



分布计算系统

徐高潮 胡亮 鞠九滨

高等教育出版社

第一章 绪论

1.1 为什么需要分布计算系统？

- ❖ 促进分布计算系统发展的两大技术：
 - 计算机硬件技术和软件技术的发展；
 - 计算机网络技术的发展。
- ❖ 两大技术改变了人们使用计算机的方式：
 - 50年代，预约上机，占用全部资源；
 - 60年代，批处理技术；
 - 70年代，分时系统，多用户同时使用一台计算机；
 - 80年代，个人计算机，每个用户有专用计算机；
 - 90年代至今，通过计算机网络使用多台计算机。

第一章 绪论

1.1 为什么需要分布计算系统？

❖ 多计算机系统环境带来的新问题：

- 在使用上，用户必须知道本地对象和远程对象的区别；
- 在管理上，管理人员不能四处奔走进行文件备份等操作。

❖ 解决上述新问题的方法是实现分布式操作系统

- 在分布计算系统中，多台计算机构成一个完整的系统，其行为类似一个单机系统。分布式操作系统是实现分布计算系统的核心。

第一章 绪论

1.2 分布计算系统的相关概念

❖ 什么是分布计算系统

➤ Andrew S. Tanenbaum教授给出的定义：分布计算系统是由多个独立的计算机系统相互连接而成的计算系统，从用户的角度来看它好像是一个集中的单机系统。

➤ 本文总结的定义：分布计算系统是由多个相互连接的处理资源组成的计算系统，它们在整个系统的控制下可合作执行一个共同的任务，最少依赖于集中的程序、数据和硬件。这些处理资源可以是物理上相邻的，也可以是在地理上分散的。

第一章 绪论

1.2 分布计算系统的相关概念

➤ 分布计算系统定义的说明：

- 1) 系统是由多个处理器或计算机系统组成。
- 2) 两类结构：这些计算资源可以是物理上相邻的、由机器内部总线或开关连接的处理器，通过共享主存进行通信；这些计算资源也可以是在地理上分开的、由计算机通信网络(远程网或局域网)连接的计算机系统，使用报文(message)进行通信。
- 3) 这些资源组成一个整体，对用户是透明的，即用户使用任何资源时不必知道这些资源在哪里。
- 4) 一个程序可分散到各个计算资源上运行；
- 5) 各个计算机系统地位平等，除了受全系统的操作系统控制外，不存在主从控制和集中控制环节。

第一章 绪论

1.2 分布计算系统的相关概念

❖ 紧密耦合与松散耦合分布计算系统

➤ 紧密耦合分布计算系统

- 1) 连接方式：内部总线或机器内互连网络；
- 2) 处理资源间距离：物理上分散，相距很近；
- 3) 处理资源：处理机；
- 4) 通信方式：共享存储器。

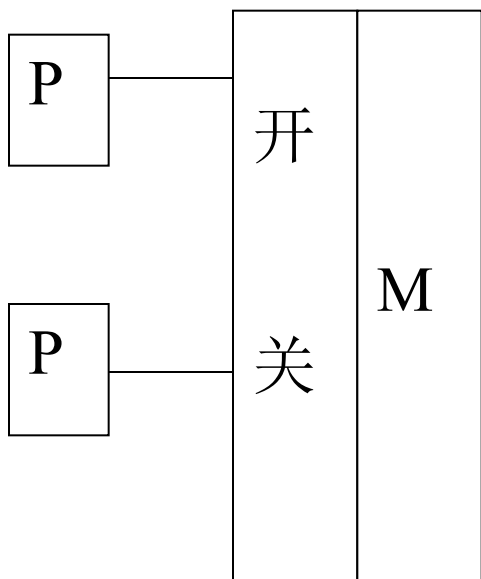
➤ 松散耦合分布计算系统

- 1) 连接方式：通信网络；
- 2) 处理资源间距离：地理上分散，相距很远；
- 3) 处理资源：计算机系统；
- 4) 通信方式：报文交换。

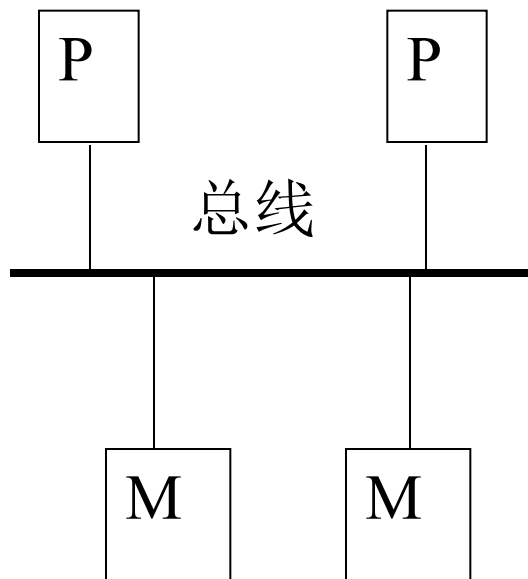
第一章 绪论

1.2 分布计算系统的相关概念

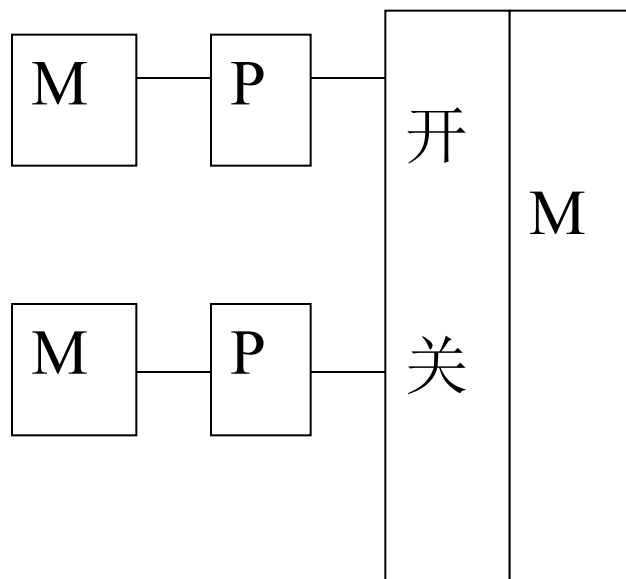
➤ 紧密耦合分布计算系统结构：



(a) 开关互连, 既有共享存储器



(b) 总线互连, 既有共享存储器



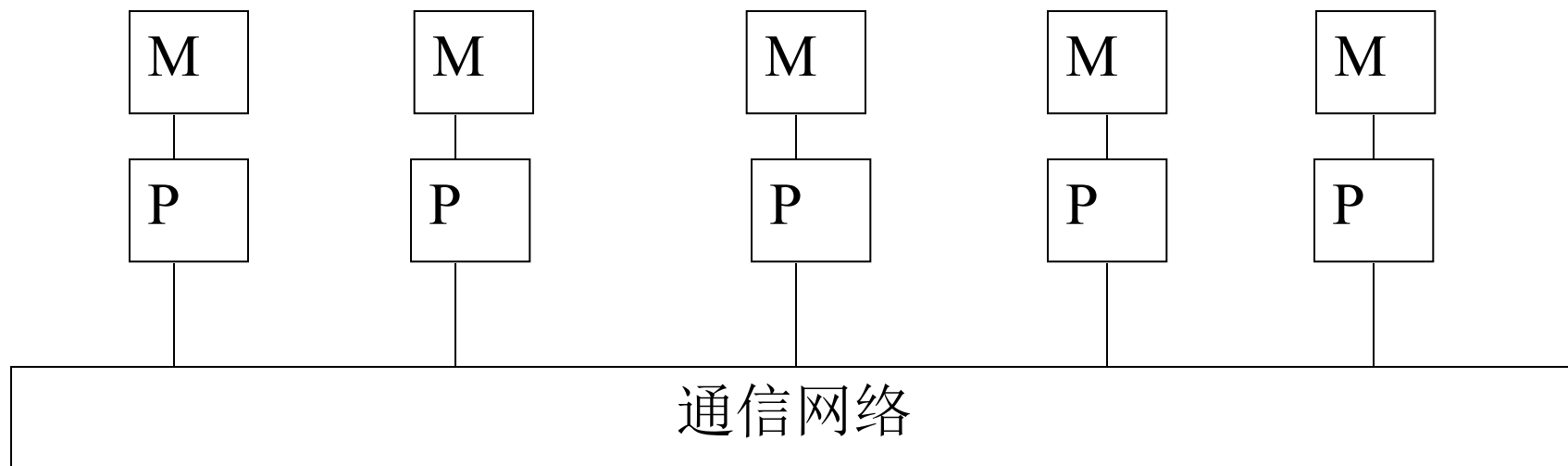
(c) 开关互连, 既有共享存储器, 又有专用存储器

P: 处理机, M: 主存储器

第一章 绪论

1.2 分布计算系统的相关概念

➤ 松散耦合分布计算系统结构：



P：处理机， M：主存储器

第一章 绪论

1.2 分布计算系统的相关概念

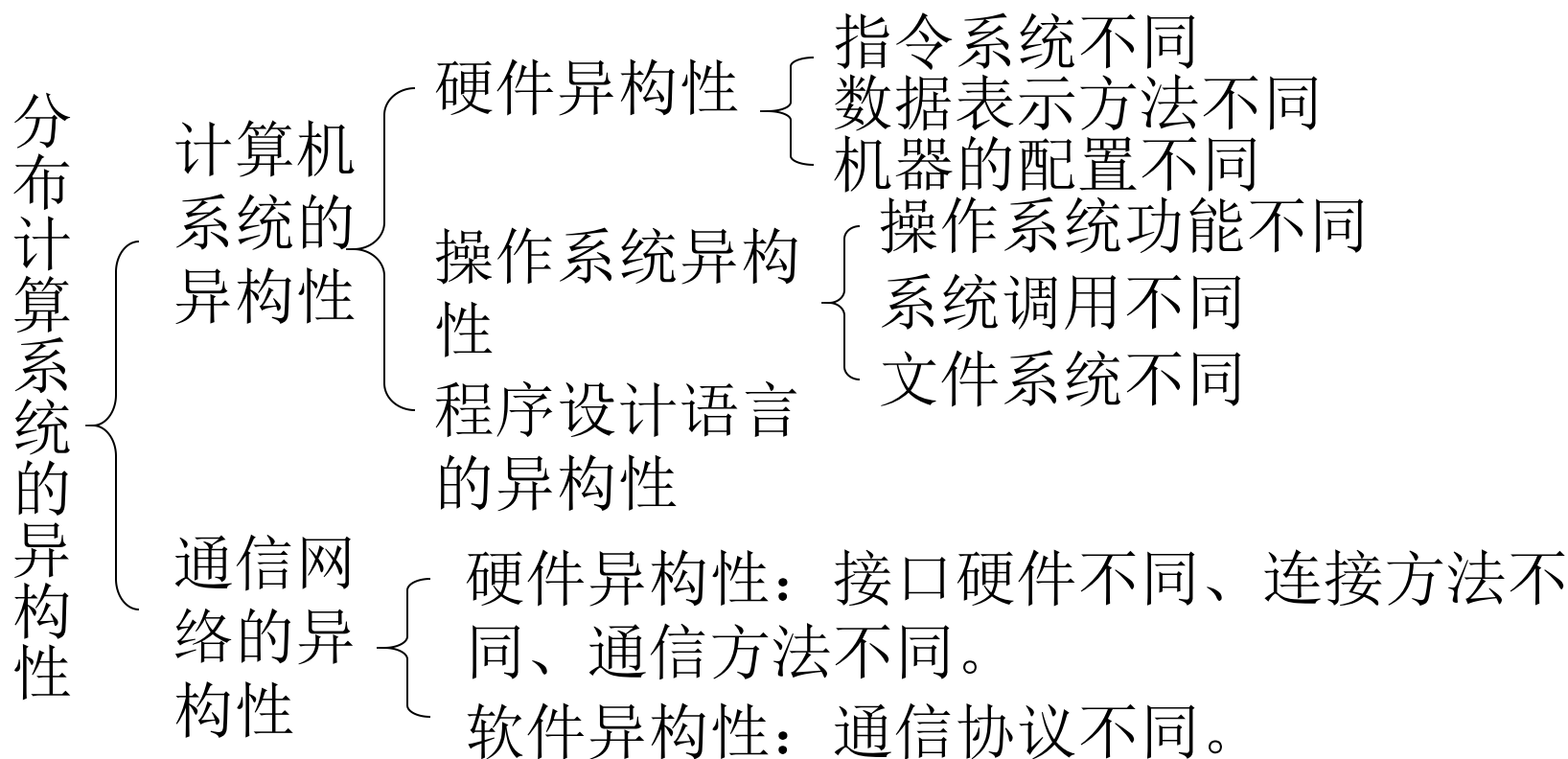
❖ 同构型与异构型分布计算系统

- 对于同构型分布式系统而言，组成该系统的计算机的硬件和软件是相同的或非常相似的，同时组成该系统的计算机网络的硬件和软件也是相同的或非常相似的。
- 对于异构型分布式系统而言，组成该系统的计算机的硬件或软件是不同的，或者组成该系统的计算机网络的硬件或软件也是不同的。

第一章 绪论

1.2 分布计算系统的相关概念

➤ 分布计算系统的异构性 的表现:



第一章 绪论

1.2 分布计算系统的相关概念

➤为什么分布计算系统的异构性是不可避免的？

- 1) 分布计算系统已成为资源共享的重要形式。随着分布计算系统资源的增多，其他用户也希望加入系统，共享其资源。这些新的系统往往同原有系统中现有的硬件和软件不同。
- 2) 由于硬件性能的提高和其价格的下降，当扩充一个分布计算系统时，人们往往会选择新型的计算机系统，而不是选择已有的设备类型。
- 3) 把不同的硬件和软件系统结合在一起，整个系统可以得到较高的性能价格比。在这样的系统中，如果配置一些专门为某种目的设计的具有特殊系统结构的处理器，则普通的计算机也可以共享这些功能。

第一章 绪论

1.3 分布计算系统的优点和新问题

❖ 分布计算系统的优点

- 可扩充性。不必像单机系统那样替换整个系统，分布计算系统容易通过扩大规模以包括更多的资源。
- 高的性能/价格比。在分布计算系统中，可以通过增加计算机的数目，提高并行程度而得到所需要的性能，从而可以获得很高的性能/价格比。
- 资源共享。系统中的硬件和软件资源如外部设备、文件系统和数据库等可以被更多的用户所共享，甚至连CPU和内存等资源也可被共享。
- 可靠性。分布计算系统具有在系统中当某个部分出现了故障的情况下继续运行的潜力。
- 支持固有的分布式应用。分布计算系统与许多应用场合相适应，如银行、铁路等本来就分散而又必须相互协调的行业。

第一章 绪论

1.3 分布计算系统的优点和新问题

❖ 分布计算系统的新问题

➤ 资源的多重性带来的问题。由于处理资源的多重性，分布计算系统可能产生的差错类型和次数都比集中式单机系统多。

资源多重性还给系统资源管理带来新的困难。

➤ 资源的分散性带来的问题。分布式的状态信息和不可预知的报文延迟使得系统的控制和同步问题变得很复杂，要想及时地、完整地搜集到系统各方面的信息是很困难的，从而使处理机进行最佳调度相当困难。

➤ 系统的异构性带来的问题。在异构性分布计算系统中，由于各种不同资源(特别是计算机和网络)的数据表示和编码、控制方式等均不相同，这样一来就产生了翻译、命名、保护和共享等新问题。

第一章 绪论

1.4 分布计算系统的透明性

❖ 透明性的概念

- 透明性：事物本来存在某种属性，但是这种属性从某种角度上来看是不可见的，称这种特性为透明性。
- 分布计算系统的透明性：用户或程序员看不见网络的存在。这样从用户或程序员的角度看来，网络中的全部机器表现为一个，用户或程序员看不到机器的边界和网络本身。用户不必知道数据放在什么地方以及进程在何处执行。
- 分布计算系统的透明性表现：

(1) 访问透明性：透明性是指用户或程序员在访问网络中的资源时，不需要知道资源的物理位置，也不需要知道资源的物理结构。透明性是指用户或程序员在访问网络中的资源时，不需要知道资源的物理位置，也不需要知道资源的物理结构。

第一章 绪论

1.4 分布计算系统的透明性

➤ 分布计算系统具有透明性时有以下一些优点：

- 1) 使软件的研制变得容易，因为访问资源的方法只有一种，软件的功能与其位置无关。
- 2) 系统的某些资源变动时不影响或较少影响应用软件。
- 3) 系统的资源冗余(硬件冗余和软件冗余)使操作更可靠，可用性更好。透明性使得在实现这种冗余的时候，各种冗余资源的互相替换变得容易。
- 4) 在资源操作方面，当把一个操作从一个地方移到若干其他地方时没有什么影响。

第一章 绪论

1.4 分布计算系统的透明性

❖影响透明性的因素

➤系统的异构性对透明性的影响：不同方法实现的异构性系统具有不同的透明性。

2a) 基于不同语言的程序。如果在一个系统中，每个程序都是用不同的语言编写的，那么，这个系统就不具有透明性。因为，不同的语言有不同的语法规则和语义，这导致程序之间的交互和通信变得非常复杂。在这种情况下，用户无法直接看到程序的内部结构，也无法理解程序的运行逻辑，因此，系统不具有透明性。

第一章 绪论

1.4 分布计算系统的透明性

➤局部自治性对透明性的影响：

分布计算系统由分散在各地点的一批计算机组成，这些地点可能希望保持对处在该地点的机器的控制权，这种局部自治性限制了全局透明性。

- 1) 资源控制方面。由分布计算系统连接的各机器是由不同的用户所操作，或由一个机关的不同部门控制，希望在资源的使用上有较大的控制权。因此，必须有一种手段解决这个问题，在透明性和局部自治性之间进行折衷。
- 2) 命名方面。即使同型号的机器，不同的用户也可能以不同的方式形成他们的目录，例如FORTRAN程序库在某个机器上使用某个名字，而在另一台机器上可能使用另一个名字，造成不透明。

第一章 绪论

1.4 分布计算系统的透明性

➤网络互连对透明性的影响：

- 1) 很多网络连接了不同厂商提供的不同系列的计算机，要实现透明性必须修改这些机器的软件，这是很不实际的。
- 2) 现在的网络一般是直接从早期网络结构发展来的，其最重要的功能是通信，并未考虑到分布计算。
- 3) 远程网络一般是很昂贵的资源，其特征是低带宽或高延迟，或者二者兼有，因此，很多人认为应当使这种资源的应用成为可见的(不透明的)。

第一章 绪论

1.5 分布计算系统与计算机网络系统

❖ 分布计算系统和计算机网络系统有什么区别呢？

➤ 如果用户能说明他在使用哪一个计算机，则他是在使用一个计算机网络系统而不是分布式系统。一个真正的分布计算系统的用户不必知道他的程序在哪个机器上运行，他的文件在哪里存放，等等。使分布计算系统具有这种性质的是它的软件：分布式操作系统。

