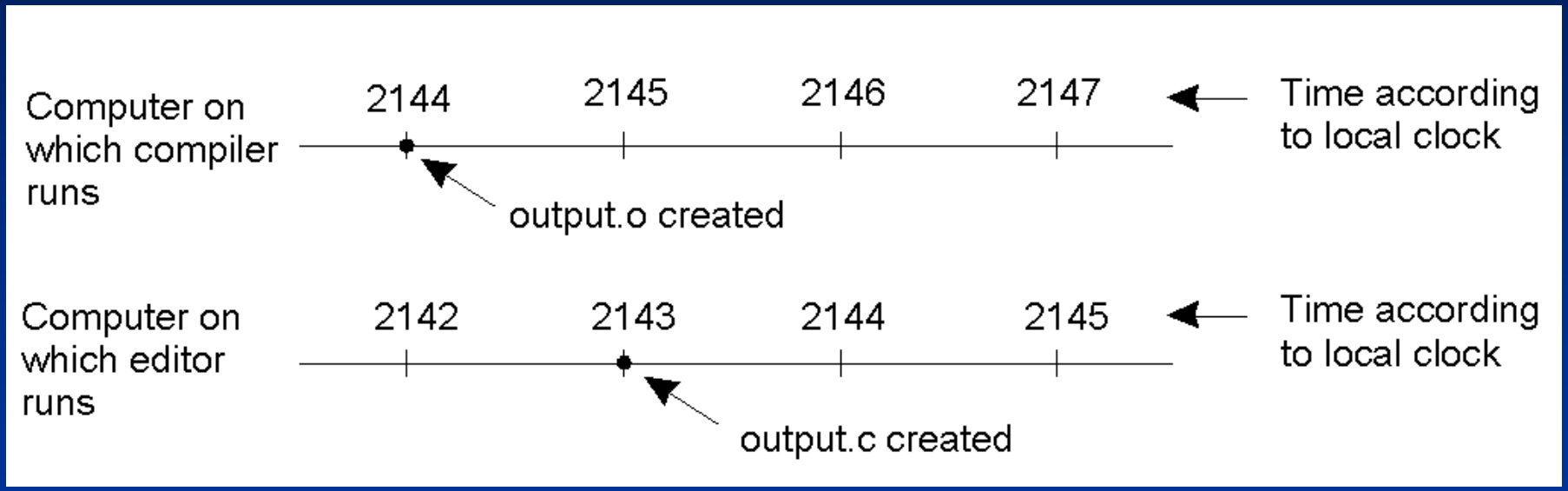
第六章 同步

- 时钟同步
- 逻辑时钟
- 选举算法
- 互斥
- 分布式事务
- 分布式系统中的死锁

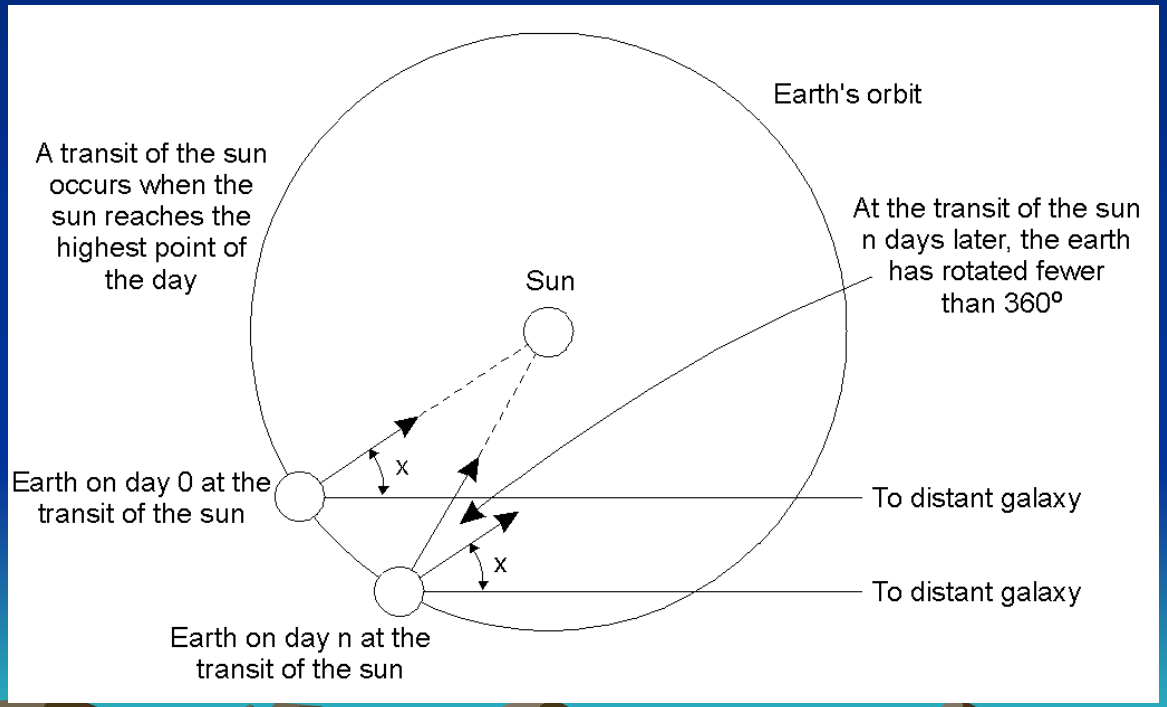
时钟同步



- 分布式系统中，不存在公共时钟或精确的全局时间
- 当每台机器都有自己的时钟时，一个发生较晚的时间可能被标上较早的时间
- 例子：Unix中的make程序

物理时钟

- 太阳日：连续的两次日中天的时间
- 太阳秒：**solar-day/86400**
- 平均太阳秒：很多天的长度平均/86400，格林威治时间



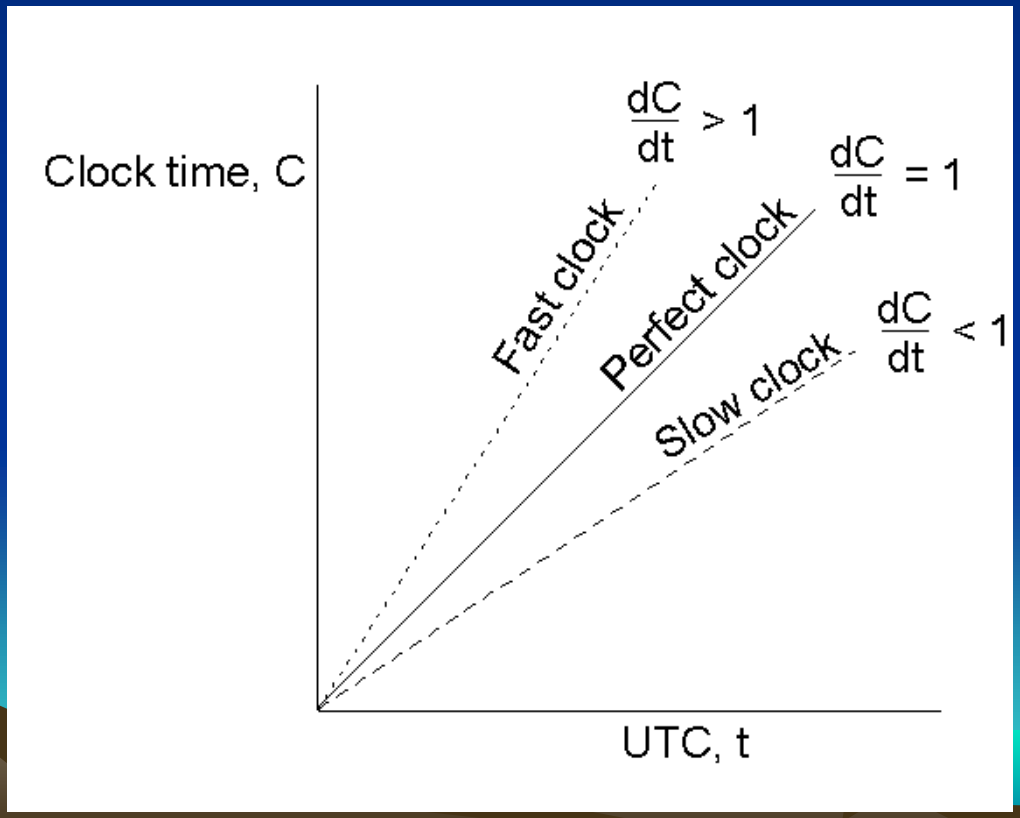
平均太阳日的计算

现实时钟

- 铯原子钟：9192631770次跃迁=1秒
- TAI秒：国际原子时间
- UTC秒：统一协调时间（在TAI秒中加入闰秒）
- 时间服务：WWV电台、GEOS卫星

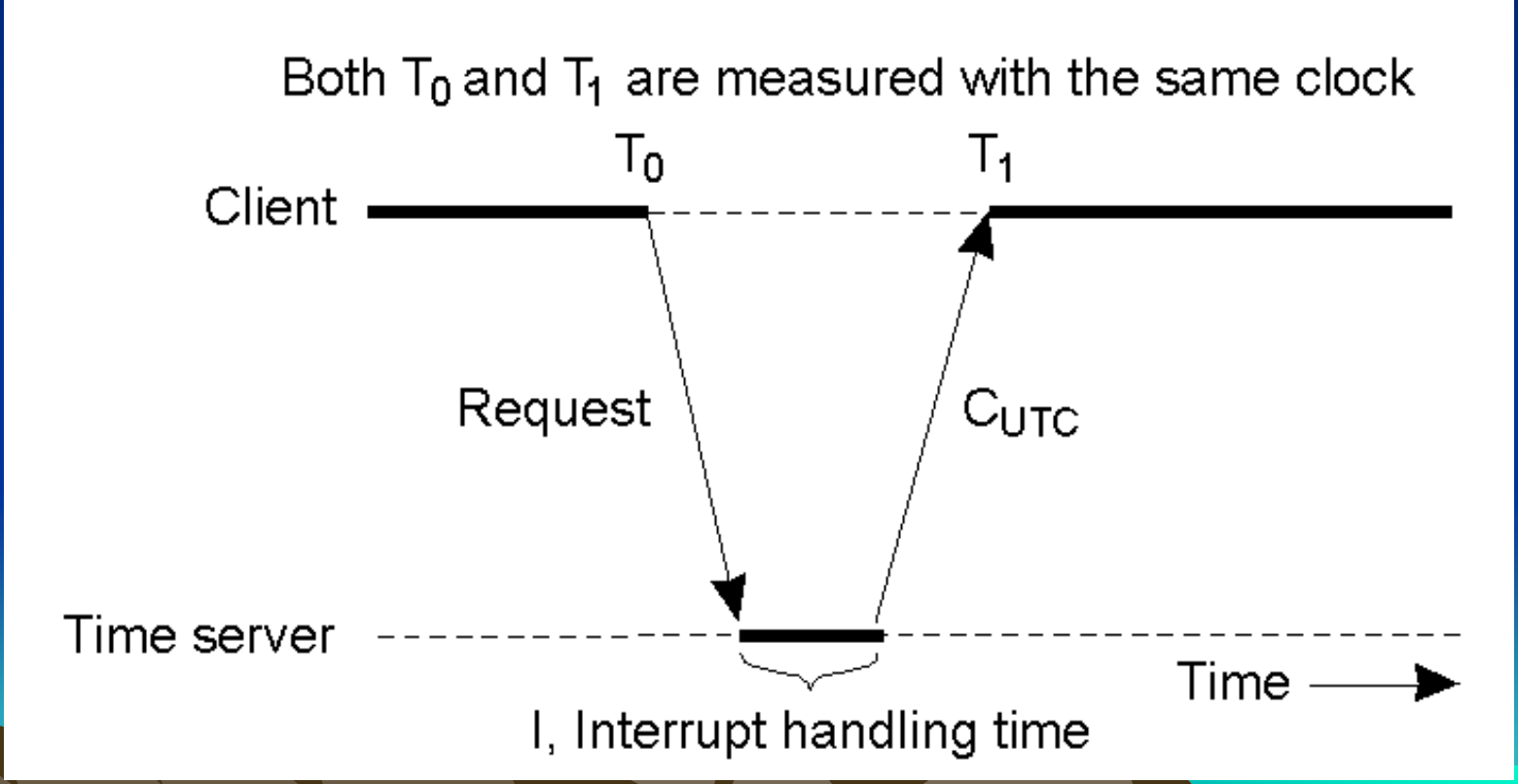
时钟同步算法

- 当时钟以不同的速率滴答时，时钟时间与UTC之间的关系：当UTC时间为t时，机器上的时间为$C_p(t)$，理想情况是$C_p(t)=t$,即$dC/dt=1$.
- $1-p<=dC/dt<=1+p$
- 为保证每两个时钟间的差值不超过a,则时钟必须至少每a/2p秒重新同步一次



