高级操作系统

2. 分布式操作系统的同步

2.3-2.5. 选举、原子事务、死锁

熊焰，yxiong@ustc.edu.cn
黄文超, huangwc@ustc.edu.cn
http://staff.ustc.edu.cn/~huangwc/advancedos

参考书目：《分布式操作系统》，《分布式系统原理与范型》

3. 选举算法

- **需求：**
  - 许多分布式算法**需要**一个进程充当：
    - 协调者、发起者、排序者...
    - 如集中式互斥算法中的协调者
  - 哪个进程充当协调者并**不重要**

3. 选举算法

- **前提假设及目标**（输入与输出）：
  - 如何区分进程?
    - 每一个进程有唯一ID，如IP等
    - （新增假设：每台机器只有一个进程）
- 如何选择进程?
  - 可选目标：选取ID最大的进程

3. 选举算法

- **前提假设及目标**（输入与输出）：
  - 实际性假设：
    - 每个进程**都知道**所有其他进程的**进程号**
      - **但不知道**哪些进程正常，哪些不正常
  - 关注的目标：
    - **何时**开始选举?
    - 在所有进程**都同意**的基础上选出协调者

3. 选举算法

- 传统选举算法
  - 欺负算法 (Bully Algorithm)
  - 环算法
- 无线系统环境中的选举算法
- 大型系统中的算法

3. 选举算法

3.1 欺负(Bully)算法

- Hector Garcia-Molina: Elections in a Distributed Computing System. *IEEE Trans. Computers 31(1)*: 48-59 (1982) （被引用次数：749）

- 核心思想：
  - 当一个进程**发现协调者不再响应请求**时，它就发起选举

3. 选举算法

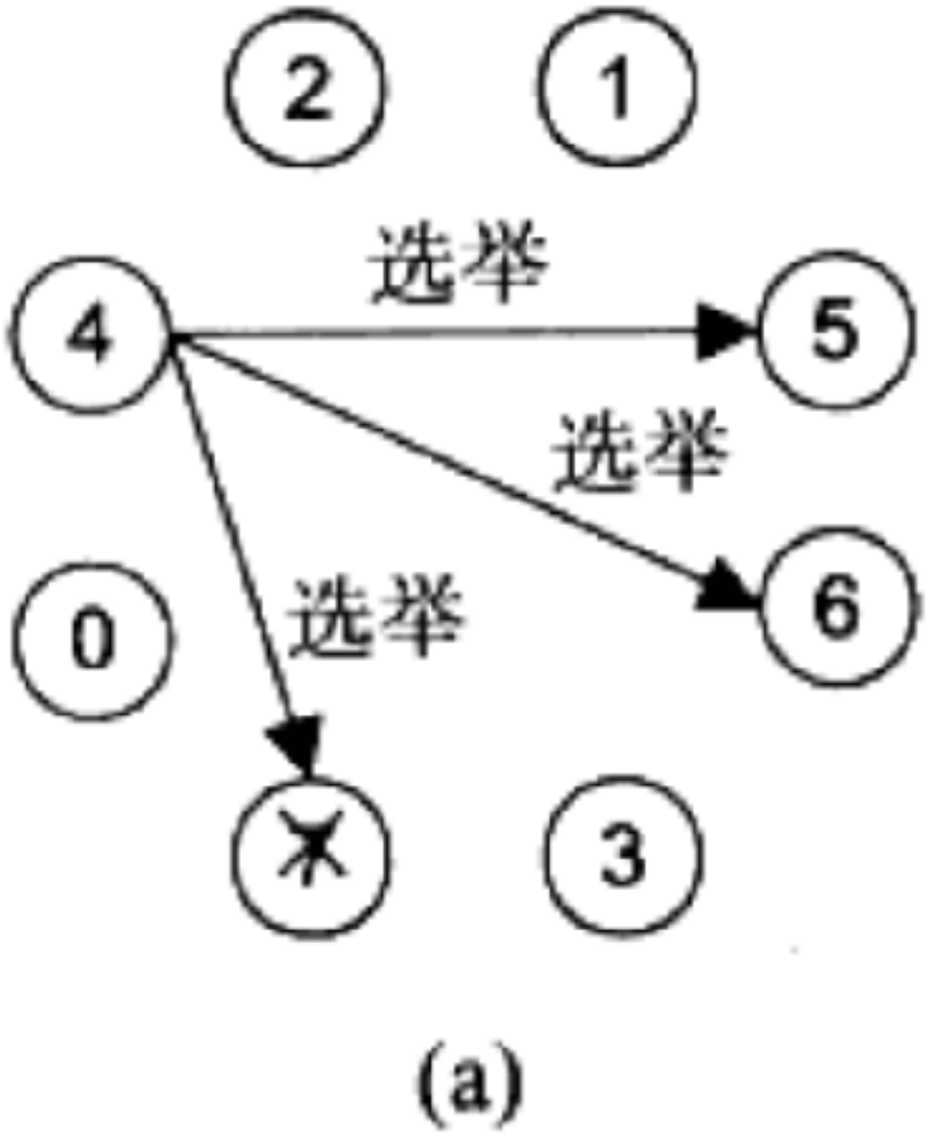
3.1 欺负(Bully)算法

- 算法描述：
  - 进程P向所有进程号比它大的进程发送选举（ELECTION）消息；
    - 若无人响应，P获胜成为协调者；
    - 若有进程号比它大的进程响应，响应者接管，P的工作完成。
- 最终结果：
  - 最大的进程总能取胜，故称之为**欺负算法**

3. 选举算法

3.1 欺负(Bully)算法

- 进程4主持选举

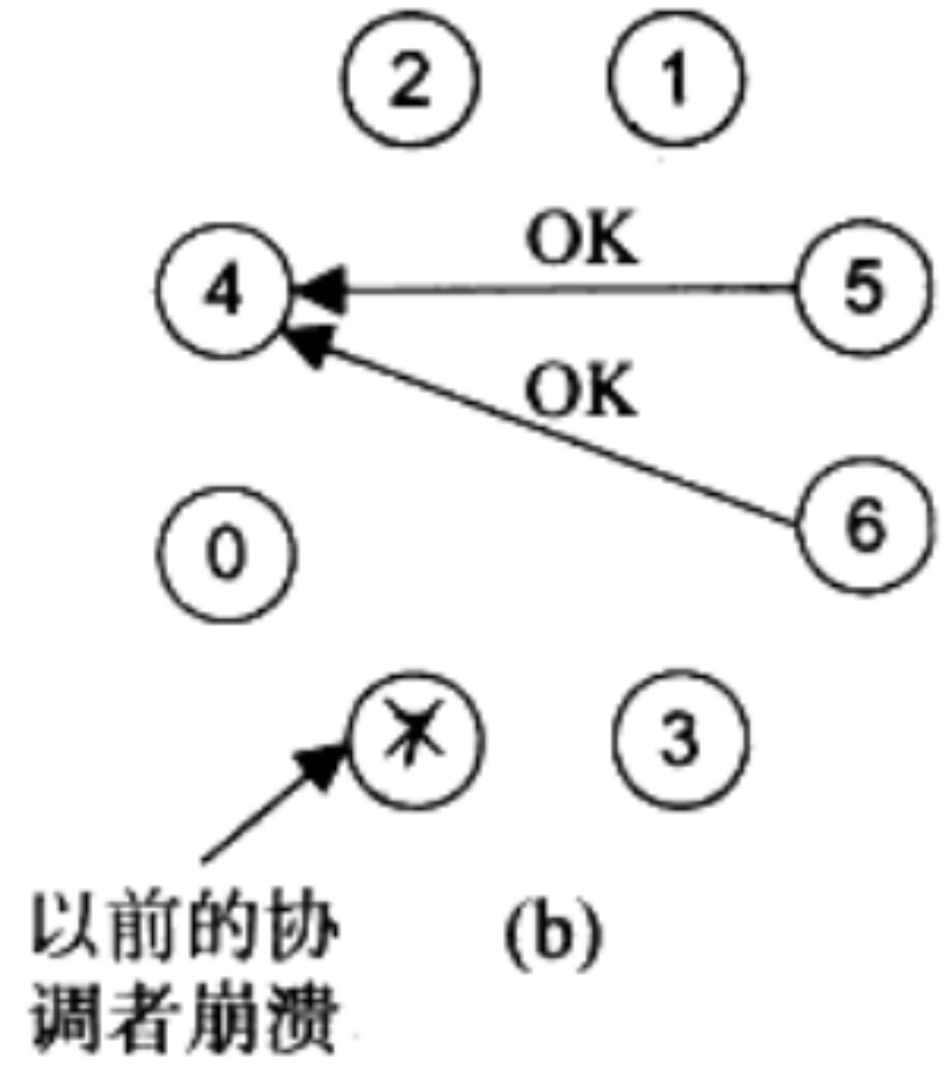


(a)