➤ 从计算机网络系统上所运行的操作系统软件来分，计算机网络系统的发展可分为三个阶段：无操作系统阶段、运行网络操作系统阶段和运行分布式操作系统阶段。

第一章 绪论

1.5 分布计算系统与计算机网络系统

➤ 网络操作系统一般具有以下特点：

- 1) 每个计算机都运行自己的操作系统，而不是运行共同的、全系统范围的操作系统或其一部分。
- 2) 每个用户通常在自己的计算机上以“远程登录”的方式或其他明确指出的方式使用不同的机器，而不是由系统给用户进程分配计算机，因而不能并行执行某个程序。
- 3) 用户知道他们的文件存放在哪里，在机器之间移动文件时必须明确地使用“文件传送”命令。
- 4) 系统没有容错能力，或者仅具有很少的容错能力。

第一章 绪论

1.5 分布计算系统与计算机网络系统

➤ 分布计算系统应达到的目标：

- 1) 程序(进程)、终端用户或程序员对全部分布资源应该有一个唯一的连贯的观点，不必明确地知道所需资源是在本地、远程或是分散的，主机之间的边界应尽可能隐匿。
- 2) 在性能方面，NOS结构的实现是有效的、可用的。本地用户进程访问本地服务时应象单机操作系统一样有效，不增加额外数目和类型的报文或系统调用。
- 3) 可扩充性。用户很容易在现存服务上增加新的服务而不必要求系统程序员增加新的驻存程序，正如在单机系统上增加新的服务不必要求修改现有的操作系统。

第一章 绪论

1.5 分布计算系统与计算机网络系统

❖ 区分计算机网络系统与分布计算系统

➤ 从文件系统的访问方法上区分：

- 1) 没有操作系统的访问方法。计算机A上的程序将B上的文件复制到A上来，然后再在A上访问此文件。
- 2) 有网络操作系统的访问方法。这种方法是把不同的文件系统连接起来，一个机器上的程序可以使用路径名打开另一个机器上的文件，只是这个路径名中包含了另一个机器的信息。

或如：`open(“/machine_name/pathname”, READ);`
或：`open(“../machine_name/pathname”, READ);`

- 3) 分布计算系统使用的方法。在这种方法中，所有各子系统的文件系统组成一个整体文件系统。

第一章 绪论

1.5 分布计算系统与计算机网络系统

➤从访问控制方面区分

UNIX和其他许多操作系统给每个用户赋予一个唯一的内部标识符（UID），以利于访问控制。

- 1) 没有网络操作系统下的情况。这种办法要求所有要访问机器X上的文件的用户先使用属于机器X的用户名在机器X上登录。
- 2) 有网络操作系统下的情况。在这种办法中，由网络操作系统对不同机器上的UID进行变换。
- 3) 在分布式操作系统下的情况。在分布计算系统中，对每个用户只设一个UID，使用它可以访问任何机器，不必经过变换。

第一章 绪论

1.5 分布计算系统与计算机网络系统

➤是否区分本地执行和远程执行方面判断：

- 1) 在没有网络操作系统的计算机网络系统中，用户要远程执行一个程序时，该用户先远程登录到一个远程机器上，然后在那里运行作业。
- 2) 在有网络操作系统的计算机网络系统中，用户在自己的终端上输入一个特殊的命令，指定一个机器运行一个程序。
如：`remote vax4 who`，这个命令是让远程计算机vax4运行程序who。
- 3) 在分布计算系统中，用户执行一个程序时，只需简单的给出要执行的程序的程序名和相关的参数，并不指出在何处执行这个程序，何处执行由操作系统决定。

第一章 绪论

1.6 分布计算系统的体系结构与设 计

❖ 分布计算系统的分层体系结构

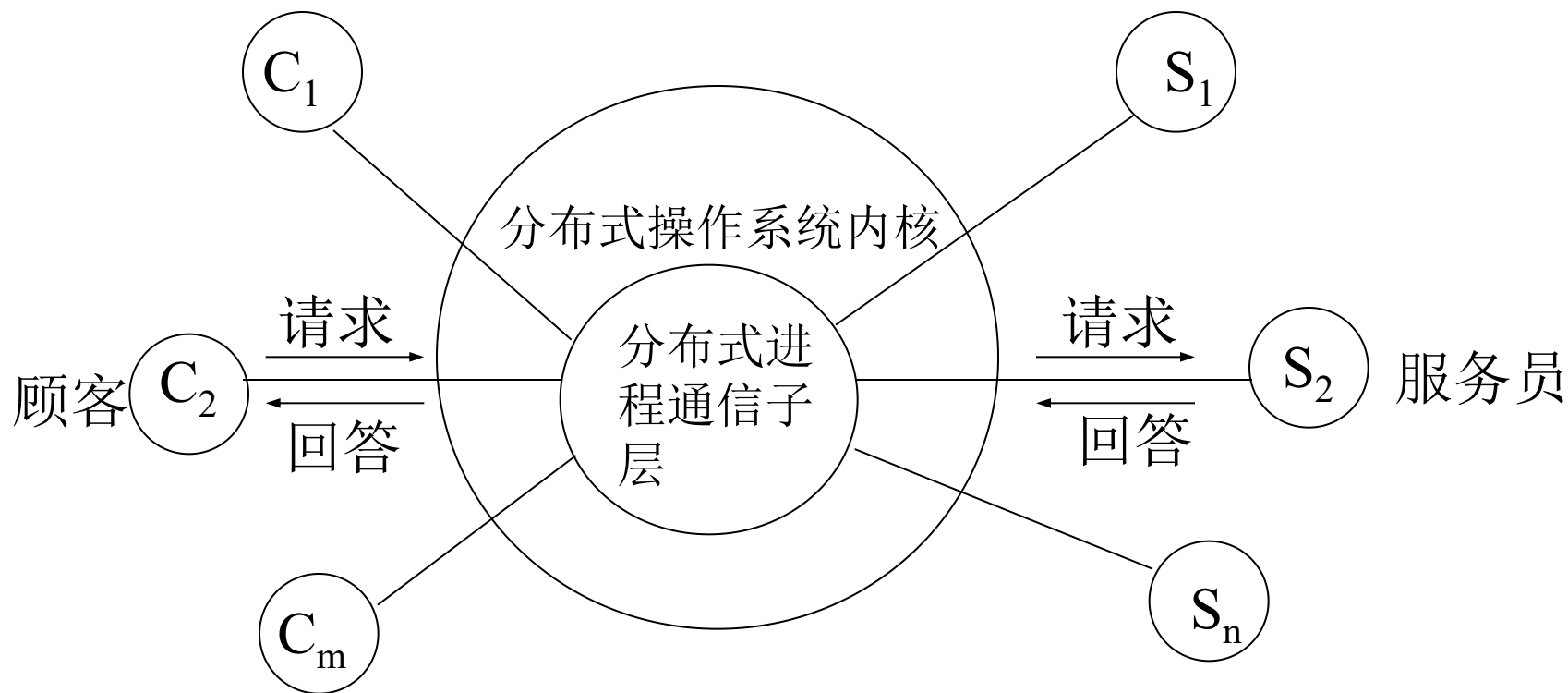
➤ 分布计算系统可以分成若干逻辑层，层与层之间称为接口，每层有两个接口。层所提供的功能还可进一步分割成若干模块，模块之间也有接口。接口由以下三部分组成：

- 1) 一套可以见到的抽象对象(资源)以及对这些对象所要求的操作和参数。
- 2) 一套控制这些操作的合法顺序的规则。
- 3) 操作和参数所需要的编码和格式化约定。

第一章 绪论

1.6 分布计算系统的体系结构与设

➤ 面向进程和报文传递的分布计算系统模型



第一章 绪论

1.6 分布计算系统的体系结构与设

计

❖ 分布计算系统的组成

分布计算系统由四层组成：第一层是由硬件或固件组成的硬核，第二层是分布式操作系统的内核，第三层是分布式操作系统的服务层，第四层是和用户有关的应用层。

第一章 绪论

1.6 分布计算系统的设计与设

❖ 分布计算系统的组成

硬件/固件层(硬核)。该层包括处理器、主存、I/O设备、键盘终端以及用于数据集和物理过程控制的各种硬件设备。

内核的最基本最重要的功能是进程通信(IPC)，除此之外，还包括进程的同步机制、进程管理、存储管理和I/O管理等功能。

这一层包括和各种应用有关的顾客服务进程。要考虑的主要问题有两方面：应用结构问题和语言问题。

服务层

应用

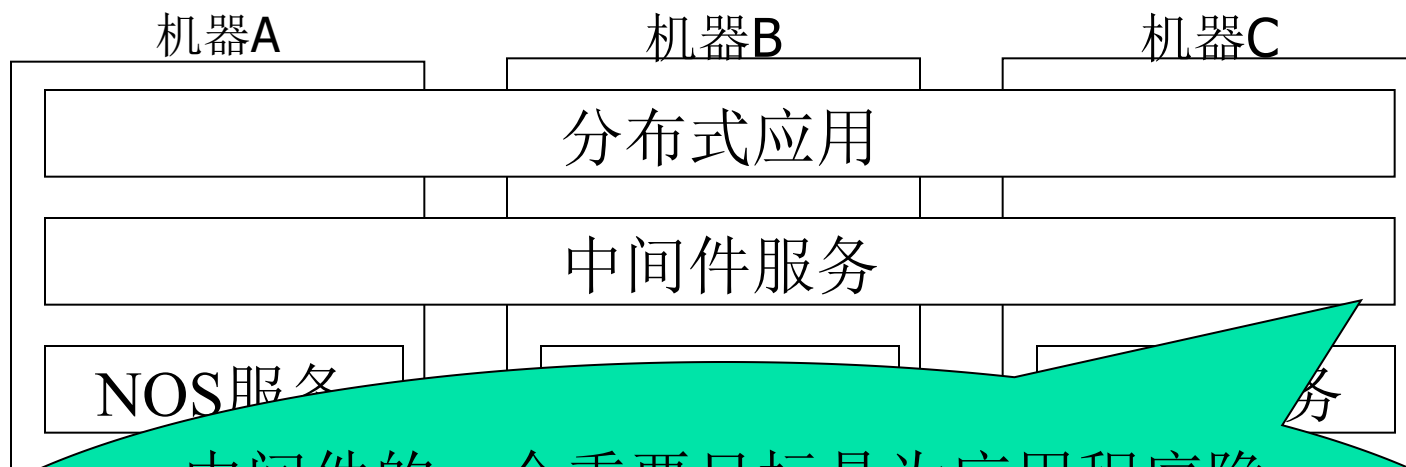
内核

网络

第一章 绪论

1.6 分布计算系统的体系结构与设

❖ 基于中间件的分布计算系统



中间件的一个重要目标是为应用程序隐匿底层平台的异构性。许多中间件系统提供某种程度的全局服务集成，并且只为用户或应用程序提供使用这些服务的接口。

网络

第一章 绪论

1.6 分布计算系统的体系结构与设

➤中间件模型一般包括如下一些类型：

- 1) 将任何资源作为文件来对待。如在Plan 9中，所有的资源，包括键盘、鼠标、硬盘、网络接口等等这些I/O设备都被当作文件看待。无论一个文件是远程的还是本地的，在本质上是没有任何区别的。因为一个文件能够被几个进程共享，进程通信可以简化到对同一个文件访问的问题。
- 2) 以分布式文件系统(DFS—Distributed File Systems)为中心的中间件模型。这种模型类似于第一种模型，但并不像Plan 9那样严格。在许多情况下，这种中间件实际上只在支持传统文件的分布透明性方面比网络操作系统前进了一步。

第一章 绪论

1.6 分布计算系统的体系结构与设

➤ 中间件模型一般包括如下一些类型：

- 3) 基于远程过程调用(RPC—Remote Procedure Call)的中间件模型。这种模型主要集中在隐匿网络通信，隐匿网络通信的方式是允许一个进程调用在一个远程机器上实现的过程。这个过程调用似乎就发生在本地，调用进程不必关心发生网络通信。
- 4) 基于分布式对象(Distributed Objects)的中间件模型。一个分布式对象往往是在拥有该对象的一个机器上实现，而它的接口在许多其他的机器上可用。当一个进程引用一个方法时，进程所在机器上的接口将此引用转换成一个报文传送给对应的对象，该对象执行所请求的方法并回送结果。同RPC的情况一样，进程完全可以不关心网络通信的细节。

第一章 绪论

1.6 分布计算系统的体系结构与设 计

➤中间件模型一般包括如下一些类型：

- 5) 基于分布式文档(Distributed Documents)的中间件模型。
如在Web模型中，信息被组织成文档，每个文档透明地存放在某个机器上，文档里包含有一些指向其他文档的链接(link)。通过一个链接，这个链接所指定的文档会被从它存放的位置取到用户的机器上并显示到用户的显示器上。

第一章 绪论

1.6 分布计算系统的体系结构与设

❖ 分布计算系统的设计问题

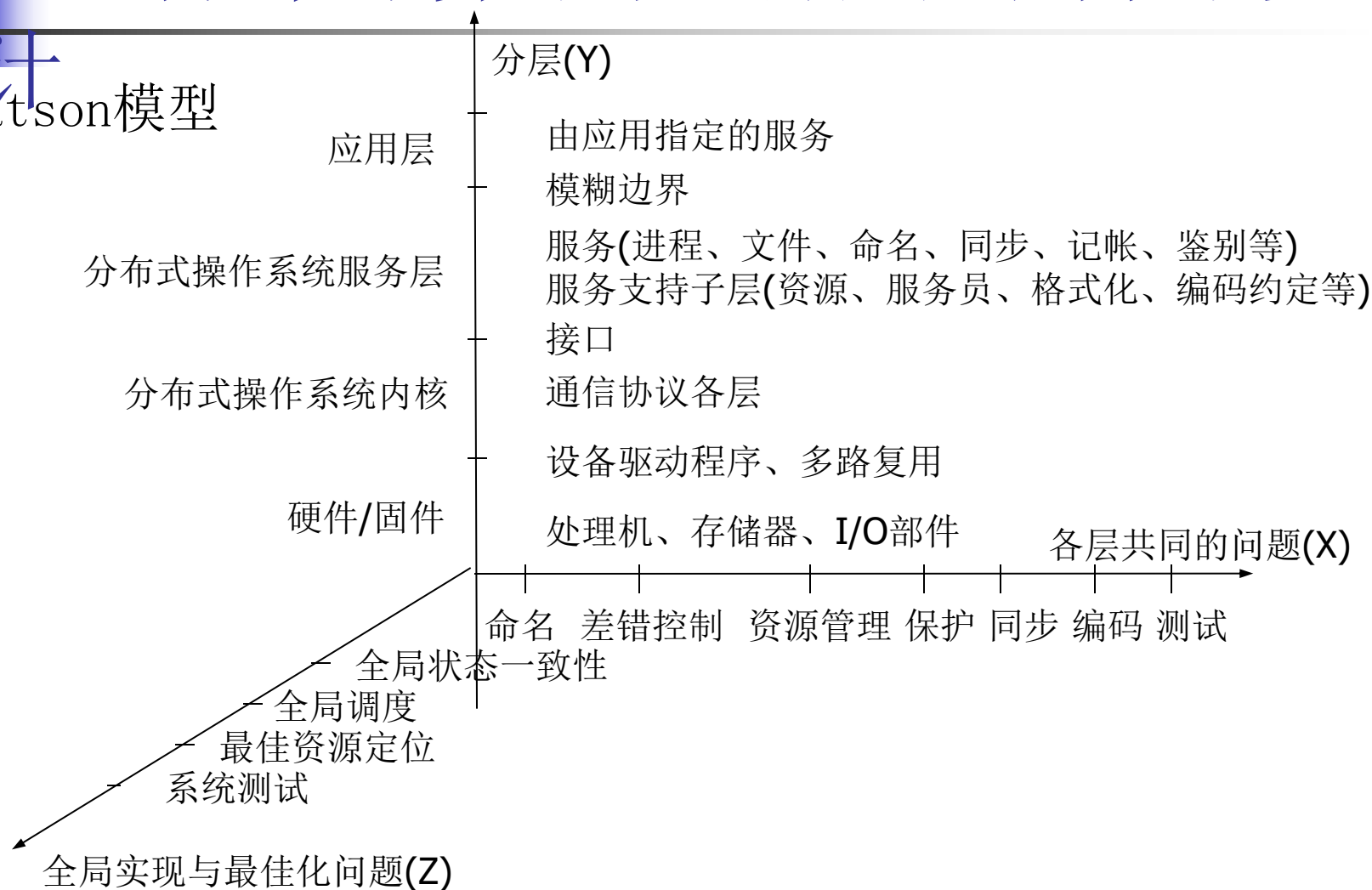
➤ 各层或很多层所共有的 设计问题：

- 1) 设计问题：如何将系统资源（如处理器、存储器、通信链路等）合理地分配到各个层或节点上，以满足系统性能要求。设计问题的复杂性随着系统规模的增大而增加，特别是在多用户、多任务、多资源的环境中，设计问题变得更加复杂。设计问题的解决需要综合考虑系统的性能、成本、可靠性、可扩展性等因素。

第一章 绪论

1.6 分布计算系统的体系结构与设

➤ Watson模型



第二章 进程通信

2.1 同一节点上的进程间通信

大多数UNIX系统提供多种进程通信方式，主要有信号(Signal)、管道(pipe)、命名管道(FIFO)、消息队列(message queue)、信号灯(semaphore)、共享内存(shared memory)和内存映象(memory mapped file)等。

❖管道

➤无名管道

无名管道只能在有亲缘关系的进程之间使用(例如父进程和子进程，子进程和子进程)，并且与创建无名管道的进程一起存在。

第二章 进程通信

2.1 同一节点上的进程间通信

无名管道的应用实例：

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#define MSGSIZE 16
```

```
char *msg1 = "hello, book #1";
```

```
char *msg2 = "hello, book #2";
```

```
char *msg3 = "hello, book #3";
```

第二章 进程通信

2.1 同一节点上的进程间通信

无名管道的应用实例：

```
main()
```

```
{
```

```
    char inbuf[MSGSIZE];
```

```
    int p[2], j;
```

```
    if(pipe(p) == -1)
```

```
    {
```

```
        perror("pipe call failed");
```

```
        exit(1);
```

```
    }
```

第二章 进程通信

2.1 同一节点上的进程间通信

无名管道的应用实例：

```
switch(pid=fork()) {  
    case -1:  
        perror("fork call failed");  
        exit(2);  
    case 0:  
        /* if child then write to pipe*/  
        write(p[1], msg1, MSGSIZE);  
        write(p[1], msg2, MSGSIZE);  
        write(p[1], msg3, MSGSIZE);  
}
```

第二章 进程通信

2.1 同一节点上的进程间通信

无名管道的应用实例：

default:

```
/*if parent then read from pipe*/  
for(j=0; j<3; j++)  
{  
    read(p[0], inbuf, MSGSIZE);  
    printf("%s\n", inbuf);  
}
```

```
wait(NULL);
```

```
exit(0);
```

```
}
```

第二章 进程通信

2.1 同一节点上的进程间通信

➤ FIFO或命名管道

命名管道作为拥有文件访问权限的目录入口而存在，所以它们可以在彼此无关的进程之间使用。

命名管道的创建与使用：

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
....
```

```
Mkfifo("/tmp/fifo", 0666);
```

```
.....
```

```
fd=open("/tmp/fifo", O_WRONLY);
```