Cristian算法

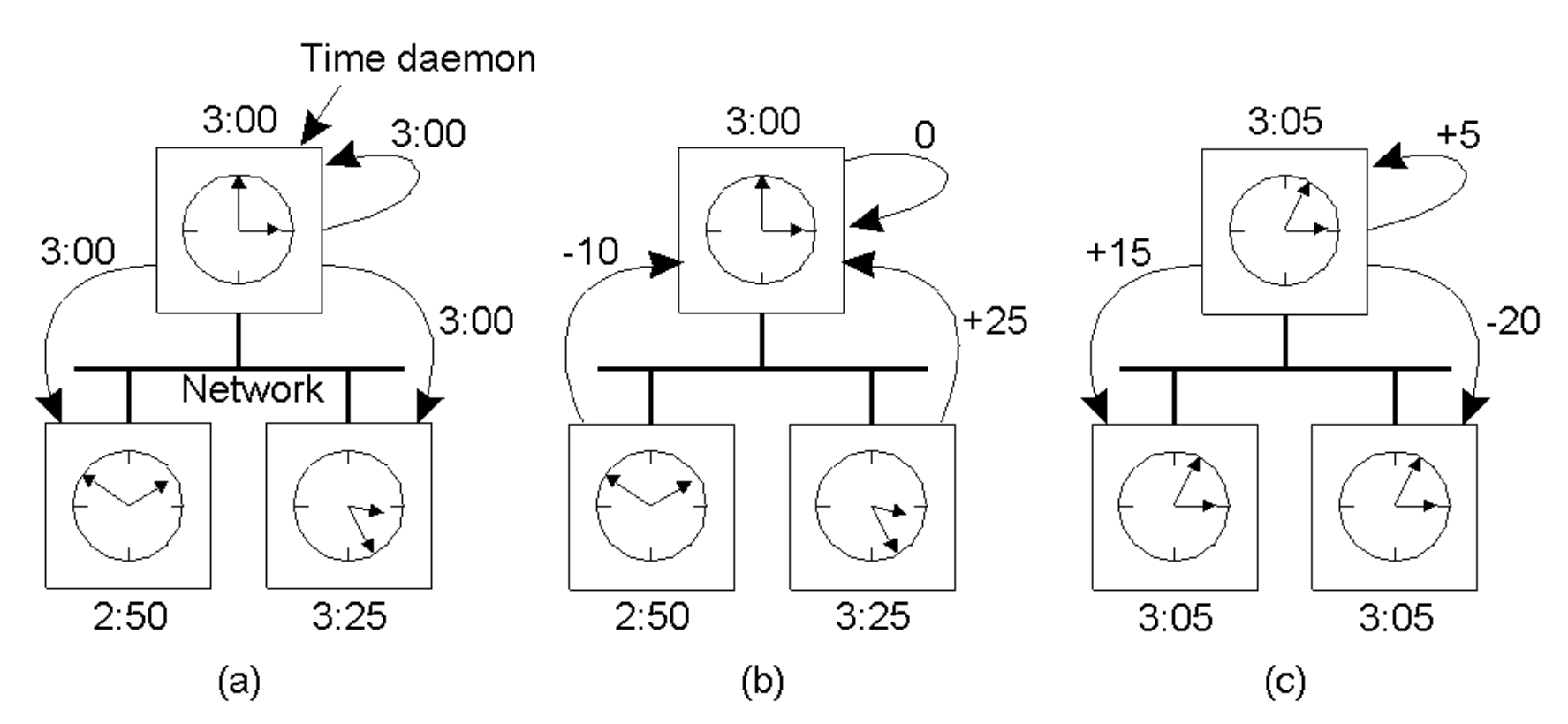
- 适合只有一台时间服务器的情况，可接收WWV的UTC时间



Cristian算法

- 不能简单接受C_{UTC}
  - 时间不能倒退，须逐步修正
    - 假设：每秒产生100次中断，每次中断将时间加10毫秒
    - 若调慢时钟，中断服务程序每次只加9毫秒；
    - 若加快时钟，则加11毫秒。
  - 存在延迟

Berkeley 算法



a) time daemon 向所有其他机器发送自己的时间，并询问其时钟值
b) 其他机器应答
c) time daemon计算出平均值，通知其他机器如何调整时钟

平均值算法

- 非集中式算法
- 时间间隔[T_0+iR, $T_0+(i+1)R$]
- 在每次时间间隔开始，每台机器广播自己的当前时间
- 之后启动本地计时器，搜集时间间隔S内到达的所有时间广播，计算时间值
  - 平均值
  - 去掉若干个最低值和最高值
  - 给每个消息加上传输时间的估计值

逻辑时钟
Lamport Timestamps

时间戳（ Time-Stamping）的算法：

- 网络上的每个系统（站点）维护一个计数器，起时钟的作用
- 每个站点有一个数字型标识,消息的格式为（m,Ti,i）,m为消息内容， Ti为时间戳， i为站点标识
- 当系统发送消息时，将时钟加一
- 当系统j接收消息时，将它的时钟设为当前值和到达的时间戳这两者的最大者加一
- 在每个站点，时间的排序遵循以下规则
  - 对来自站点i的消息x和站点j的消息y,如果
    - Ti<Tj或
    - Ti=Tj，且i<j
  - 则说消息x早于消息y

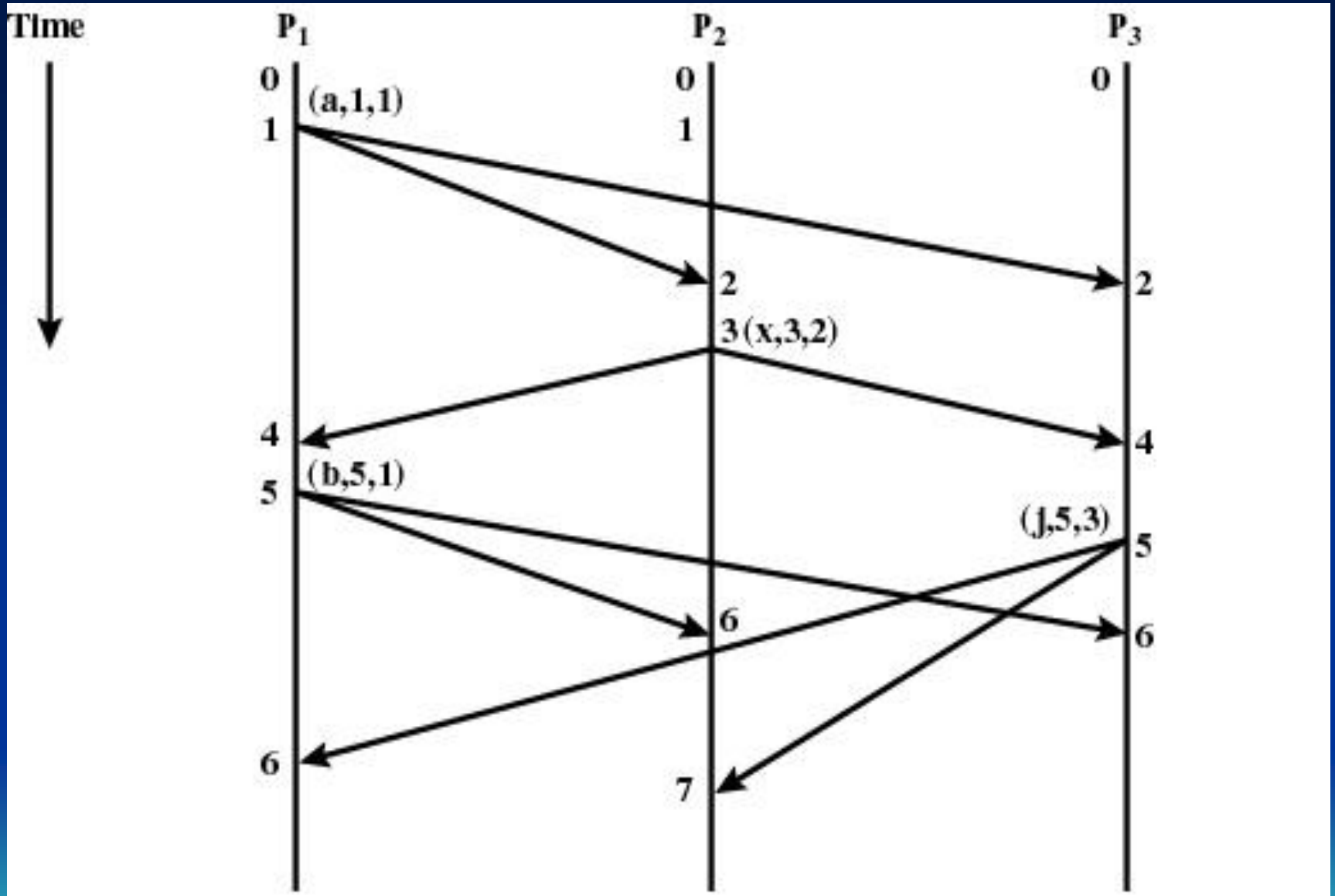


Figure 14.8 Example of Operation of Timestamping Algorithm

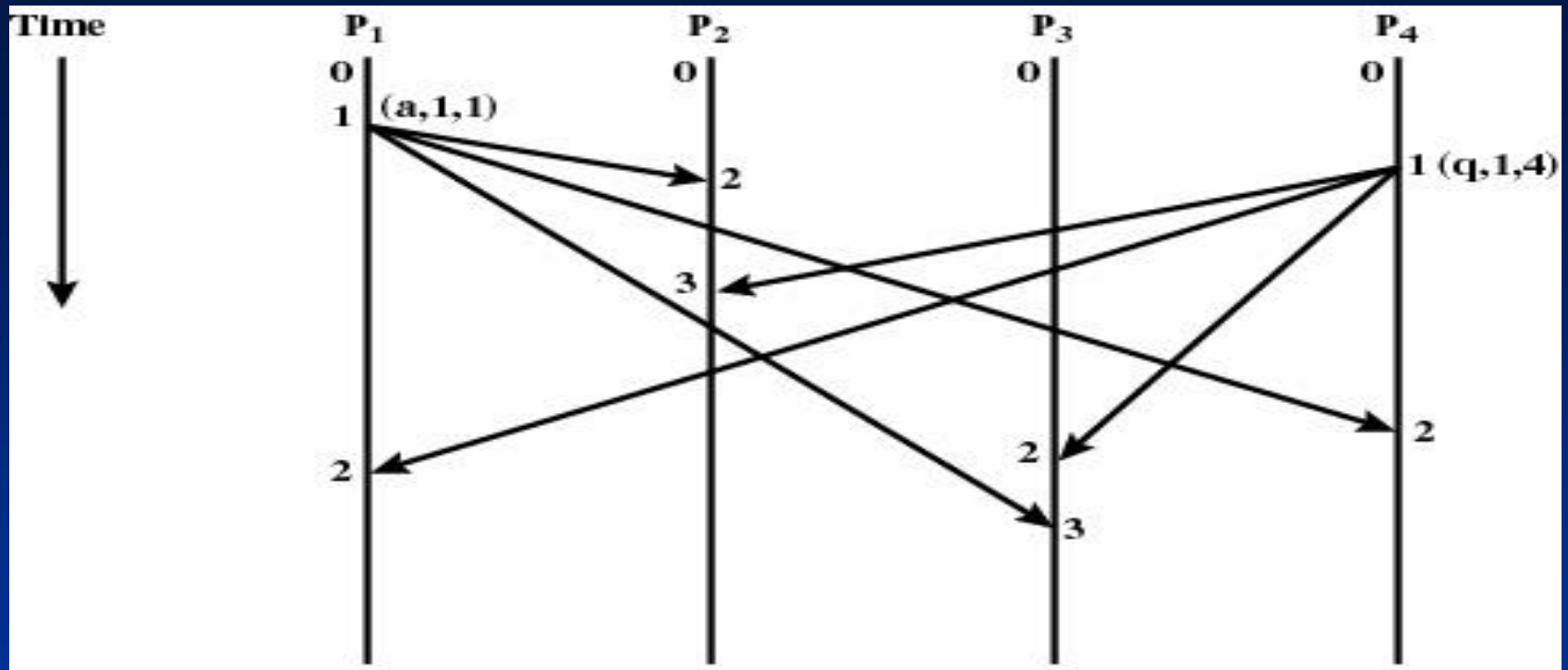


Figure 14.9 Another Example of Operation of Timestamping Algorithm

算法不考虑系统之间传输时间上的差别

哪个事件在实际上首先发生并不重要，重要的是所有进程对事件的发生顺序意见一致

选举算法

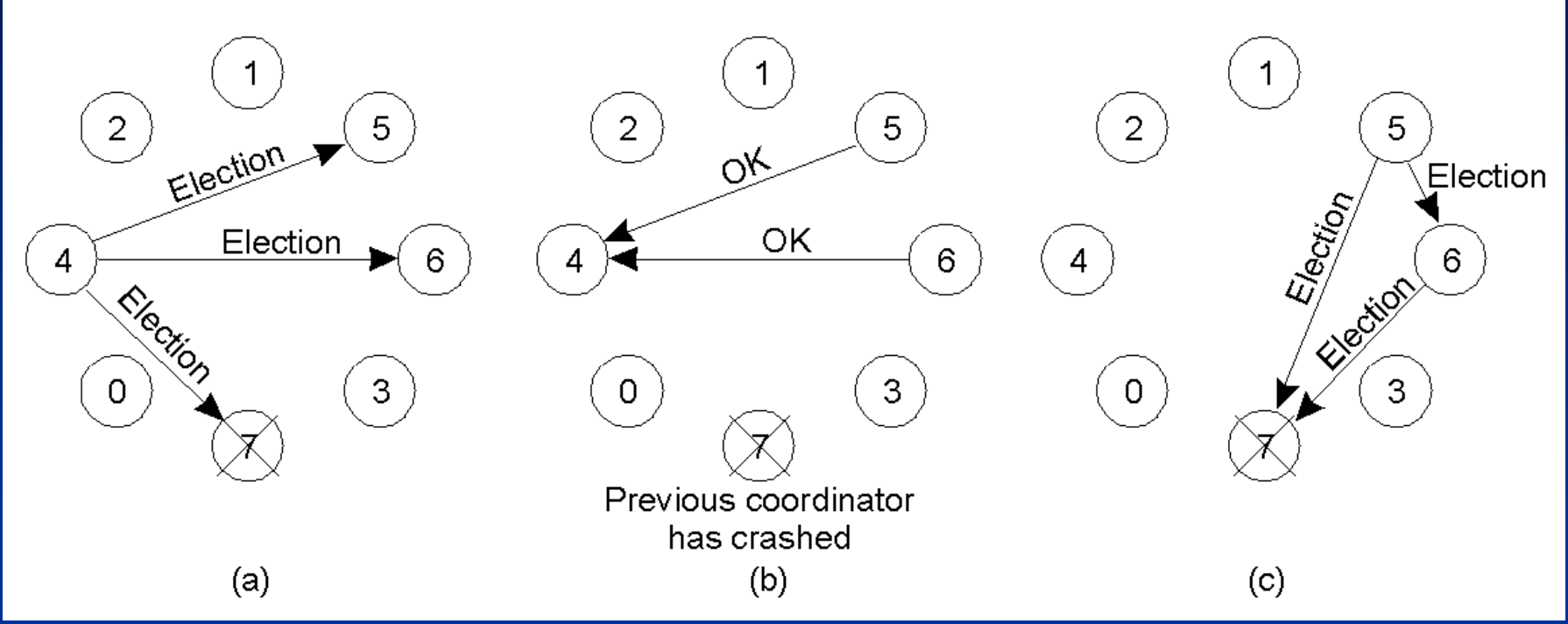
- 选择一个进程作为协调者、发起者或其他特殊角色，一般选择进程号最大的进程（假设每个进程都知道其他进程的进程号，但不知道是否还在运行）
- 目的：保证在选举之行后，所有进程都认可被选举的进程
- 相关算法：
  - Bully算法
  - 环算法
  - 无线环境下的选举算法
  - 大型系统中的选举算法