3. 选举算法

3.1 欺负(Bully)算法

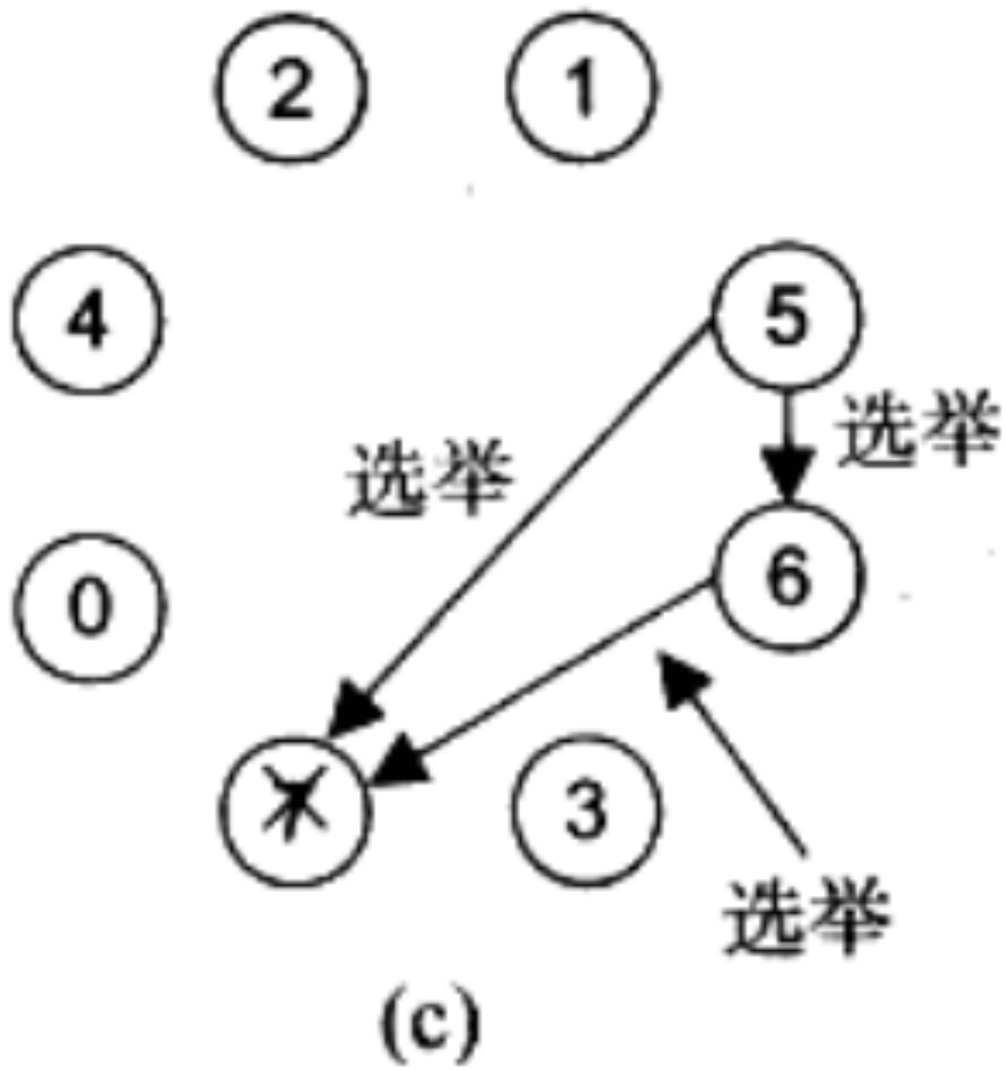
- 进程5和进程6应答，通知进程4停止



3. 选举算法

3.1 欺负(Bully)算法

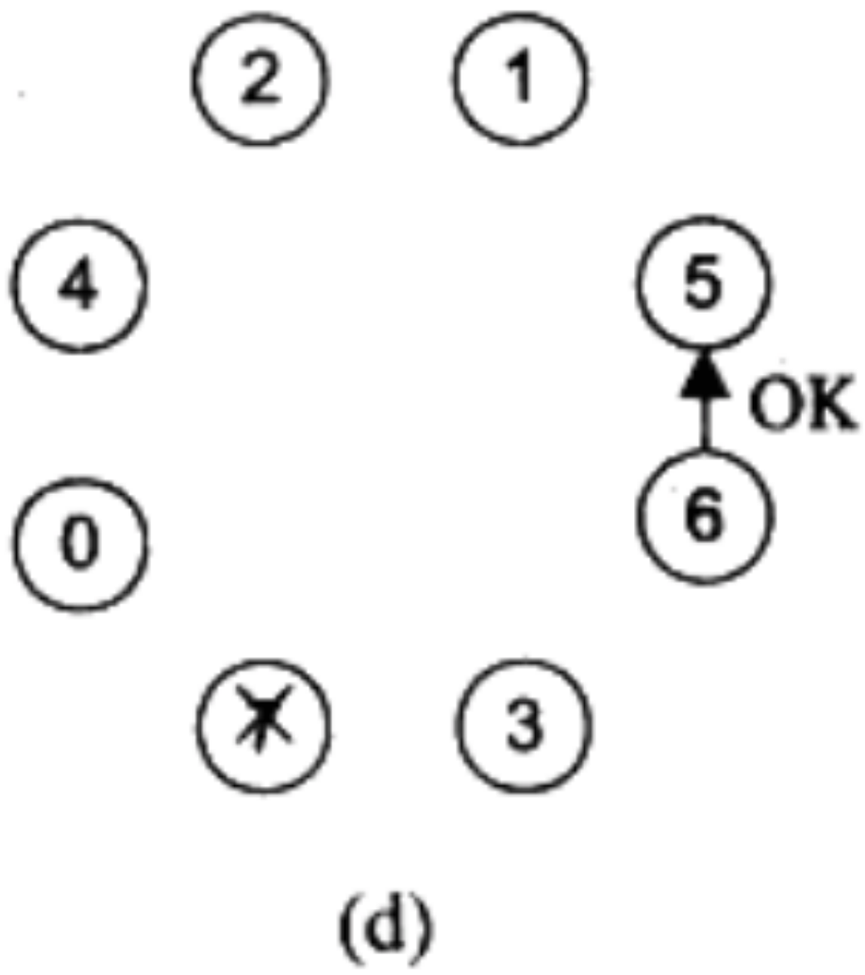
- 进程5，进程6分别主持选举



3. 选举算法

3.1 欺负(Bully)算法

- 进程6通知进程5停止

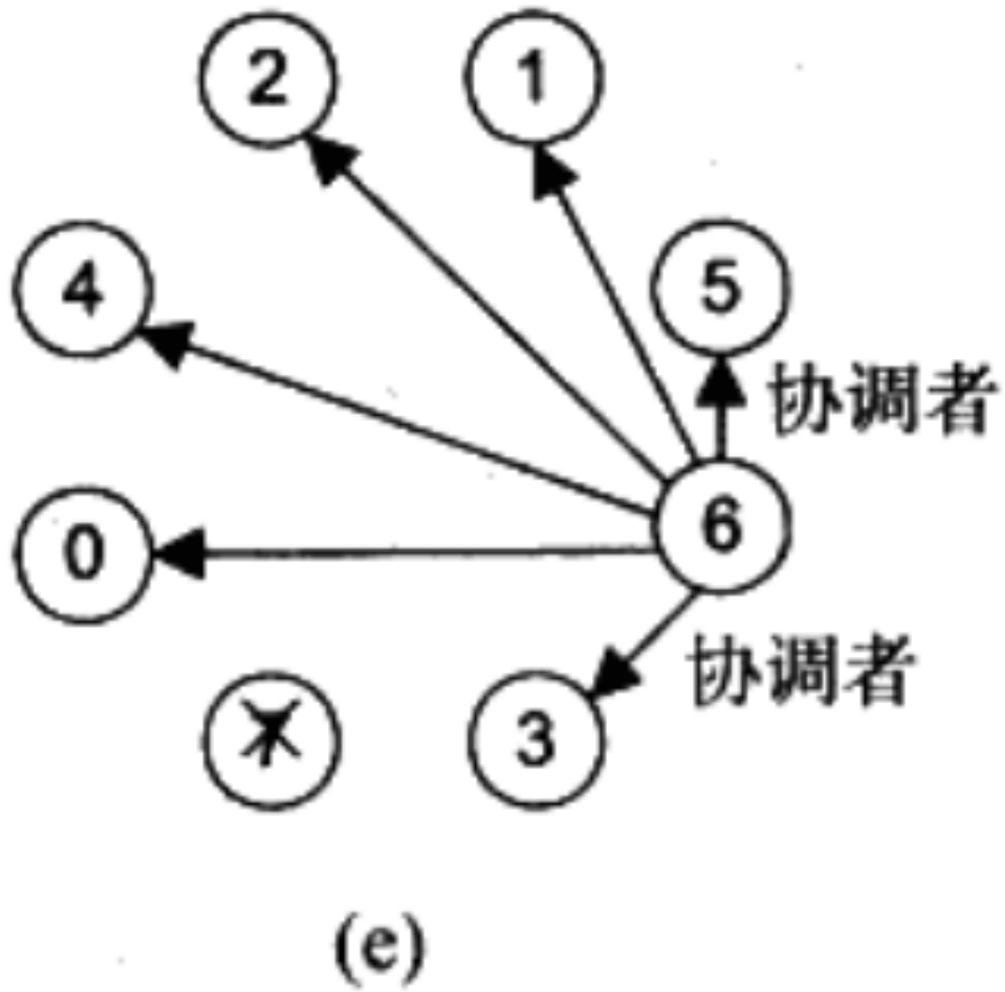


(d)

3. 选举算法

3.1 欺负(Bully)算法

- 进程6获胜并通知所有进程



(e)

3. 选举算法
3.2 环算法

- 新增假设：
- 没有令牌的环：
  - 所有的进程是按物理或逻辑环排序
  - 每个进程都知道谁是它的后继

3. 选举算法

3.2 环算法

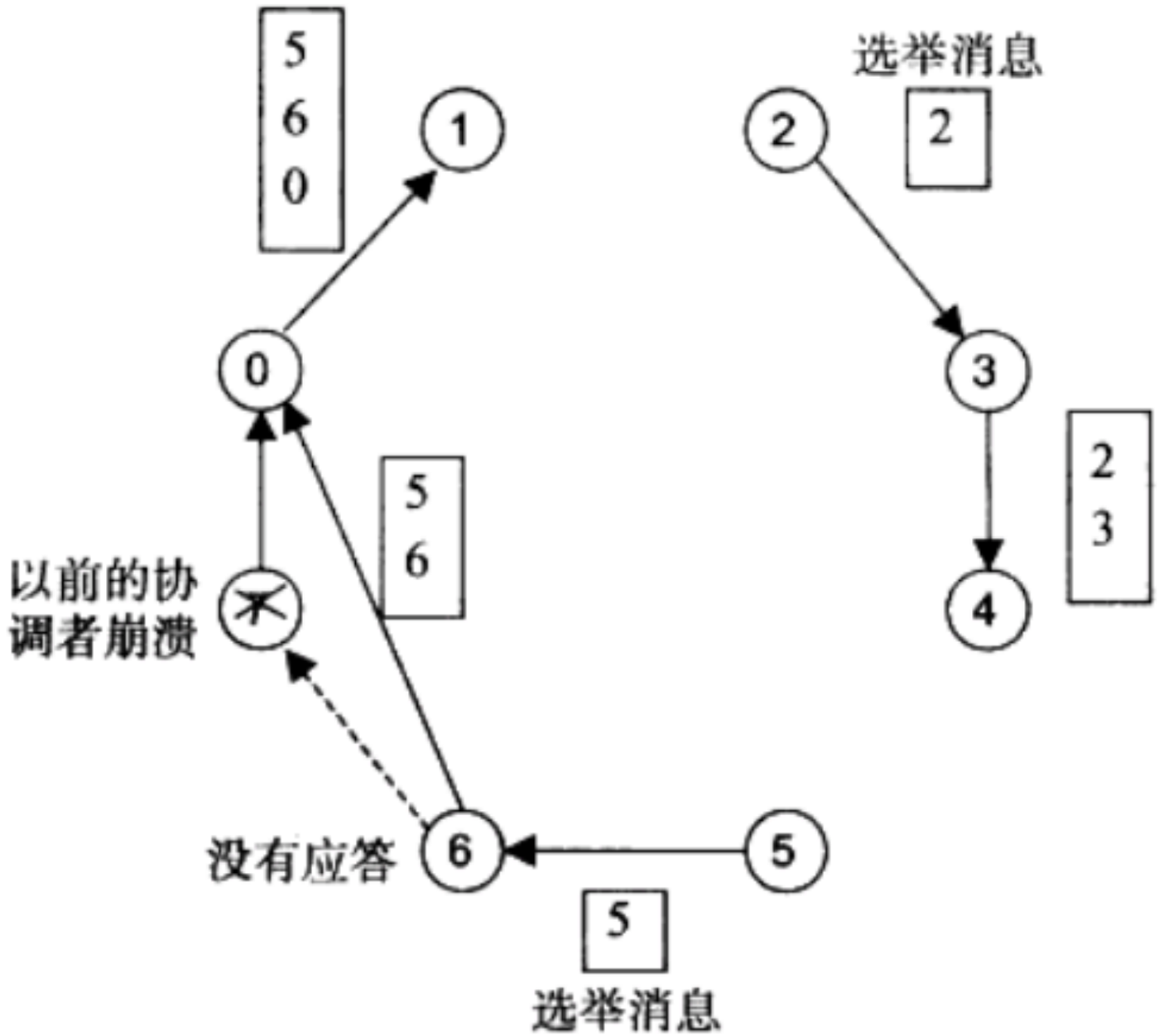
- 算法描述
  - 当**任一进程**发现协调者不再起作用时
    - 创建一个包含它自身进程号的选举消息发送给它的后继
  - 如果下邻居失效
    - 消息将绕过它到达它的后继，或者再下一个，直到找到一个运行进程。
  - 每一个发送者都将自己的进程号加入到消息表中

3. 选举算法
3.2 环算法

- 算法描述
    - 最后，消息到达了始发者手中
        - 始发者接收到包括自己进程号的消息后
            - 将消息的类型转化为协调者消息，该消息将再一次绕环运行，向所有的进程通知谁是协调者（在成员表中进程号最大的那个）。
- 当消息循环一周后，被销毁，每个进程都恢复工作。

3. 选举算法

3.2 环算法



3. 选举算法

3.3 无线系统环境中的选举算法

- 传统算法的某些假设不切实际 （在大多数无线网络环境中，如MANET）
- 消息传送是可靠的
- 拓扑结构不会改变

3. 选举算法

3.3 无线系统环境中的选举算法

- Vasudevan提出MANET中能处理**节点失效和网络分区**的解决方法

  - Sudarshan Vasudevan, James F. Kurose, Donald F. Towsley: Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks. *ICNP 2004*: 350-360. （被引用次数：204）

