不可能在同一时刻观察一个分布式系统的全局状态

必须找到某种可以依赖的性质：

- 时间回溯
- 因果相关

§ 4.2 因果关系

■ 假设分布式系统构成：

$P = \{P_1, P_2, \dots, P_n\}$ ：处理器集合

\mathcal{E} ：全体事件的集合

$\mathcal{E}_p \subseteq \mathcal{E}$, \mathcal{E}_p 表示发生在 p 上的所有事件

■ 次序 $e_1 < e_2$ ：事件 e_1 发生 e_2 在之前（亦记： $e_1 \rightarrow e_2$ ）

$e_1 <_I e_2$ ：事件 e_1 发生 e_2 在之前， I 为信息源

■ 定序 有些 \mathcal{E} 中事件很容易定序：

❖ 发生在同一节点 p 上的事件满足全序：

若 $e_1, e_2 \in \mathcal{E}_p$, 则 $e_1 <_p e_2$ 或 $e_2 <_p e_1$ 成立

❖ e_1 发送消息 m , e_2 接收 m , 则 $e_1 <_m e_2$

§ 4.2 因果关系

■ Happens-before关系 (\langle_H)

该关系是节点次序和消息传递次序的传递闭包：

❖ 规则1：若 $e_1 \langle_p e_2$ ，则 $e_1 \langle_H e_2$

❖ 规则2：若 $e_1 \langle_m e_2$ ，则 $e_1 \langle_H e_2$

❖ 规则3：若 $e_1 \langle_H e_2$ ，且 $e_2 \langle_H e_3$ ，则 $e_1 \langle_H e_3$

在集合 X 上的二元关系 R 的传递闭包是包含 R 的 X 上的最小的传递关系。

$e_1 \langle_H e_3$ 表示存在1个事件因果链，使 e_1 发生 e_3 在之前

Note: \langle_H 是一种偏序关系，即

存在 e 和 e' ，二者之间无这种关系

❖ 并发事件：若两事件不能由 \langle_H 定序

§ 4.2 因果关系

■ Happens-before关系 (\prec_H)

- ❖ 规则1表述的是同一处理器上两个事件之间的因果关系；
- ❖ 规则2表述了不同处理器上两个事件之间的因果关系
- ❖ 规则3阐述了传递律

Note:

Happens-Before关系完整表述了执行中的因果关系。如果一个执行中的事件按照其相对顺序重新进行排序，而不改变他们的happens-before关系的话，那么其结果仍是一个执行，并且对处理器而言，该执行与原执行并无区别。

§ 4.2 因果关系

■ 举例

1) 因事件 e_1 , e_4 和 e_7 均发生在 P_1 上, 故: $e_1 <_{P_1} e_4 <_{P_1} e_7$

2) 因 e_1 发送1个msg到 e_3 , 故: $e_1 <_m e_3$, 类似地 $e_5 <_m e_8$

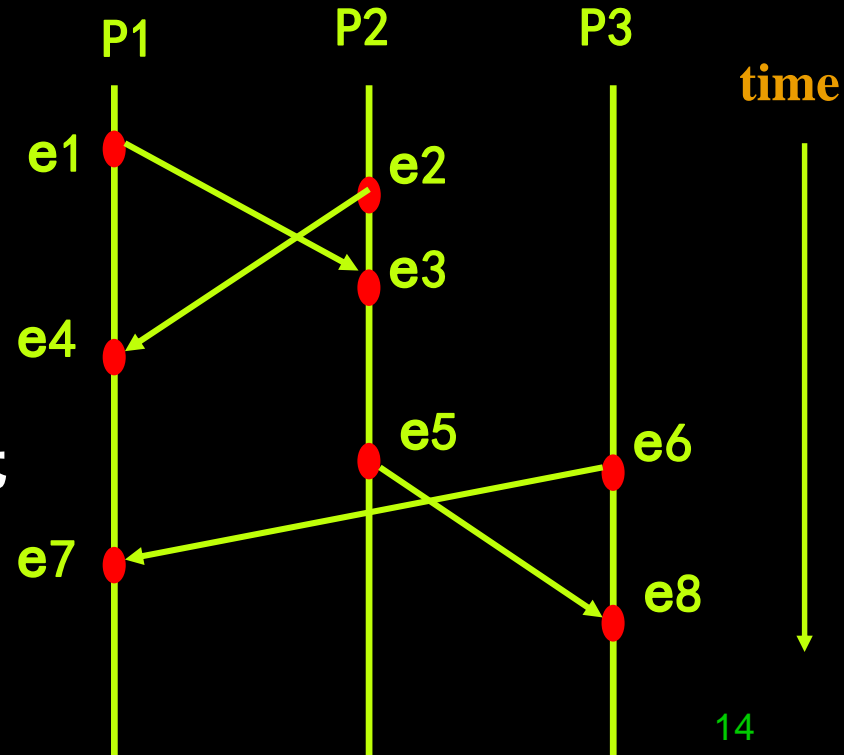
3) 应用规则1和2得:

$$e_1 <_H e_4 <_H e_7, e_1 <_H e_3, e_5 <_H e_8$$

4) 由 $<_H$ 的传递闭包性质得:

$$e_1 <_H e_8$$

5) e_1 和 e_6 是并发的: e_1 和 e_6 之间无路径



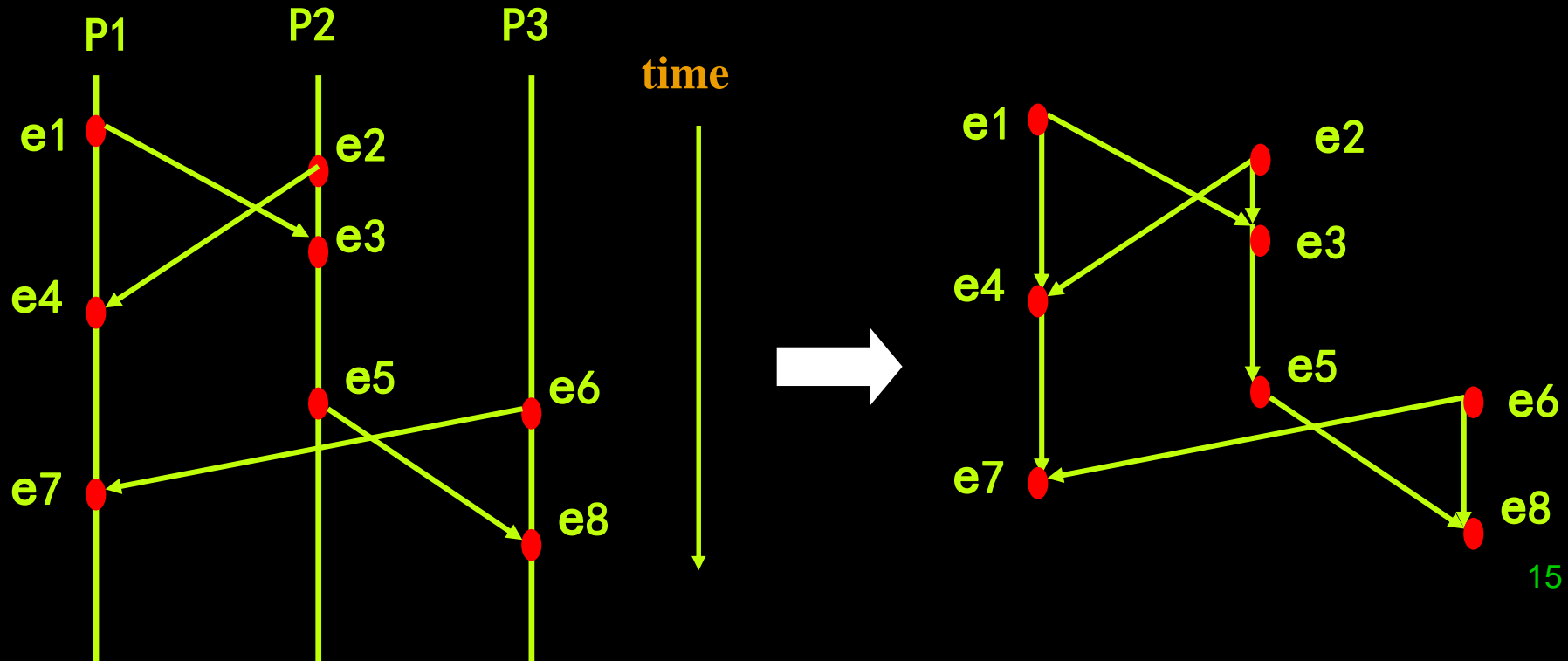
§ 4.2 因果关系

■ H-DAG

有时将Happens-before关系描述为一个有向无环图

❖ **顶点集** V_H 是事件集 \mathcal{E} : $e \in V_H$ 当且仅当 $e \in \mathcal{E}$

❖ **边集** E_H : 若 $(e_1, e_2) \in E_H$ 当且仅当 $e_1 <_P e_2$ 或 $e_1 <_m e_2$



§ 4.2.1 Lamport时间戳

■ 系统有序性的重要性

若分布式系统中存在全局时钟，则系统中的事件均可安排为全序。例如，可以更公平地分配系统资源。

■ 全序对事件的影响和由H关系确定的偏序对事件的影响是一致的

■ 如何通过H关系确定的偏序关系来建立一个“一致”的全序关系？

❖ 在 \langle_{H} 的DAG上拓扑排序

❖ On the fly: Lamport提出了动态即时地建立全序算法

§ 4.2.1 Lamport时间戳

■ Lamport算法的思想

每个事件 e 有一个附加的时戳： $e.TS$

每个节点有一个局部时戳： my_TS

每个msg有一个附加时间戳： $m.TS$

节点执行一个事件时，将自己的时戳赋给该事件；

节点发送msg时，将自己的时戳赋给所有发送的msg。

§ 4.2.1 Lamport时间戳

■ Lamport算法的实现

Initially: $my_TS=0$;

On event e :

if (e 是接收消息 m) then

$my_TS = \max (m.TS, my_TS)$;

//取msg时戳和节点时戳的较大者作为新时戳

my_TS++ ;

$e.TS=my_TS$; **//给事件 e 打时戳**

if (e 是发送消息 m) then

$m.TS=my_TS$; **//给消息 m 打时戳**

§ 4.2.1 Lamport时间戳

■ Lamport算法赋值的时戳是因果相关的

若 $e_1 <_H e_2$, 则 $e_1.TS < e_2.TS$

∵ 若 $e_1 <_P e_2$ 或 $e_1 <_M e_2$, 则的 e_2 时戳大于 e_1 的时戳

∴ 在因果事件链上, 每一事件的时戳大于其前驱事件的时戳

■ 问题: 系统中所有事件已为全序?