第二章 进程通信

2.1 同一节点上的进程间通信

❖ 消息队列

队列中的消息能够以随意的顺序进行检索，因为可以通过消息的类型把消息从队列中检索出来。

➤ 创建和打开一个消息队列：

```
int open_queue(key_t key)
{
    int qid;
    if((qid=msgget(key,IPC_CREAT|0660))==-1)
    {
        return(-1);
    }
    return(qid);
}
```

第二章 进程通信

2.1 同一节点上的进程间通信

➤向队列中写入消息:

```
int send_message(int qid, struct msgbuf *qbuf)
{
    int result, length;
    length=sizeof(struct msgbuf)-size(long);
    if((result=msgsnd(qid,qbuf,length,0))==-1)
    {
        return(-1);
    }
    return(result);
}
```

第二章 进程通信

2.1 同一节点上的进程间通信

➤ 读取队列中的消息：

```
int read_message(int qid, long type, struct msgbuf *qbuf)
{
    int result, length;
    length=sizeof(struct msgbuf)-sizeof(long);
    if((result=msgrcv(qid, qbuf, length, type, 0))!=-1)
    {
        return(-1);
    }
    return(result);
}
```

第二章 进程通信

2.1 同一节点上的进程间通信

➤检查符合条件的消息是否到来：

```
int peek_message(int qid, long type)
{
    int result, length;

    if((result=msgrcv(qid,NULL,0,type,IPC_NOWAIT))==-1)
    {
        if(errno==E2BIG) return(TRUE);
    }
    return(FALSE);
}
```

第二章 进程通信

2.1 同一节点上的进程间通信

➤ 删除消息队列：

```
int remove_queue(int qid)
{
    if(msgctl(qid, IPC_RMID, 0)==-1)
    {
        return(-1)
    }
    return(0);
}
```

第二章 进程通信

2.1 同一节点上的进程间通信

❖共享内存

使用共享内存技术，不同进程可以同时访问相同的内存区域。能够通过共享它们地址空间的若干部分，然后对存储在共享内存中的数据进行读和写，从而实现彼此直接通信。

➤共享内存技术实现 进程通信

- 1) 由一个进程来创建 / 分配一个共享内存段，其大小和访问权限在其生成时设置。系统调用shmget建立一个新的共享内存区或者返回一个已经存在的共享内存区。
- 2) 进程可挂接此存储段，同时将其映射到自己当前的数据空间中。每个进程通过其挂接的地址访问共享内存段。系统调用shmat将一个共享内存区附加到进程的虚拟地址空间上。

第二章 进程通信

2.1 同一节点上的进程间通信

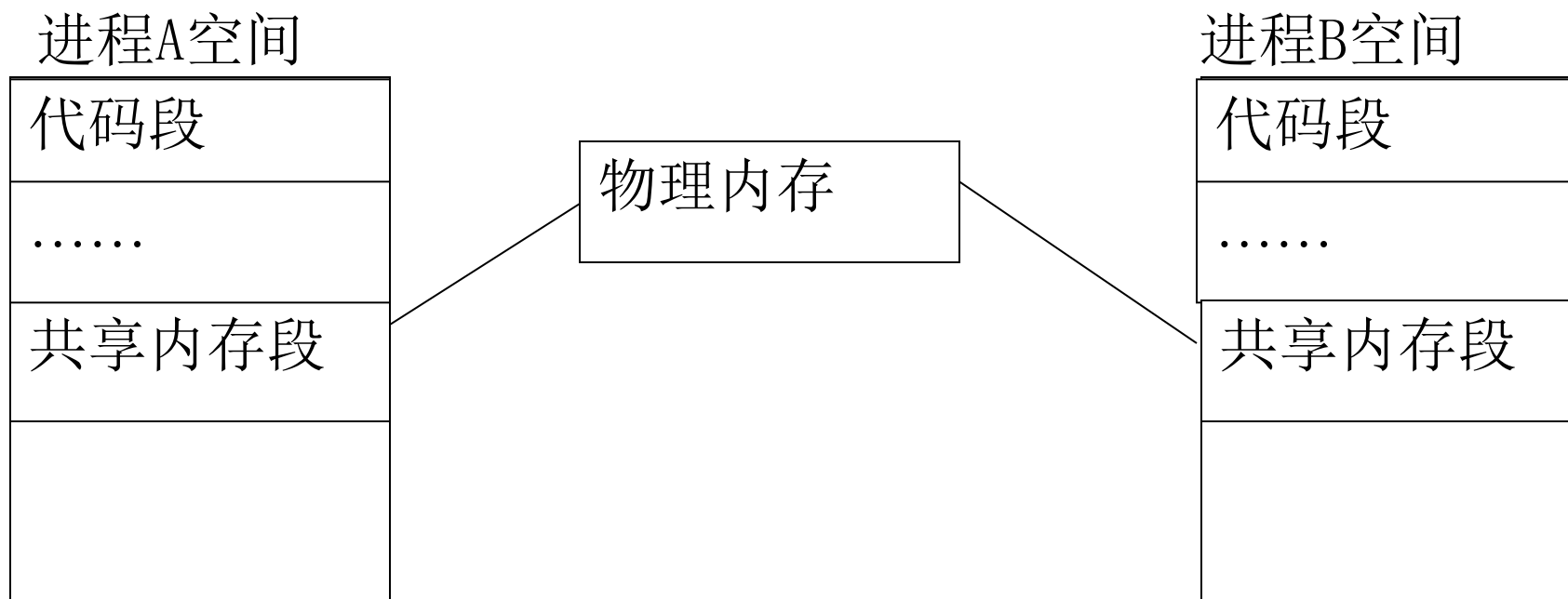
➤ 共享内存技术实现 进程通信

- 3) 当一个进程对共享内存段的操作已经结束时，可以从虚拟地址空间中分离该共享内存段。系统调用`shmdt`从进程的虚拟地址空间分离一个共享内存区。
- 4) 进程对该共享内存段进行操作。系统调用`shmctl`对与共享内存相关联的各种参数进行操作。
- 5) 当所有进程完成对该共享内存段的操作时，通常由共享内存段的创建进程负责将其删除。

第二章 进程通信

2.1 同一节点上的进程间通信

➤ 共享内存技术实现 进程通信



第二章 进程通信

2.1 同一节点上的进程间通信

➤共享内存的操作函数:

1) `int shmget(key_t key, int size, int shmflg);`

函数shmget通过key值获取或者创建一块共享内存。如果shmflg&IPC_CREAT不为0，则在共享内存不存在时将创建它，新创建的共享内存的大小由size指示。

2) `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`

函数shmctl可以用来查询一些共享内存的使用情况和进行一些控制操作。例如，`shmctl(shm_id, IPC_RMID, 0);`，这条语句将删除shm_id指示的共享内存。

第二章 进程通信

2.1 同一节点上的进程间通信

➤共享内存的操作函数:

3) `void *shmat(int shmid, const void *shmaddr, int shmflg);`

函数shmat将根据shmid得到指向共享内存段的指针, 即这个共享内存段被附加到进程虚拟地址空间中。进程可以像使用一般的指针一样使用这个指针, 例如进行强制

4) `int shmdt(const void *shmaddr);`

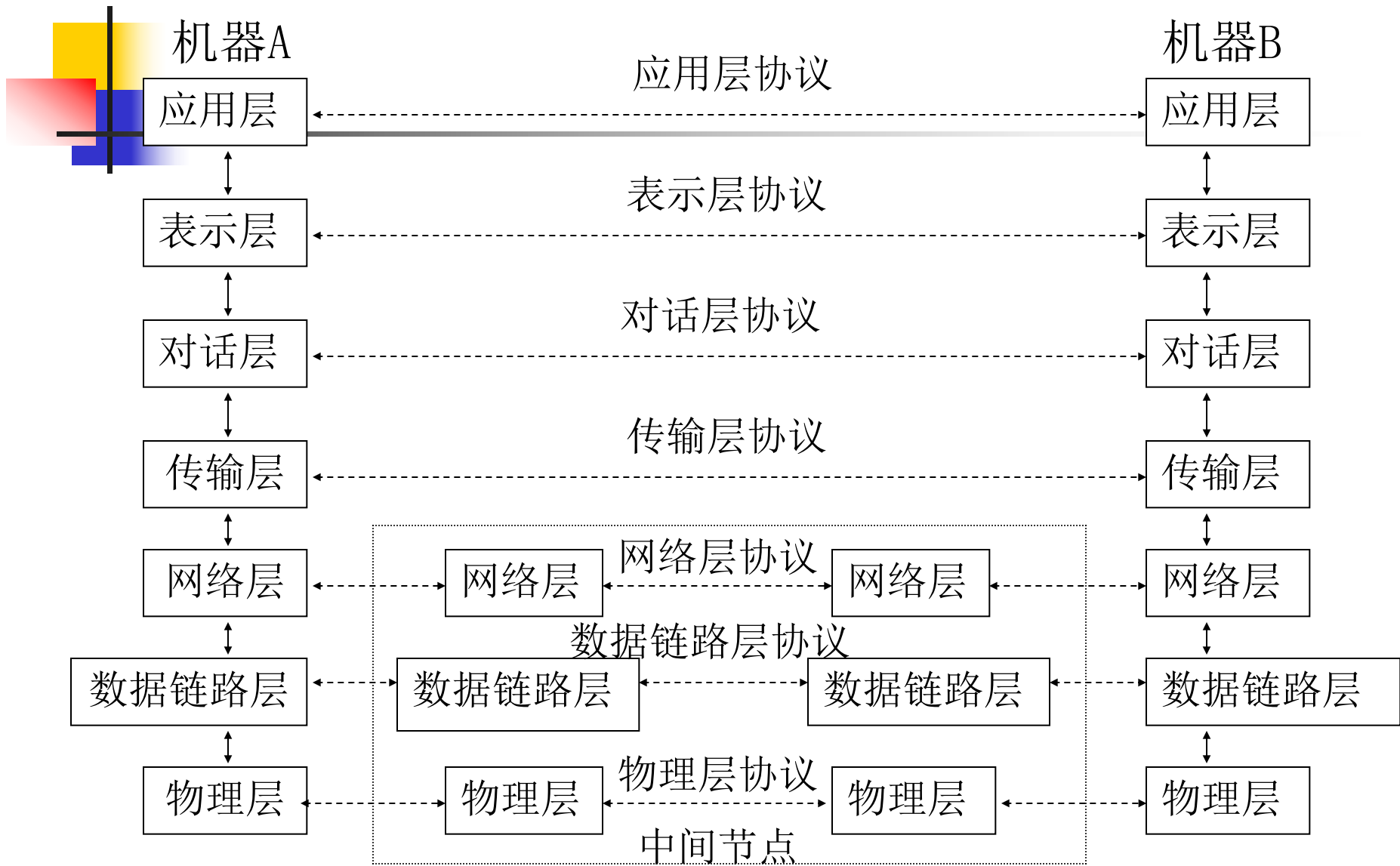
函数shmdt是将共享内存段从调用进程的虚拟地址空间中分离出去。参数shmaddr是通过函数shmat得到的指向共享内存的指针。

第二章 进程通信

2.2 不同节点上的进程间通信

❖ 网络通信分层结构模型

- ISO OSI/RM的分层与协议



第二章 进程通信

2.2 不同节点上的进程间通信

❖ 网络通信分层结构模型

- ISO OSI/RM数据传输方式

发送进程

Data

接收进程

应用层

应用层

表示层

表示层

对话层

对话层

传输层

传输层

网络层

网络层

数据链路层

数据链路层

物理层

物理层

发送主机

接收主机

AH

Data

PH

AH

Data

SH

PH

AH

Data

TH

SH

PH

AH

Data

NH

TH

SH

PH

AH

Data

DH

NH

TH

SH

PH

AH

Data

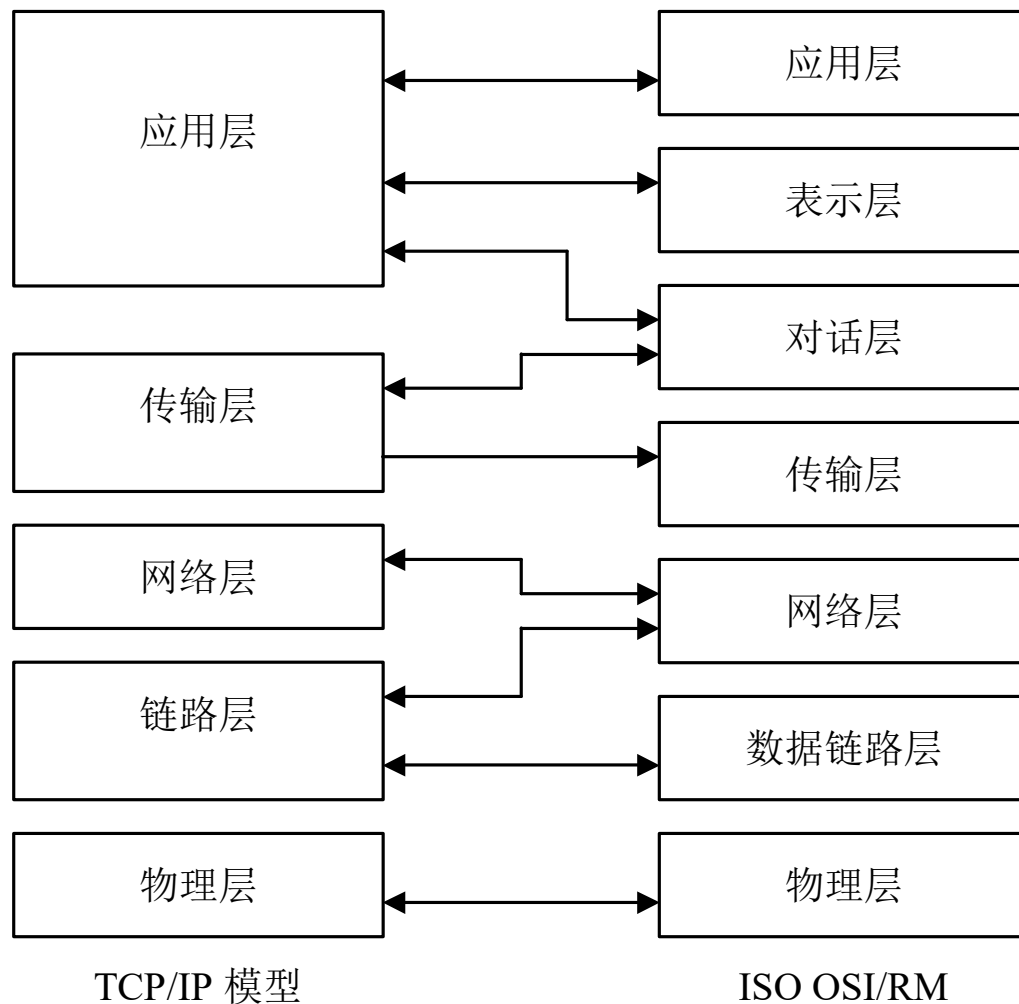
DT

比特序列

第二章 进程通信

2.2 不同节点上的进程间通信

➤ TCP / IP 的分层与协议



第二章 进程通信

2.2 不同节点上的进程间通信

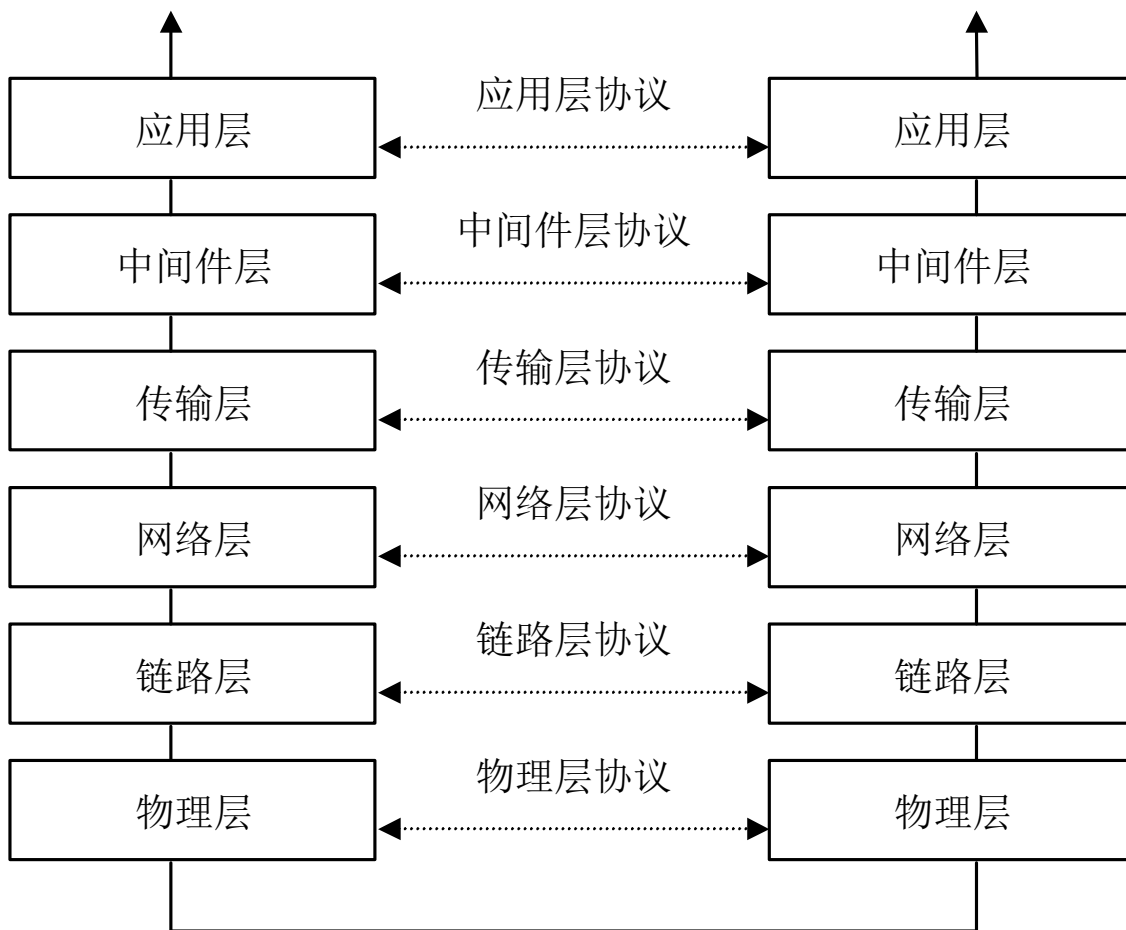
➤ 分布计算系统的中间件协议：

- 1) 认证(Authentication) **协议**，有各种方法建立认证机制来确认身份。
- 2) 访问控制协议能保证经过认证的用户和进程仅能访问那些它有访问权限的资源。
- 3) 分布式提交协议，用于当一组合作的进程执行一个特定操作时，达成要么所有进程都完成了该操作，要么该操作根本没执行的效果。
- 4) 分布封锁协议，用于防止分布在多个机器上的多个进程同时访问同一个资源。
- 5) 中间件通信协议支持高层通信服务，例如，中间件协议可以支持一个进程以透明的方式调用另一个机器上的一个过程，或者支持一进程以透明的方式访问远程机器上的一个对象。

第二章 进程通信

2.2 不同节点上的进程间通信

➤ 分布计算系统的通信模型：



第二章 进程通信

2.2 不同节点上的进程间通信

❖ 进程通信原语

两种基本的实现进程通信的方法是：报文传递和远程过程调用RPC。

➤ 报文传递原语： `send(b, msg)` 和 `receive(a, msg)`；

- 1) 阻塞原语。阻塞性报文通信原语也称为同步(Synchronous)原语。阻塞原语不立即将控制权返回给调用该原语的进程，也就是说 `send` 一直被阻塞直到发送的信息被接收方收到并得到接收方的应答。同样地， `receive` 一直被阻塞，直到要接受的信息到达并被接收。

第二章 进程通信

2.2 不同节点上的进程间通信

a) 无缓冲的阻塞原语。

进程A

进程B

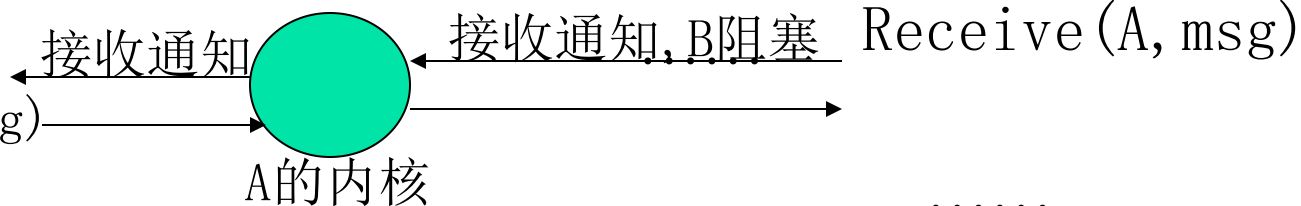
·Send (B, msg) (阻塞)

·Send (B, msg) (阻塞, 超时重发)

·Send (B, msg) (阻塞, 超时重发)

Send (B, msg)

.....