- 重要特性：

  - 可以选举出最佳的领导者（协调者）

3. 选举算法

3.3 无线系统环境中的选举算法

- 算法描述
  - 初始：**任意节点**可以通过往其紧邻节点发送**ELECTION**消息来开始一个选举
  - 当节点R**第一次收到**ELECTION消息时
    - 将该发送者Q指定为**父节点**
    - 将ELECTION消息发给所有紧邻节点（父节点Q除外）
    - R等待来自其他节点的确认消息
    - R响应Q，并向Q报告其他信息
      - 如， 电池寿命、其他资源容量

3. 选举算法

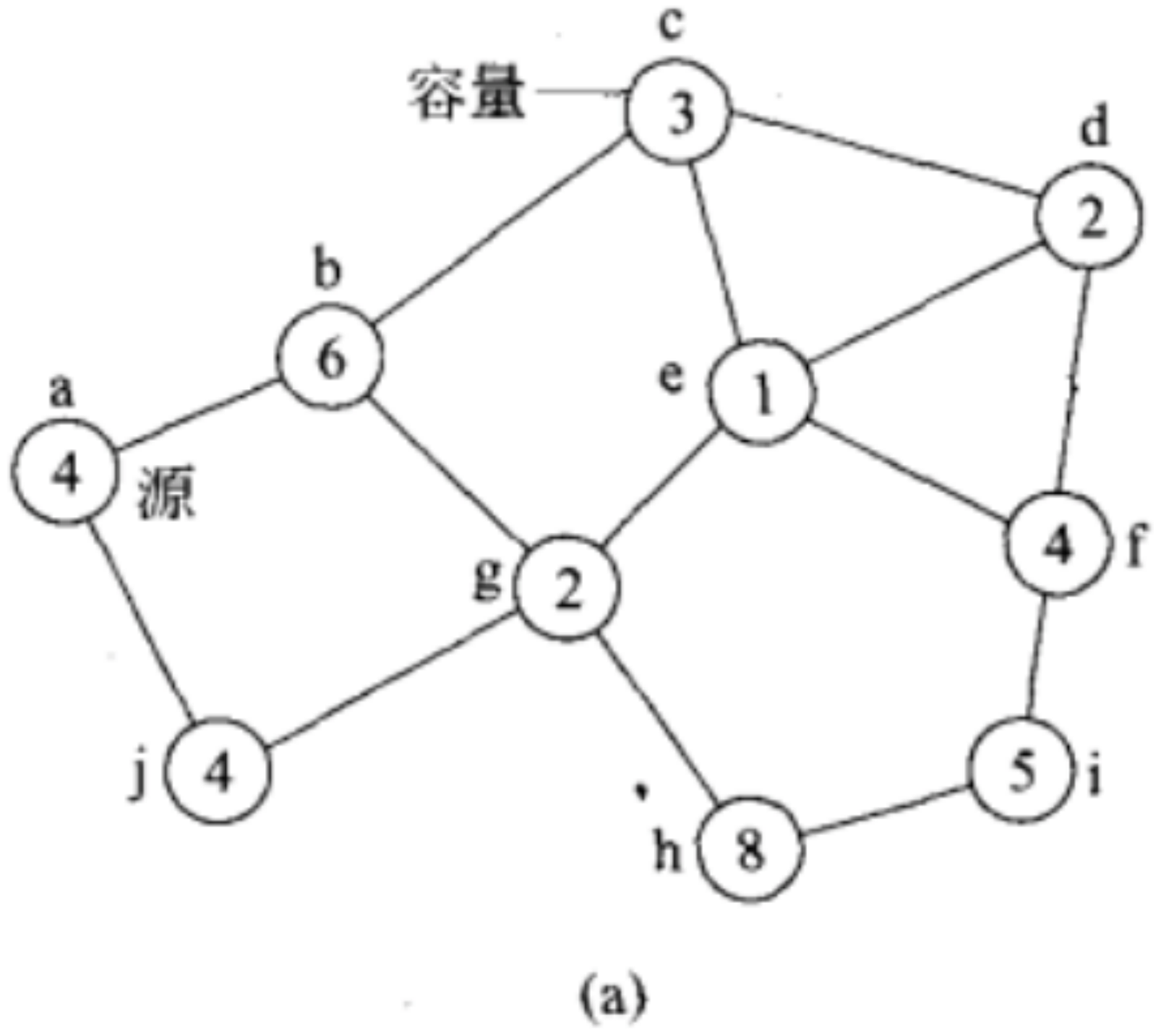
3.3 无线系统环境中的选举算法

- 算法描述（续）
  - Q利用返回信息：
    - 将R的容量与其他下游节点比较
      - 选最合格的作为领导候选者
      - 将候选者发给其父节点P
- 依次递推，**源节点**选择最终领导者
  - 将该信息广播给所有其他节点

3. 选举算法

3.3 无线系统环境中的选举算法

- a向b, j 广播ELECTION消息

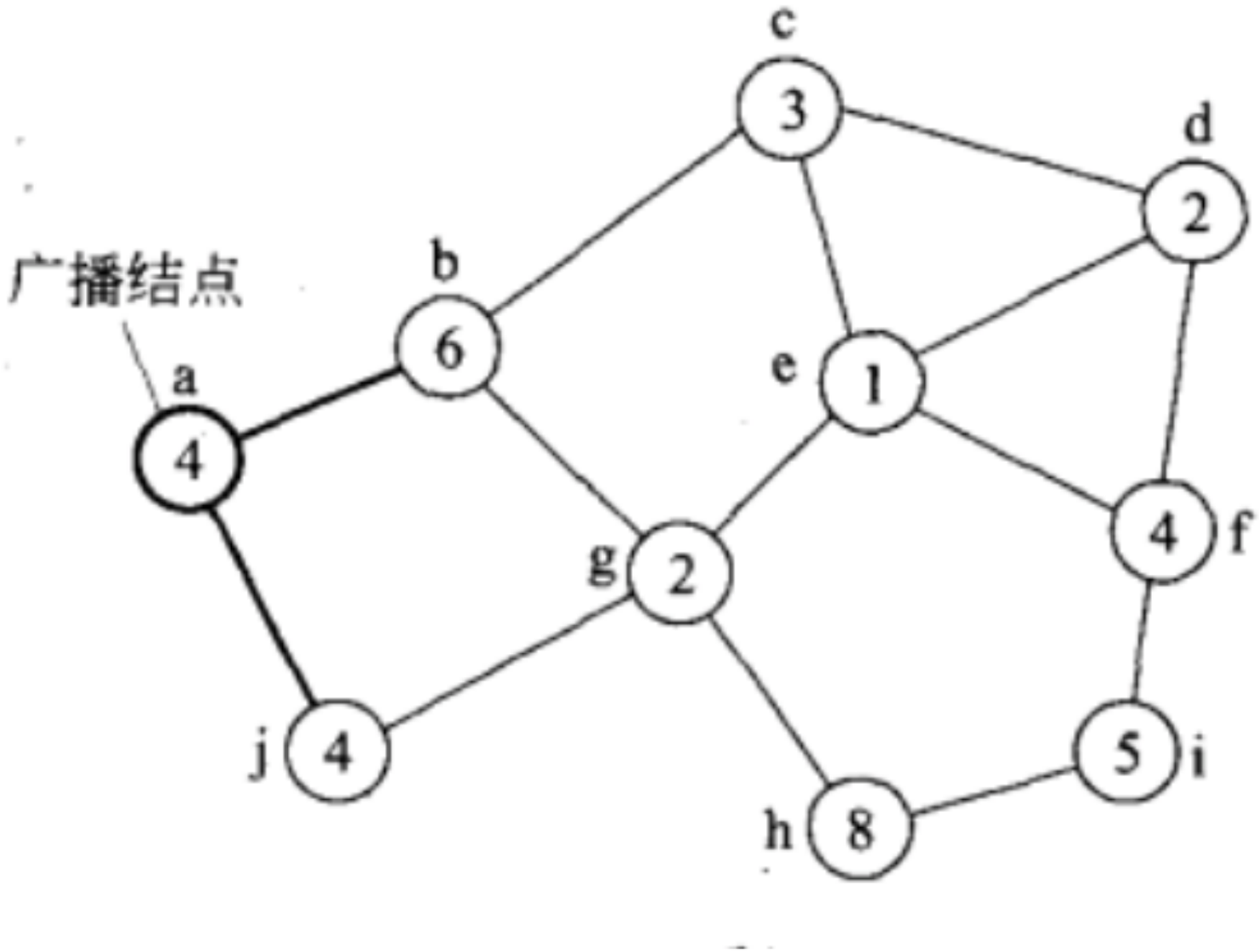


(a)