不同的事件可能有相同的时戳 (并发事件)

■ Lamport算法改进

因为并发事件的时戳可以任意指定先后
故可用节点地址作为时戳的低位

§ 4.2.1 Lamport时间戳

■ 改进的Lamport时戳

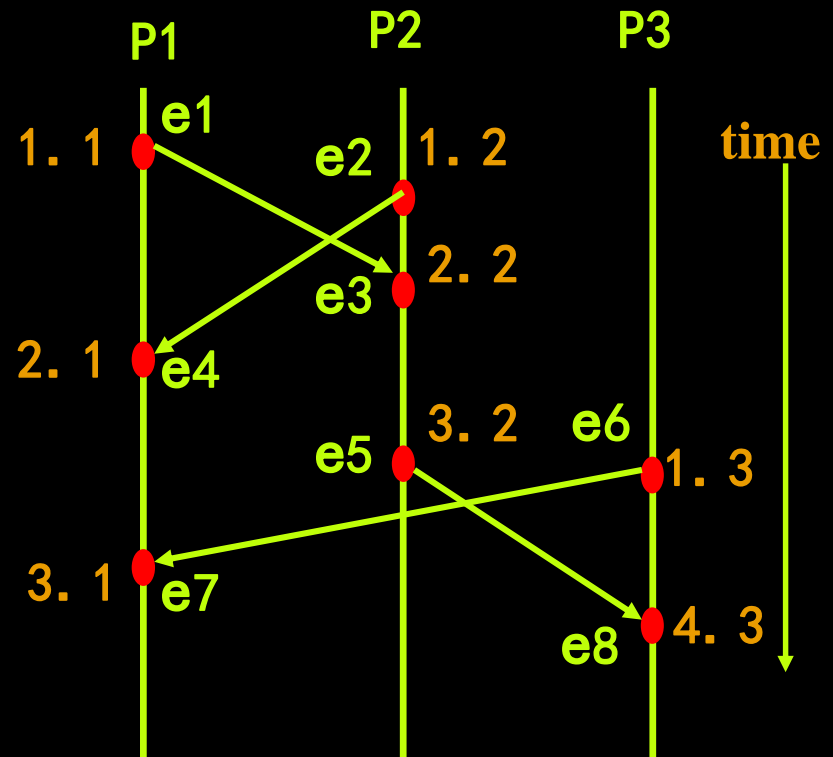
事件标号：时戳. id

事件e8为4.3:

$$\begin{aligned} \text{my_TS} &= \max(\text{m. TS}, \text{my_TS}) \\ &= \max(3, 1) = 3 \end{aligned}$$

按字典序得全序:

1.1, 1.2, 1.3, 2.1, 2.2, 3.1, 3.2, 4.3



■ 算法特点：分布、容错、系统开销小

- Lamport算法的迷人之处在于：任何进程在发送消息前，先将自己的本地计数器累加1；接收进程总是计算自己的本地计数器和接受到计数器中较大值加上1的结果

§ 4.2.2 向量时间戳

■ Lamport时戳缺点

若 $e_1 \prec_H e_2$, 则 $e_1 \cdot TS < e_2 \cdot TS$; 反之不然。

例如: $1.3 < 2.1$, 但是 $e_6 \prec e_4$ 不成立

原因: 并发事件之间的次序是任意的

不能通过事件的**时戳判定**两事件之间是否是**因果相关**

■ 判定事件间因果关系的重要性

例子: 违反因果关系检测

在一个分布式对象系统中, 为了负载平衡, 对象是可移动的, 对象在处理器之间迁移是为了获得所需的调用的进程或资源。如下图:

§ 4.2.2 向量时戳

1) P1持有对象O，决定迁移到P2

为获取资源，P1将O装配在消息M1中发送给P2

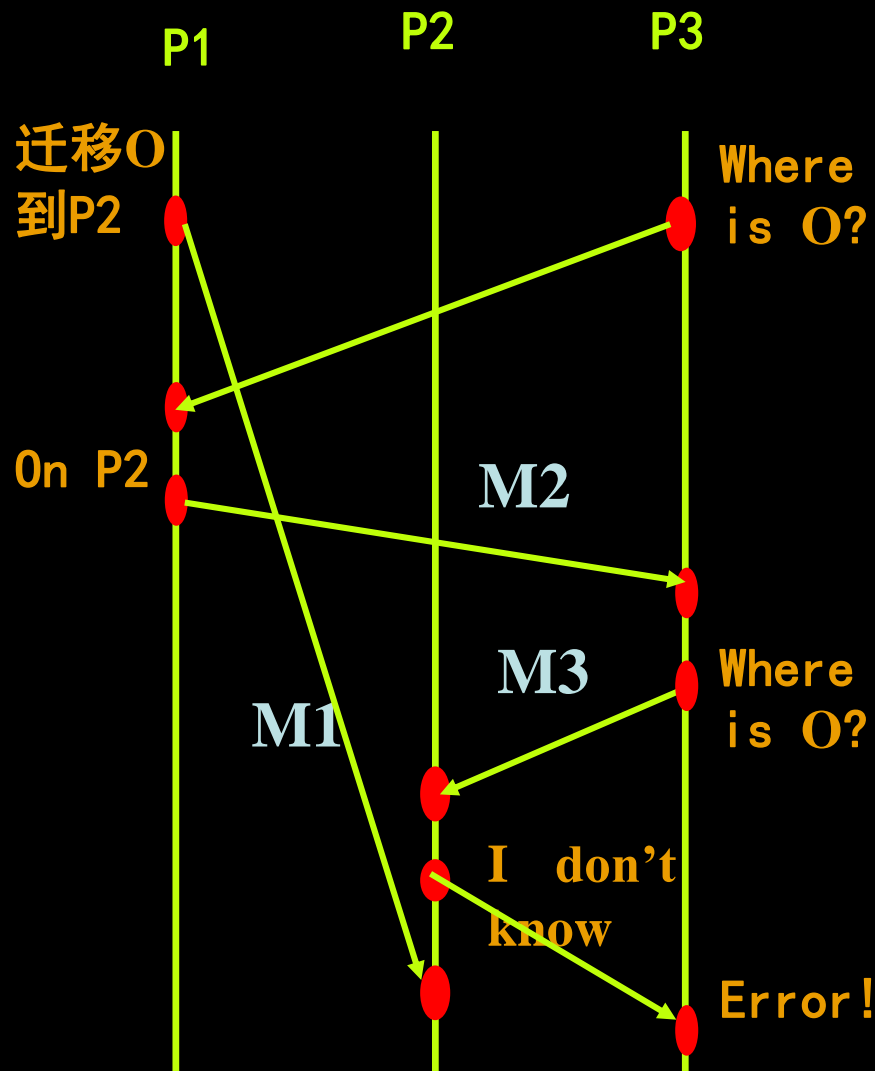
2) P1收到P3访问O的请求

P1将O的新地址P2放在消息M2中通知P3

3) P3在M3中请求访问P2的O

当M3达到P2时，O不可用，故回答一个出错消息。

- 问题：当debug该系统时，会发现O已在P2上，故不知错在哪？



§ 4.2.2 向量时间戳

■ 错误原因：违反因果序

P3请求O是发生在从P1迁移到P2之后，但该请求被处理是在迁移达到P2之前。形式地，

设：s(m)是发送m的事件

r(m)是接收m的事件

若 $s(m1) <_H s(m2)$ ，则称消息m1因果关系上先于m2，记做 $m1 <_C m2$

若 $m1 <_C m2$ ，但 $r(m2) <_P r(m1)$ ，则违反“因果关系”：

即，若m1先于m2发送，但在同一节点P上m2在m1之前被接收

例如，在上例中有：

$M1 <_C M3$ ，但 $r(M3) <_{P2} r(M1)$

§ 4.2.2 向量时间戳

■ 违反因果序检测

❖ **定义**：若时戳VT具有比较函数 $<_V$ 满足：

$$e1 <_H e2 \text{ iff } e1.VT <_V e2.VT$$

则我们能够检测出是否违反因果关系

❖ VT性质

- 1) 因 $<_H$ 是偏序，故 $<_V$ 也是偏序；
- 2) 因为必须知道在因果关系上每一节点中哪些事件是在事件e之前，故e.VT中必须包含系统中每一个其它节点的状态。