Bully算法

当进程P注意到需要选举一个进程作协调者时：

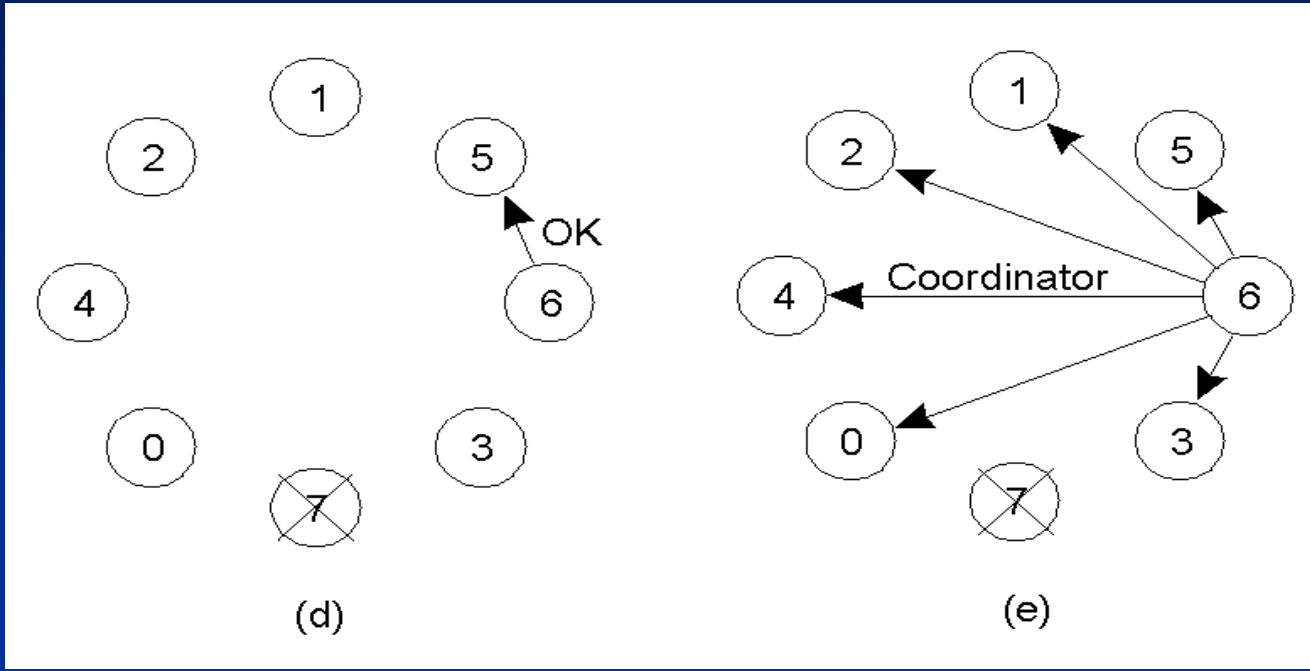
- 向所有进程号比它高的进程发ELECTION消息
- 如果得不到任何进程的响应，进程P获胜，成为协调者
- 如果有进程号比它高的进程响应，该进程接管选举过程，进程P任务完成
- 当其他进程都放弃，只剩一个进程时，该进程成为协调者
- 一个以前被中止的进程恢复后也有选举权

Bully算法



- 进程 4 启动选举
- 进程 5 和进程 6 响应，接管选举，成为协调者

Bully算法

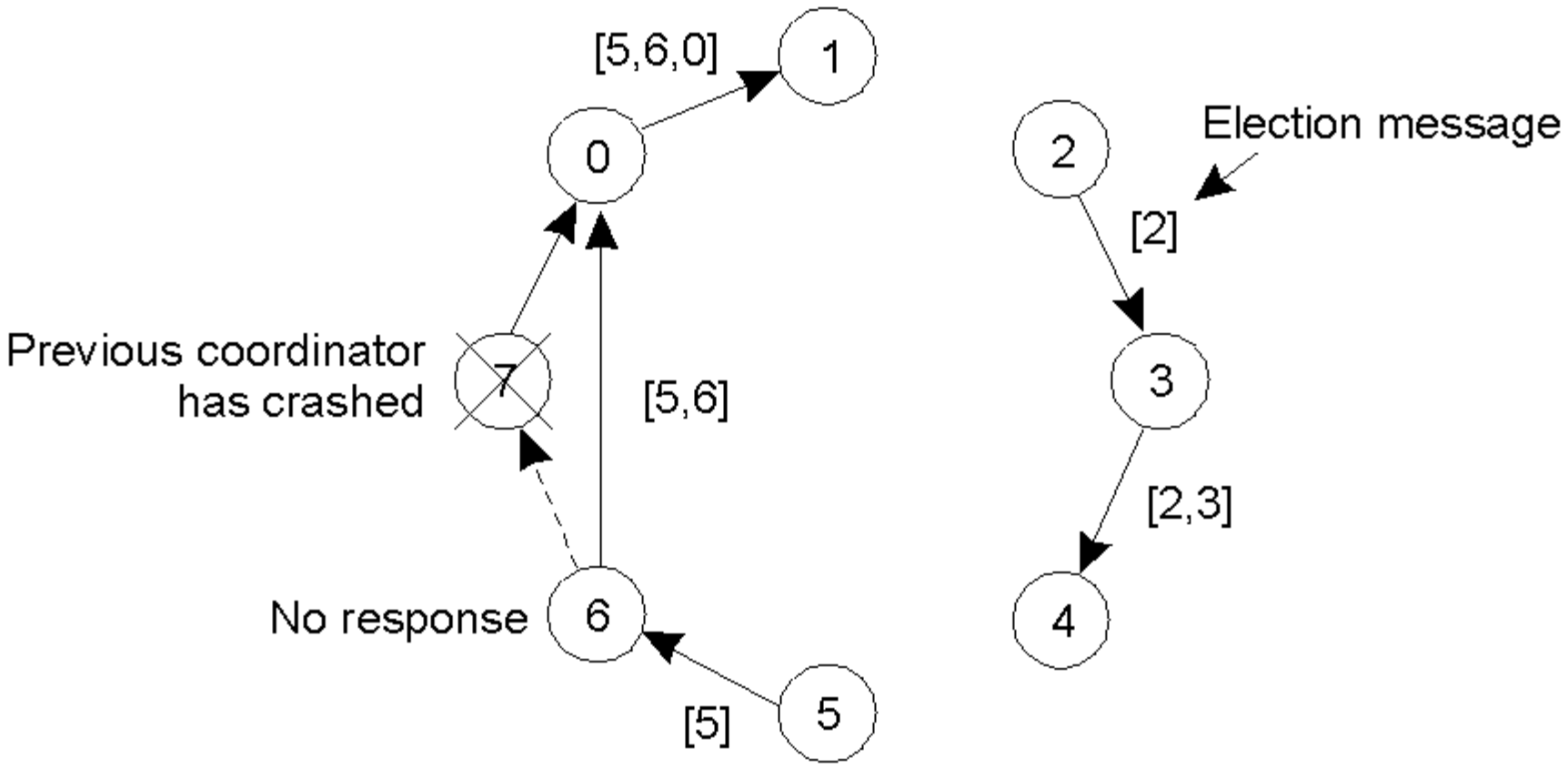


- 进程 6 响应进程 5 的消息，接管选举，进程 6 成为协调者，通知所有进程

环算法

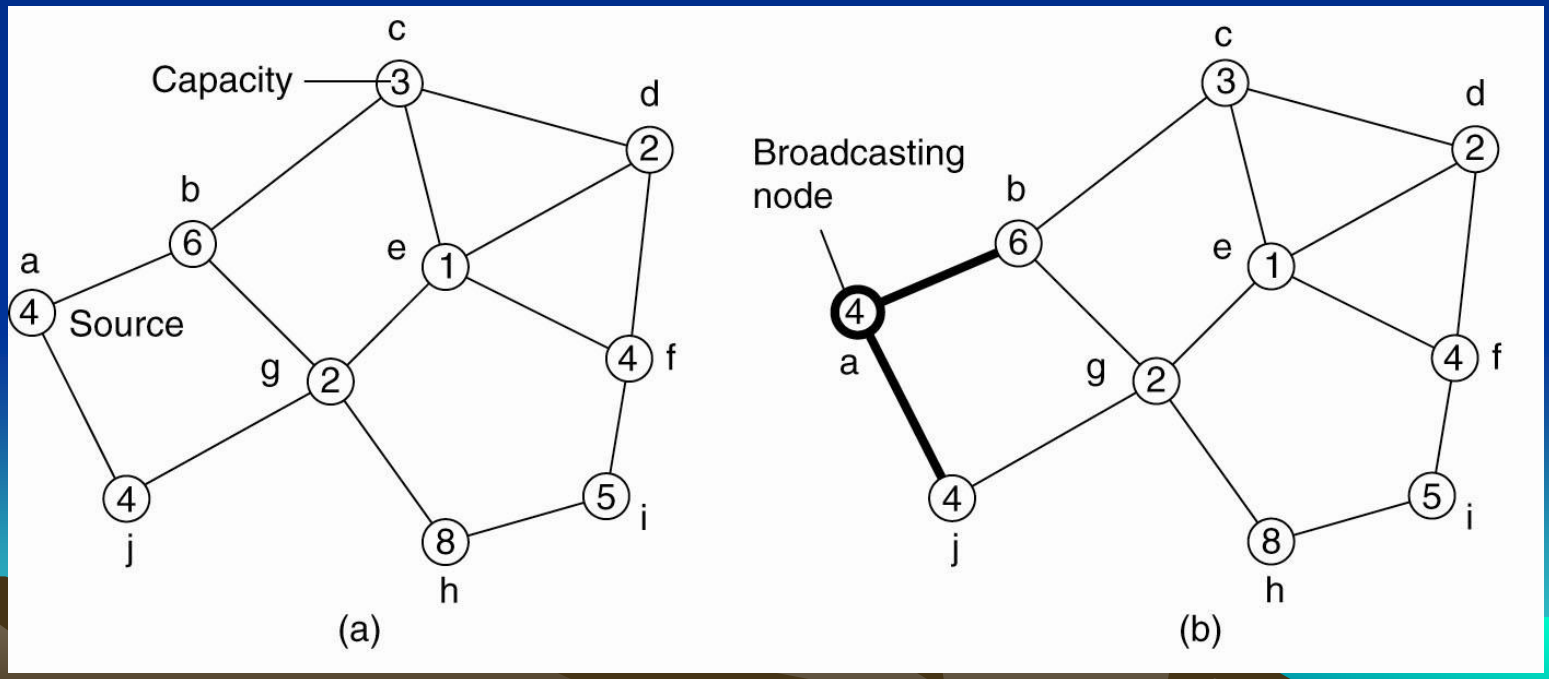
- 不使用令牌
- 按进程号排序，每个进程都知道自己的后继者
- 当进程P注意到需要选举一个进程作协调者时：
  - 就创建一条包含该进程号的ELECTION消息，发给后继进程
  - 后继进程再将自己的进程号加入ELECTION消息，依次类推
  - 最后回到进程P，它再发送一条COORDINATOR消息到环上，包含新选出的协调者进程（进程号最大者）和所有在线进程