3. 选举算法

3.3 无线系统环境中的选举算法

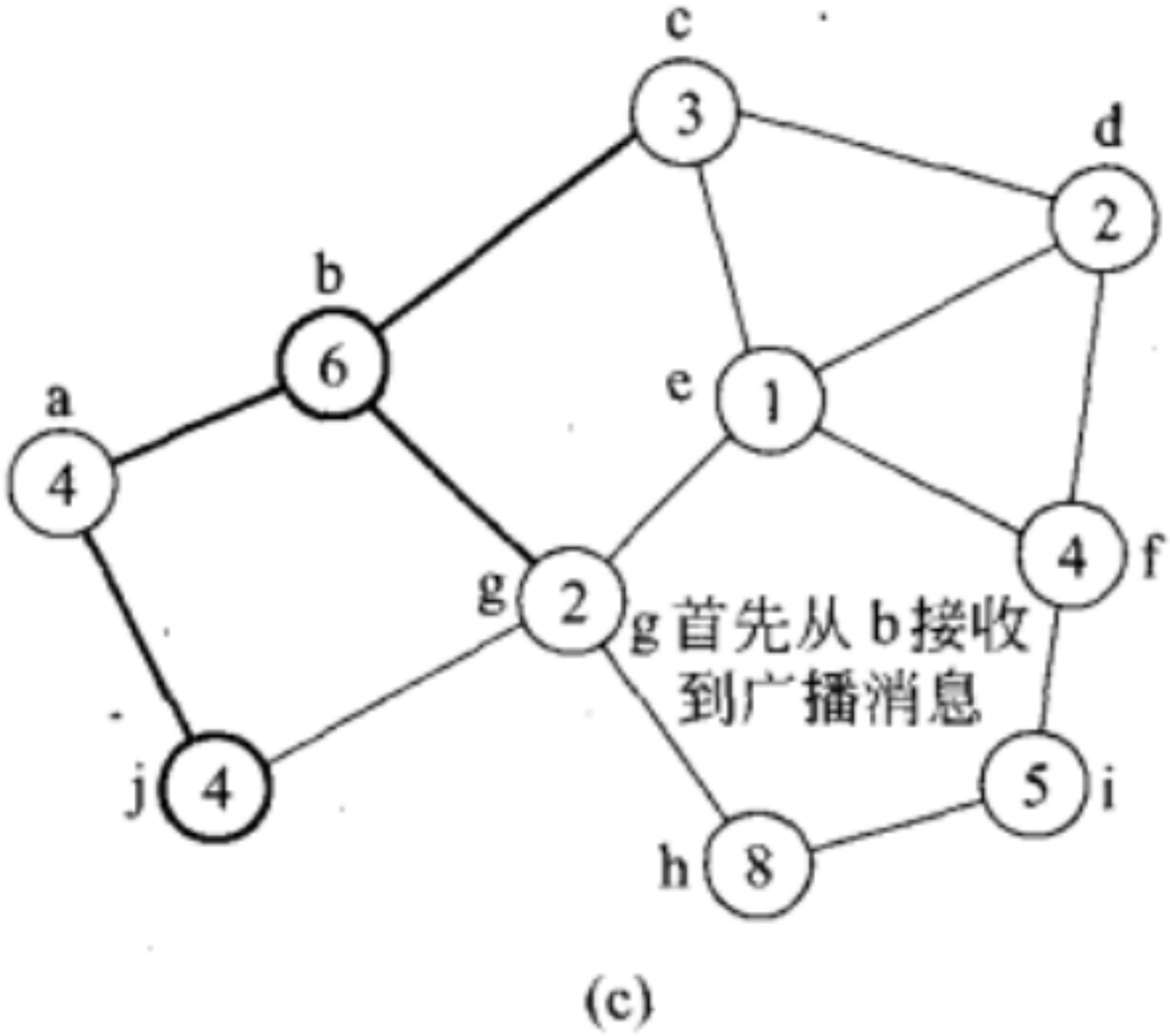
- 消息被传播至b,j



3. 选举算法

3.3 无线系统环境中的选举算法

- 消息被传播至b,j,g,c

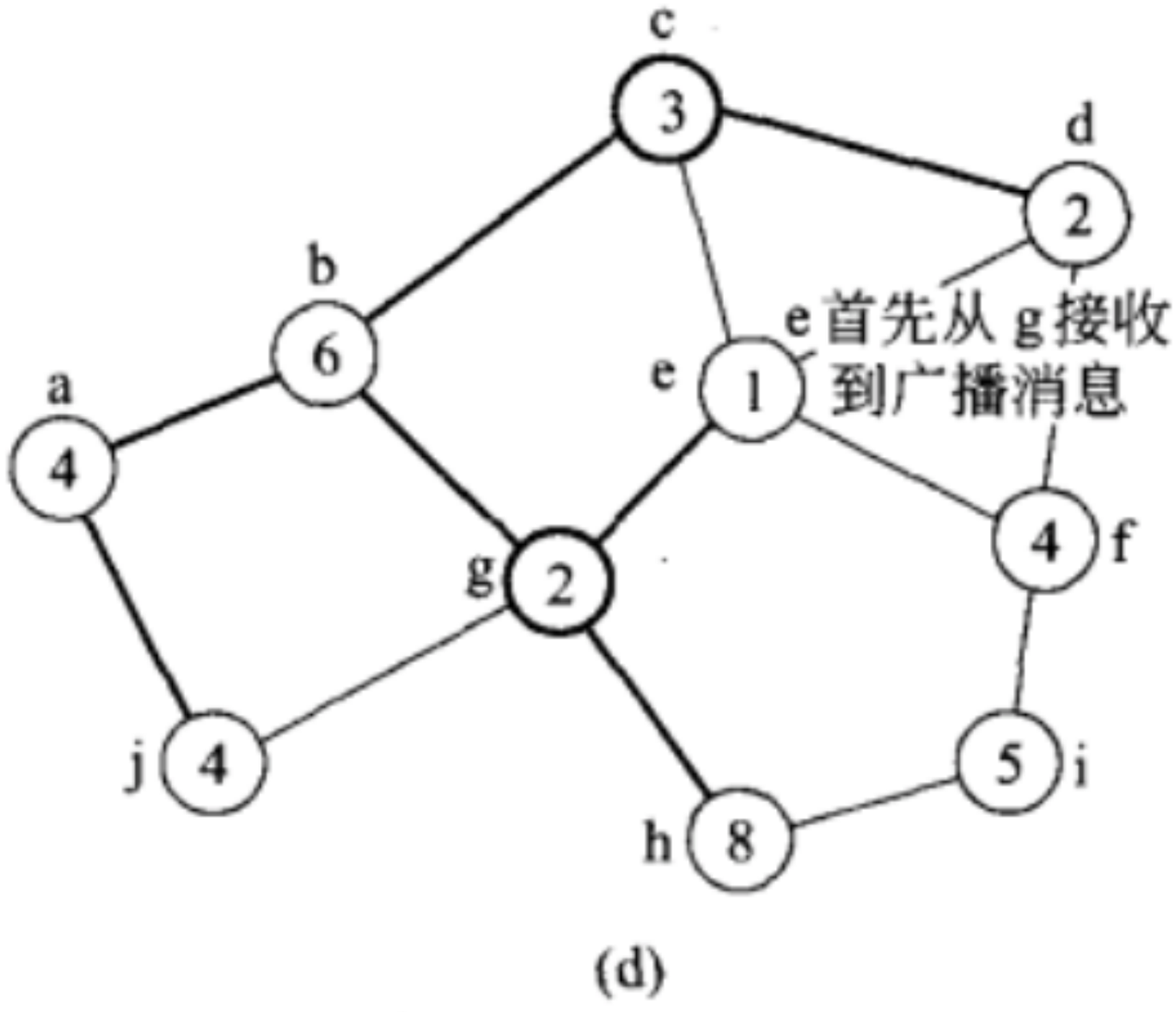


(c)

3. 选举算法

3.3 无线系统环境中的选举算法

- 消息被传播至b,j,g,c,e

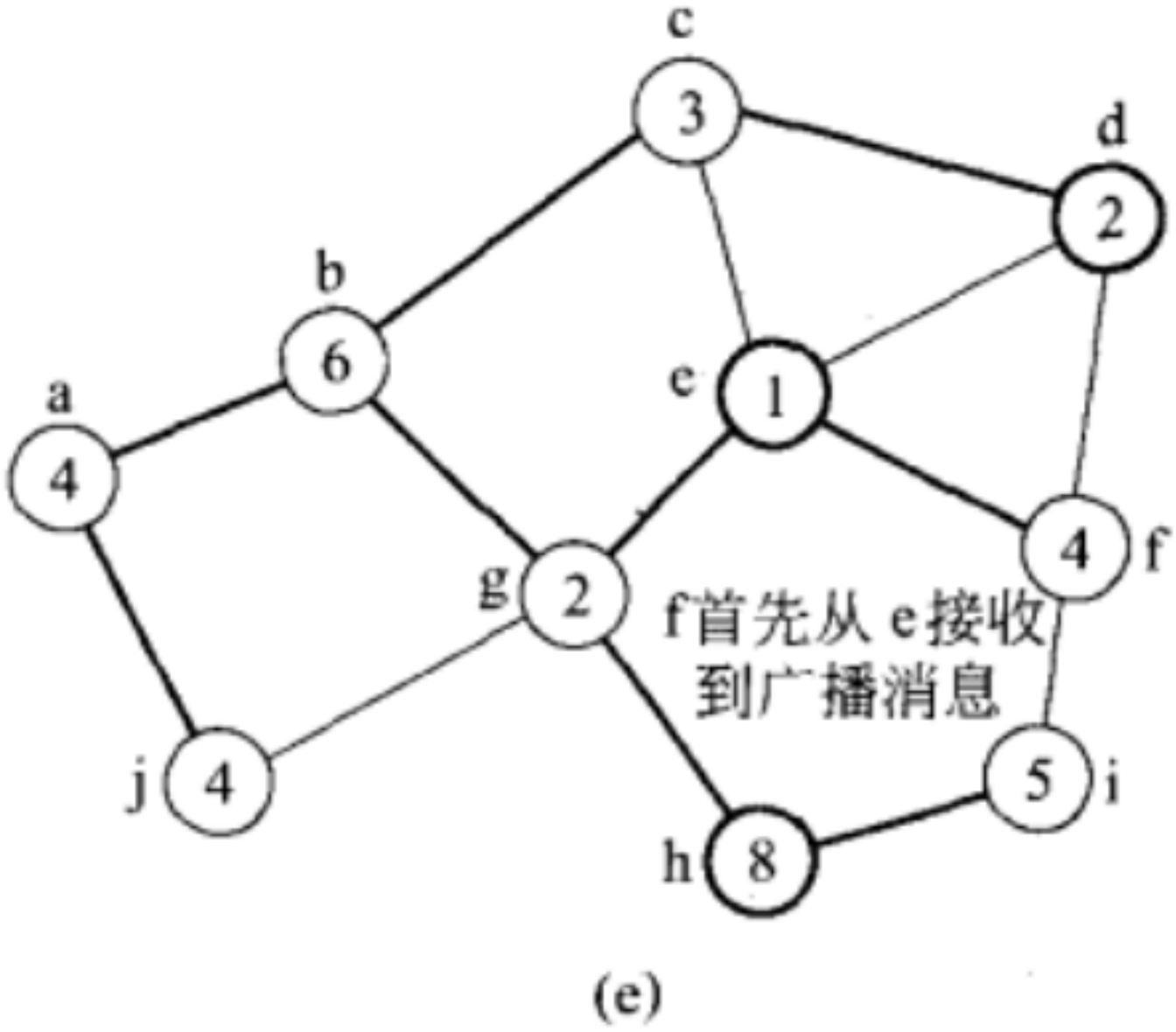


(d)

3. 选举算法

3.3 无线系统环境中的选举算法

- 消息被传播至b,j, g,c,e, f,h,i

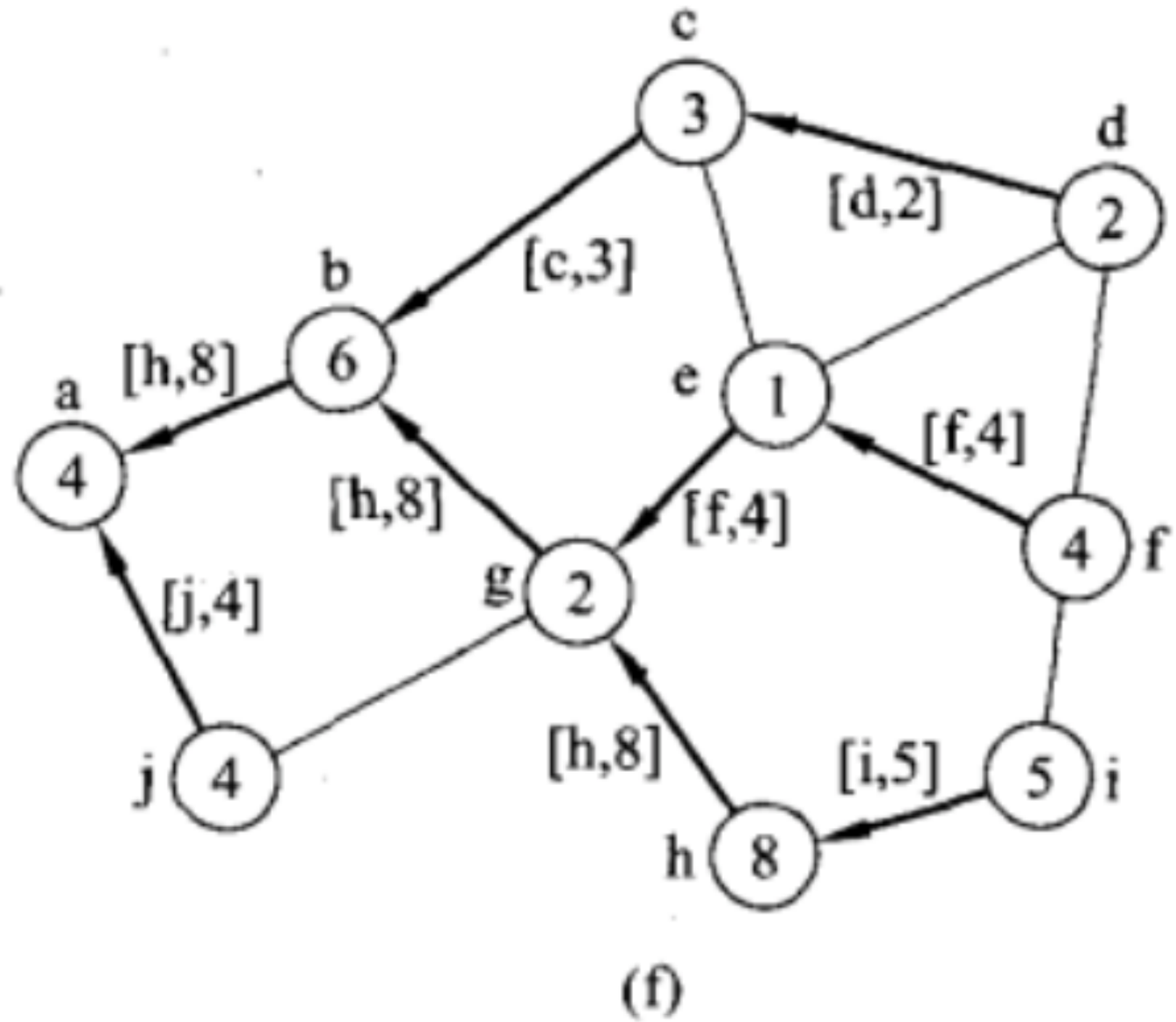


(e)