.....

第二章 进程通信

2.2 不同节点上的进程间通信

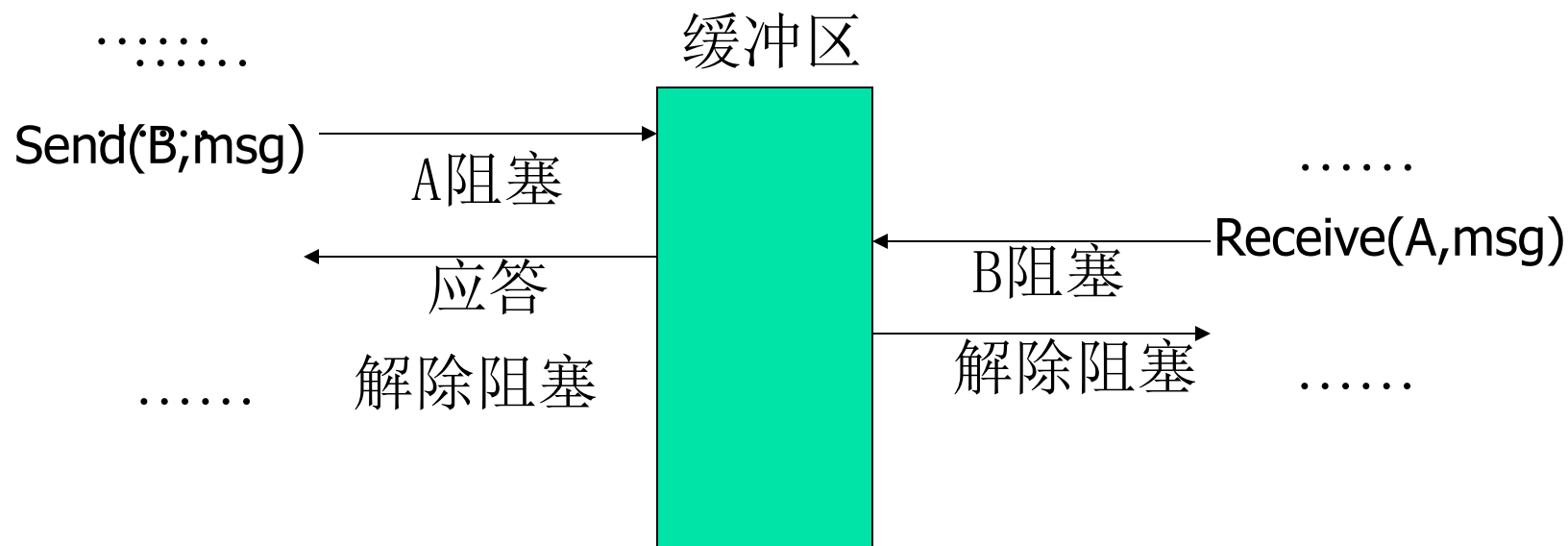
b) 有缓冲的阻塞原语。

进程A

进程B

.....

.....



第二章 进程通信

2.2 不同节点上的进程间通信

➤ 远程过程调用

远程过程的调用者发出远程过程调用后被阻塞等待返回值, 而不象阻塞的报文传递那样等待的仅仅是一个应答。设计和实现远程过程调用主要考虑的问题包括以下方面:

- 1) 参数类型。RPC中有三中参数传递类型: 第一种是输入参数, 这种参数只用于顾客向服务员传递信息; 第二种是输出参数, 这种参数只用于服务员向顾客传递信息, 顾客不能使用它向服务员发送信息; 第三种既作输入又作输出的参数, 顾客能用这种类型的参数向服务员传递信息, 服务员同样能用这个参数向顾客传递信息。

第二章 进程通信

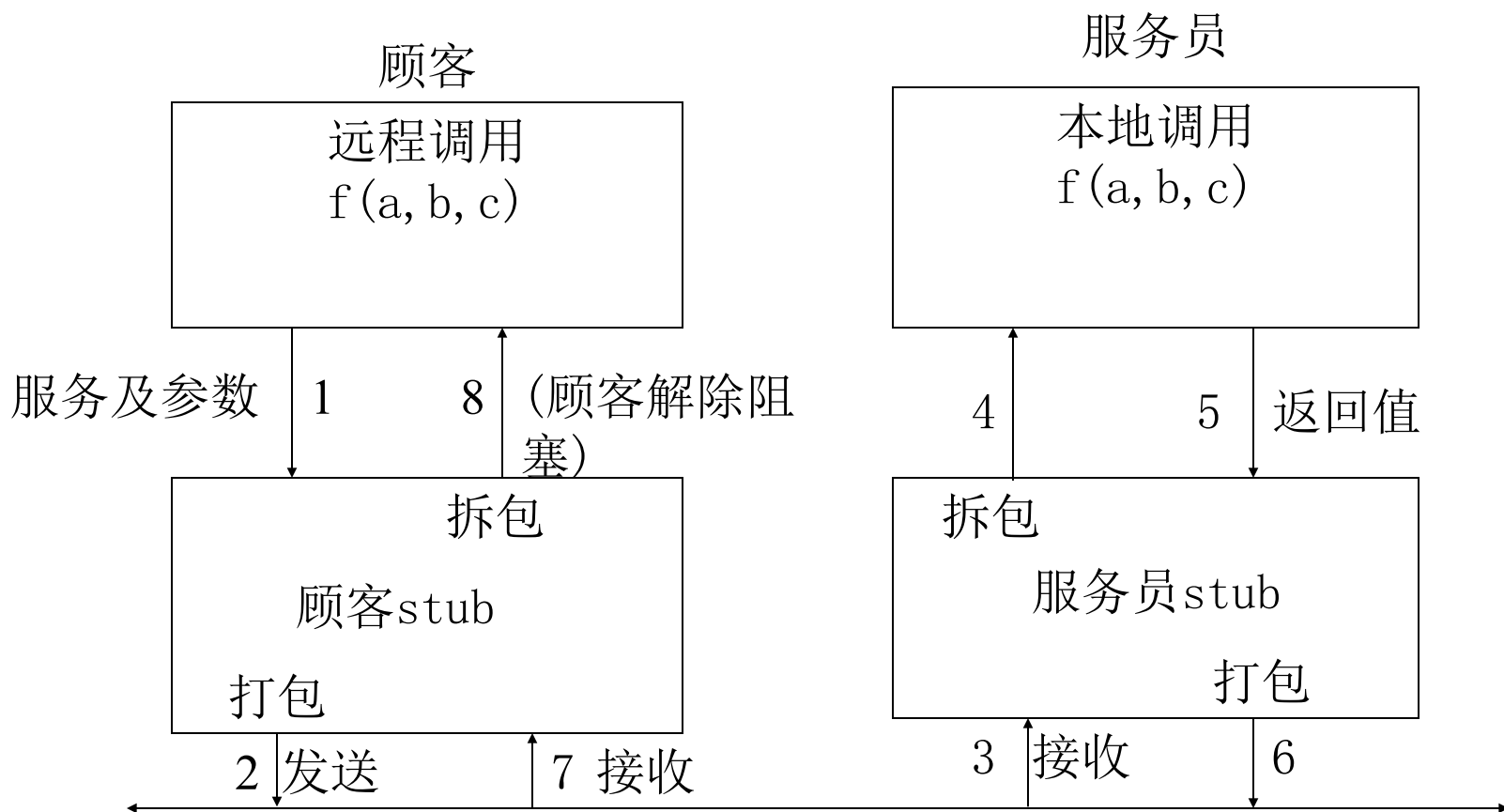
2.2 不同节点上的进程间通信

设计和实现远程过程调用主要考虑的问题包括以下方面：

- 2) 数据类型的支持。数据类型的支持指的是在远程过程调用中哪些数据类型可以作为参数使用。同各种程序设计语言一样，RPC也有可能限制参数的复杂程度。一般的RPC对参数的个数进行了限制而允许较复杂的数据类型。例如只使用一个参数，但是参数可以是一个复杂的结构，这样以来，程序设计者可以通过比较方便的办法绕过这种限制。
- 3) 参数打包(Parameter Marshalling)。为了进行有效的通信，参数和较大的数据结构需要进行打包，接收方能够正确地进行拆一个远程过程调用的执行过程如下：

第二章 进程通信

2.2 不同节点上的进程间通信



第二章 进程通信

2.2 不同节点上的进程间通信

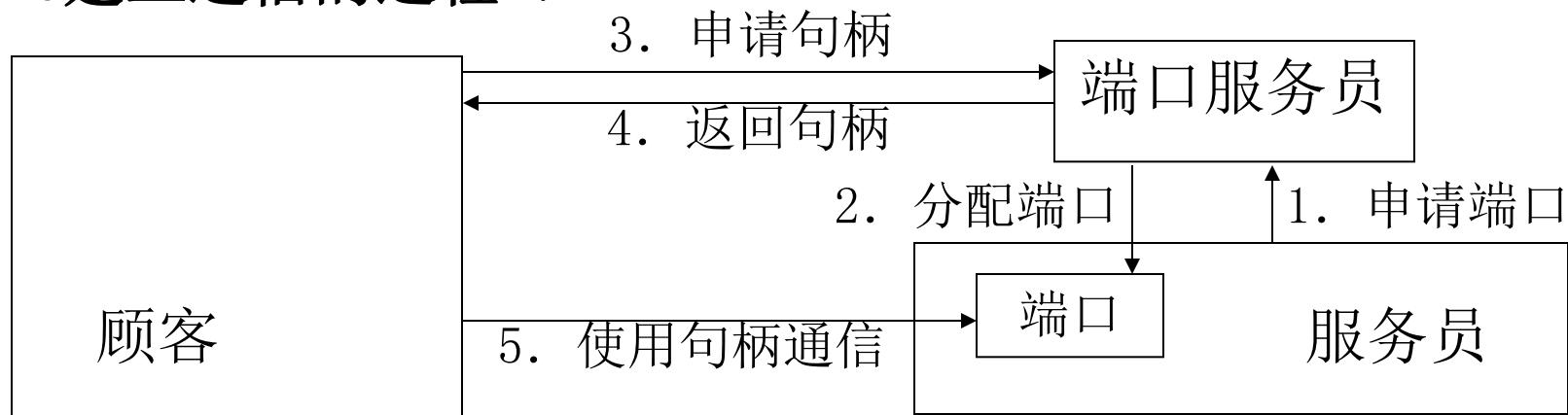
设计和实现远程过程调用主要考虑的问题包括以下方面：

- 4) RPC顾客和服务员的结合(Binding)。一个顾客向一个服务员发送一个远程过程调用前，服务员必须是存在的，并进行了注册，注册时向系统内核的端口管理员(port mapper)申请一个通信端口。服务员将监听这个端口和顾客进行通信。顾客通过访问端口管理员而得到用于访问这个服务员的“句柄”(handle)，这个句柄用于指引和低层的socket结合。这整个过程对程序员来说是透明的。

第二章 进程通信

2.2 不同节点上的进程间通信

RPC建立通信的过程：



第二章 进程通信

2.2 不同节点上的进程间通信

设计和实现远程过程调用主要考虑的问题包括以下方面：

- 5) RPC认证。在一个分布计算系统中，顾客可能需要对服务员的身份进行认证，或者服务员希望对顾客的身份进行认证。
- 6) RPC调用语义。调用语义确定了同一个调用的多次重复请求所造成的后果。

第一种是恰好一次语义(Exactly-Once)。

第二种是最多一次语义(At-Most-Once)。

第三种是至少一次语义(At-Least-Once)。

第四种是多次中最后一次语义(Last-of-Many-Call)。

第五种是幂等语义(Idempotent)。

第二章 进程通信

2.2 不同节点上的进程间通信

❖ 报文传递实例1: socket进程通信

➤ socket地址

```
struct in_addr{
    u_long s_addr;          /* 32_bit netid/hosted */
                             /* network byte ordered */
};

struct sockaddr_in{
    short sin_family;        /*AF_INET*/
    u_short sin_port;        /*16_bit port number*/
                             /*network byte ordered*/
    struct in_addr sin_addr; /*32_bit netid/hostid*/
                             /*network byte ordered*/
    char sin_zero[8];        /*unused*/
};
```

第二章 进程通信

2.2 不同节点上的进程间通信

❖ 报文传递实例1: socket进程通信

➤ Socket原语

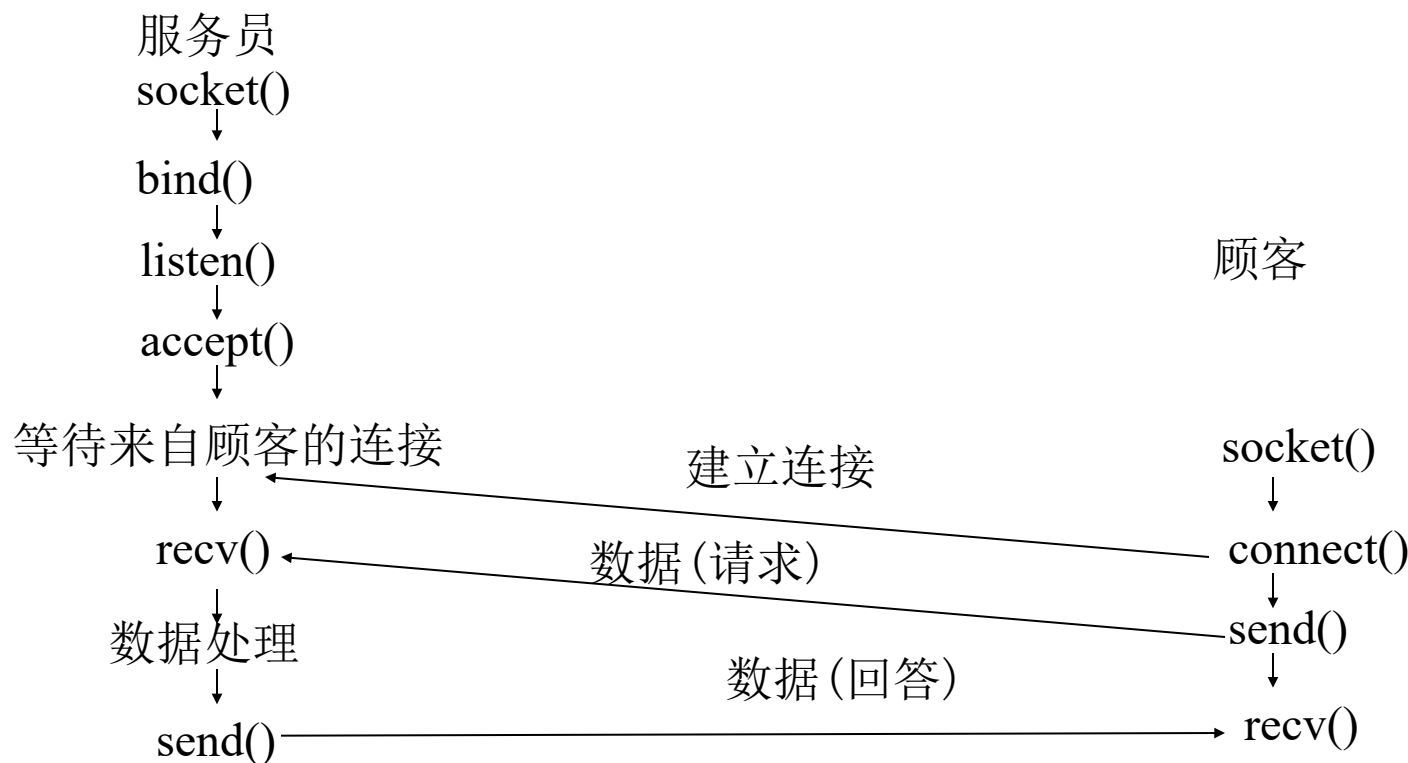
- (1) `s=int socket(domain, type, protocol)。`
- (2) `int bind(s, localaddr, addrlen)。`
- (3) `int connect(s, server, serverlen)。`
- (4) `listen(s, n)。`
- (5) `new_sock=int accept(s, from, fromlen)。`
- (6) `send(s, buf, buflen, flags)。`
- (7) `recv(s, buf, buflen, flags)。`
- (8) `close(s)。`
- (9) `shutdown(s, how)。`
- (10) `sendto(s, buf, buflen, flags, to, tolen)。`
- (11) `recvfrom(s, buf, buflen, flags, from, fromlen)。`
- (12) `select(nfds, readfds, writefds, exceptfds, timeout)。`

第二章 进程通信

2.2 不同节点上的进程间通信

❖ 报文传递实例1: socket进程通信

➤ Socket通信模型: TCP



第二章 进程通信

2.2 不同节点上的进程间通信

❖ 报文传递实例1: socket进程通信

➤ Socket通信模型: UDP



第二章 进程通信

2.2 不同节点上的进程间通信

❖ 报文传递实例2: MPI进程通信

➤ MPI中进程地址

MPI中通信总是发生在一个已知的进程组中，每个进程组都被赋予了一个组标识符(groupID)，进程还有一个进程标识符(processID)。(groupID, processID)唯一地指出了报文的源地址或目的地址，不使用传输层地址。在一个计算中可能发生多个进程组重叠的现象，并且这多个进程组可以同时执行。