环算法

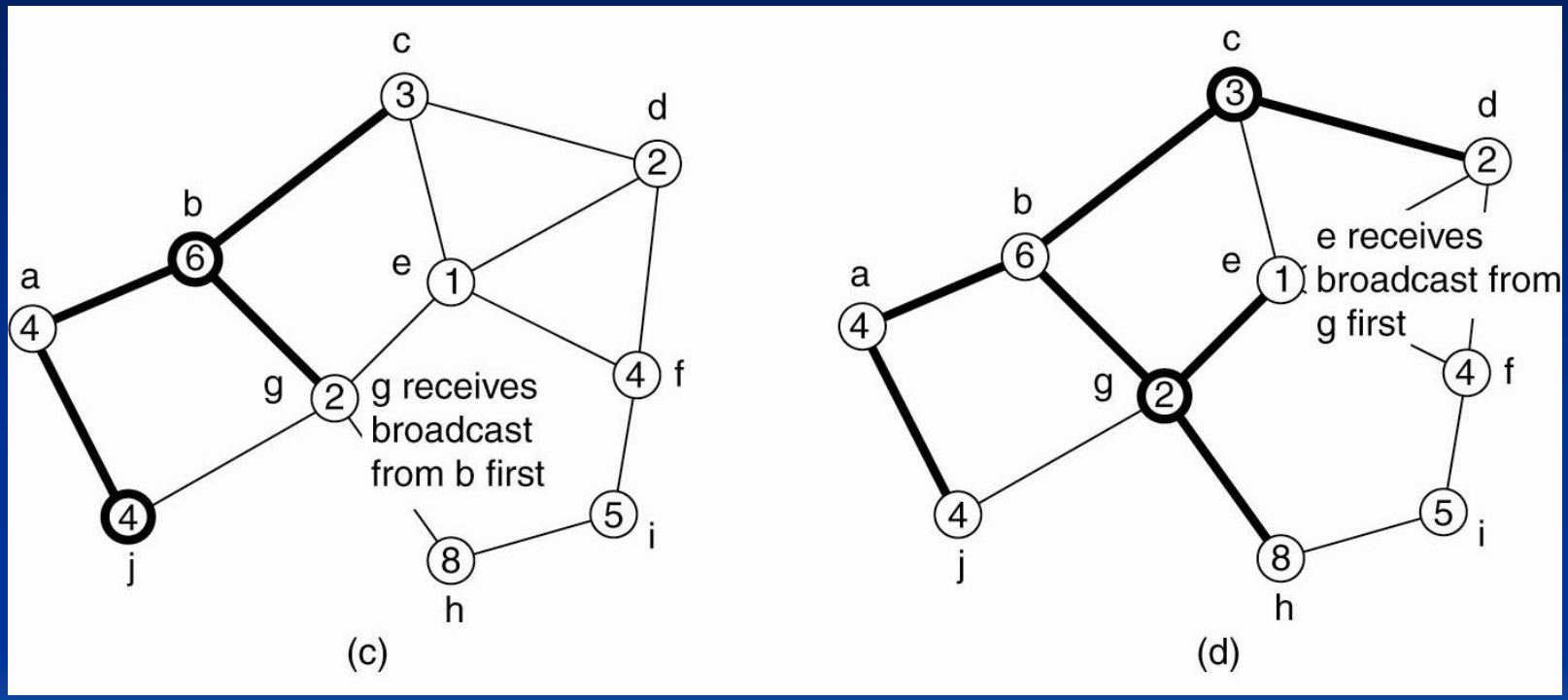

[5,6,0]
1
0
Election message
2
[2]
Previous coordinator
has crashed
7
[5,6]
3
[2,3]
No response
6
4
[5]
5

无线环境下的选举算法

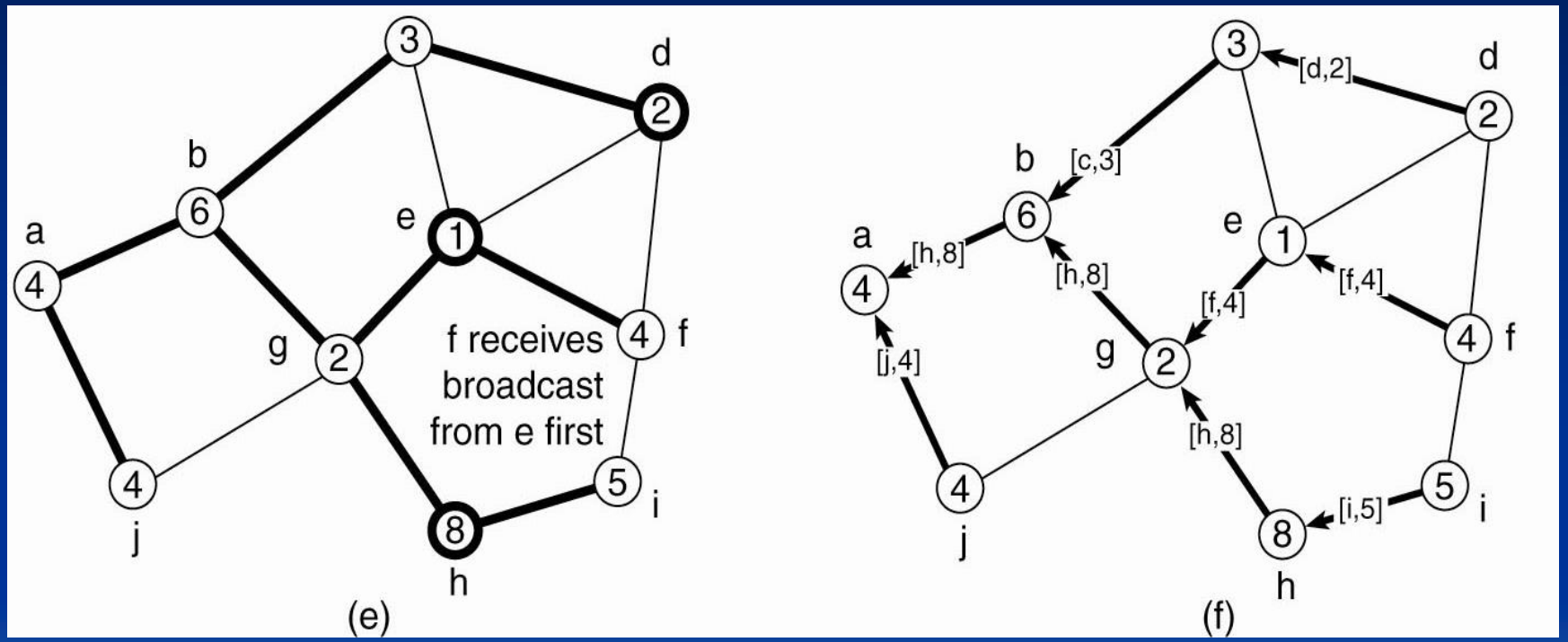
- 不能保证其消息传送是可靠的以及网络拓扑结构不改变
- 源节点向其相邻节点发送ELECTION消息开始一个选举
- 当节点第一次收到ELECTION消息时，会将发送者作为其父节点，然后将ELECTION消息发给其相邻节点（父节点除外）。等到其他节点的确认消息都收到后，再向父节点确认（消息中包含资源容量）。
- 而从某个节点再次收到ELECTION消息时，只是确认。



无线环境下的选举算法


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

无线环境下的选举算法

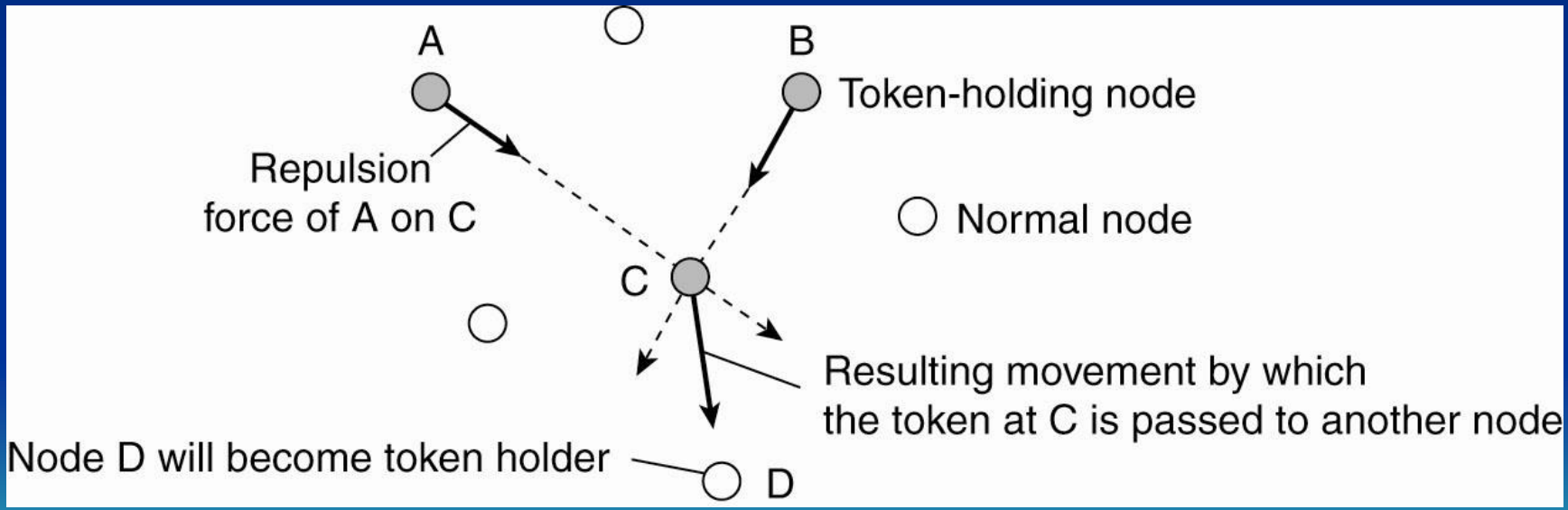

3
d
2
b
6
e
1
a
4
g
2
f receives
broadcast
from e first
4
f
4
5
i
j
8
h
(e)
3
[d,2]
d
2
b
[c,3]
6
e
1
a
[h,8]
[h,8]
[f,4]
[f,4]
4
f
4
[j,4]
g
2
[h,8]
4
5
[i,5]
i
j
8
h
(f)

大型系统中的选举算法

- 可能需要选举多个节点（如超级节点）
- 要求：
  - 一般节点访问超级节点的延时要低
  - 超级节点平均地分布在覆盖网络上
  - 相对于覆盖网络中的所有节点，应有一部分预定义的超级节点
  - 每个超级节点不应为超过固定数目的一般节点服务

使用推动力在二维空间移动令牌

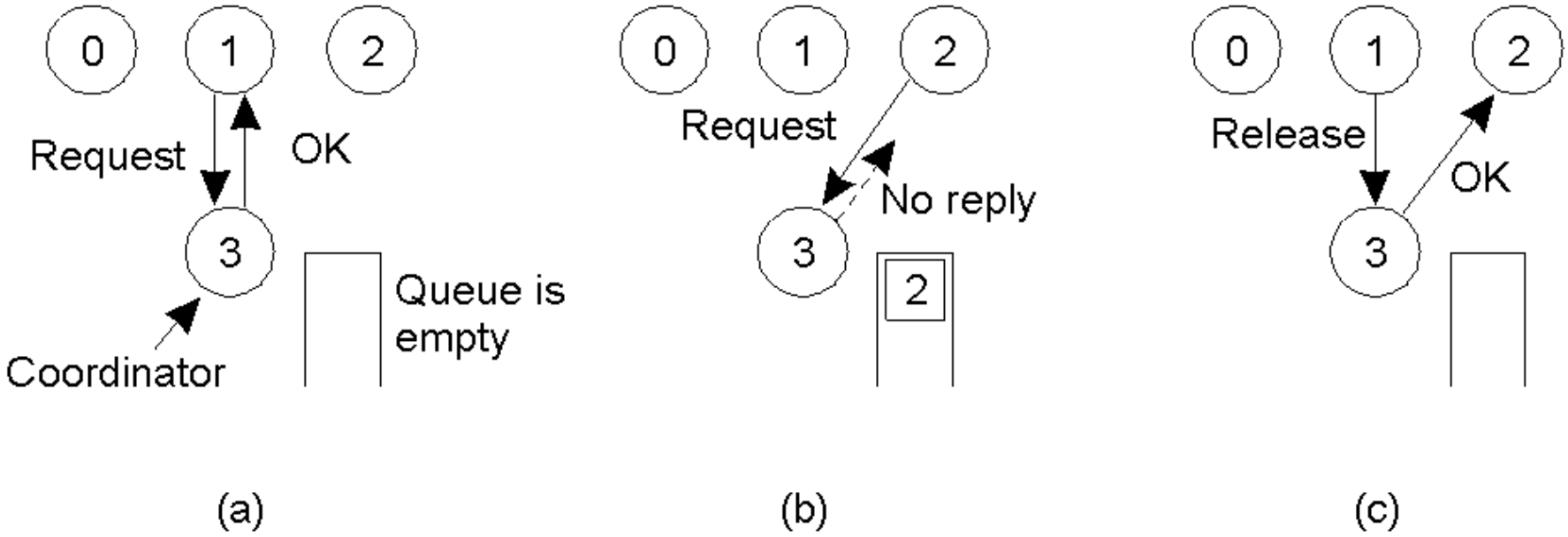
- 如果某个节点发现总的推力超过某个阈值，就会将令牌向其合力方向移动
- 当令牌被一个节点拥有超过定长时间后，该节点就会把自己提升为超级节点



互斥

- 基于令牌的解决方案
  - 拥有令牌者获得使用资源的权限
  - 可以避免饿死和死锁
  - 令牌可能会丢失
- 基于许可的解决方案
  - 获得其它进程的许可来使用资源