3. 选举算法

3.3 无线系统环境中的选举算法

- 每个节点向父节点报告具有最佳容量的节点



(f)

3. 选举算法

3.4 大型系统中的选举算法

- 传统算法只关注**单个节点**的选举
- 如果选举多个节点：
  - 点对点网络中
    - 超级点(superpeer)

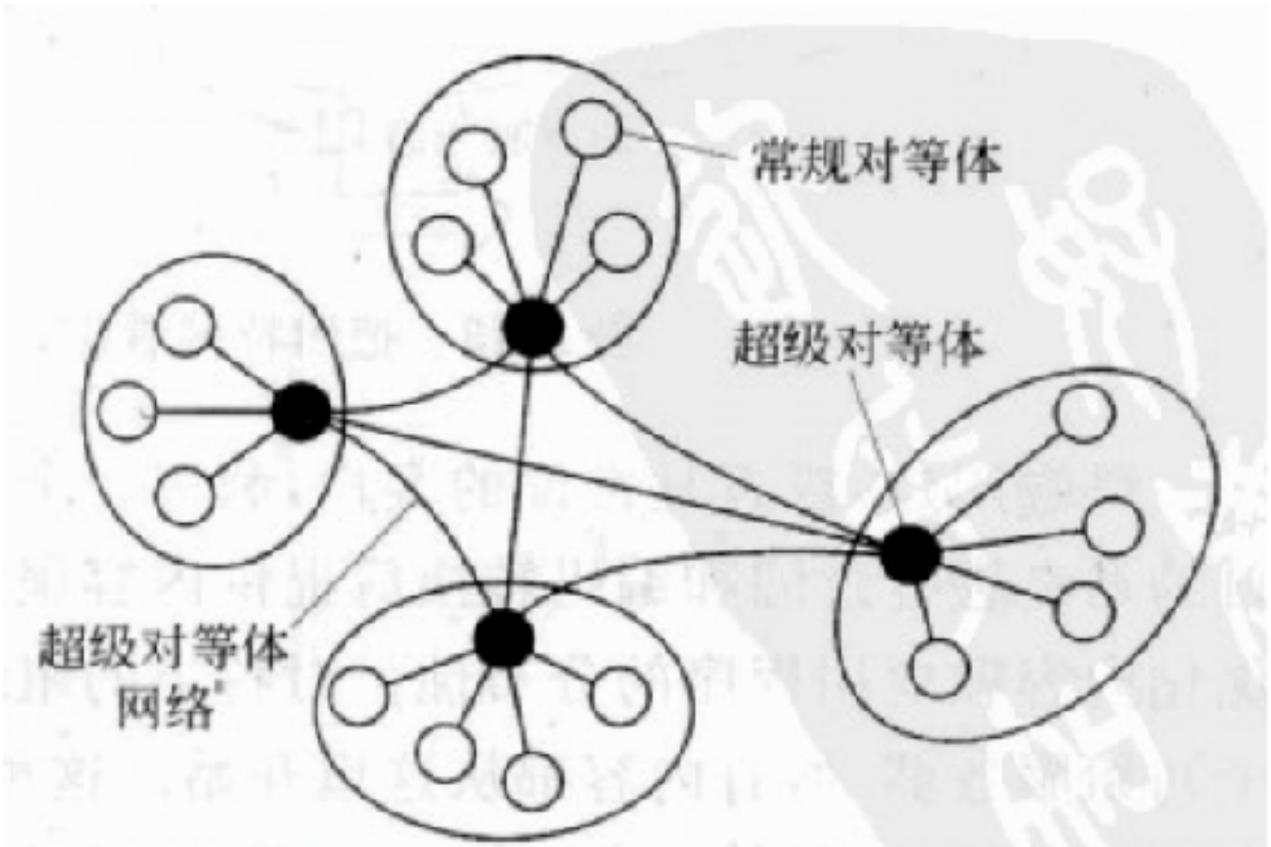


图 2.12　超级对等体网络中结点的分层组织结构

3. 选举算法

3.4 大型系统中的选举算法

- 超级点选举的**需求**
  - 超级点的访问**延时低**
  - 超级点应**平均覆盖**网络
  - 有一部分**预定义**的超级点
  - 超级点**无需**为**固定数目**的一般节点服务

3. 选举算法

3.4 大型系统中的选举算法

- 方法一：Chord系统所采用方案
  - 为超级点**预留**一部分标识符
    - 为每个节点分配一个m位标识符
      - 预留m位中的前k位来表示超级点
- 例：假设m=8, k=3
  - 如要查找负责标识为p的节点，则把查询请求发给下面节点：
    - p AND 11100000

3. 选举算法

3.4 大型系统中的选举算法

- 方法二：在几何空间中定位节点
  - 目标：在整个覆盖网络中**均匀放置N个超级点**
    - 总共**N个令牌**分布在N个随机选择的节点中
    - 每个令牌都会推动其他令牌移开
    - 最终，令牌均匀分布在几何空间中

3. 选举算法

3.4 大型系统中的选举算法

- 方法二：在几何空间中定位节点

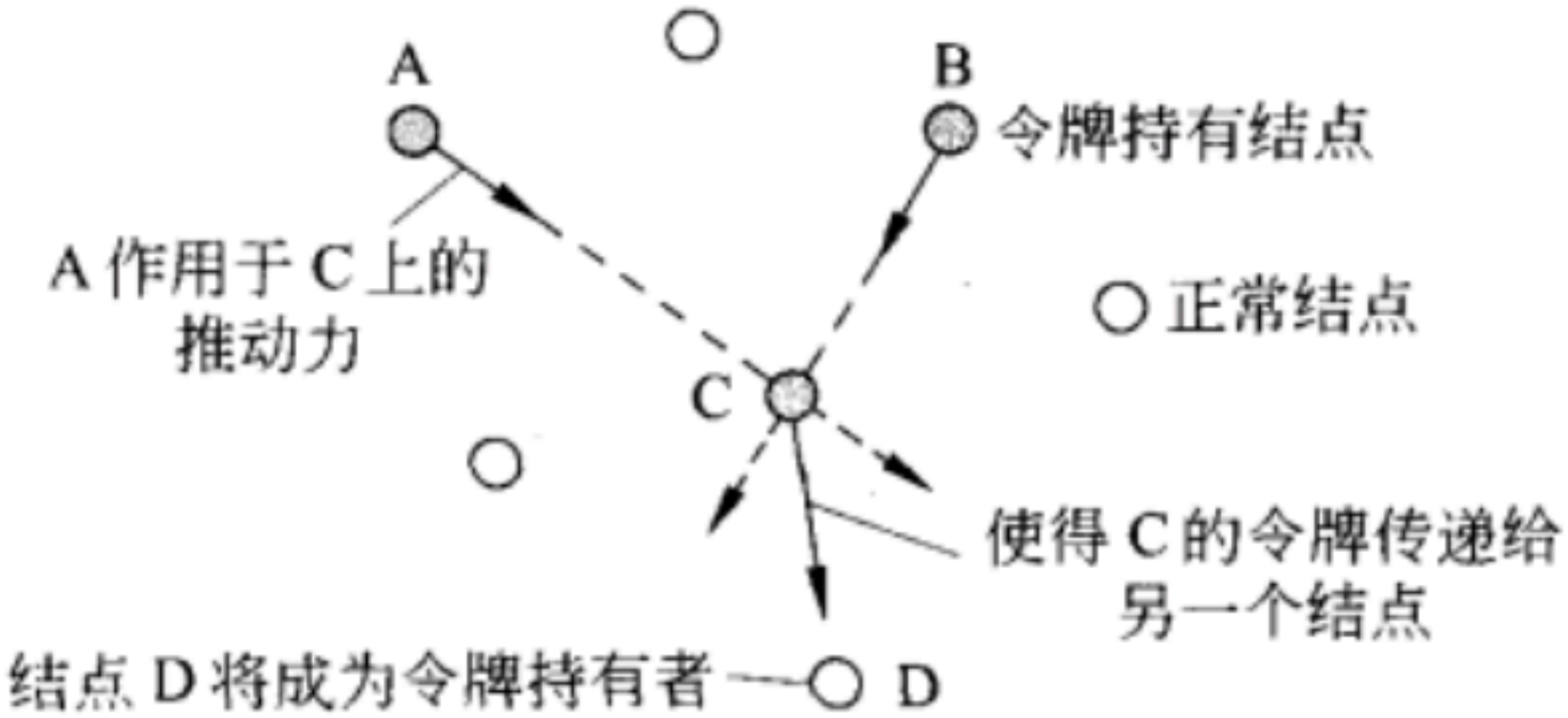


图 6.23　使用推动力在二维空间中移动令牌

4. 原子事务
4.1 原子事物简介

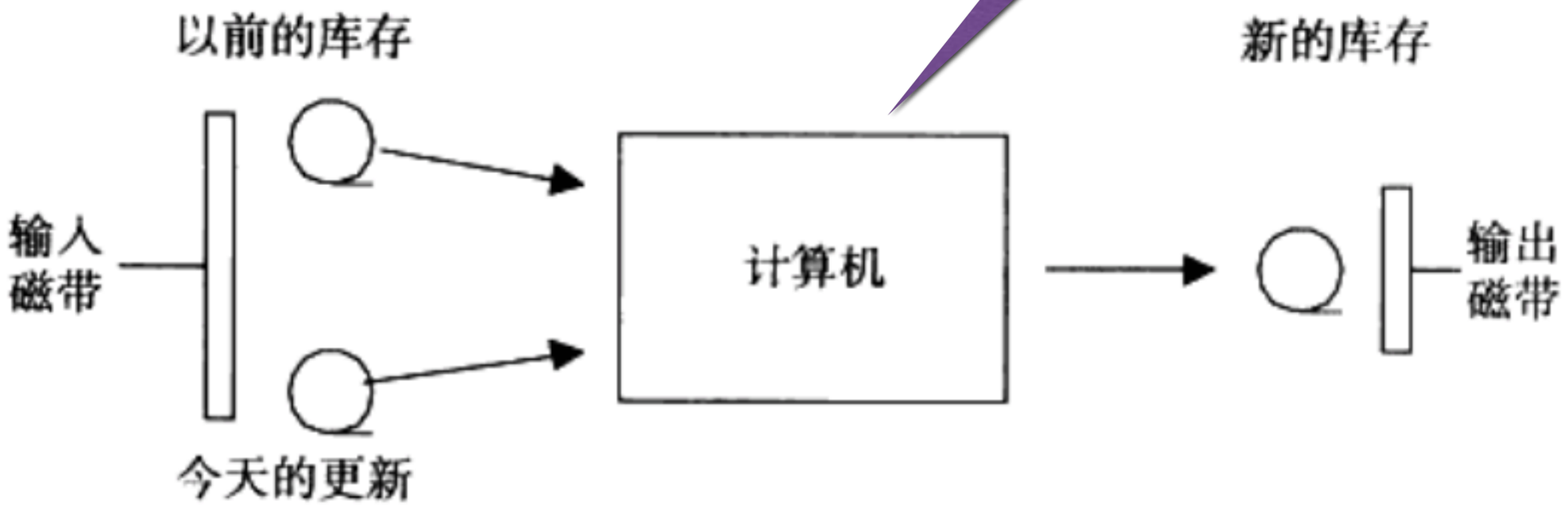
- 底层技术：
  - 互斥、同步
    - 信号量、死锁。。。
- 高层抽象：
  - **原子事务** （简称**事务**）