第二章 进程通信

2.2 不同节点上的进程间通信

❖ 报文传递实例2: MPI进程通信

➤ MPI进程通信原语:

- (1) MPI_bsend原语。
- (2) MPI_send原语。
- (3) MPI_ssend原语。
- (4) MPI_sendrecv原语。
- (5) MPI_isend原语。
- (6) MPI_issend原语。
- (7) MPI_recv原语。
- (8) MPI_irecv原语。

第二章 进程通信

2.2 不同节点上的进程间通信

❖RPC实例1: SUN RPC

➤ SUN RPC的特点:

- (1) SUN RPC支持最多一次的调用语义和幂等调用语义。
- (2) 支持广播RPC和无响应(no-response or batching)RPC。无响应RPC不需要返回值, 并常用于修改记录。
- (3) SUN RPC对参数数目进行了限制, 它只允许两个参数, 一个是输入参数, 一个是输出参数。但是C语言支持结构数据类型, 可以将多个参数集合到一个数据结构中作为一个参数传递给远程过程。
- (4) SUN RPC在三个层次上支持认证, 最高层次的认证称为安全RPC并使用DES加密技术。

第二章 进程通信

2.2 不同节点上的进程间通信

❖RPC实例1: SUN RPC

➤ 实现RPC的过程:

(1) 编写一个RPC说明文件square.x。

```
1      struct square_in {          /* input parameter */
2          long arg1;
3      };
4      struct square_out {         /* output parameter */
5          long res1;
6      };
7      program SQUARE_PROG {
8          version SQUARE_VERS {
9              square_out SQUAREPROC(square_in)=1; /*过程号=1 */
10             }=1;          /* 版本号=1 */
11     }=0x31230000;
```

第二章 进程通信

2.2 不同节点上的进程间通信

❖ RPC实例1: SUN RPC

➤ 实现RPC的过程:

(2) 编写一个顾客程序client.c, 该程序调用远程过程。

```
1      #include "unpipc.h"    /*our header*/
2      #include "square.h"    /*generated by rpcgen*/
3      int
4      main(int argc, char **argv)
5      {
6          CLIENT *cl;
7          square_in in;
8          square_out *outp;
9          if(argc!=3)
10             err_quit("usage: client <hostname> <integer-value>");
```

第二章 进程通信

2.2 不同节点上的进程间通信

❖RPC实例1: SUN RPC

➤ 实现RPC的过程:

(2) 编写一个顾客程序client.c, 该程序调用远程过程。

```
11      cl=clnt_creat(argv[1], SQUARE_PROG, SQUARE_VERS,  
    “tcp” );  
12      in.arg1=atol(argv[2]);  
13      if((outp=squareproc_1(&in, cl))==NULL)  
14          err_quit(“%s”, clnt_sperror(cl, argv[1]));  
15      printf(“result:%ld\n”, outp->res1);  
16      exit(0);  
17  }
```

第二章 进程通信

2.2 不同节点上的进程间通信

❖RPC实例1: SUN RPC

➤ 实现RPC的过程:

(3) 编写一个服务员程序。服务员的主程序由rpcgen自动生成, 只需要编写服务过程程序。如下所示是过程程序文件

server.c:

```
1      #include "unpipc.h"
2      #include "square.h"
3      square_out *
4      squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
5      {
6          static square_out out;
7          out.res1=inp->arg1*inp->arg1;
8          return(&out);
9      }
```

第二章 进程通信

2.2 不同节点上的进程间通信

❖RPC实例1: SUN RPC

➤ 实现RPC的过程:

(4) 编译。有如下编译任务:

a) 使用rpcgen对RPC说明文件进行编译。

```
Solaris% rpcgen -C square.x
```

通过编译生成4个文件: square.h、square_clnt.c、square_xdr.c、square_svc.c。

b) 用cc编译生成顾客程序client。

```
Solaris% cc -o client client.c square_clnt.c square_xdr.c  
libunipc.a -lnsl
```

c) 用cc编译生成服务员程序server。

```
Solaris% cc -o server server.c square_svc.c square_xdr.c  
libunipc.a -lnsl
```

第二章 进程通信

2.2 不同节点上的进程间通信

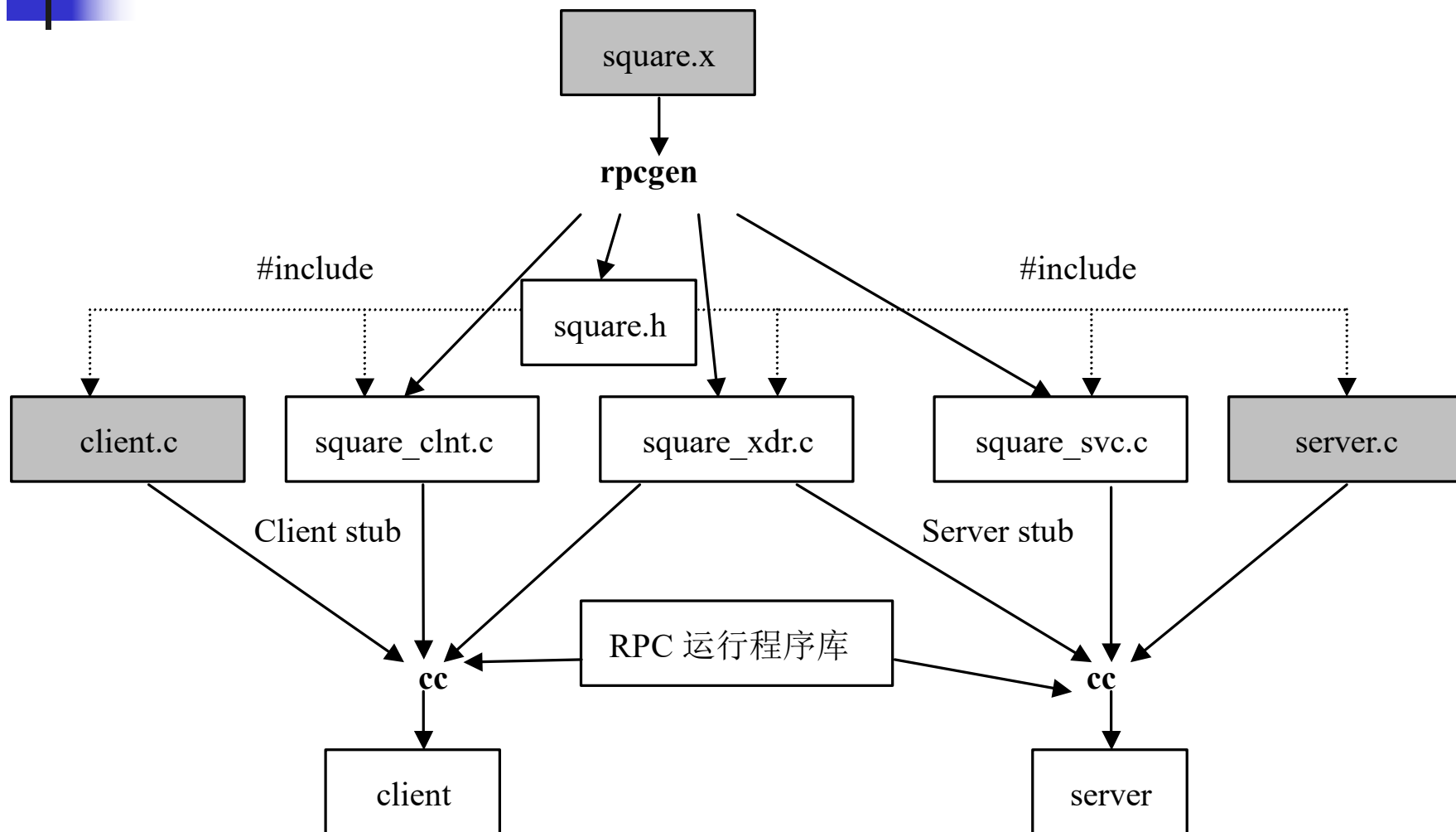
❖RPC实例1: SUN RPC

➤ 实现RPC的过程:

(5) 执行程序。在服务员机上执行server程序，在顾客机上执行client程序。

第二章 进程通信

2.2 不同节点上的进程间通信



第二章 进程通信

2.2 不同节点上的进程间通信

❖ RPC实例2: DCE RPC

➤ DCE简介：

DCE是由OSF开发的，它是一个中间件系统，它作为一个抽象层运行在现有的网络操作系统和分布式应用程序之间。它的主要意图是让用户将现有的机器连接起来，再加上DCE软件就可以执行分布式应用程序，并且不干扰现有的应用。

DCE的编程模型都是顾客—服务员模型。所有顾客和服务员之间的通信都是使用RPC方式实现的。

DCE的基本服务主要有：

(1) 分布式文件服务；(2) 目录服务；(3) 安全服务；(4) 分布式时间服务。

第二章 进程通信

2.2 不同节点上的进程间通信

❖RPC实例2: DCE RPC

➤ DCE RPC的目标：

- (1) 第一个目标，也是最重要的目标就是能够让顾客能够简单地调用一个本地过程就能访问到远程服务。
- (2) 第二个目标是由RPC系统对顾客隐匿通信的实现细节。
- (3) 第三个目标是顾客和服务员之间的高度独立性。例如顾客可以用JAVA语言编写，而服务员则用C语言编写，反过来也是这样。

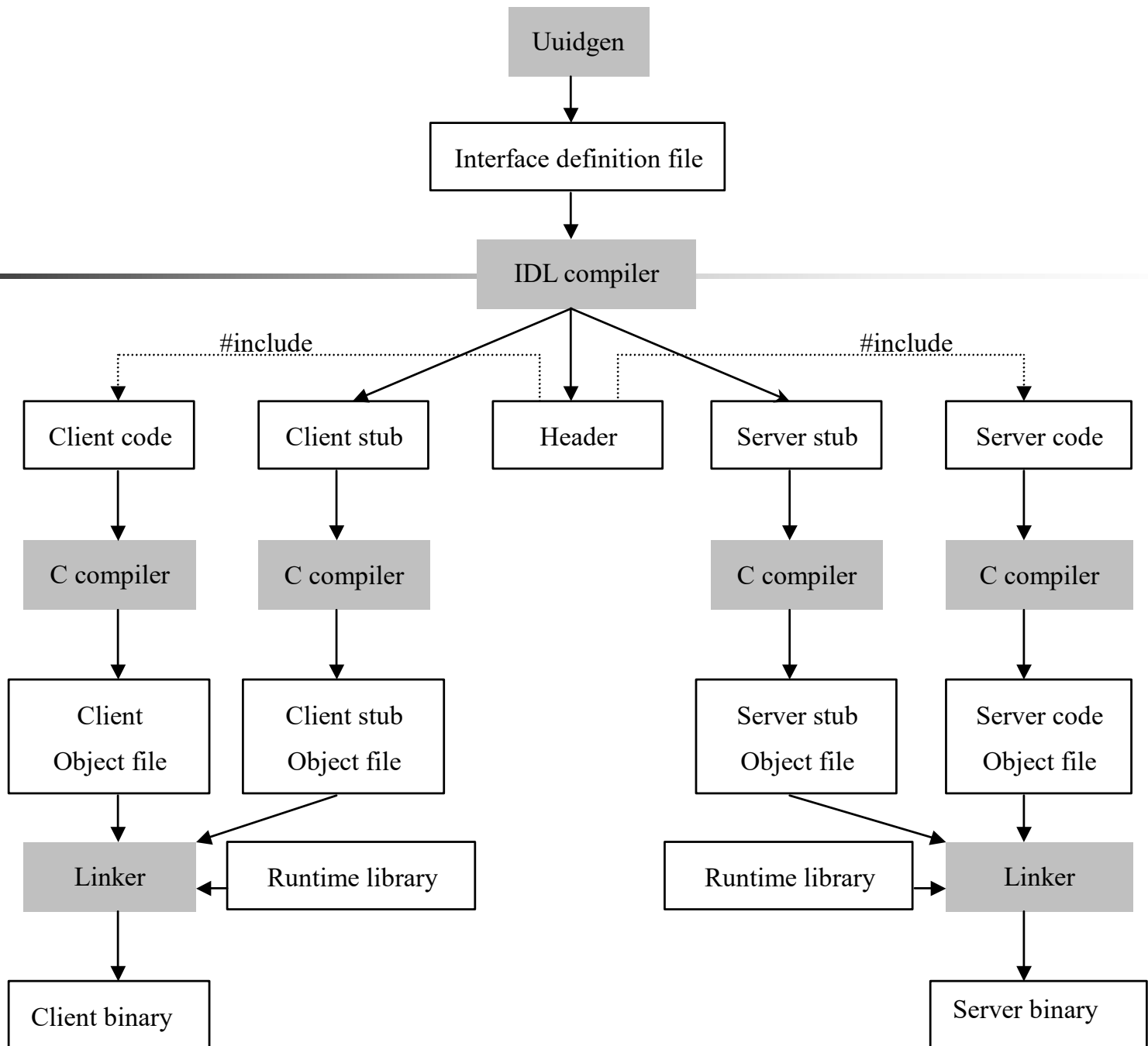
第二章 进程通信

2.2 不同节点上的进程间通信

❖RPC实例2: DCE RPC

➤ 编写顾客和服务员程序过程：

- 1) 调用uuidgen程序，由该程序生成一个IDL范例文件。
- 2) 编辑uuidgen程序生成的IDL文件，在该文件中添加远程过程的名字和它们的参数。
- 3) 编译IDL文件。当IDL文件编辑完毕，使用IDL编译器进行编译，编译后生成如下三个文件：一个头文件（如interface.h）；一个顾客代理(client stub)和一个服务员代理(server stub)。
- 4) 编写顾客和服务员程序代码，编译和连接，生成顾客可执行代码和服务员可执行代码。



第二章 进程通信

2.3 组通信

❖ 组通信的概念

- 所谓组通信指的是一个报文能够被发送到多个接收者的通信，组通信可分为三种情况：
 - 1) 一到多 (One-to-Many) 通信。在这种情况下，对于一个报文来说，它只有一个发送者，但是却有多个接受者。
 - 2) 多到一 (Many-to-One) 通信。在这种情况下，有多个发送者但是只有一个接收者。
 - 3) 多到多 (Many-to-Many) 通信。在这种情况下，有多个发送者和多个接收者。

第二章 进程通信

2.3 组通信

❖ 组通信的概念

➤ 组通信的接收语义：

- 1) 捎带顺序语义。这种语义保证了如果报文捎带了标识这些报文之间关联的一些信息，那么报文就能够以一种正确的顺序接收。
- 2) 一致顺序语义。所有接收者按完全相同的顺序接受报文，但是，一组报文的发送顺序和被接收的顺序可能是不同的。
- 3) 全局顺序语义。这种语义要求所有的接收者严格地按照报文的发送顺序接收。这就需要一个全局时钟，也就是说不同系统中进程的时钟是必需经过同步或使用全局时间戳。

第二章 进程通信

2.3 组通信

❖ 组通信的设计问题

- 1) 封闭组(closed group)和开放组(open group)。进程组是封闭的，是指只有这个组的成员才允许向这个组发送报文，组外的进程不能向整个组发送报文，但是可以向组内的某个成员发送报文。进程组是开放的，是指系统中的所有进程都允许向这个组发送报文。
- 2) 对等组(peer group)和分级组(hierarchical group)。所谓对等组是指组内的所有进程是平等的，没有一个进程处于主导地位，任何决定都是所有进程集体作出的。在分级进程组中，组中进程存在着级别，例如一个进程为协调者(coordinator)，其它进程为工作者(worker)。

第二章 进程通信

2.3 组通信

❖ 组通信的设计问题

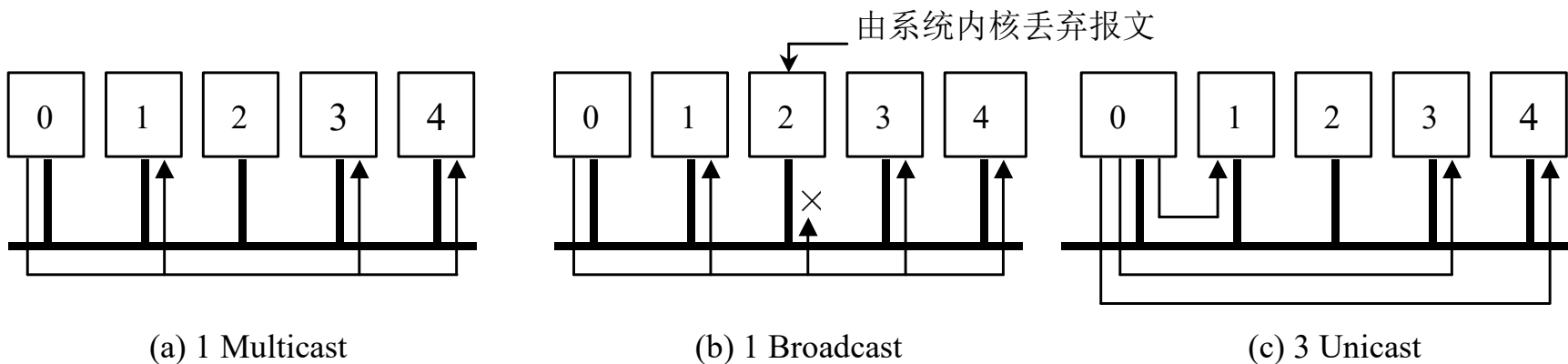
- 3) 组成员的管理。当需要进行组通信的时候，需要某种方法建立和删除一个组，同时需要提供一种方法允许一个进程加入到某个进程组或者离开某个进程组，处理进程组和组成员的崩溃问题。
- 4) 组寻址(group addressing)。每个进程组必须有一个地址，正如一个进程必须有一个地址一样。组寻址的实现方法分为三类：由系统内核实现的方法、由发送进程实现的方法和预测寻址的实现方法。

第二章 进程通信

2.3 组通信

❖ 组通信的设计问题

a) 系统内核实现的方法



(a)用 Multicast 实现 (b)用 Broadcast 实现 (c)用 Unicast 实现

第二章 进程通信

2.3 组通信

❖ 组通信的设计问题

- b) 由发送进程实现的方法。由发送进程维持一张表，该表明确指出接收进程组的所有进程的目地址。
- c) 预测寻址(predicate addressing)的实现方法。每个报文中包含一个预测表达式，它是一个布尔表达式，这个布尔表达式涉及到接收者的机器号码、局部变量或其它因素，接收的机器如果计算得到表达式的值为真，则接收此报文，否则丢弃此报文。