集中式算法



使用协调者

- 进程1请求协调者允许它进入临界区，得到同意
- 进程2也请求协调者允许它进入临界区，协调者不应答
- 当进程1退出临界区时，协调者对进程2作出应答

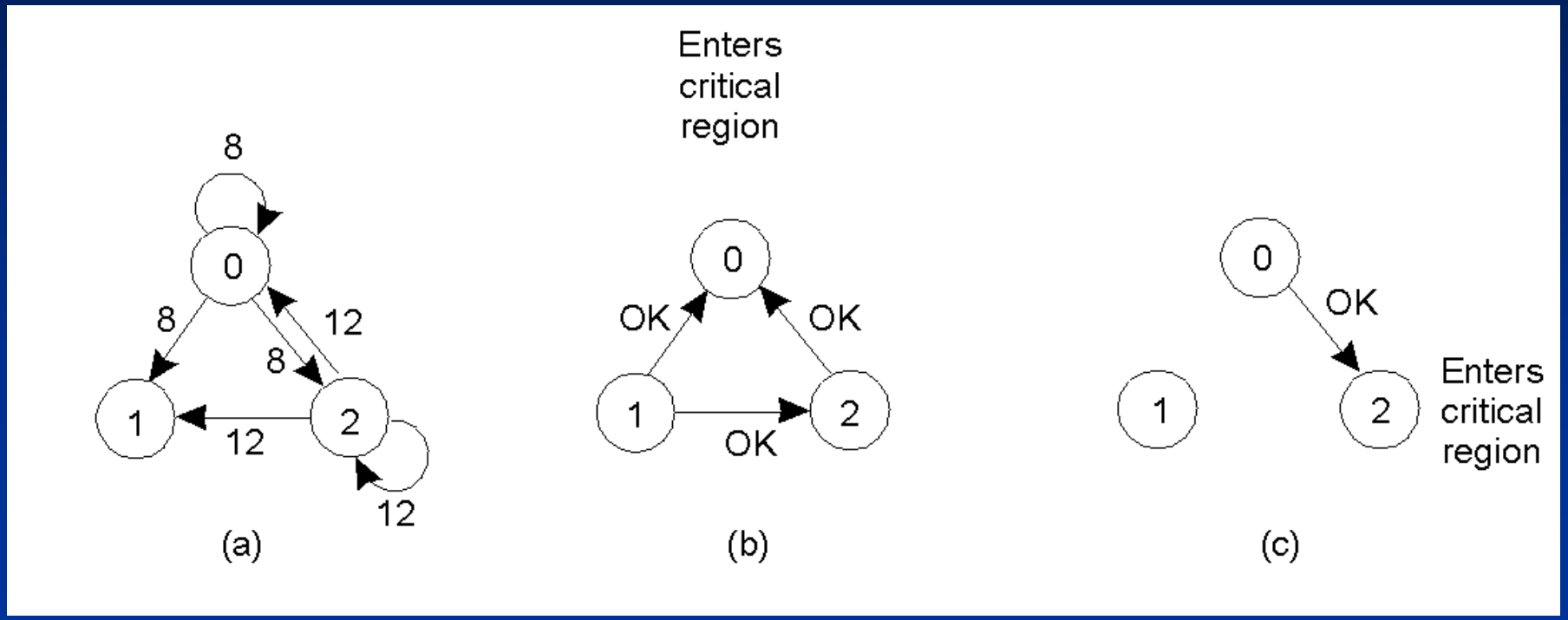
集中式算法

- 保证互斥的实现
- 公平
- 协调者
  - 瓶颈
  - 单点崩溃

互 斥：分布式算法

- 当进程想进入临界区时，它向所有其他进程发一条打了时间戳的消息Request
- 当收到所有其他进程的Reply消息时，就可以进入临界区了
- 当一个进程收到一条Request消息时，必须返回一条Reply消息：
  - 如该进程自己不想进入临界区，则立即发送Reply消息
  - 如该进程想进入临界区，则把自己的Request消息时间戳与收到的Request消息时间戳相比较，
    - 如自己的晚，则立即发送Reply消息
    - 否则，就推迟发送Reply消息

分布式互斥算法



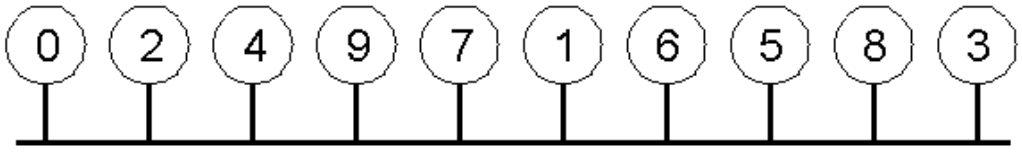
a) 进程 0 和 2 都想进入临界区
b) 进程 0 的时间戳低，抢先进入临界区
c) 进程 0 退出临界区后，发应答给进程2，进程2 随后进入临界区

- N个故障点
  - 修正：请求到达时，无论请求还是拒绝都发送应答
- 如使用多播通信，需维护成员列表
- 要求所有进程都参与决定共享资源的访问许可
  - 修正：获得大多数进程的许可即可

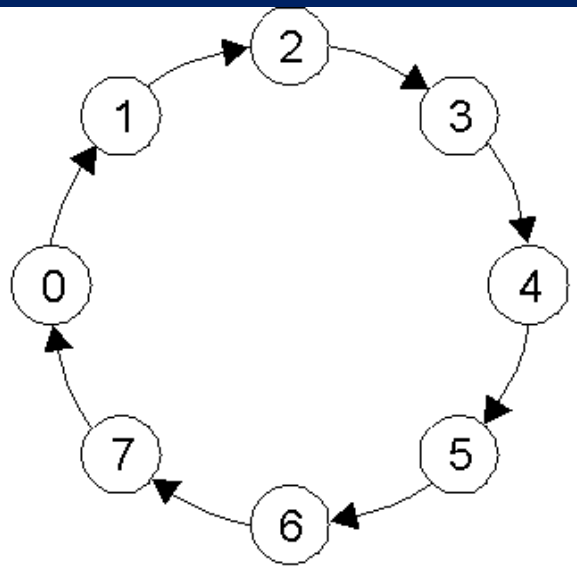
令牌环算法

- 构造进程逻辑环
- 令牌在环上顺序循环传播
- 获得令牌的当前进程
  - 若想进入临界区，则进入；退出时将令牌向后传；
  - 若不想进入临界区，则直接将令牌向后传；
- 令牌丢失时产生新令牌
- 某进程崩溃时，绕过该进程

令牌环算法



(a)



(b)

三种互斥算法的比较

算法	每次进/出临界区所需消息次数	进入前的延迟(消息次数)	问 题
集中式	3	2	协调者崩溃
分布式	2 (n – 1)	2 (n – 1)	任一进程崩溃
令牌环	1 to ∞	0 to n – 1	丢失令牌，进程崩溃

分布式事务

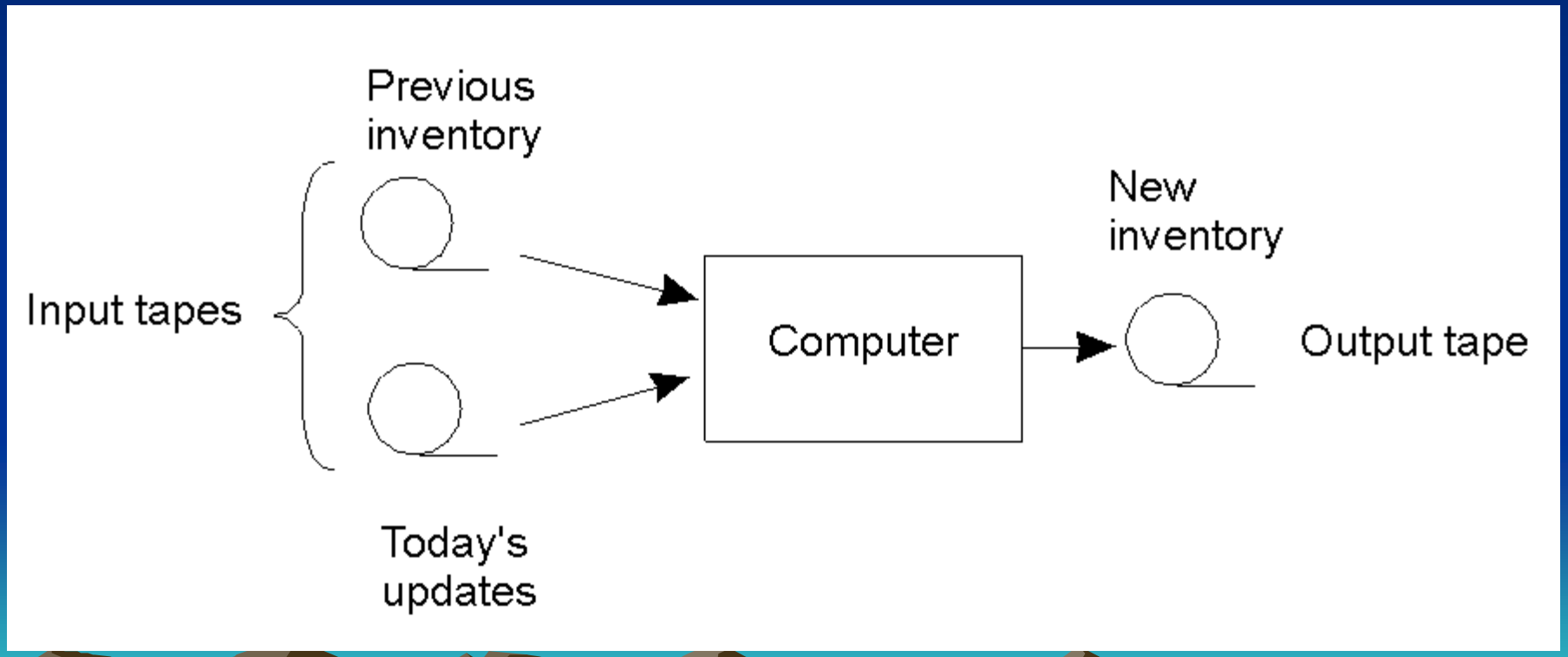
- 事务模型
- 事务分类
- 事务实现
- 并发控制
  - 串行化
  - 悲观的时间戳排序
  - 乐观的并发控制

分布式事务

- 事务模型
- 事务分类
- 实现

事务模型 (1)

- 更新一个主库存磁带是具有容错性的



事务模型 (2)

- 事务原语示例

原语	描述
BEGIN_TRANSACTION	开始事务
END_TRANSACTION	中止事务并尝试提交
ABORT_TRANSACTION	取消事务并恢复原值
READ	从文件、表或其他地方读数据
WRITE	向文件、表或其他地方写数据

事务模型 (3)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

(b)

a) 预定三个航班的事务得以提交
b) 当订不到第三个航班时，事务中止

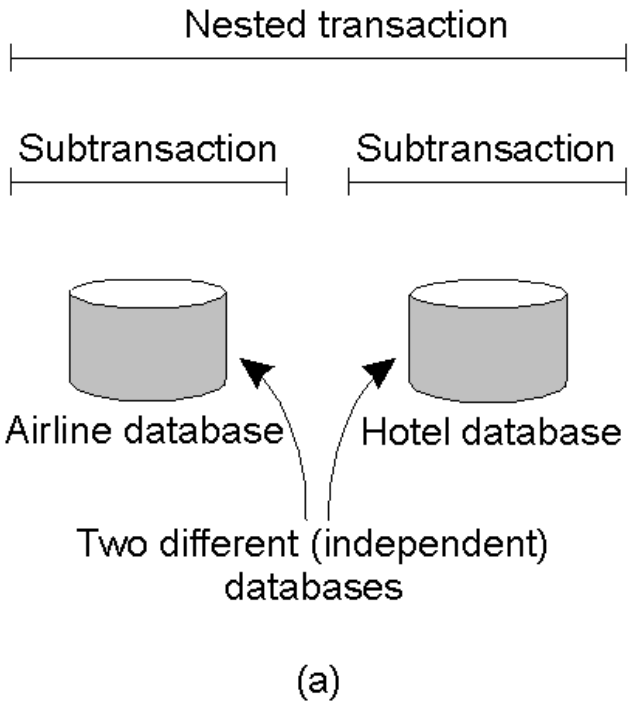
事务属性

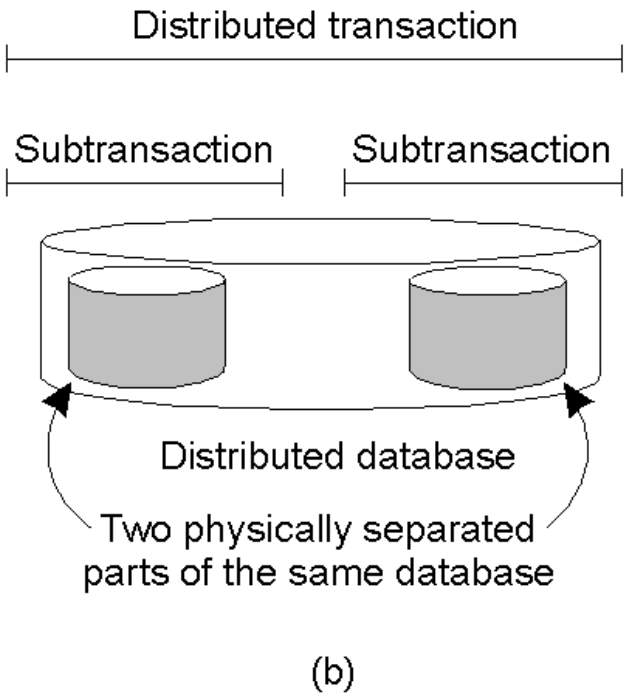
事务属性ACID：

- 原子性(atomic)：事务的执行不可分割
- 一致性(consistent)：事务不能破坏系统的恒定性
- 独立性(isolated，串行性)：并发的事务不会互相干扰
- 持久性(durable)：一旦事务被执行，所作的修改就永远生效

事务分类

- 单层事务（A flat transaction ）：不允许提交或取消部分结果
- 嵌套事务（A nested transaction）：按逻辑关系分成独立的许多子事务（可分布到不同机器上运行），子事务提交结果对父事务和后续子事务是可见的；父事务中止会导致所有子事务的中止
- 分布式事务（A distributed transaction）：单层、不可分割的事务，操作对象是分布式的数据。使用分布式算法锁定数据和提交整个事务