这两个性质导致了**向量时戳VT**的引入

§ 4.2.2 向量时戳

■ 向量时戳VT

VT是一个整数数组：

e.VT[i]=k表示在节点i（或 P_i ）上，因果关系上e之前有k个事件（可能包括e自己）。

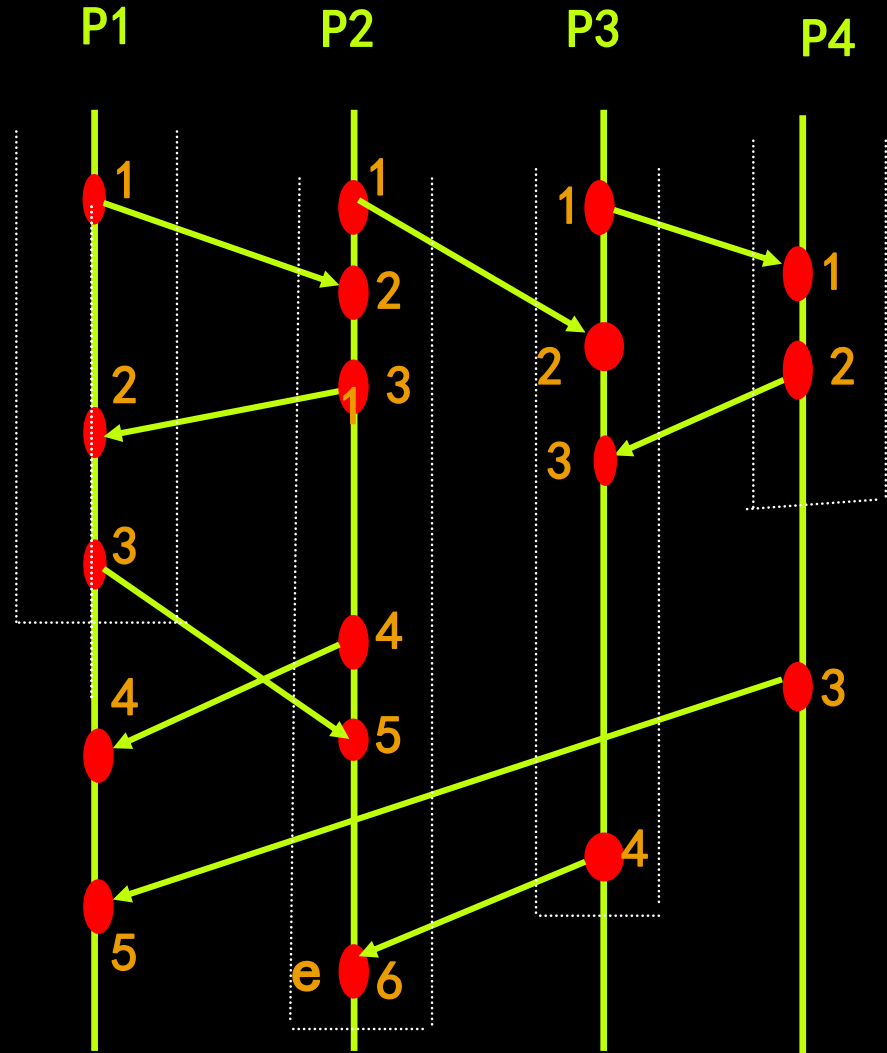
e.VT=(3,6,4,2)表示因果序：

在P1上，有3个事件须在e之前

在P2上，有6个事件须在e之前

在P3上，有4个事件须在e之前

在P4上，有2个事件须在e之前



§ 4.2.2 向量时间戳

■ 向量时戳的意义

在因果关系上， $e1.VT \leq_v e2.VT$ 表示 $e2$ 发生在 $e1$ 及 $e1$ 前所有的事件之后。更精确的说，向量时钟的次序为：

$e1.VT \leq_v e2.VT$ iff $e1.VT[i] \leq e2.VT[i]$, $i=1, 2, \dots, M$

$e1.VT <_v e2.VT$ iff $e1.VT \leq_{vT} e2.VT$ 且 $e1.VT \neq e2.VT$

■ 向量时戳算法

my_VT: 每个节点有局部的向量时戳

e.VT: 每个事件有向量时戳

m.VT: 每个msg有向量时戳

节点**执行**一个事件时，将自己的时戳赋给该**事件**；

节点**发送**msg时，将自己的时戳赋给所有**发送的msg**。

注意 \leq_v , \leq_{vT} 以及
 $<_v$ 之间的区别：
 v 代表因果序，
而VT代表时戳比较

§ 4.2.2 向量时间戳

■ 算法实现

Initially: $my_VT=[0,\dots,0]$;

On event e :

if (e 是消息 m 的接收者) then

for $i=1$ to M do //向量时戳的每个分量只增不减

$my_VT[i] = \max(m.VT[i], my_VT[i]);$

$my_VT[self]++;$ //设变量 $self$ 是本节点的名字

$e.VT=my_VT;$ //给事件 e 打时戳

if (e 是消息 m 的发送者) then

$m.VT=my_VT;$ //给消息 m 打时戳

§ 4.2.2 向量时间戳

■ 算法性质

1) 若 $e <_H e'$, 则 $e.VT <_{VT} e'.VT$

∴ 算法确保对于每个事件满足:

若 $e <_P e'$ 或 $e <_M e'$, 则 $e.VT <_{VT} e'.VT$

2) 若 $e \prec_H e'$, 则 $e.VT \prec_{VT} e'.VT$

pf: 若 e 和 e' 因果相关, 则有 $e' <_H e$, 即 $e'.VT <_{VT} e.VT$

若 e 和 e' 是并发的, 则在 H-DAG 上, 从 e 到 e' 和从 e' 到 e 均无有向路径, 即得:

$$e.VT \prec_{VT} e'.VT \text{ 且 } e'.VT \prec_{VT} e.VT$$

当且仅当 $e.VT$ 和 $e'.VT$ 是不可比时, 称向量时间戳是捕获并发的! ²⁸

§ 4.2.2 向量时戳

■ 向量时戳比较

$$e1.VT=(5,4,1,3)$$

$$e2.VT=(3,6,4,2)$$

$$e3.VT=(0,0,1,3)$$

1) e1和e2是并发的

$$\because e1.VT[1] > e2.VT[1]$$

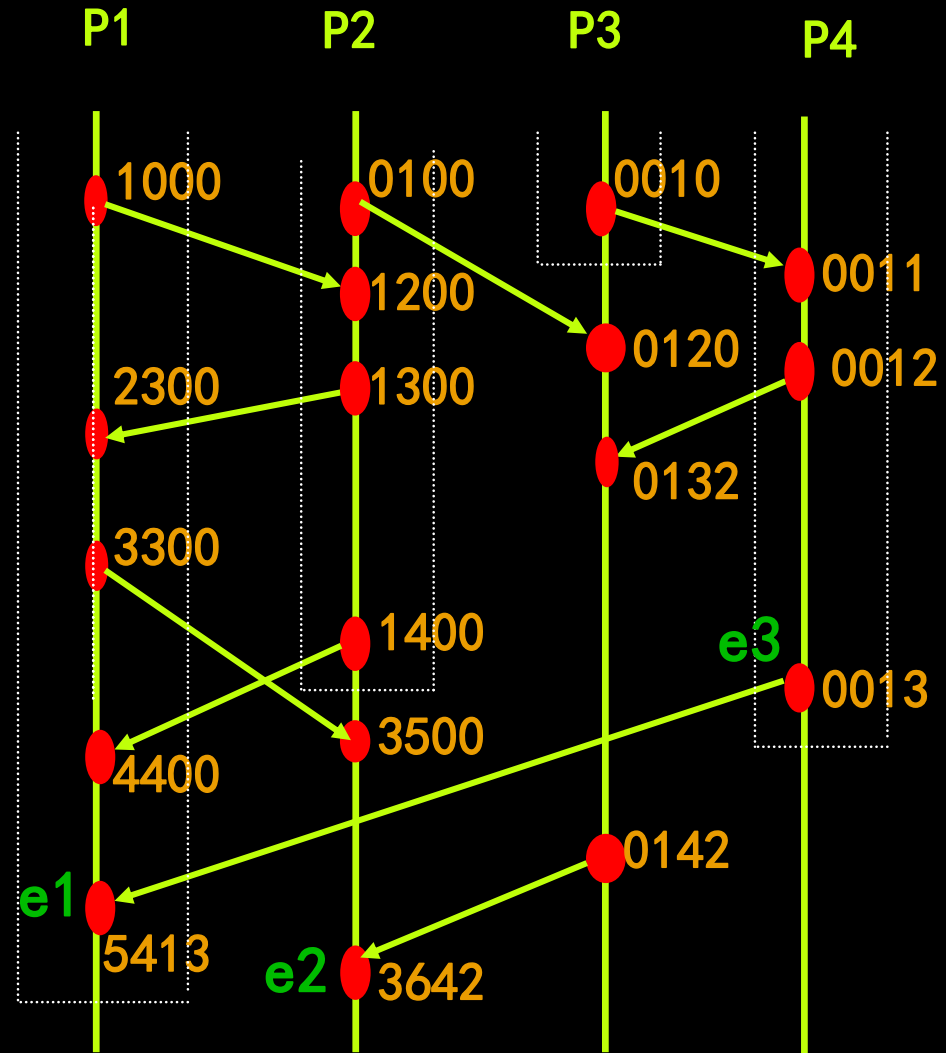
$$e1.VT[2] < e2.VT[2]$$

\therefore e1到e2及e2到e1均无路径

2) e3在因果序上先于e1

$$\text{即: } e3.VT <_v e1.VT$$

e1的前驱事件见方框



§ 4.2.2 向量时戳

■ 因果序检测

1) 消息时戳间比较

在P2上，先到达的M3的时戳为(3,0,3)，后到达的M1的时戳为(1,0,0)。但是：

$\because (1,0,0) <_v (3,0,3)$

\therefore M1在因果序上先于M3

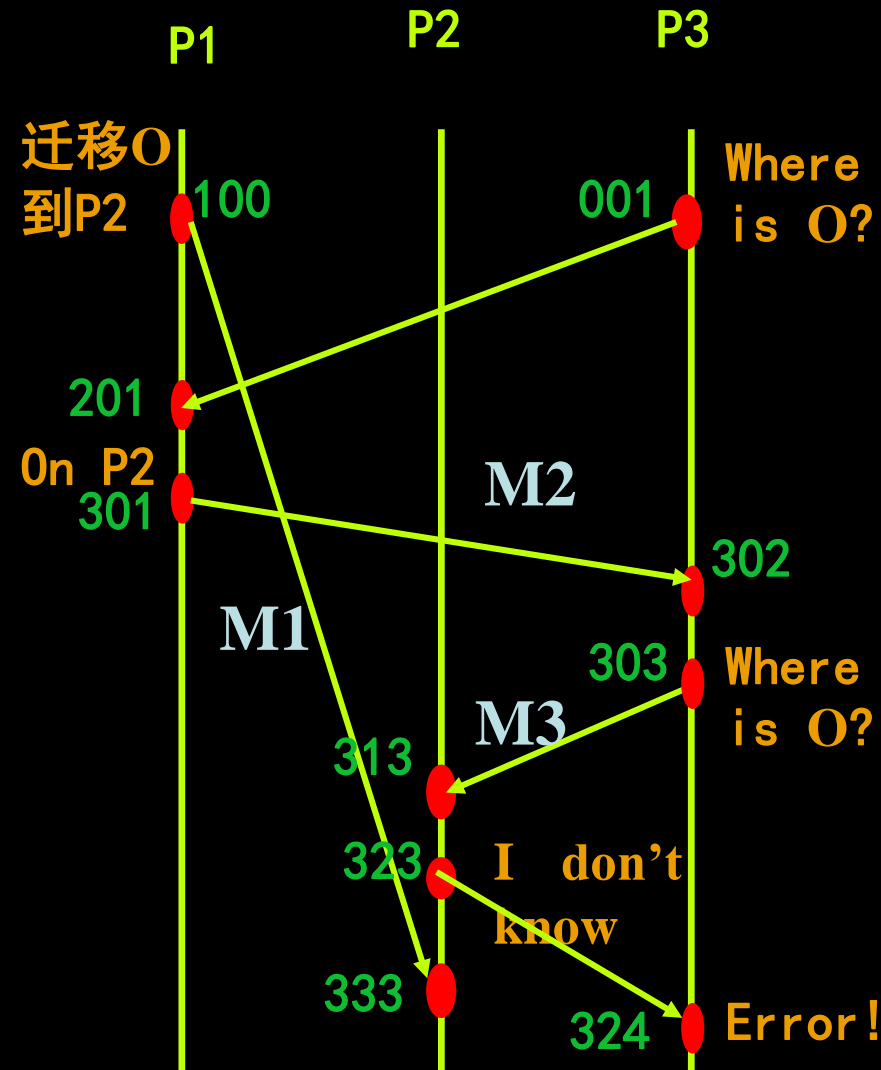
故M1后于M3到达违反因果序

2) 消息时戳和局部时戳比较

当时戳为(1,0,0)的M1到达P2时，P2的时戳是(3,2,3)。但：

$\because (1,0,0) <_v (3,2,3)$

\therefore M1在因果序上应先于(3,2,3)对应的事件



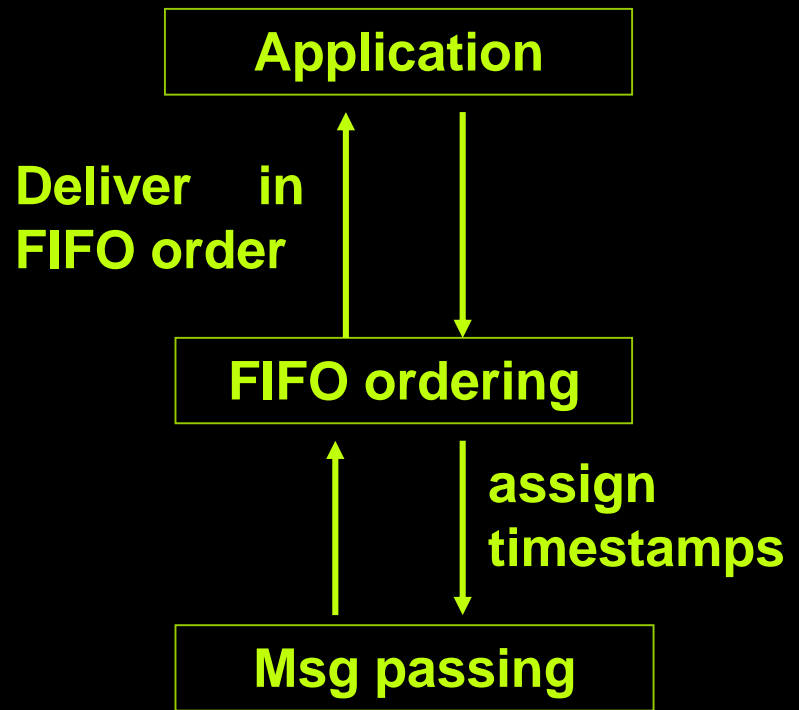
§ 4.2.3 因果通信

■ 如何保证通信不违反因果关系？

处理器不能选择msg达到的次序，但能抑制过早达到的msg来修正传递（指提交给应用）次序。

■ FIFO通信（如TCP）

由msg传递协议栈里的一层负责确保FIFO通信



§ 4.2.3 因果通信

■ FIFO通信

源处理器给每个发送的msg顺序编号，目的处理器知道自己所收到的msg应该有何顺序的编号，若目的处理器收到一个编号为x的msg但未收到较小编号的msg时，则延迟传递直至能够顺序传递为止。

■ 因果通信

因果通信与FIFO通信类似

源：附上时戳

目的地：延迟错序msg



§ 4.2.3 因果通信

■ 因果通信实现思想

❖ 抑制从P发送的消息 m ，直至可断定没有来自于其它处理器上的消息 m' ，使 $m' \prec_v m$ 。

❖ 在每个节点P上：