第二章 进程通信

2.3 组通信

❖ 组通信的设计问题

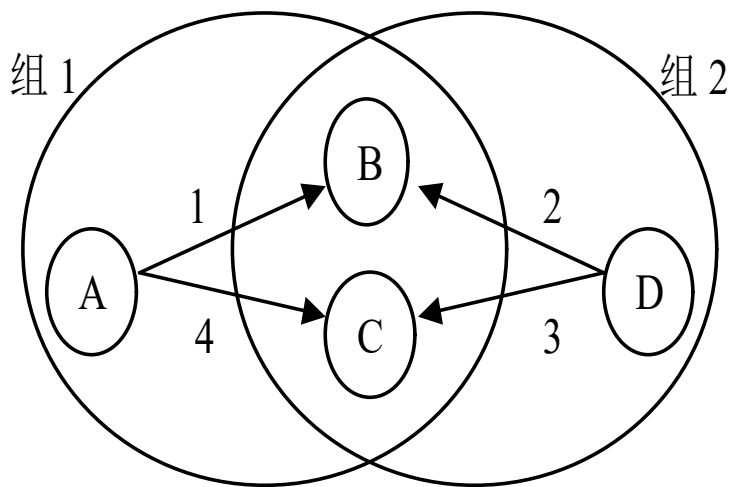
- 5) 发送和接收原语。发送原语同样可以是有缓冲的或是无缓冲的、阻塞的或是无阻塞的、可靠的或是不可靠的。同样地，接收原语可以是阻塞的或是无阻塞的。有的系统专门提供了组通信原语group_send和group_receive。
- 6) 原子性(atomicity)。对于组通信来说，当一个报文要发送到一个进程组，那么这个报文要么被组内的所有进程正确接收，要么没有一个进程接收。某些进程接收报文，而某些进程不接收的情况是不允许的。组通信的这种特性称为原子性。

第二章 进程通信

2.3 组通信

❖ 组通信的设计问题

- 7) 组重叠 (overlapping groups)。一个进程可能同时是多个进程组的一个成员，这种情况可能会导致进程接收报文的顺序不一致。



第二章 进程通信

2.3 组通信

❖ ISIS中的组通信

➤ ISIS中定义的同步形式：

同步系统(synchronous system)、松弛同步系统(loosely synchronous system)和虚同步系统(virtually synchronous system)。

- 1) 同步系统中，任何一个事件按照严格的顺序发生，对于每个事件所需的完成时间本质上来说可以假定为0，发送的报文立即到达接收进程。从一个外部观察者的角度来看，系统是由一系列的离散事件组成的，没有任何的事件在时间上会和其它事件重叠。这种特征有利于理解系统的行为。

第二章 进程通信

2.3 组通信

❖ ISIS中的组通信

➤ ISIS中定义的同步形式：

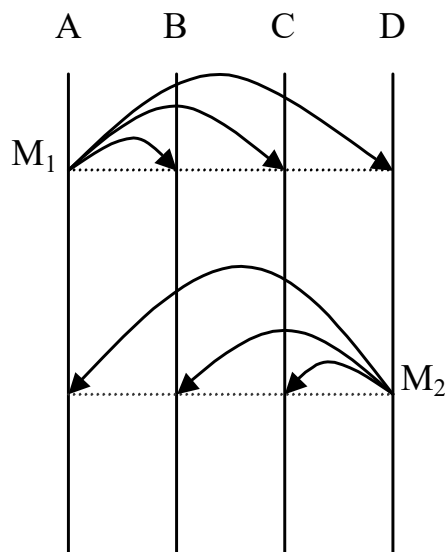
- 2) 松弛同步系统中，一个事件会花费一定的时间，但所有的事件对所有的参加者来说表现出同样的顺序。
- 3) 虚同步系统来说，如果两个报文是因果相关的，那么所有的进程必须按同样的顺序接收这两个报文。如果两个报文是并发的，那么不同的进程可以按不同的顺序接收这两个报文。

第二章 进程通信

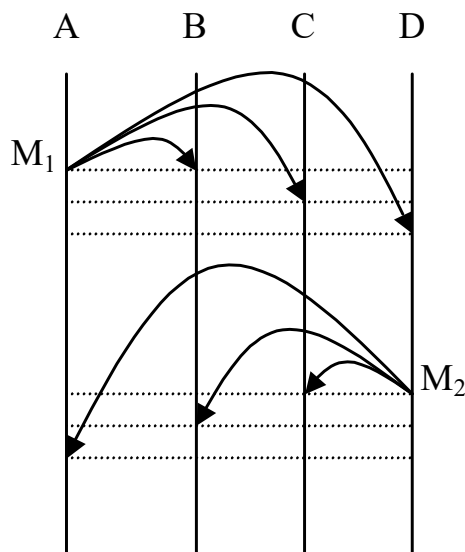
2.3 组通信

❖ ISIS中的组通信

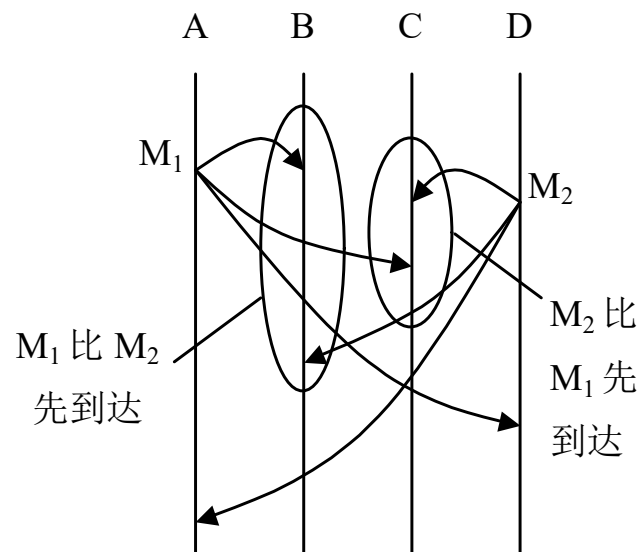
➤ ISIS中定义的同步形式：



(a)同步系统



(b)松弛同步



(c)虚同步

第二章 进程通信

2.3 组通信

❖ ISIS中的组通信

➤ ISIS中定义的通信原语：

ISIS中定义了三种广播语义，它们分别是：ABCAST、CBCAST、GBCAST。ABCAST提供松弛的同步通信，用于向进程组成员传输数据；CBCAST提供虚同步通信，也用于发送数据；GBCAST的语义和ABCAST一样，只是它用于组成员的管理，而不是用于发送一般的数据。

第二章 进程通信

2.3 组通信

❖ ISIS中的组通信

➤ ISIS中定义的通信原语：

- 1) ABCAST 。ABCAST使用两阶段提交协议(two-phase commit protocol)的形式进行工作。首先，发送者A向进程组中的所有成员申请一个时间戳(timestamp)，并将报文发送到进程组的所有成员。进程组的所有成员向发送者分配一个时间戳，该时间戳要大于所有该成员所发送的和所接收的报文的时间戳。A接收到所有成员分配的时间戳之后，从中选择一个最大的时间戳作为要发送报文的时间戳，然后向所有组成员发送一个带有该时间戳的提交报文。提交报文按照时间戳的顺序传递给应用程序。使用该协议能够保证所有报文按照同样的顺序传递到所有进程。

第二章 进程通信

2.3 组通信

❖ ISIS中的组通信

➤ ISIS中定义的通信原语：

2) CBCAST。使用CBCAST原语，只能保证因果相关的报文被所有的进程按同样的顺序接收。利用一个向量来调整具有因果关系的报文的接收顺序。

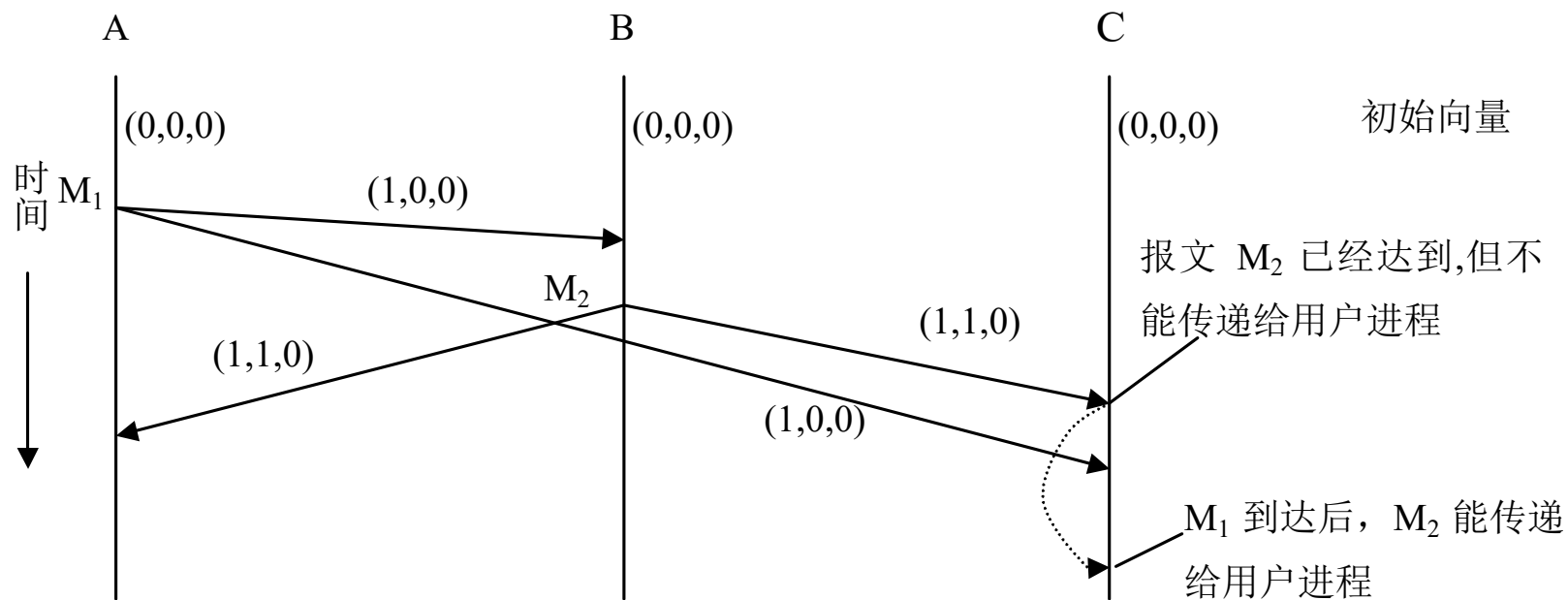
如果一个进程组有 n 个成员，则每个成员保持一个有 n 个元素的向量，向量中的第 i 个元素是该成员从进程 i 那里收到的最后一个报文的号码。当一个进程需要发送一个报文时，它首先将向量中对应自己的元素增1，对于发送者来说，因为自己是该组的成员，发向该组的报文也意味着发向自己。发送者将修改后的向量随报文一起发送。

第二章 进程通信

2.3 组通信

❖ ISIS中的组通信

➤ ISIS中定义的通信原语：



第三章 分布式程序设计语言

3.1 分布式程序设计语言概述

❖ 分布式应用程序的分类

- 并行、高性能应用程序。通过并行性达到加速是在分布计算系统上运行应用程序的最主要的原因。
- 容错应用程序。而分布计算系统具有允许部分失效的特性，即由于各处理机具有自治性，一个处理机的故障不影响其他处理机的正常工作，所以可靠性高。程序和数据也可在若干处理机上复制而进一步增加可靠性。
- 具有专用功能的应用程序。一些应用程序可以被构造成一组专用的服务程序。例如文件服务、打印服务、进程服务、终端服务、时间服务等。
- 固有的分布式应用程序。有些应用程序本身就是分布的，在这种情况下，可以把工作站的集合看成一个分布计算系统，这种应用程序必须在分布式硬件上运行。

第三章 分布式程序设计语言

3.1 分布式程序设计语言概述

❖ 分布式程序设计与顺序程序设计的区别

- 使用多个处理机。对分布式程序设计支持的第一个要求就是系统应该具有把一个程序的不同部分分配到不同处理机上执行的能力。
- 处理机合作。各个进程必须能相互通信和同步，这是对分布式程序设计支持的第二个要求。
- 处理部分失效。在分布计算系统中一些CPU失效时，其他CPU照样工作。所以对分布式程序设计支持的第三个要求是能对系统的部分失效进行检测并恢复。

第三章 分布式程序设计语言

3.1 分布式程序设计语言概述

❖ 分布式程序设计语言的分类

➤ 按并行模型来分

- 1) 顺序进程并行语言。这类语言使用的最基本模型是一组顺序进程，它们并行运行，并且通过报文传递进行通信。大部分是流行的C(或C++)和FORTRAN的扩展。
- 2) 具有内在并行性的语言。一些研究者认为算法语言不是处理并行性的最好语言，因为算法语言是内在顺序式的，许多研究者研究具有内在并行性的语言，如函数式语言、逻辑语言和面向对象语言。

第三章 分布式程序设计语言

3.1 分布式程序设计语言概述

❖ 分布式程序设计语言的分类

➤ 按通信模型来分

- 1) 在物理分布的硬件上运行逻辑上分布的软件。相互使用SEND和RECEIVE原语通信，在网络上发送报文。
- 2) 在物理非分布的硬件上运行逻辑上分布的软件。用共享主存方法实现报文传递来模拟物理报文传递通信。
- 3) 在物理分布的硬件上运行逻辑上非分布的软件。使用分布式共享存储器通信。
- 4) 在物理非分布的硬件上运行逻辑上非分布的软件。使用物理共享存储器通信。

第三章 分布式程序设计语言

3.1 分布式程序设计语言概述

❖ 分布式程序设计语言的分类

➤ 容错模型和技术

故障的处理模型：

- 1) 系统对程序员隐匿全部处理机故障。
- 2) 给程序员提供高层机制，使得程序员能够描述哪些进程和数据是重要的，以及发生崩溃后怎样恢复。

实现可靠性的方法有两种：程序设计容错和通信容错。

- 3) 程序设计容错技术有三类：向前恢复试图确定错误所在并基于这个知识改正包含错误的系统状态；向后恢复通过把系统恢复到错误发生前的状态来改正系统状态；错误屏蔽，利用同一个算法独立开发几个版本，一个最后投票系统用于对这 n 个版本产生的结果进行投票并确定一个正确的结果。
- 4) 通信容错处理进程通信中发生的故障，通信容错依赖于使用的通信方式和故障的类型。

第三章 分布式程序设计语言

3.2 并行性的支持

❖ 并行性的概念

- 并行性。因为分布计算系统有多个处理机，所以可把程序分成若干部放到多个处理机上同时运行，这就是所谓的并行性。
- 伪并行性(pseudo parallelism)，即把程序表示为一组并行运行的进程但不管它们是否在不同的处理机上同时运行。
- 并行粒度。并行单位可以是进程(如并发C)，也可以是表达式(如Par A1f1)。一般说来，通信代价越大，则并行的粒度就应该越大。

第三章 分布式程序设计语言

3.2 并行性的支持

❖ 并行性的表示

- 进程并行。一般说来，一个进程是一个逻辑处理机，顺序地执行代码，具有自己的状态和数据。在语言中，进程或进程类型是要被说明的，就像过程或过程类型一样。进程的创建可以由说明隐式地完成，也可以通过创建某种结构显式地完成。
- 对象并行。用下述方法扩充顺序对象模型可获得并行性：(1) 允许对象不必在收到报文时才活动；(2) 允许接收对象在返回结果后继续执行；第(3)一次向几个对象发送报文；(4) 允许报文发送者继续和接收者并行工作。

第三章 分布式程序设计语言

3.2 并行性的支持

❖ 并行性的表示

➤ 语句并行

PAR

S1

S2

PAR j=0 FOR n

A[j]: =A[j]+1

第三章 分布式程序设计语言

3.2 并行性的支持

❖ 并行性的表示

➤ 函数并行

例如表达式 $h(f(3, 4), g(8))$ ，先计算 f 或 g 是没有关系的，从而可以并行计算 f 和 g 。

➤ 子句的并行

下面的程序给出谓词 A 的两个子句：

(1) $A: \neg B, C, D$

(2) $A: \neg E, F$

存在两个并行性的机会：

(1) A 的两个子句可并行工作只到有一个成功或两个都失败。

(2) 每个子句中的子定理可并行工作直到它们全都成功，或其中一个失败。

前一种并行性叫做OR并行性，后一种叫做AND并行性。

第三章 分布式程序设计语言

3.3 进程通信与同步的支持

❖ 报文传递

进程通信的表示方法：报文传递和共享数据

➤ 设计报文传递的通信方式应考虑的问题：

- 1) 可靠的报文传递和非可靠的报文传递：可靠的报文传递需要承认报文。
- 2) 显式接收和隐式接收：显式接收时，接收者执行某一类 `accept` 语句指明接收哪些报文，以及当报文到达时采取什么行动。使用隐式接收时，在接收者内自动调用程序，通常在接收进程中创建一个新的线程。
- 3) 直接命名和间接命名：直接命名用于指示一个指定的进程，名字可以是该进程的静态名字或是一个表达式。间接命名包括一个中间对象，通常叫做邮箱，发送者把报文送给它，接收者从它那接收。

第三章 分布式程序设计语言

3.3 进程通信与同步的支持

❖ 报文传递

进程通信的表示方法：报文传递和共享数据

➤ 设计报文传递的通信方式应考虑的问题：

- 4) 对称命名和非对称命名。如果发送者和接收者相互命名，则基于直接命名的方案是对称的。在非对称方案中，仅发送者找接收者，在此情况下，接收者要与任何发送者相互作用。注意，使用隐式接收报文的相互作用在命名方面总是非对称的。

第三章 分布式程序设计语言

3.3 进程通信与同步的支持

❖ 报文传递

➤ 报文传递通信模式有：

- 1) 同步和异步点到点报文。在同步报文传送方式中，发送者在接收者接收报文前一直阻塞。这样，双方不仅交换了数据而且还达到同步。在异步报文传送方式中，发送者并不等待接收者准备好接收其报文，发送者在送出报文后立即继续工作。
- 2) 会合。在Ada中会合模型基于三个概念：项说明、项调用和接受语句。项说明和接受语句是服务员程序的一部分，项调用在顾客端。当进程S调用进程R的一项，R为此项执行accept语句时，在S和R之间发生了相互作用，叫做会合。

```
accept incr(X: int; Y: out int)
do Y:=X+1;
end
```

第三章 分布式程序设计语言

3.3 进程通信与同步的支持

❖ 报文传递

➤ 报文传递通信模式有：

- 3) 远程过程调用(RPC)。它是双向通信的另一个原语。当进程S调用进程R的过程P时，由S提供的P的输入参数被送给R。当R收到调用请求时，执行过程P，然后把输出参数送回给S。执行P期间S阻塞，直到输出参数返回。这和会合机构不同，在会合机构中，一旦accept语句已执行，则调用者就不阻塞。
- 4) 一到多报文传送。很多用于分布计算系统的网络支持快速的广播或组通信设施。

第三章 分布式程序设计语言

3.3 进程通信与同步的支持

❖ 共享数据

如果两个进程访问同一个变量，可以实现另一种通信方式：一个进程对此变量进行设置，另一个进程对它进行读。如果两个进程在同一个机器上运行，变量在此机器上存储，则可直接通信。分布进程的共享数据方法有：分布式数据结构和共享的逻辑变量。