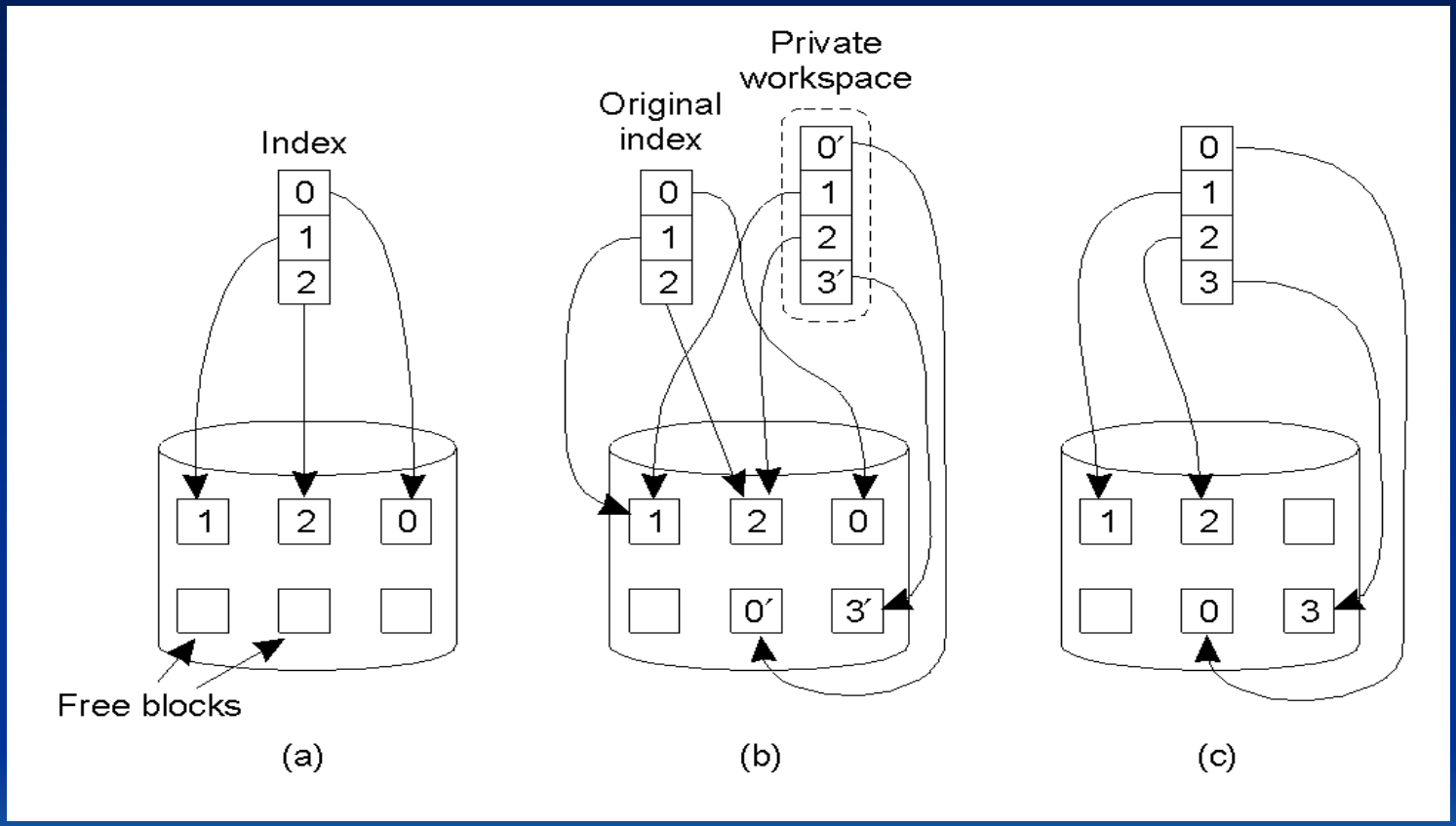




事务实现（一）

- 私有工作空间
  - 为进程提供一个私有工作空间，包含进程要访问的所有对象
  - 进程的读写操作在私有工作空间进行，而不对实际的文件系统进行
  - 开销大，可以进行优化使之可行
    - 读操作不复制
    - 写操作时复制

私有工作空间



a) 包含三个块的文件及其索引
b) 块0被修改，块3被添加后的情况
c) 事务提交之后

私有工作空间

- 如果事务中止，私有工作空间被释放，指向的私有块被删除
- 如果事务提交，私有索引被移到父辈空间，不再被访问的块被释放掉

事务实现（二）

- 写前日志（writeahead log）：先写日志，再做实际修改
- 日志内容：哪个事务在对文件进行修改，哪个文件和数据被改动，新值和旧值是什么…
- 日志写入后，改动才被写入文件
- 事务中止，使用写前日志回退到原来的状态
- 借助稳定存储器中的写前日志：当系统崩溃后，完成事务或取消事务

写前日志

x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0/1]	[x = 0/1]	[x = 0/1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

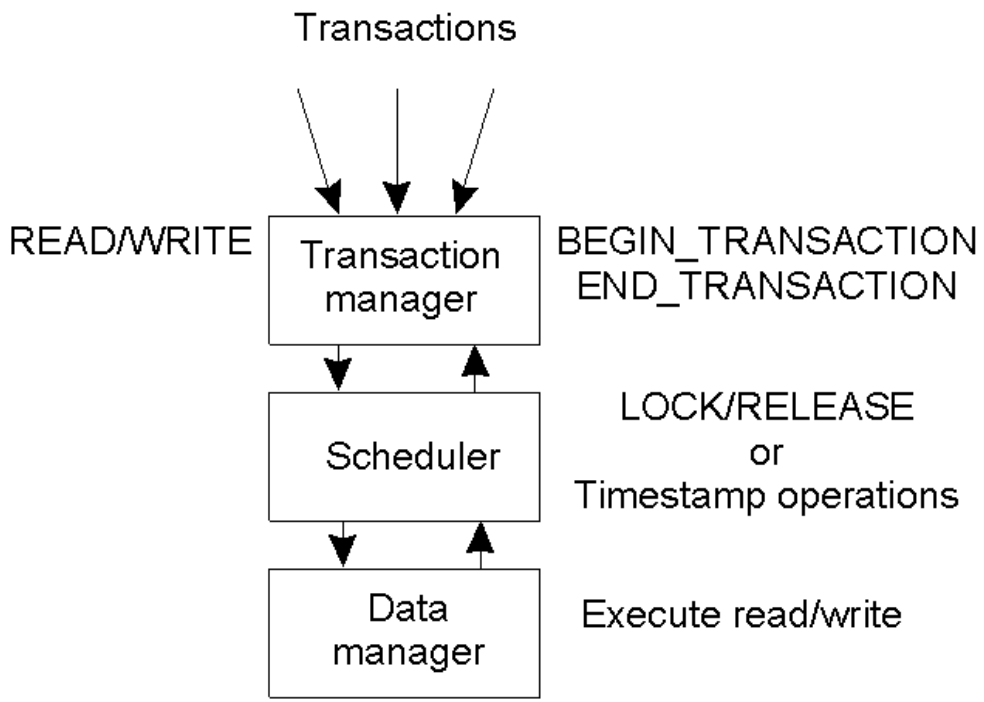
a) 一个事务

b) – d) 语句执行前的日志

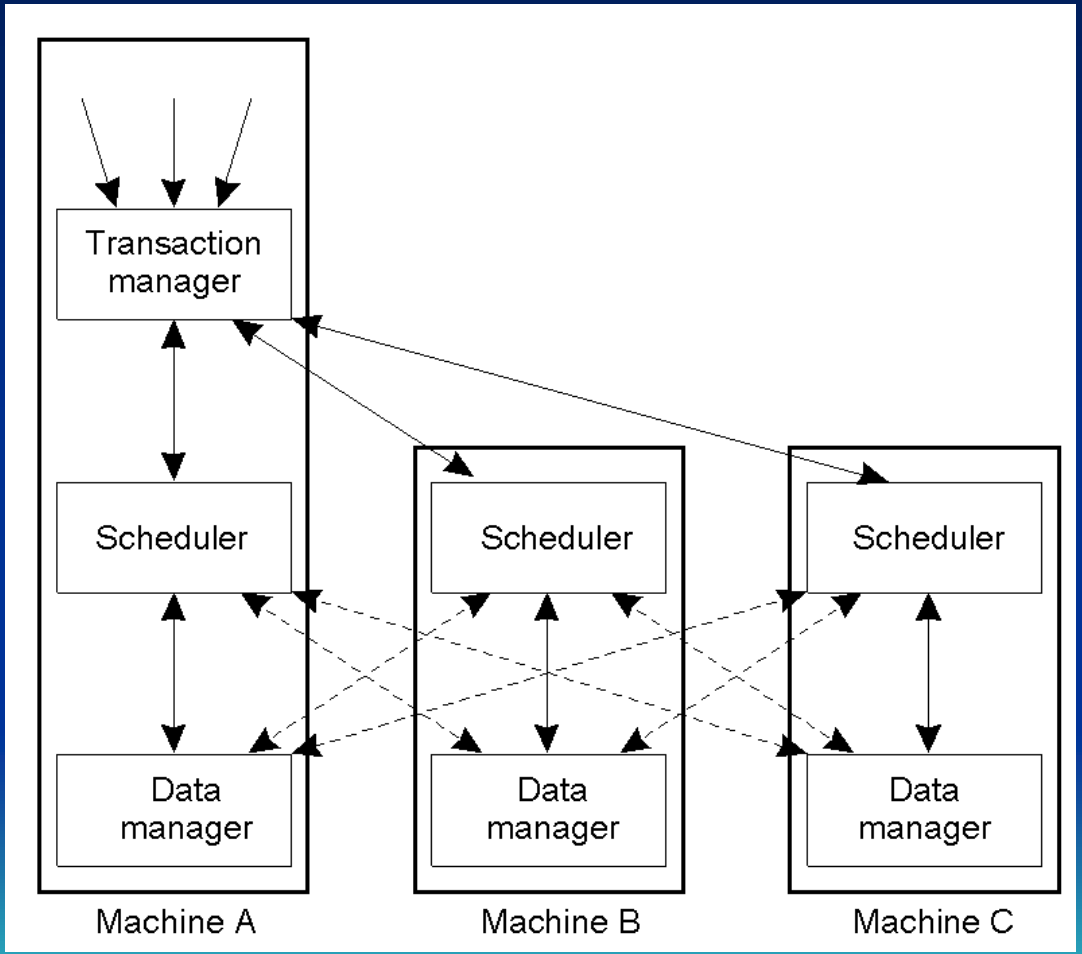
并发控制 (1)

- 通过正确地控制并发事务（同时对共享数据进行操作的事务）的执行基本上可以解决一致性和独立性问题
- 事务管理器：保证事务的原子性
- 调度管理器：正确地控制并发，决定哪个事务在何时被允许将读、写操作传给数据管理器
- 数据管理器：数据的读写

处理事务的管理器组织=>



并发控制 (2)



分布式事务管理器组织

串行化

串行化：多个事务同时执行并保持独立，最终的执行结果与事务以某种特定顺序一个接一个串行执行得到的结果相同。

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	合法
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	合法
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	非法

(d)

a) – c) 三个事务 T_1, T_2, 和 T_3

d) 可能的调度

并发控制算法

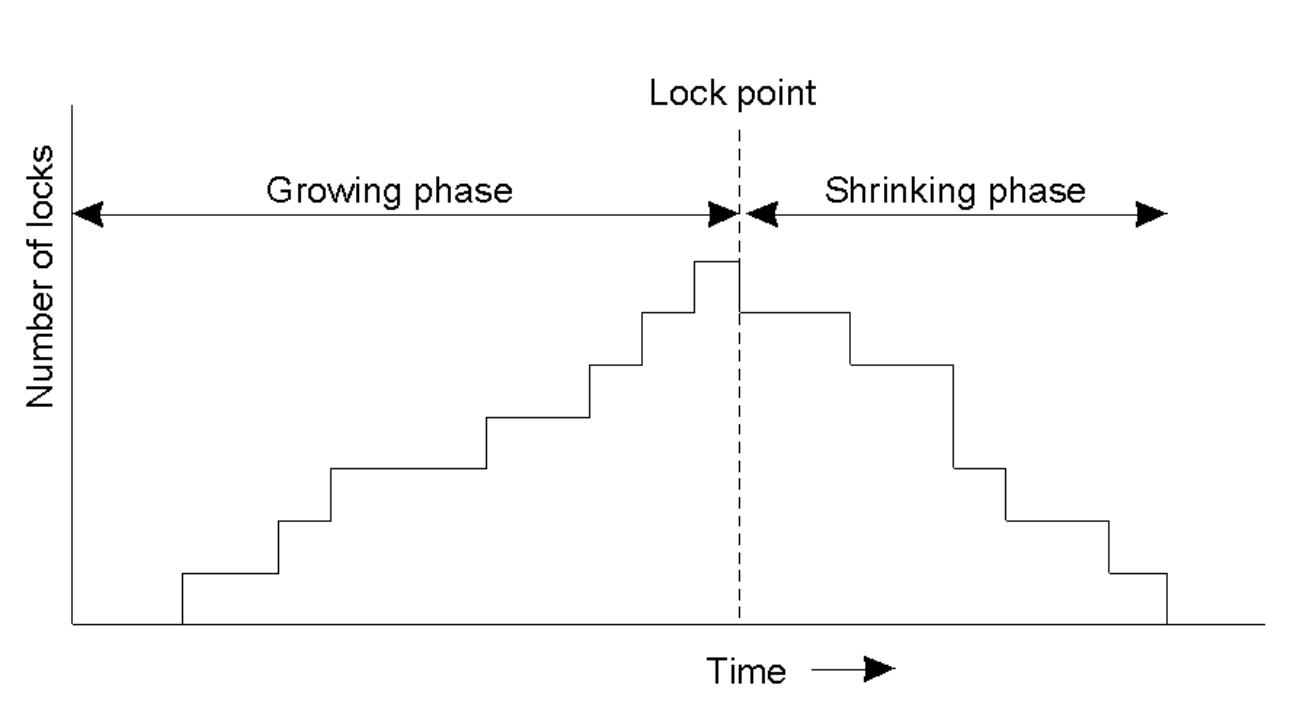
- 并发控制的思想：正确调度相冲突的操作（读写和写写）
- 按读写操作同步的方式分为
  - 共享数据上的互斥机制，如锁定
  - 显示地使用时间戳排序

锁机制

- T1读D的时候，T2不可以写D，但可以读D；
- T1写D的时候，T2既不可以写D，又不可以读D。
- 一个事务可以对D实施读锁和写锁：
- 不同的事务可以因读而同时封锁同一个D，所以，读锁又叫共享锁（Sharing Lock）；
- 不同的事务不能因写而同时封锁同一个D，所以，写锁又叫排它锁（Exclusive Lock）；
- 根据上述分析，我们可得如下锁协议：
  - 1、T ReadLock(D)，T′ 也可以ReadLock(D)，如果T′ WriteLock(D)，则T′ 被挂起，直到 T UNLock（D）；
  - 2、T WriteLock(D)，如果T′ ReadLock(D)或者WriteLock(D)，则T′ 被挂起，直到 T UNLock（D）；
  - 3、T使用完D之后，UNLock(D)。
- 问题：不能保证可串行化。