4. 原子事务

4.1 原子事物简介

- 来源：商业社会
  - 谈判—>合同签订
- 事务的使用： 20世纪60年代
- 自动盘点系统的超级市场

优点：对任何原因引起的运行错误，所有磁带都可以倒卷(rewound)



4. 原子事务

4.1 原子事物简介

- 例子：在线更新数据库：
  - 将账户1的钱转账至账户2
    - (1) Withdraw （amount, account1）
    - (2) Deposit (amount, account2)

如果在第一步之后，在第二步之前中断？ —>用事务解决该问题

4. 原子事务
4.2 事务模型

- 假设
  - 系统由一些相互独立的进程组成
  - **每个**进程都会随机**出错**
  - 通信消息会丢失，但通信错误已被透明处理
  - **稳定存储器**

4. 原子事务
4.2 事务模型

- 存储器种类：
  - 普通RAM存储器——断电或系统崩溃后丢失信息
  - 磁盘存储器——磁头错会导致信息丢失
  - **稳定存储器**
    - 只有大灾难（如洪水地震）才产生影响

4. 原子事务
4.2 事务模型

- 稳定存储器

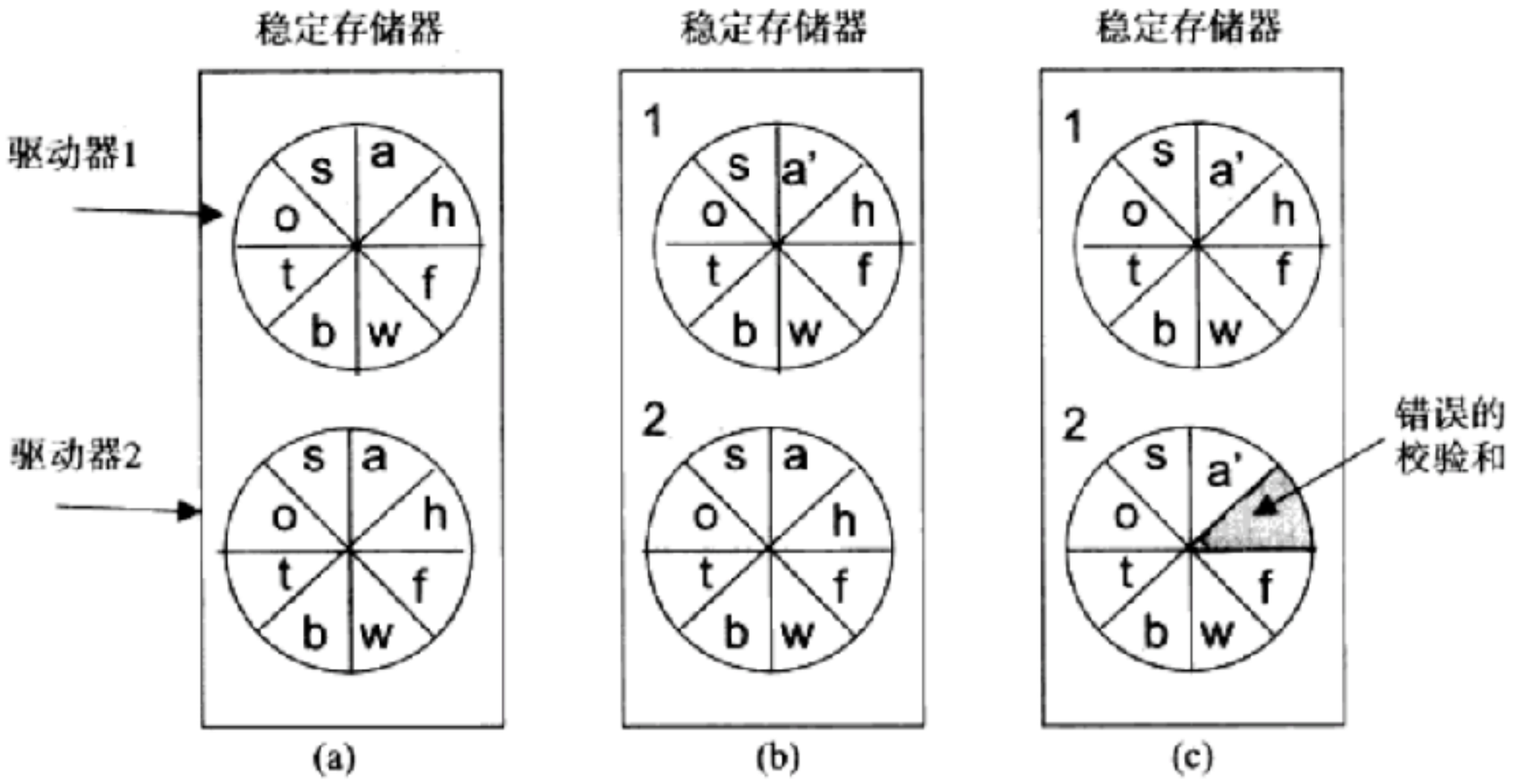


图 3.14　(a) 稳定存储器；(b) 驱动器 1 更新后崩溃；(c) 错误的地方

4. 原子事务
4.2 事务模型

- 事务原语

BEGIN_TRANSACTION	标记一个事务的开始
END_TRANSACTION	结束事务并设法提交
ABORT_TRANSACTION	取消事务；恢复旧值
READ	从一个文件（或其他对象）读取数据
WRITE	将数据写入一个文件（或其他对象）

4. 原子事务
4.2 事务模型

- 例：航空订票系统

BEGIN_TRANSACTION
reserve 合肥-阜阳
reserve 阜阳-武汉
reserve 武汉-长沙
END_TRANSACTION

BEGIN_TRANSACTION
reserve 合肥-阜阳
reserve 阜阳-武汉
武汉-长沙 已满 —>
  ABORT_TRANSACTION

4. 原子事务
4.2 事务模型

- 事务的特性 (ACID)

  - **原子性**（Atomic）：对外部世界来说，事务的发生是不可分割的；

  - **一致性**（Consistent）：事务不会破坏系统的恒定 (Invariants)；

  - **孤立性**（Isolated）：并发的事务不会互相干扰；

  - **持久性**（Durable）：一旦一个事务提交，改变就是永远存在的。

4. 原子事务

4.2 事务模型

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

(a)

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

(b)

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

(c)

时间 ⟶

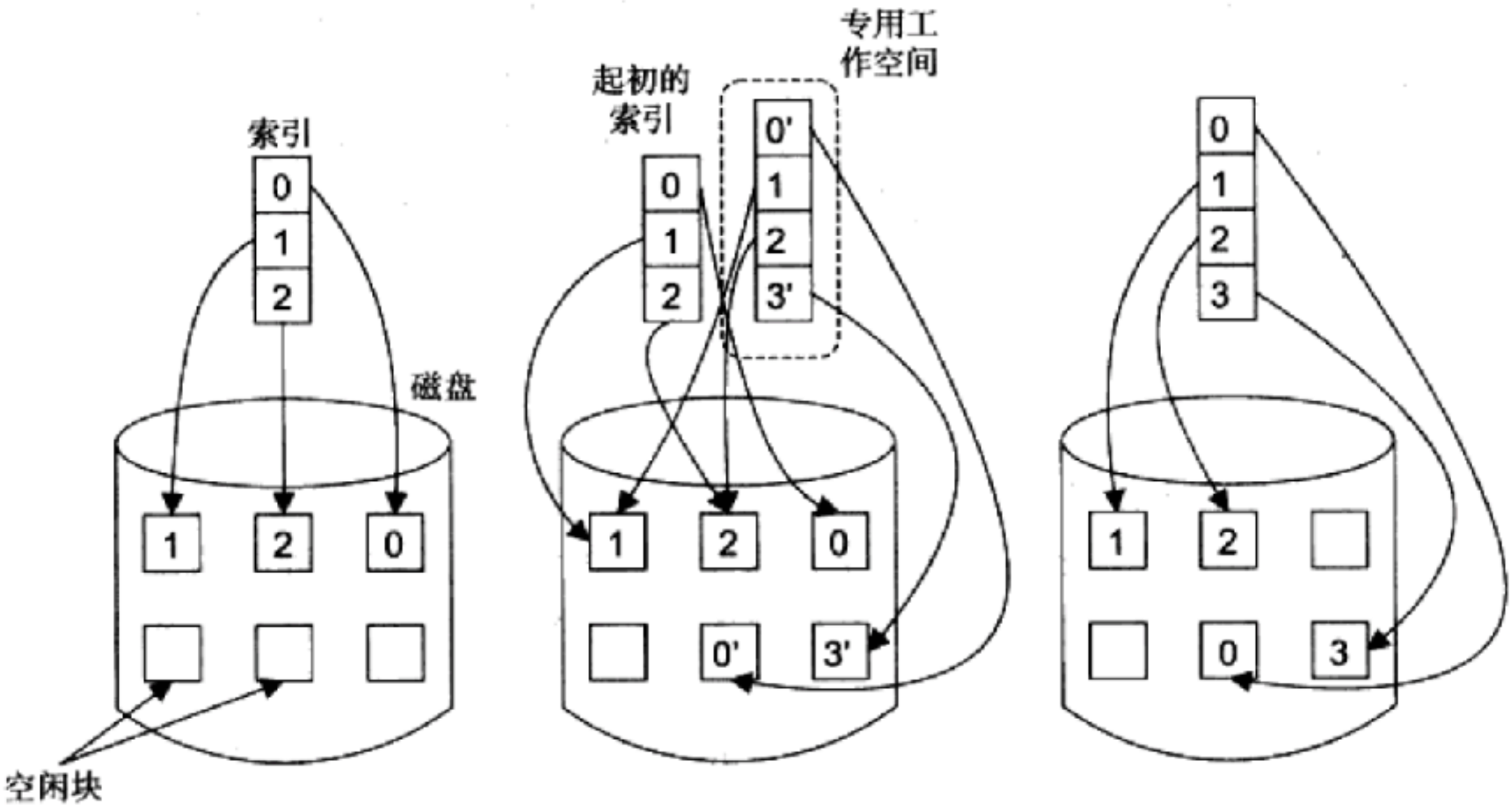
调度1	x=0; x=x+1; x=0; x=x+2; x=0; x=x+3;	合法
调度2	x=0; x=0; x=x+1; x=x+2; x=0; x=x+3;	合法
调度3	x=0; x=0; x=x+1; x=0; x=x+2; x=x+3;	不合法

(d)

图 3.16　(a) ~ (c)三个事务；(d) 可能的调度

4. 原子事务
4.3 实现

- 方法1: 专用工作空间



4. 原子事务
4.3 实现

- 方法2: 写前日志

```
x=0;
y=0;
BEGIN_TRANSACTION
    x=x+1;
    y=y+2;
    x=y*y;
END_TRANSACTION
```

(a)

日志

x = 0/1

(b)

日志

x = 0/1
y = 0/2

(c)

日志

x = 0/1
y = 0/2
x = 1/4

(d)