earliest[1..M]：存储不同节点当前能够传递的消息时戳的下界

earliest[k]表示在P上，对节点k能够传递的msg的时戳的下界

blocked[1..M]：阻塞队列数组，每个分量是一个队列

■ Alg. Causal Msg delivery

定义时戳 1_k ：若使用Lamport时戳，则 $1_k = 1$ ；

若用向量时戳，则 $1_k = (0, \dots, 1, 0, \dots, 0)$ ， k^{th} 位为1

初始化

1: $\text{earliest}[k] = 1_k$ ， $k=1, \dots, M$

2: $\text{blocked}[k] = \{ \}$ ， $k=1, \dots, M$ //每个阻塞队列置空

§ 4.2.3 因果通信

```
3: On the receipt of msg m from node p:
4:   delivery_list={};
5:   if (blocked[p]为空) then
6:     earliest[p]=m.timestamp;
7:   将m加到blocked[p]队尾; //处理收到的消息
8:   while (  $\exists k$ 使blocked[k]非空 and 对每个 $i=1, \dots, M$ (除k和self外),
           not_earliest( earliest[i], earliest[k], i )) { //处理阻塞队列
           //对非空队列k, 若其他节点i上无比节点k更早的msg要达到本
           //地, 则队列k的队首可解除阻塞
9:     将blocked[k]队头元素m'出队, 且加入到delivery_list;
10:    if (blocked[k]非空) then
11:      将earliest[k]置为m'.timestamp;
12:    else increment earliest[k] by  $1_k$  } //end while
13: deliver the msgs in delivery_list; //按因果序
```

§ 4.2.3 因果通信

■ 向量时戳比较

```
not_earliest( proc_i_vts, msg_vts, i ) { //前者不早于后者时为真
    if ( msg_vts[i]<proc_i_vts[i] )
        return true;
    else return false;
}
```