两阶段锁定

- 两阶段锁定：为了保证并发调度的可串行化，要求T在执行过程中，有一个时间点t，在t之前，T不执行UNLock操作，并且在t之后，T不再执行ReadLock(D)和WriteLock(D)操作。
- 进程在增长阶段先请求它需要的所有锁，然后在收缩阶段释放它们。
- 可以证明（Eswaran等，1976）如果所有的事务都使用两阶段加锁法，那么通过交错事务进行的所有调度都是串行的。



悲观的时间戳排序
Pessimistic Timestamp Ordering

- 思想：
  - 每个事务指定一个时间戳，文件都有相关的读时间戳和写时间戳
  - 如果事务的进程试图访问文件时，文件的读时间戳和写时间戳都比事务的时间戳更早（小），这种关系是正常的
  - 反之，说明当前事务提交太晚，应该终止。

乐观的并发控制
optimistic concurrency

- 处理同时运行多个事务的另一种方法是乐观并发控制法（Kung and Robinson，1981）。
- 这种方法的思想惊人的简单：
  - 尽管放心去做你想做的，不用在意其他人正在做什么。如果有问题出现，那么以后再考虑吧。
  - 在实际情况中，冲突相对来说非常少，所以这个策略大部分时间都可以正常工作。

乐观的并发控制—冲突的处理

- 尽管冲突会非常少，但存在的可能性还是有的，因此还需要一些处理冲突的方法。
- 乐观并发控制算法所做的只是
  - 记录下有哪些文件曾经被读写过。
  - 在提交时刻，检测其他的事务以判断在本事务开始后它的文件是否被其他事务修改过。
    - 如果被修改过，那么本事务将被中止。
    - 如果没有修改过，那么本事务就可以提交了。

- 乐观并发控制算法最适合于基于私有工作空间的情况。
  - 每个事务都独立地修改各自的文件，不会涉及其他的事务。
  - 在结束的时候，新的文件要么被提交要么被释放。
- 乐观并发控制算法的最大优点在于
  - 避免了死锁，而且允许最大的并行度（进程不需要去等待一个锁）
- 它的缺点是：
  - 有时可能会失效，这时所有事务都必须退回重新运行
  - 在重负载的情况下，算法失效的可能性将会直线上升，这使得乐观并发控制算法成了一个很糟糕的选择。

分布式系统中的死锁

- 分布式系统中的死锁类似单处理机系统中的死锁，只是情况更糟。
  - 它们更难于避免、预防或者检测，即使在检测到以后也很难处理，因为所有的相关信息都分散在多台机器上。

策略的分类

讨论死锁问题的策略有很多种。四个最著名的策略：

1. 鸵鸟算法（忽略问题）
2. 预防（静态的，使死锁在机制上是不可能发生的）
3. 避免（通过仔细的分配资源以避免死锁，需要（事先）知道每个进程最终到底需要多少资源。而这样的信息即使有，也非常的少。）
4. 检测与恢复（允许死锁发生，在检测到后想办法恢复）

分布式死锁预防

- 死锁预防是由细致的系统设计构成的，因此死锁从机制上来说是不可能的。
- 一些已有的办法在实践中都不太方便，例如：
  - 在某一时刻只允许进程占有一个资源
  - 要求进程在初始阶段请求所有的资源
  - 当进程请求新资源时必须释放所有资源。
  - 或者要求进程必须预定资源，并以严格增序请求资源。
    - 即一个进程不可能既占有了一个高序资源又去请求一个低序资源，这就使得环路不可能出现了。

两种基于时间戳的算法

- 在拥有全局时间和原子事务的分布式系统中，另外两种实用的算法也是可能的。
  - 这两种算法都是基于在一个事务开始时给它分配一个全局时间戳的思想。
  - 同许多基于时间戳的算法一样，在这两种算法中保证不会有两个事务分配了完全一致的时间戳。
  - **Lamport**的算法有效的保证了时间戳是唯一的。

基本思想

- 这两种算法的基本思想是：

  当一个进程因等待一个正被其他进程占用的资源而要阻塞时，进行检查以判断哪个进程的时间戳更大（即更晚）。

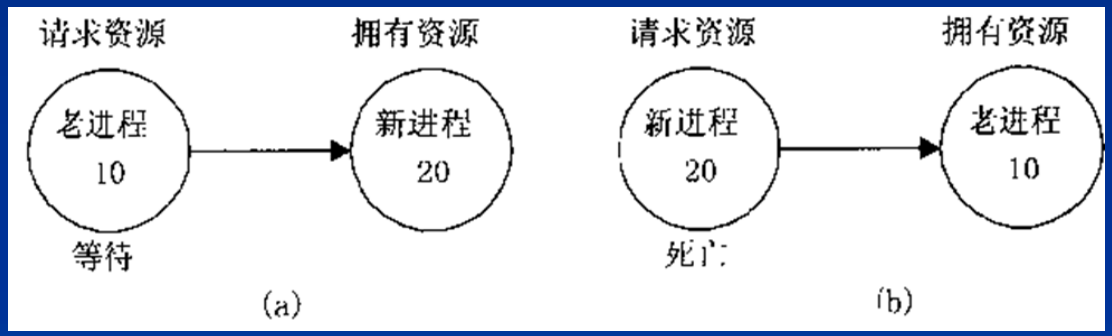
  - 只有当等待进程的时间戳小于（早于）被等待进程的时间戳，才允许等待发生（只允许老进程等待），否则中止
    - 沿着等待进程链，时间戳递增，不可能发生环路
  - 或只有当等待进程拥有大于（晚于）被等待进程的时间戳时，才允许等待发生（只允许新进程等待），否则中止
    - 沿着等待进程链，时间戳递减

老进程？新进程？

- 尽管两种方法都能预防死锁，但是给予老的进程以优先权更明智些。？？？
  - 它们已经运行了较长时间，系统对它们的投入会更大一些，它们占有的资源也就更多一些。
  - 另外，这种选择消除了饿死现象
    - 一个被中止的新进程最终成为系统中最老的进程，从而得到等待的机会，进而获得资源。

等-死算法（wait-die）

- 由于使用了时间戳，当请求被占用的资源时，只可能有两种情况：
  - 老进程请求被新进程占用的资源，
  - 或者，新进程请求被老进程占用的资源



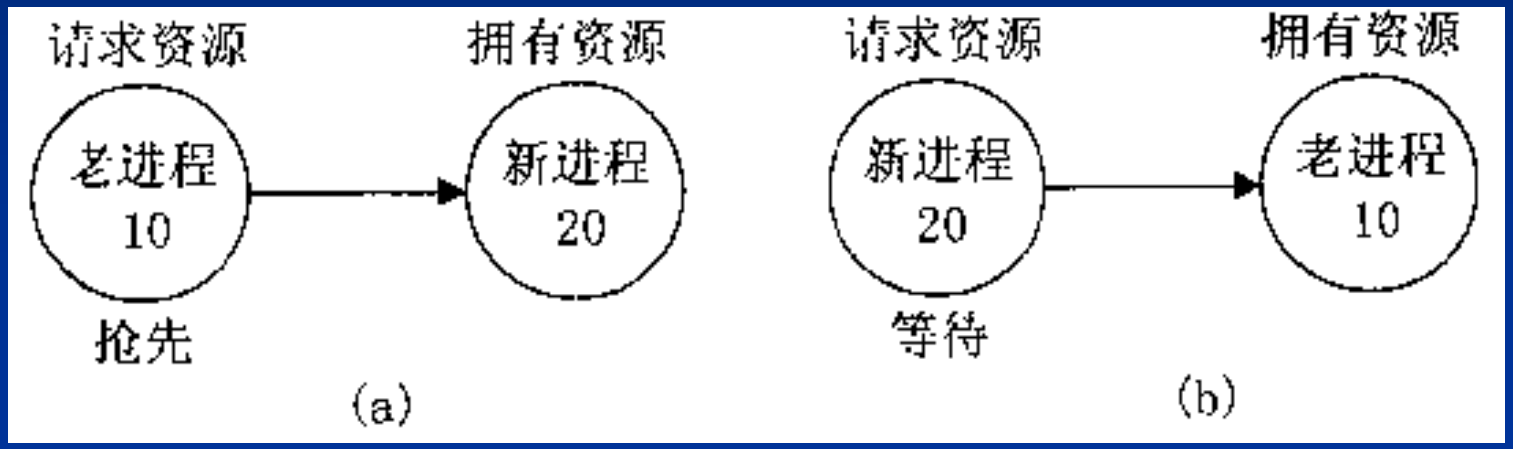
- 一种情况应该允许进程等待，另一种情况应该中止进程。
  - 也即等-死算法

资源不可剥夺？

- 一旦我们假设了事务的存在，我们就可以做一些在以前是被禁止的事情：从运行进程中夺走资源。
- 当冲突发生的时候，我们不需要中止提出请求的进程，我们可以中止资源拥有者。
- 如果没有事务，中止一个进程可能会有严重的后果，例如进程可能已经修改了文件。有了事务后，当事务取消时，这些效果会消失。

伤-等算法（wound-wait）

- 伤-等算法允许抢占：假设只允许老进程抢占新进程，图a被标记为抢先，图b为等待。



- 这种算法称为伤-等算法（wound-wait），因为一个事务可能会受到伤害（被中止运行，变成等待）。

等-死算法与伤-等算法的比较

- 等-死算法中，
  - 若一个老事务想得到一个正被新事务占用的资源，那么它会很礼貌的等待。
  - 反之，若一个新事务想得到一个被老事务占用的资源，它将被中止。尽管它还会重新开始，但很可能又会立即被中止。在老事务释放资源之前，这个循环可能要重复多次。
- 伤-等算法没有这么糟糕的特性。

分布式死锁检测

- 在分布式系统中找出一般的死锁预防和避免的解决方法是相当困难的，因此许多研究人员都只是尝试为更简单的死锁检测问题找出一种解决方法，而不是想办法去禁止死锁的发生。

- 在一些分布式系统中原子事务的提出使得在概念上有了极大的不同。
  - 在普通的操作系统中检测到死锁后，解决方法是中止掉一个或几个进程，但这必然会使一些用户感到不满。
  - 在基于原子事务的系统中检测到死锁后，解决方法是中止掉一个或几个事务。但事务允许出现中止。
  - 当一个事务因为产生死锁而被中止的时候，
    - 首先让系统恢复到事务开始前的状态，以后事务可以从这一点重新开始。若运气好，事务在第二次执行时就应该能成功。
- 使用事务与不使用事务的差别在于：
  - 使用事务时中止一个进程的后果要比不使用事务时的后果小的多得多。

集中式的死锁检测

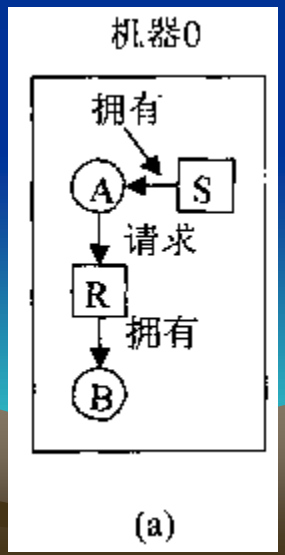
- 作为第一个尝试，我们使用集中式的死锁检测算法来尽量模仿非分布式的算法。
- 集中式的死锁检测算法
  - 每台机器都有一幅资源图以描述自己所拥有的进程和资源
  - 有一台中心机器拥有整个系统（所有资源图的集合）的资源图。
  - 当协调者检测到了环路时它就中止一个进程以解决死锁。

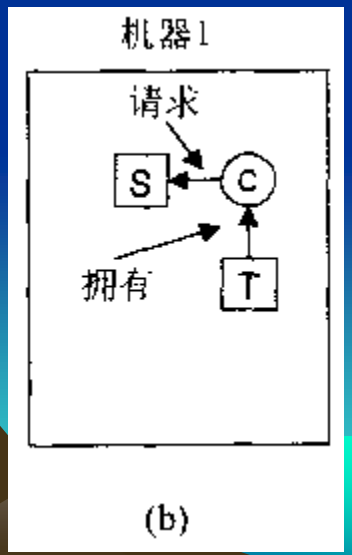
全局资源图信息的维护

- 在分布式系统中需要精确维护全局资源图。
  - 每台机器的资源图中只包含它自己的进程和资源。
  - 需要适当的方法维护全局资源图信息。
- 方法1：每当资源图中加入或删除一条弧时，相应的消息就发送给协调者以提供更新。
- 方法2：每个进程周期性的把从上次更新后新添加的和删除的弧的列表发送给协调者。
  - 这种方法比第一种方法发送的消息要少。
- 方法3：协调者在需要的时候主动去请求信息。

反例

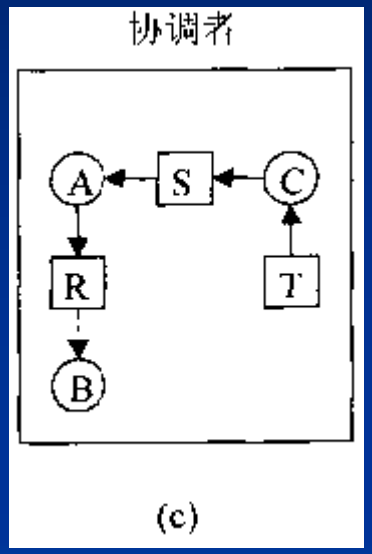
- 不幸的是上述方法的效果都不太好。例如有这样一种系统
  - A和B运行在机器0上，C运行在机器1上。
  - 共有三种资源S，R和T。
  - 如图，一开始A拥有S并想请求R，但B正在使用R；C拥有T并想请求S。