图 3.18　(a) 一个事务; (b) ~ (d)每条语句执行前的日志

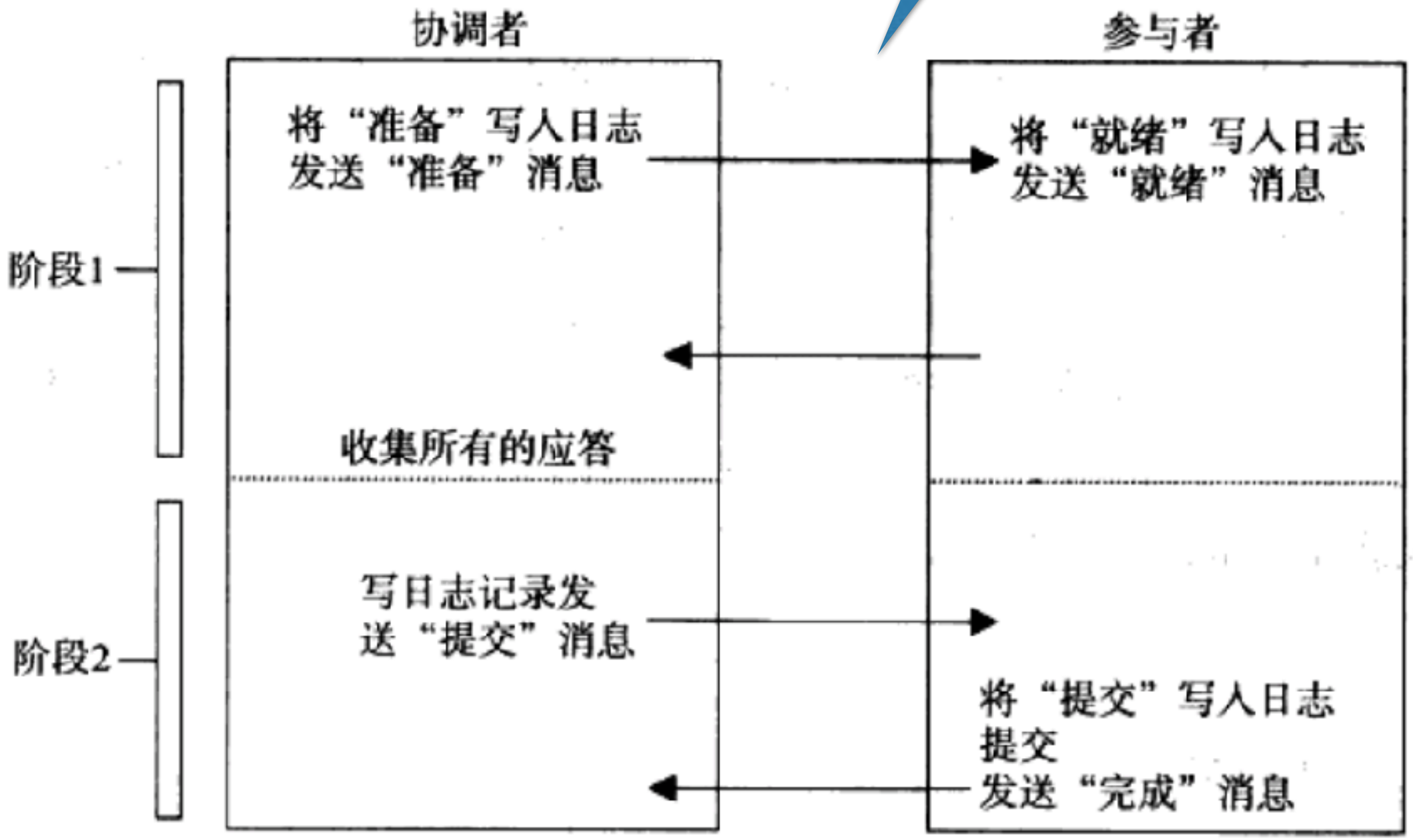
优点：　利于系统崩溃时的恢复

4. 原子事务

4.3 实现

保证分布式系统中的原子性

- 方法3: 两阶段提交协议



4. 原子事务
4.4 并发控制

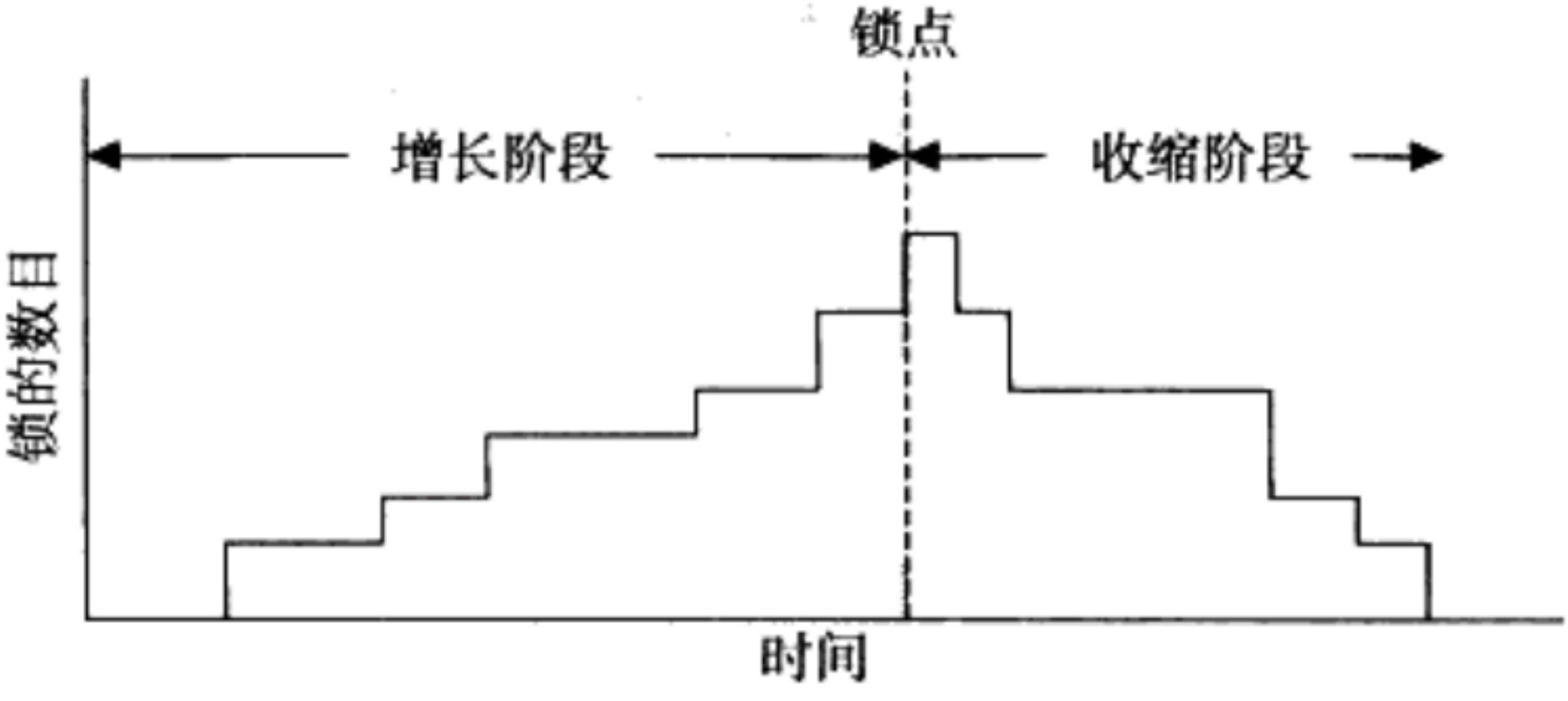
- 当**多个事务**在不同的进程（在不同的处理机上）中**同时执行**时
  - 需要一些机制以保证它们**互不干扰**。
  - 称为**并发控制算法**
- 方法
  - （1）加锁法 （2）乐观并发控制 （3）时间戳

4. 原子事务
4.4 并发控制

- （1）加锁法
  - 当一个进程需要读或写一个文件时，首先给文件加锁
    - 注：锁一般由事务系统管理，无需编程人员操作
- 许多用加锁实现的事务都用**两阶段加锁法**

4. 原子事务
4.4 并发控制

- 两阶段加锁



4. 原子事务
4.4 并发控制

- （2）乐观并发控制
  - 思想（许多政治家的策略）：
    - 尽管去做，不用在一其他人在做什么；
    - 如果有问题出现，则再解决
- 适用于基于**专用工作空间**的情况

4. 原子事务
4.4 并发控制

- （3）时间戳
  - 在BEGIN_TRANSACTION的时候分配一个时间戳
    - 使用Lamport算法，确保时间戳是唯一的
  - 如果事务都很短小且时间间隔大，
    - 则已发生进程的文件读写的时间的时间戳低于当前事务的时间戳

4. 原子事务

4.4 并发控制

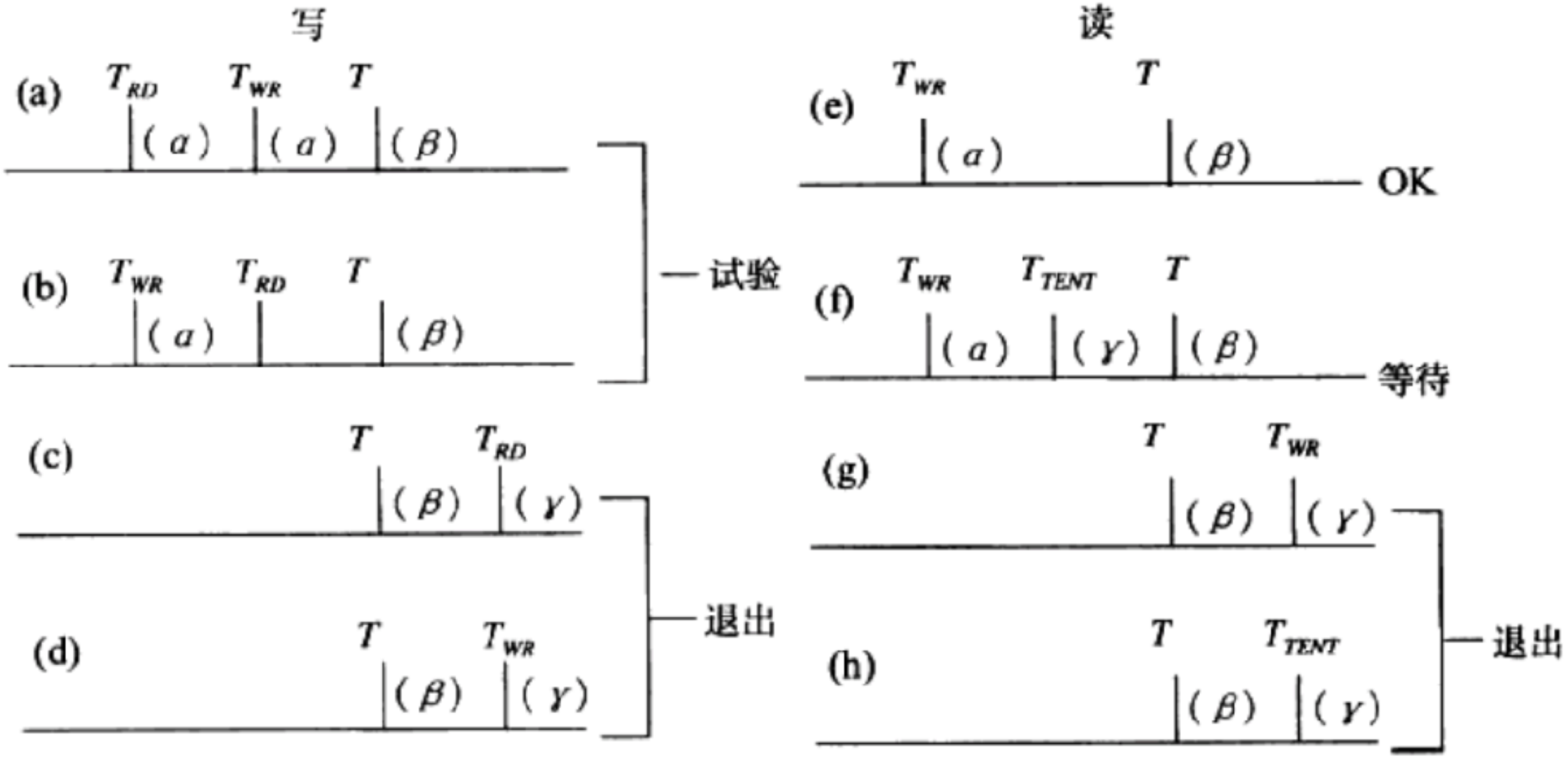


图 3.21　使用时间戳进行并发控制

5. 分布式系统中的死锁

- 分布式系统中的死锁
  - 类似于单处理机的死锁，但情况更糟
- 处理死锁的策略
  - 鸵鸟算法
  - 检测（允许死锁发生，之后恢复）
  - 预防 （静态判断）
  - 避免 （动态分配） （不现实，如银行家算法）