■ 分析 使用向量时戳较好，不会假定序。

1) **初始化**: 在本地节点 (self) 上，能够**最早传递**的来自于节点k的msg的时戳存储在本地的earliest[k]中，line1初始化正确

2) **处理接收的消息**: 当本地节点接收来自于p的消息m时

行5,6: 若blocked[p]中无msg，则earliest[p]被置为m的时戳，这是因为从p上不会有更早的msg达到本地，更早的已处理过。

§ 4.2.3 因果通信

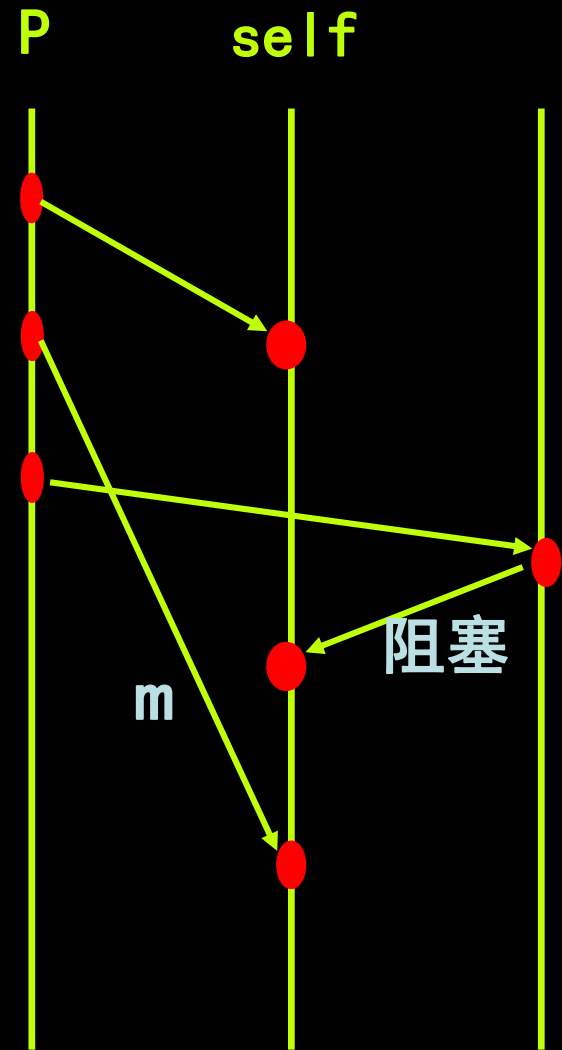
■ 分析

行7：将m放入阻塞队列blocked[p]中，直到可安全传送为止。

3) 处理阻塞：因m的达到能够将earliest[p]中的时戳更新（变大），故可能会使若干个阻塞的msg变为非阻塞；当然m本身可能是安全的，可直接传递给应用。

最终，一个阻塞msg变为非阻塞msg后，也可能使其他阻塞msg变为非阻塞msg。

行8：while循环检查阻塞队列，对于非空队列k进行处理。



§ 4.2.3 因果通信

■ 分析

什么样的msg是非阻塞的？

当阻塞队列k（指blocked[k]）非空时，检测其能否解除阻塞：
设队头msg是m，在self上若其他节点i（除self和k之外）无更早时戳earliest[i]（即比m.timestamp=earliest[k]更小）的msg可能被传递，则消息m是非阻塞的。此时，m可安全传递，m出队（行9）。

4) 问题

上述算法可能会发生死锁：若一节点长时间不发送你要的msg，会发生死锁。

因此，上述因果通信算法通常被用于组播的一部分。

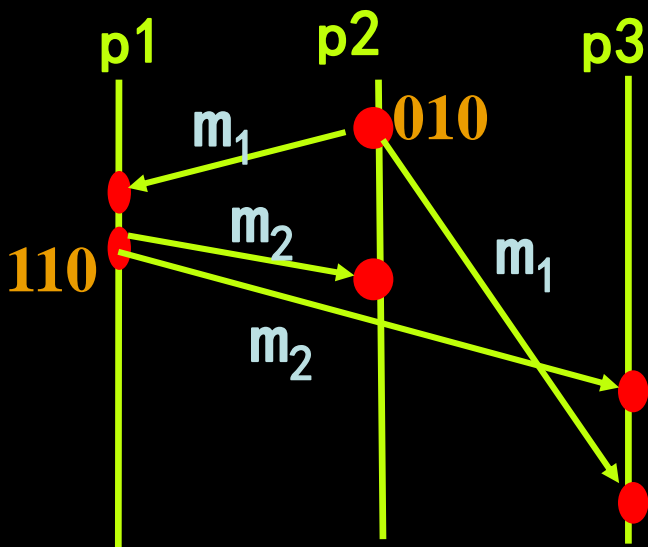
§ 4.2.3 因果通信

■ 算法执行例子 (Causal Multicast)

因为协议只对发送事件的因果序感兴趣，故一节点只有**发送**msg时对向量时戳做**增量**操作。

①**处理接收消息**：当p3从p1接收m2时，**修改**earliest[1]为(1,1,0)，m2入阻塞队列blocked[1]；//line6, 7

②**处理阻塞**：blocked[1]≠Φ，故**while**循环中k=1, self=3, i=2, **not_earliest**(earliest[2], earliest[1], 2)，前者早于后者，表示p2上有一个更早的msg还没有达到p3，返回**假**。m2被**阻塞**



在p3上

earliest[1]	earliest[2]	earliest[3]
(1,0,0)	(0,1,0)	(0,0,1)

接收m2 //k=1, i=2, **阻塞m2**
m2
 $(1,1,0) \geq (0,1,0)$ (0,0,1)

§ 4.2.3 因果通信

①处理接收消息：当p3从p2接收m1时，earliest[2]为(0,1,0)(不变)，m1入阻塞队列blocked[2]；//line6, 7

②处理阻塞：while检测各阻塞队列是否有阻塞的msg。

k=2, blocked[2]≠Φ，其中的m1不依赖其他事件，故放入传递表(行9)，m1出队后blocked[2]=Φ，earliest[2][2]+1(行12)后earliest[2]=(0,2,0)；

k=1, blocked[1]≠Φ，self=3, i=2, not_earliest(earliest[2], earliest[1], 2)，前者不早于后者，表示p2上无更早的msg会达到p3，返回真。m2从blocked[1]出队入传递表(行9)，blocked[1]=Φ，earliest[1][1]+1(行12)后earliest[1]=(2,1,0)。

earliest[1] earliest[2]

接收m1

m2	m1	//k=2, i=1
(1,1,0)	>	(0,1,0)
		(0,0,1)

deliver m1

m2		//k=1, i=2
(1,1,0)	<	(0,2,0)
		(0,0,1)

deliver m2

(2,1,0)		(0,2,0)		(0,0,1)
---------	--	---------	--	---------