1) 分布式数据结构。这种数据结构可由若干进程同时处理。

Linda语言使用元组空间(tuple space)的概念实现分布式数据结构。

例如[“jones”, 31, true]是一个有三个段的元组：一个字符串、一个整数和一个布尔值。对TS定义了三个原子操作：out操作向TS加入一个元组，read读TS中的一个元组，in读TS中的一个元组并删除它。

第三章 分布式程序设计语言

3.3 进程通信与同步的支持

❖ 共享数据

- 2) 共享的逻辑变量。逻辑变量具有“单赋值”性质，最初，它们是未赋值的，但一旦它们接收一个值就不能改变它们。这些变量被用于进程之间的通信通道。如下三个目标：

$\text{goal_1}(X, Y), \text{goal_2}(X, Y), \text{goal_3}(X)$

进行逻辑乘，用进程 P_1 、 P_2 、 P_3 并行求解。变量 X 是这三个进程的通信通道，最初是未赋值的。如果三个进程中的某个给 X 赋值，则其它两个进程可使用此值。类似地， Y 是 P_1 和 P_2 的通信通道。

第三章 分布式程序设计语言

3.3 进程通信与同步的支持

❖ 非确定性的表示和控制

进程之间的相互作用模式并不总是确定性的，有时还决定于运行时条件。因此，表示和控制非确定性模型被提出。选择语句和保护(guarded)Horn子句 是两种表示和控制非确定性的模型。

1) 选择语句。它是由如下形式的一组保护命令组成的：

保护 \rightarrow 语句

其中保护(guard)由一个布尔表达式和某一类“通信请求”组成。布尔表达式必须无副作用，因为它可能在执行该选择语句过程中被计算多次。

第三章 分布式程序设计语言

3.3 进程通信与同步的支持

❖ 非确定性的表示和控制

2) 保护的Horn子句。逻辑程序本质上就不是确定性的。

并行逻辑语言不是对一给定的谓词一个又一个地试验子句，失败时回溯，而是并行地搜索所有那些子句，并且在这些并行执行期间直到有一个并行执行提交前不允许任何赋值对外部是可见的，这叫做OR并行性。但是，这不能无限地进行，因为并行工作的搜索路径随证明的长度而指数地增长。

很普遍的控制OR并行性技术是提交选择非确定性，它非确定地选择一个可选择的子句，取消其他子句。它是基于保护的Horn子句，形式如下：

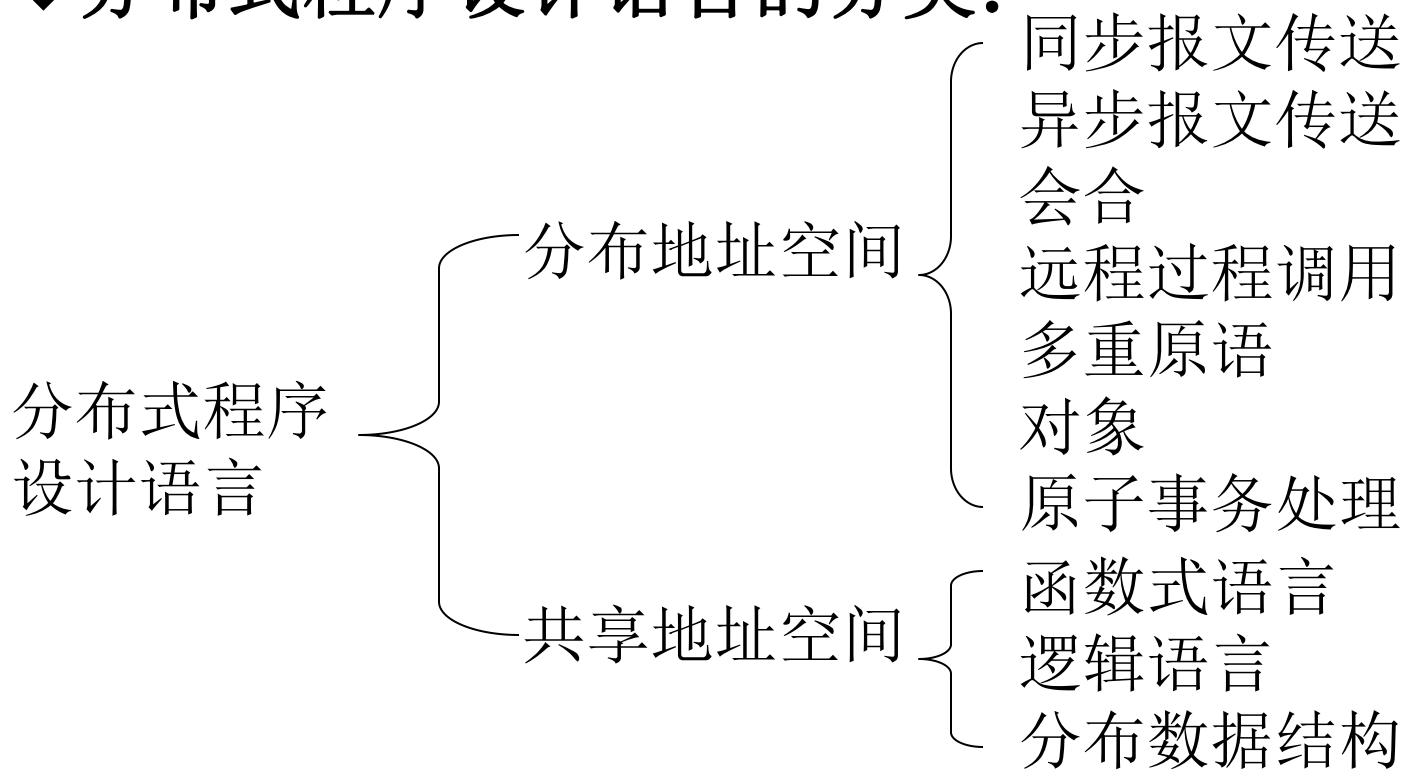
$$A: -G_1, \dots, G_n \mid B_1, \dots, B_m \quad n \geq 0, m \geq 0$$

目标 G_i 的合取(与操作)叫做保护，目标 B_i 的合取叫做体(body)。提交操作符“ \mid ”也是一个合取操作符。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 分布式程序设计语言的分类:



第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 同步式报文传递语言：

1) 创建并行进程：

如CSP提供简单的并行命令创建固定数目的并行进程。进程包含名字、逻辑变量和一系列语句(进程体)。CSP可以创建一组相似的进程，但其数目必须在编译时是个常数。例如并行语句

$$[\text{writer}::X:\text{real}; \dots \parallel \text{reader}(i:1..2)::\dots]$$

创建三个进程，叫作“writer”、“reader(1)”和“reader(2)”。Writer有一个局部变量X。下标量i可在reader进程的体中使用。

2) 通信：

CSP进程不能使用全局变量相互通信，只能使用同步的receive和send。执行send或receive的进程受阻一直到其对方执行完互补的语句为止。例如

$$[X::Y! 3 \parallel Y::n:\text{integer}; X? n]$$

在进程X的语句中，把值3发送给Y。在进程Y的语句中，从进程X读取输入，并存放到局部变量n中。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 同步式报文传递语言：

3) 可传递的数据类型：

简单数据和有结构的数据均可传送与赋值，只要发送的值与接收它的变量类型相同。可给有结构的数据一个名字(构造符)，如下例中的pair：

```
[X::Y! pair(35, 60) || Y::n, m:integer; X? pair(n, m)]
```

可使用空构造符对两个进程进行同步但不传送任何实际数据。

4) 非确定性的表示：

CSP中使用alternative结构表示非确定性，它由一组保护(后面跟着待执行的动作)组成。保护可包含布尔表达式和一个输入语句。CSP允许进程根据当前通信的输入和名字段的信息有选择地接收。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 异步式报文传递语言：

- 1) 并行性单位：NIL中的并行性是基于所谓进程模型。进程不仅是并行性的单位，也是模块化的单位。进程到处理机变换是实现上的问题，由编译和运行时系统处理。
- 2) NIL可动态地进行进程间通信路径的配置：NIL中的信口是一个排队的通信通道。在给定时间，一个信口有一个指定的所有者。所有者关系可以转让给其他进程，可以把信口作为报文的一部分传送，或把信口作为一个新创建进程的初始化参数传送。进程可以连接其拥有的输入口和输出口。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 异步式报文传递语言：

- 3) 通信类型：NIL既支持同步通信也支持异步通信，可把单个输入口连接到几个输出口，所以在输入口可以有多个挂起的报文，因而必须排队。
- 4) 非确定表示：NIL提供一个保护命令风格的语句用于在任何输入口上等待报文。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 基于会合的语言：

➤ Ada：

- 1) 并行性表示：其并行性是基于顺序进程，叫作任务(task)，每个任务具有一定的类型。任务由说明部分(说明其他任务如何与其通信)和一个体(包含它的可以执行的语句)组成。
- 2) 通信：任务通常通过会合机制通信，也通过共享变量通信，会合机制基于项说明、项调用和接受语句。
- 3) 非确定性表示：Ada使用select语句表示非确定性。这个语句用于三个目的：从一组未处理的请求中非确定地选择一个项调用；有条件地调用一项(即仅当被调用的任务准备好立即接受它)和为一个项调用设置时限。
- 4) 容错：Ada有一个异常处理机制处理软件故障，但语言定义未说明硬件故障问题。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 基于会合的语言：

➤ 并发C：

- 1) 进程创建：它使用create原语显式地创建进程，并可向创建的进程传送参数。可赋给新进程一个优先权，以后新进程或其他进程还可以改变此优先权。
- 2) 通信：进程通过会合机构相互通信。并发C中的事务处理与Ada中的项不同，可以返回一个值，并支持异步事务处理(但并不返回值)。并发C支持一个比Ada中的功能更强的accept语句，它根据事务处理参数的值，可以有条件地接受一些事务处理。
- 3) 非确定性表示：使用select语句表示非确定性。
- 4) 容错：基于进程复制。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 基于远程过程调用的语言：

- 1) DP的进程：每个处理机专用于执行一个进程，但每个进程可包含几个处理线程，这些线程以伪并行方式运行。
- 2) 通信：DP进程相互调用对方的公用过程进行通信，用如下形式调用：

Call P. f(exps, vars)

这里P是被调用进程的名字，f是由P说明的过程名字，表达式是输入参数，返回值赋予变量vars。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 多重通信原语：

- 1) SR是由一个或多个资源(resource)组成。资源是运行在一个物理节点(单处理机或共享存储器多处理机)的一个程序模块,可动态创建,并可选择地分配到指定机器上运行。
- 2) 资源可包含多个进程,它们共享数据。资源可包含一个初始进程和终结进程,它们隐式地被创建和运行。
- 3) SR使用类似于select语句的结构处理非确定性。
- 4) SR操作的定义类似过程的定义,可看成一个过程或入口点(entry point)。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 多重通信原语：

- 5) 把操作的两种服务方式和两种调用方式结合起来就有四种进程通信方法。

	call(同步)	send(异步)
entry(同步)	会合	报文传送
process(异步)	RPC	Fork

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 基于对象的语言：

- 1) 对象：Emerald把所有实体都看成对象。对象可以是主动的或被动的。
- 2) 并行性：Emerald中的并行性表现在主动对象的同时执行上。一些对象可从一个处理机上迁移到另一个上。
- 3) 分布式系统中，很多对象可以并行运行，Emerald为本地和远程调用提供相同的语义。
- 4) 对象的迁移可由编译程序或程序员使用几个简单原语发动。对象可作为远程操作中的参数传送。对该参数对象的每次访问都会产生另一个远程调用。为了使这类调用最佳化，先把参数对象传送到目的处理机，后把该对象传送回来。因为这种情况经常发生，所以引入一个新的参数传送类型，叫做传送调用(call_by_move)以便有效地完成这种操作。

第三章 分布式程序设计语言

3.4 逻辑上分布地址空间的语言

❖ 基于原子事务处理的语言：

➤ Argus：

- 1) 主要特点:guardian(保护者)和action(活动)。保护者是能从崩溃中幸存下来的模块，而活动是一组原子执行。
- 2) 为了在保持原子语义下允许活动的并行，使用原子对象，它是一种原子数据类型。Argus提供一些原子类型，用户也可自己定义一些。
- 3) Argus提供两级同步机制：用于伪并行进程的和用于并行活动的。
- 4) mutex类型提供对保护者内各进程所共享的对象的互斥访问。
- 5) 在容错方面，可把某些保护者对象说明成stable，存放到坚固存储器中，如果某个节点崩溃了，则可在坚固存储器中检索并得到恢复。

第三章 分布式程序设计语言

3.5 逻辑上共享地址空间的语言

❖ 并行函数式语言：

- 1) ParAlf1利用隐式函数并行性。函数并行性通常是细粒度的。由于可能有比处理机数多得多的任务要并行执行，所以使用变换方法指定哪个表达式在哪台处理机上计算。如： $(f(x) \text{ on } (\$self-1)) + (g(y) \text{ on } (\$self+1))$
- 2) 通信和同步是隐式的，所以不需要显式语言结构。某个计算需要另一个计算的结果但还未出来时则受阻。
- 3) 语义是基于迟缓计算(lazy evaluation)，即一个表达式仅当其结果被要求时才进行计算。一般说来，程序员不需关心计算次序，但为了有效性要对计算次序进行控制。

第三章 分布式程序设计语言

3.5 逻辑上共享地址空间的语言

❖ 并行逻辑语言：

➤ 并发PROLOG：

- 1) 并行性来自合取的各目标的AND并行计算和保护Horn子句的各保护的OR并行计算。
- 2) 并发PROLOG中的并行进程使用共享逻辑变量通信。同步是基于在只读变量上暂停的办法。
- 3) 并发PROLOG使用保护Horn子句处理非确定性。

第三章 分布式程序设计语言

3.5 逻辑上共享地址空间的语言

❖ 并行逻辑语言：

➤ PARLOG:

- 1) AND/OR并行性由程序员控制。有两种不同的合取操作符：“.” 并行计算各合取；“&” 串行计算各合取(自左至右)。
- 2) 进程通过共享变量进行通信，而同步方法是在无界共享变量上挂起。PARLOG有个机构用来指定哪些进程可以为某变量产生赋值。如果输入自变量未被赋值，则相应的合一将挂起，当某其他进程为该变量赋值时此合一将继续。
- 3) PARLOG使用保护的Horn子句用于非确定性。PARLOG中的保护可测试任何输入变量并为子句的局部变量赋值，但不能给在输入自变量中传送的变量赋值。

第三章 分布式程序设计语言

3.5 逻辑上共享地址空间的语言

❖ 基于分布数据结构语言：

➤ Linda:

- 1) Linda的目标是将程序员从并行计算和并发事件的思考中解脱出来，从而使并行程序设计在概念上类似于顺序程序设计。
- 2) Linda使用简单原语eval创建顺序进程，但不为程序设计人员提供方法把进程变换到处理机上，实际上并不需要，因为每个处理机执行一个进程。
- 3) Linda使用元组空间通信模型。进程通信要向TS插入新元组、读和移去现存的元组。进程同步方法是使用阻塞read和in操作等待元组可用。
- 4) Linda的容错网络内核是基于TS的复制上的。

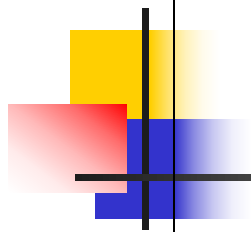
第三章 分布式程序设计语言

3.5 逻辑上共享地址空间的语言

❖ 基于分布数据结构语言：

➤ Orca：

- 1) 这种语言的并行性是基于顺序进程。使用显式的fork原语派生新的子进程并把参数传送给它。参数可以是数值，也可以是由该子进程的说明部分指出共享的抽象数据类型。
- 2) 进程之间通信通过共享数据对象间接地进行。每个对象都属于抽象数据类型，每个抽象数据类型的定义由说明部分和实现部分组成。说明部分列出可对该给定类型的对象进行的各种操作。
- 3) 一个操作的实现可由一个或多个保护语句组成。如果是这样，一个操作的调用受阻直到至少有一个保护成功，接着非确定性地选择一个为真的保护，执行其语句不再受阻。
- 4) 共享数据对象模型在分布式系统中有效的实现方法是复制对象。如果共享对象不经常改变，可在经常读它的处理机上维持副本，对本地副本进行读操作，就可减少通信开销。

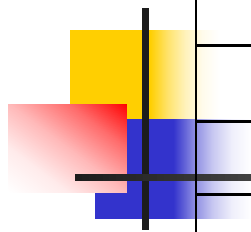


原语		语言的例子
并行性	表示并行性	
	进程	Ada、并发 C、Linda、NIL
	对象	Emerald、并发 Smalltalk
	语句	Occam
	表达式	ParAlfl、FX-87
变换	子句	并发 PROLOG、PARLOG
	静态	Occam、StarMod
	动态	并发 PROLOG、ParAlfl
	迁移	Emerald
通信	报文传递	
	点到点报文	CSP、Occam、NIL
	会合	Ada、并发 C
	远程过程调用	DP、并发 CLU、LYNX
	一对多报文	BSP、StarMod
	数据共享	
	分布式数据结构	Linda、Orca
	共享逻辑变量	并发 PROLOG、PARLOG
	非确定性	
	选择语句	CSP、Occam、Ada、并发 C、SR
部分失效	保护 Horn 子句	并发 PROLOG、PARLOG
	故障检测	Ada、SR
	原子事务处理	Argus、Aeolus、Avalon
	透明容错	NIL

第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖DCDL中的通用符号：



\square	选择
$*$	重复
\parallel	并行
\rightarrow	条件
$;$	顺序
send	输出
$: =$	赋值
receive	输入
$::$	定义
$[$	开始
$\vee]$	结束
\forall	任意(全称量词)
\exists	存在(存在量词)
$=$	相等
\neq	不等
\vee	或
\wedge	且
\neg	反

第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ DCDL中并行性表示：

DCDL中的并行单元是语句。

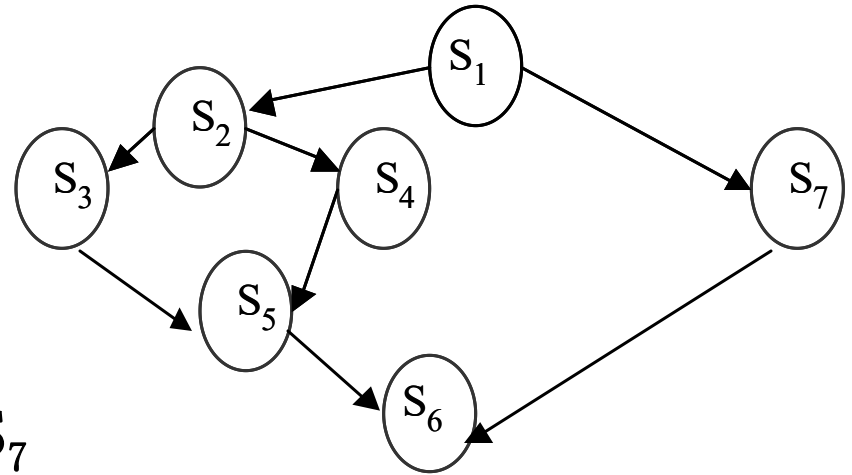
一组并行语句表示为：

$$S_1 || S_2 || \dots || S_n$$

而一组顺序语句表示为：

$$S_1; S_2; \dots; S_n$$

一组语句可以用语句优先图表示

$$S_1; [S_2; [S_3 || S_4]; S_5; S_6] || S_7$$


第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖选择语句：

一个选择语句表示为：

$$[G_1 \rightarrow C_1 \square G_2 \rightarrow C_2 \square \dots \square G_n \rightarrow C_n]$$