5. 分布式系统中的死锁

5.1 分布式死锁检测

- 死锁的预防和避免是**相当困难**的!
- 而死锁检测的代价是中止一些进程
  - 可使用**原子事务**来减小中止进程的后果
- 方法:
  - 集中式死锁检测
  - 分布式死锁检测

5. 分布式系统中的死锁
5.1 分布式死锁检测

- 集中式死锁检测
  - 使用**一台中心机器**维护全局的资源图
  - 问题： 需要维护全局时间
    - 否则，会出现假死锁的情况

5. 分布式系统中的死锁

5.1 分布式死锁检测

- 集中式死锁检测

Lamport算法可以解决，但开销太大

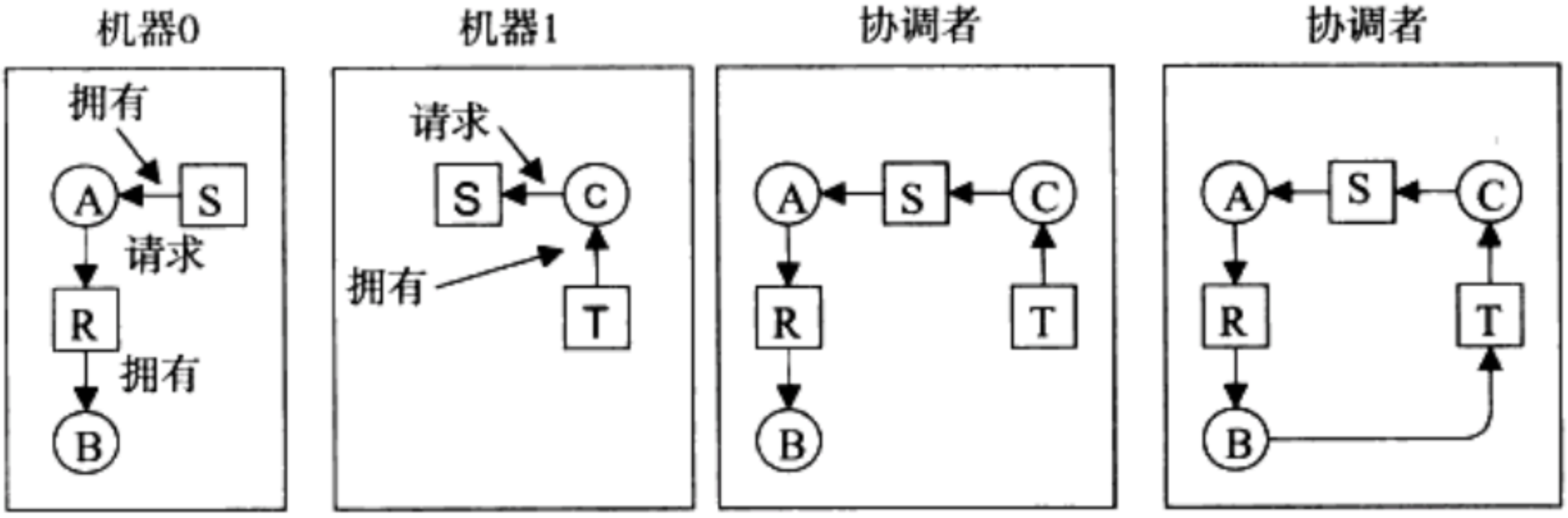


图 3.22　(a) 机器0初始资源图; (b) 机器1初始资源图; (c) 协调者对系统的观察; (d) 延迟信息后的系统情况

5. 分布式系统中的死锁
5.1 分布式死锁检测

- 分布式死锁检测
  - 典型的算法：Chandy-Misra-Haas算法
    - K. Mani Chandy, Jayadev Misra, Laura M. Haas: Distributed Deadlock Detection. *ACM Trans. Comput. Syst.* 1(2): 144-156 (1983) (被引用次数：523)
  - 特点：
    - 允许进程一次请求多个资源
    - 允许多个请求同时进行

5. 分布式系统中的死锁

5.1 分布式死锁检测

- Chandy-Misra-Haas算法

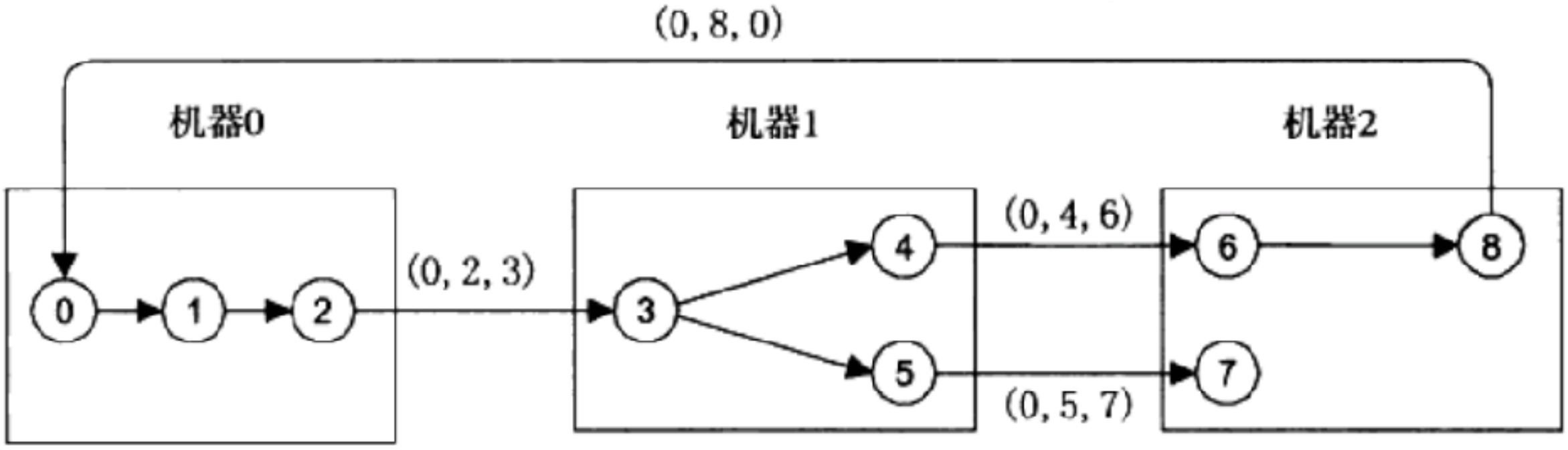


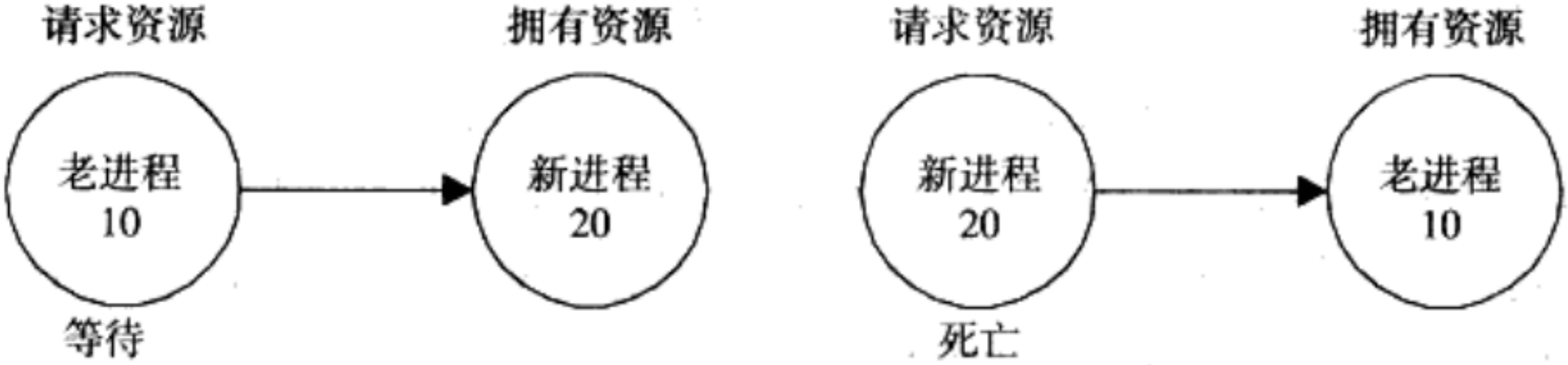
图 3.23　Chandy-Misra-Haas 分布式死锁检测算法

5. 分布式系统中的死锁
5.2 分布式死锁的预防

- 使用死锁预防的两个前提条件（需求）：
    - 全局时间戳
        - 判断新老进程
    - 原子事务
        - 减少中止进程造成的影响

5. 分布式系统中的死锁
5.2 分布式死锁的预防

- 算法1: 等-死死锁预防算法

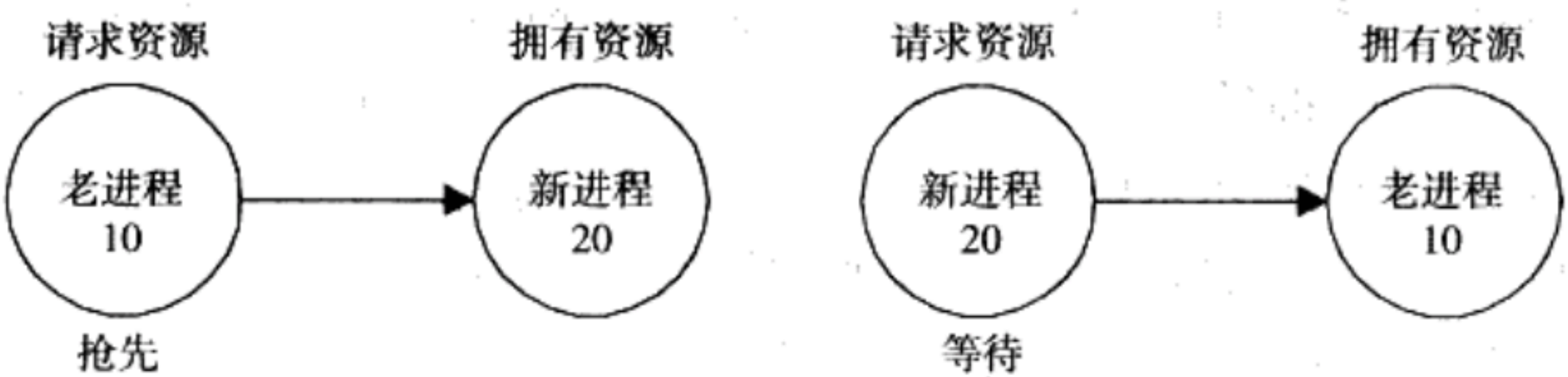


沿着箭头方向时间戳不断增大

问题：新进程会不断重启

5. 分布式系统中的死锁
5.2 分布式死锁的预防

- 算法2: 伤-等死锁预防算法



沿着箭头方向时间戳不断减小