反例（cont'd）

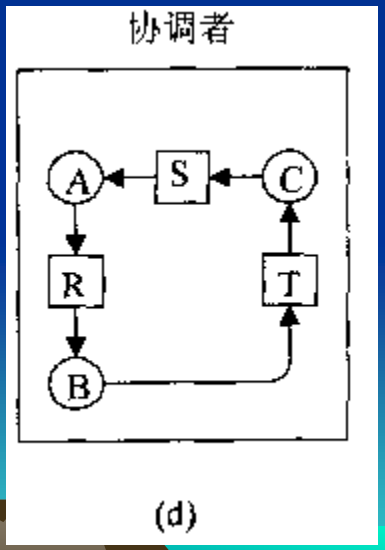
- 协调者看到的情况如图c所示。



- 这种配置是安全的。一旦B结束运行，A就可以得到R然后结束，并释放C所等待的S。

反例（cont'd）

- 过一会儿，B释放R并请求T，这是一个完全合法的安全操作。
  - 机器0向协调者发送一条消息声明它释放R
  - 机器1向协调者发送了一条消息声明进程B正在等待它的资源T。
- 不幸的是，机器1的消息首先到达，这导致协调者生成了一幅如图d所示的资源图。



假死锁问题

- 根据上图中的信息，协调者将错误的得出死锁存在的结论，并中止某个进程。
- 这种情况称为假死锁。
- 由于信息的不完整和延迟，使得分布式系统中的许多死锁算法产生了类似的假死锁问题。

解决假死锁问题

- 一种可能的解决方法是使用Lamport算法以提供全局时间。
  - 既然从机器1到协调者的消息是由机器0的请求发出的，那么从机器1到协调者的消息的时间戳就应该晚于从机器0到协调者的消息的时间戳。
  - 当协调者收到了从机器1发来的有导致死锁嫌疑的消息后，给每台机器发送一条消息
    - “我刚刚收到一条会导致死锁的消息，带有时间戳T，若有任何小于T的消息要发给我，请立即发送。”

解决假死锁问题（cont'd）

- 当每台机器或肯定或否定的响应之后，协调者就会看到从R到B已经消失了，因此系统仍然是安全的。

- 尽管这种方法消除了假死锁，但它需要全局时间，而且开销很大。
- 其他的一些消除假死锁的方法也很困难。

分布式的死锁检测

- **Chandy-Misra-Haas**算法（**Chandy**等，**1983**）允许进程一次请求多个资源（如锁）而不是一次一个。
  - 通过允许多个请求同时进行使得事务的增长阶段加速。
  - 这使得一个进程可以同时等待两个或多个进程。

资源图

- 下图是一种改进的资源图，图中只给出进程。
  - 每条弧穿过一个资源，为简单起见从图中删除了资源
- 可以看到
  - 机器1上的进程3正在等待两个资源，一个由进程4占有，一个由进程5占有。
  - 一些进程正在等待本地资源，例如进程1。
  - 一些进程，如进程2在等待其他机器上的资源。

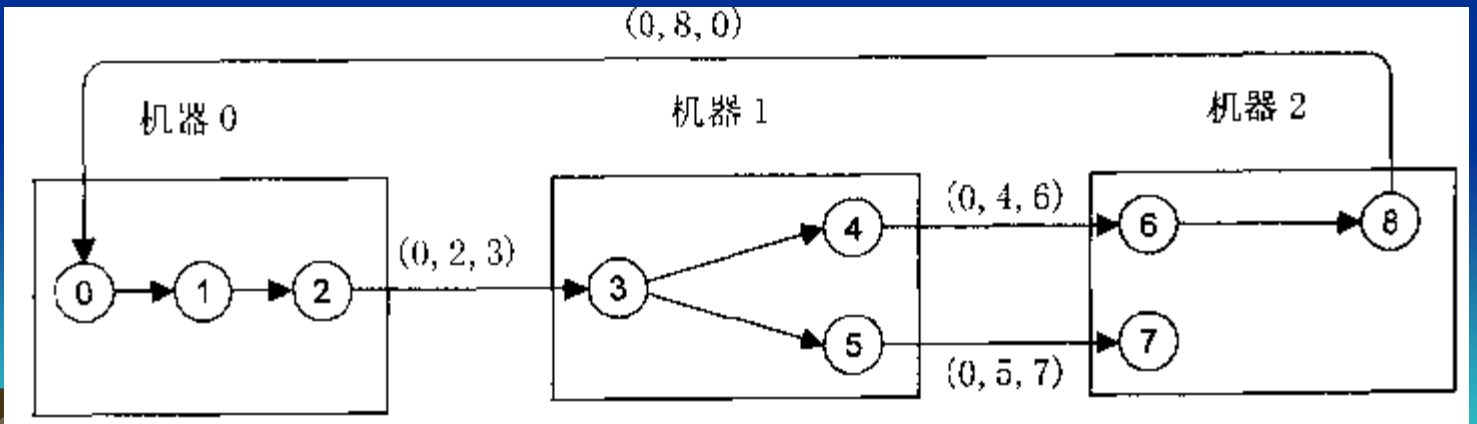


图 3-23　Chandy-Misra-Haas 分布式死锁检测算法

Chandy-Misra-Haas算法

- 当某个进程等待资源时，例如P0等待P1，将调用Chandy-Misra-Haas算法。
  - 生成一个探测消息并发送给占用资源的进程。
    - 消息由三个数字构成：阻塞的进程，发送消息的进程，接受消息的进程。
    - 由P0到P1的初始消息包含三元组（0，0，1）。

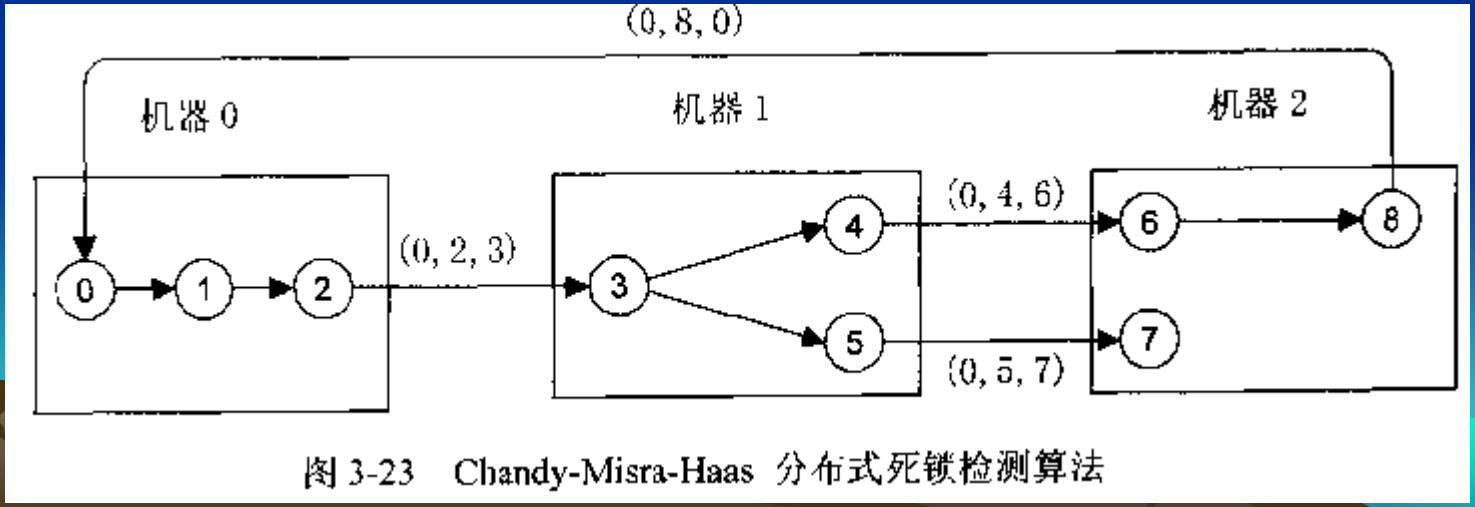


图 3-23　Chandy-Misra-Haas 分布式死锁检测算法

Chandy-Misra-Haas算法（cont'd）

- 消息到达后，接受者检查以确认它自己是否也在等待其他进程。
  - 若是，就更新消息，字段1保持不变，字段2改成当前进程号，字段3改为等待的进程号。
  - 然后消息接着被发送到等待的进程。
  - 若存在多个等待进程，就要发送多个不同的消息。

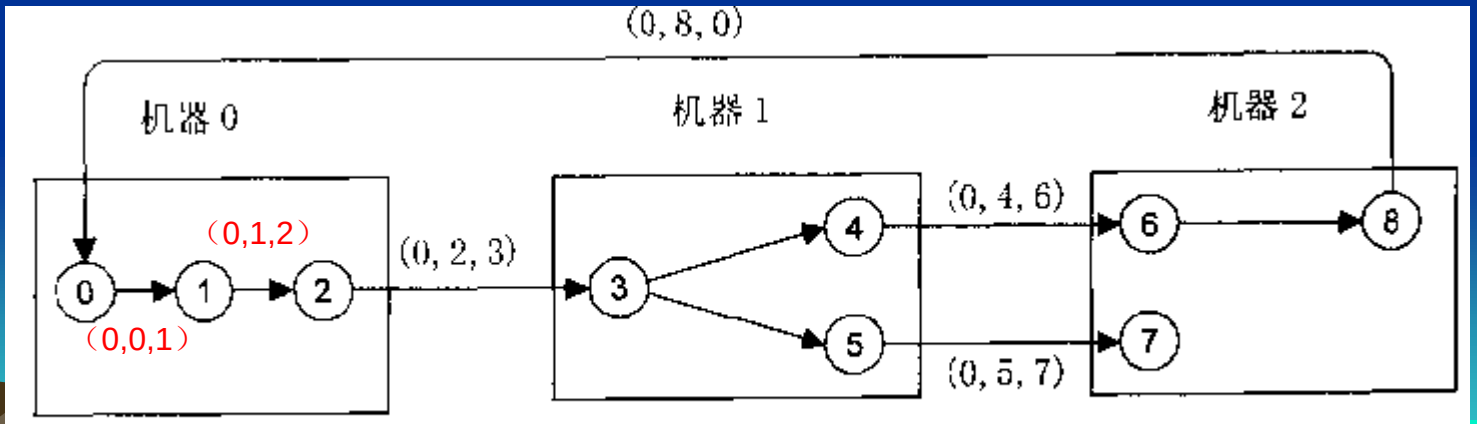


图 3-23　Chandy-Misra-Haas 分布式死锁检测算法

Chandy-Misra-Haas算法（cont'd）

- 不论资源在本地还是在远程，该算法一直继续下去。
  - 图中（0，2，3），（0，4，6），（0，5，7）和（0，8，0）都是远程消息。
- 若消息转了一圈后又回到最初的发送者，即字段1所列的进程，就说明存在一个有死锁的环

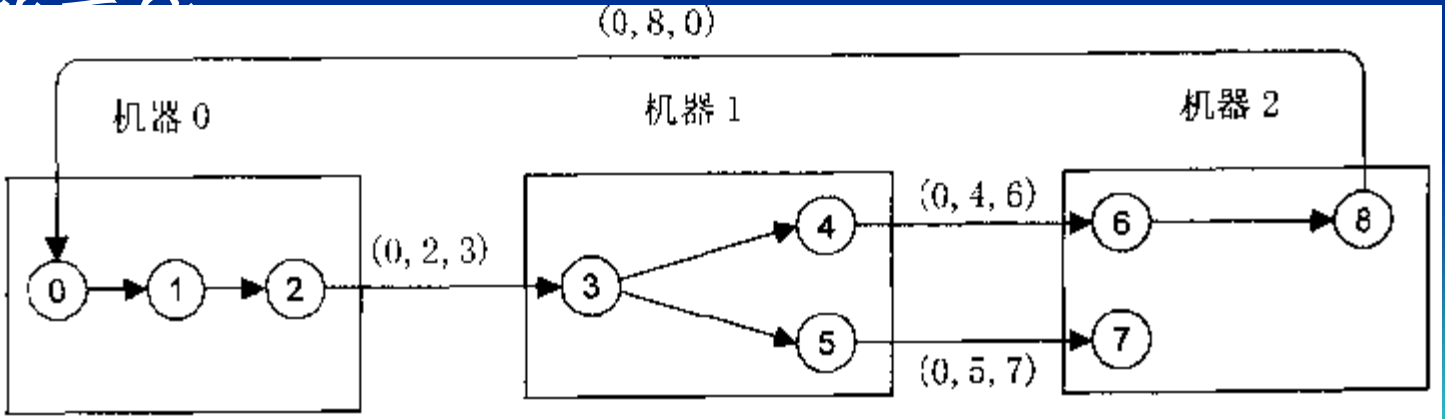


图 3-23　Chandy-Misra-Haas 分布式死锁检测算法

打破死锁的方法（1）

可以有不同的方法打破死锁：

- 一种方法是使最初发送探测消息的进程自杀。
  - 如果有多个进程同时调用了此算法，那就会出现问题。
  - 例如在上例中假设进程0~6同时阻塞，而且都初始化了探测消息。那么每个进程最终都会发现死锁，并且因此而自杀，然而这是不必要的。中止掉一个进程就足够了。

打破死锁的方法（2）

- 另一种算法是将每个进程的标识符添加到探测消息的末尾，这样当它返回到最初的发送者时完整的环路就可以列出来了。于是发送者就能看出哪个进程编号最大，可以将它中止或者发送一个消息给它请求其自杀。
- 无论如何，如果多个进程同时发现了同一个环路，它们就一定会选择同一个牺牲者。

小结

- 时钟同步
- 逻辑时钟
- 选举算法
- 互斥
- 分布式事务
- 分布式系统中的死锁

习 题

- 分布式互斥算法中，建议所有的请求都被应答（同意或否定），这样可以检测出崩溃的进程，但是否还有其他问题？
- 许多分布式算法需要协调者，与集中式协调者相比，分布性体现在哪里？