选择语句选择其组成的被保护的命令之一执行。如果多余一个命令可被选择，选择将是不确定的。如下的选择语句

$$[x \geq y \rightarrow m := x \square y \geq x \rightarrow m := y]$$

表示如果 $x \geq y$ ，将 x 赋予 m ；如果 $y \geq x$ ，将 y 赋予 m ；如果 $x \geq y$ 并且 $y \geq x$ ，则将 x 或 y 之一赋予 m 。

第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ 重复语句：

一个重复语句指定其组成选择语句的交互次数，这些语句带保护或不带保护，它的形式有如下三种：

1. $*[\text{带保护的选择语句}]$
2. $*[\text{不带保护的选择语句}]$
3. $(n)[\text{选择语句}]$

在第一种情况下，当所有的保护都经过时，重复语句终止。在第二种情况下，执行不终止。第三种是一个特别的重复语句，其重复的次数最多为 n 。

第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ 重复语句：

例1： 给出一个确定的数组 $b[1:m][1:n]$ ，其中 $1 < m$ ， $1 < n$ ，找出数组 b 中的一个确定值 x 。

$i := 1;$

$j := 1;$

$*[i \leq m \wedge x \neq b[i, j] \rightarrow$

$\quad [j := j + 1;$

$\quad [j \leq n \rightarrow \text{skip} \square j > n \rightarrow i := i + 1; j := 1]$

$\quad]$

$]$

第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ 重复语句：

例2：确定一个 $m \times n$ 的矩阵 $a[1:m][1:n]$ 中某一行的所有元素是否全部为0

$i := 1; p := m + 1;$

$*[i \neq p \rightarrow$

$[j := 1;$

$q := n + 1;$

$*[j \neq q \rightarrow$

$[a[i, j] = 0 \rightarrow j := j + 1 \square a[i, j] \neq 0 \rightarrow q := j]$

$];$

$[j = n \rightarrow p := i \square j \neq n \rightarrow i := i + 1]$

$]$

$]$

$found := (i \neq m + 1)$

第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ 语句并发(或并行)的条件

当两个语句并发执行时，可能产生与顺序执行不同的结果。让我们先定义两个符号：

(1) $R(S_i)$, S_i 的读集，即在 S_i 中被引用的所有变量的集合。

(2) $W(S_i)$, S_i 的写集，即在 S_i 中被修改的所有变量的集合。

Bernstein提出了以下三个条件，对于两个并发执行的语句 S_1 和 S_2 ，必须满足这三个条件才能使它们并发执行的结果与它们以任意次序顺序执行的结果相同。

$$(1) R(S_1) \cap W(S_2) = \Phi$$

$$(2) R(S_2) \cap W(S_1) = \Phi$$

$$(3) W(S_1) \cap W(S_2) = \Phi$$

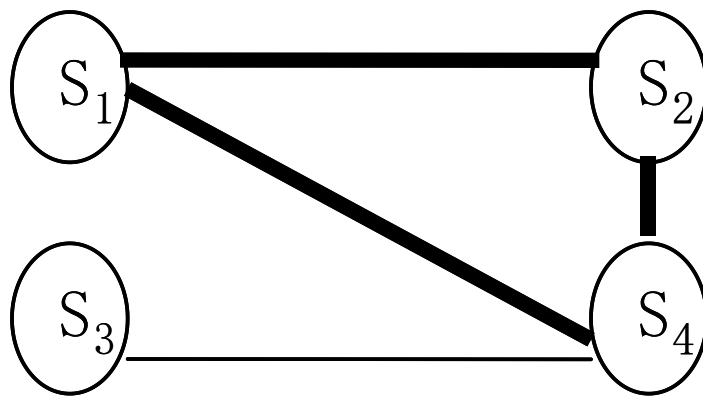
第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ 语句并发(或并行)的条件

我们可以用Bernstein条件来寻找语句中可以并行执行的最大子集。为此我们定义了一个无向图，节点集由给定语句集组成，如果 $S_i \parallel S_j$ ，则节点 S_i 和 S_j 相连，可以并行执行的最大的语句子集对应于最大的完全子图。

例如： $S_1: a := x + y$, $S_2:$
 $b := x \times z$, $S_3: x := y + z$, $S_4:$
 $c := y - 1$ 。显然， S_1, S_2, S_4 形成最大的完全子图，也就是说， $S_1 \parallel S_2 \parallel S_4$ 。



第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ DCDL中的通信

1) 输出命令的形式为:

`send message_list to destination`

其中destination是一个进程名(一对一通信)或代表所有其他进程(一对所有通信)的关键字all。

2) 输入命令的形式为:

`receive message_list from source`

其中source是一个进程名, 这个输入命令支持显式和隐式的报文接收。

隐式的报文接收表示为:

`receive message_list`

第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ DCDL中的通信

例1: 用如下递归的方法计算 $f(n)=f(n-1) \times n^2$, $n>1$ 并且 $f(1)=1$ 。

```
p(i:1..n)::=[receive m from p(i-1)→  
    [[m=0→send 1 to p(i-1)□m>0→send m-1 to p(i+1)];  
    receive r from p(i+1);  
    send m×m×r to p(i-1)  
    ]  
    ]  
p(0)::=send n to p(1);  
receive result from p(1)
```

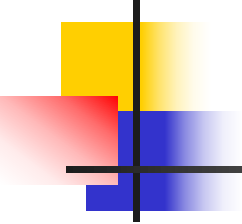
第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ DCDL中的通信

例2: Fibonacci数列是由递推公式 $F(i) = F(i-1) + F(i-2)$ ($i > 1$) 定义的一个整数数列, 其初始值 $F(0) = 0$, $F(1) = 1$ 。这里有两种算法。

算法一: 定义一系列进程: $f(i)$ 用于计算 $F(n-i+1)$, 如果 $(n-i+1)$ 大于1, $f(i)$ 从 $f(i-1)$ 接收 $(n-i+1)$ 并把 $(n-i)$ 传递给 $f(i+1)$ 。然后 $f(i)$ 等待 $f(i+1)$ 和 $f(i+2)$ 的结果, 把它们相加, 并把相加的结果传递给 $f(i-1)$ 和 $f(i-2)$ 。



```
f(0) ::= send n to f(1);  
        receive p from f(2);  
        receive q from f(1);  
        ans := q
```

```
f(i) ::= receive n from f(i-1);  
        [n > 1 → [send n-1 to f(i-1);  
                  receive p from f(i+2);  
                  receive q from f(i+1);  
                  send p+q to f(i-1);  
                  send p+q to f(i-2) ]  
        □ n = 1 → [send 1 to f(i-1);  
                  send 1 to f(i-2) ]  
        □ n = 0 → [send 0 to f(i-1);  
                  send 0 to f(i-2) ]  
        ]  
f(-1) ::= receive p from f(1)
```

在上述算法中， $f(0)$ 是用户进程， $f(-1)$ 是虚进程。

算法二。这个算法使通信只限于邻居之间，即 $f(i)$ 只能和 $f(i-1)$ 和 $f(i+1)$ 通信。

$f(0) ::= [n > 1 \rightarrow [\text{send } n \text{ to } f(1);$

receive p from $f(1);$

receive q from $f(1);$

$\text{ans} := p]$

$\square n = 1 \rightarrow \text{ans} := 1$

$\square n = 0 \rightarrow \text{ans} := 0$

$]$

$f(i) ::= \text{receive } n \text{ from } f(i-1);$

$[n > 1 \rightarrow [\text{send } n-1 \text{ to } f(i+1);$

receive p from $f(i+1);$

receive q from $f(i+1);$

send $p+q$ to $f(i-1);$

send p to $f(i-1)]$

$\square n = 1 \rightarrow [\text{send } 1 \text{ to } f(i-1);$

send 0 to $f(i-1)]$

$]$

第三章 分布式程序设计语言

3.6 分布式控制描述语言DCDL

❖ DCDL中的通信容错

容错是通过检测故障并随之对系统进行重新配置而实现的。以下是用DCDL描述的故障检测过程。

```
sender ::= [setup time(t);  
            send diagnostic_signal to receiver;  
            [receive ack from receiver → status := normal  
              □ timeout(t) → status := abnormal  
            ]  
          ]
```

个有故障的处理机将被发送方节点通过检查状态变量的值发现。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字、标识符和地址

- 1) 命名是给各种服务、对象和操作起个名字，并提供一种手段把这些名字变换成它们所代表的实体本身。
- 2) 在分布计算系统中，命名系统的实现本身就是分布式的，是跨越多个机器而实现的。命名系统的分布实现方式是影响命名系统有效性和可扩充性的关键因素。
- 3) 名字的两形式：地址和标识符。
- 4) 地址：是一个特殊类型的名字，它用来指出一个实体的访问点
- 5) 标识符：一个真正的标识符具有如下性质：
 - a) 一个标识符仅用于表示一个实体；
 - b) 一个实体只用一个标识符来表示；
 - c) 一个标识符总是只用来表示同一个实体，也就是说，它从不被重复使用。
- 6) 对一个对象进行操作或访问时，往往需要将它的标识符变换为它的地址，变换的过程中需要用到变换表，这个变换表叫做上下文(context)。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 分布式系统中的名字

- 1) 名字应该能有效地支持系统的面向事务处理和面向对话这两类服务和应用。
- 2) 应允许在系统中以分散的方式产生全局唯一的名字，以提高效率和可靠性。
- 3) 命名是实现分布计算系统透明性的关键部分，要达到名字透明和位置透明。
- 4) 命名机制应支持对本地或远程资源的访问，命名应与系统拓扑结构或具体的物理连接方式无关，尽可能隐匿各部分的边界。
- 5) 为了支持资源的迁移，命名系统应至少支持两种形式的名字，即标识符和地址，并且可动态地结合(binding)。
- 6) 同一对象可以有用户定义的多个局部名字，需要一种机制把局部名字和全局标识符结合起来。
- 7) 在分布计算系统中，一个实体可能包含多个不同的对象，就需要一个进程组标识符。这样就可以支持广播或小组标识符。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字的结构

1) 名字按结构可分成绝对名字和相对名字两种：

绝对名字和给出名字的“上下文”无关，也就是和发出此名字的地点、用户、应用程序无关，它总是代表同一对象，有利于资源共享，因为可以使用整个系统共用的名字指出对象。

相对名字和给出名字的上下文有关，例如和网络有关的邮箱名，以及UNIX操作系统中的文件名。

2) 地址结构也有两种，即平面地址和分层地址：

分层地址由若干段组成，这些段反映了网络的层次结构。

平面地址与物理位置或任何其他层次没有任何关系，可以想象平面地址的分配可以使用一个单一的全系统范围的计数器进行，任何时候需要一个新地址时读此计数器并且将计数器加1，这样，地址是唯一的，但和位置无关。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字的结构

3) 两种地址结构的优缺点：

分层地址的优点：分层地址使得路由选择容易；容易创建新的地址，即可在每一个网络内单独决定主机号码；自动给出高位号码，即在主机内不用给出它所属的网络号，正如在城市内拨电话时不必拨国家和地区号码一样。

分层地址缺点：当一个进程迁移到另一个机器上时不能使用原来的地址。

平面地址的优点：当进程迁移时仍可使用原来的地址；

平面地址的缺点：路由选择比较困难，因为每个路由选择节点必须保持一个能对所有可能的地址进行变换的路由选择表，地址的赋值也比较复杂，必须确信每个地址是唯一的。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字空间

- 1) 名字空间：是名字的一种有机组织形式。
- 2) 名字空间的表示：用一个带标号的有向图来表示，这个图中有两类节点，一类是叶节点(leaf node)，另一类是目录节点(directory node)。一个叶节点代表一个命名的实体，叶节点没有输出弧(outgoing edge)，只有输入弧(incoming edge)。每个节点有一个标识符，每个弧有个名字。
- 3) 叶节点：一般用来存放一个实体所表示的信息，例如这些信息可以包含这个实体的地址，顾客使用这个地址可以访问到这个实体；还例如这些信息还可以包含这个实体的状态，如果一个叶节点代表一个文件，它不仅包含整个文件，还包含这个文件的状态。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字空间

- 4) 目录节点：每个目录节点保存了一个表，表中的一项代表了一条输出弧，每条输出弧由(弧标号、节点标识符)表示，这样的表被称为目录表(directory table)。
- 5) 根节点：有一个特殊的节点，该节点只有输出弧而没有输入弧，该节点被称为命名图中的根节点(root node)，或简称为根。一个命名图中可能会有多个根。
- 6) 路径：一个路径是由一串弧的标号组成的，例如：
 $N: \langle \text{label-1}, \text{label-2}, \dots, \text{label-n} \rangle$ ，这里N表示这个路径中的第一个节点，这样一个标号串称为路径名。如果路径名中的第一个节点是命名图中的根，则这个路径名被称为绝对路径名，否则称为相对路径名。

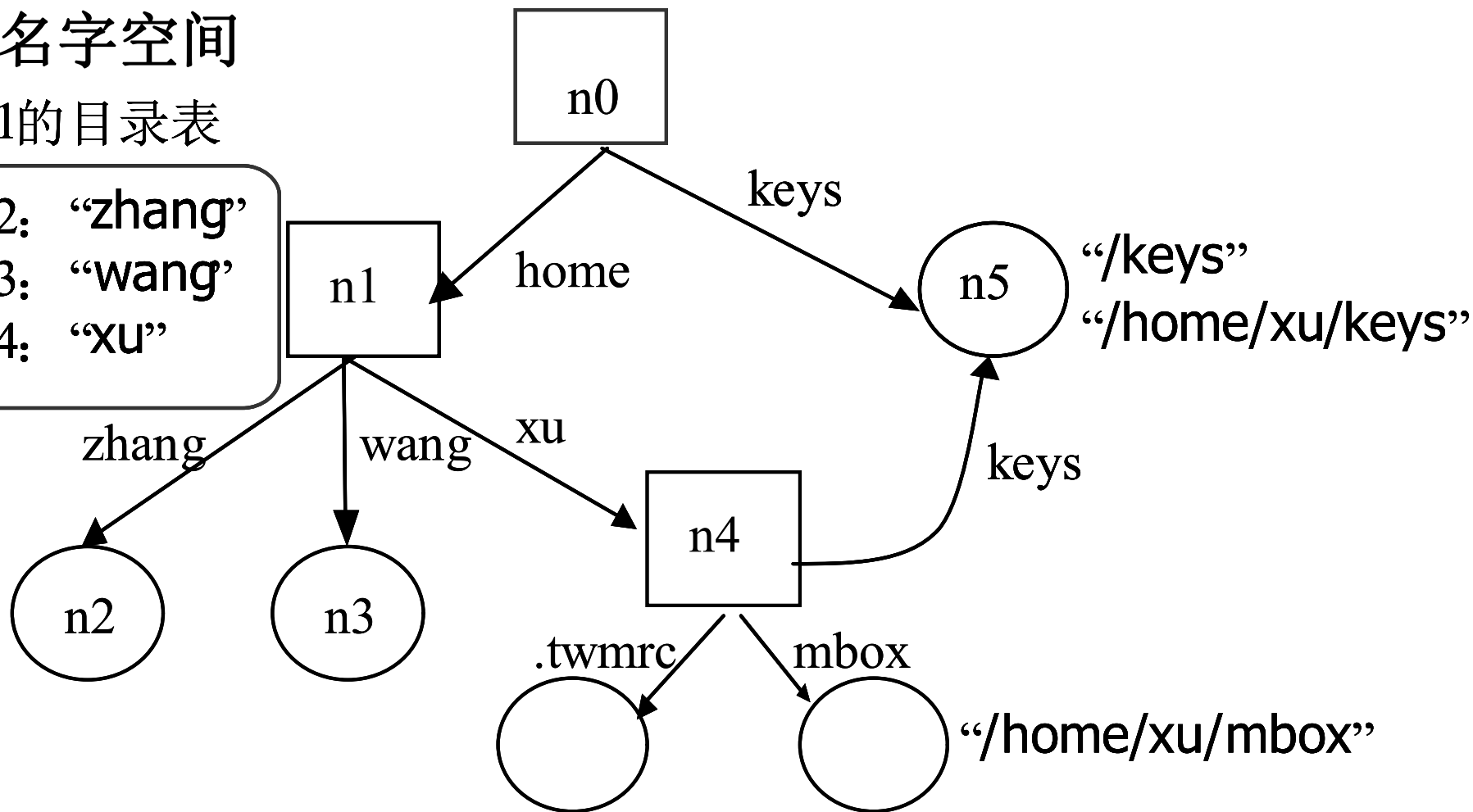
第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字空间

n1的目录表

n2: “zhang”
n3: “wang”
n4: “xu”

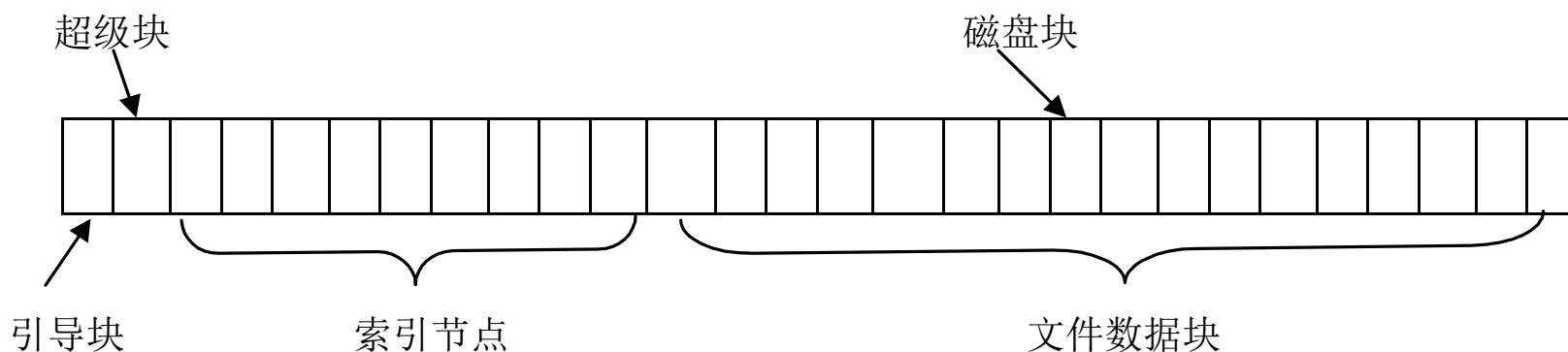


第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字空间

UNIX文件系统命名图的实现



通过一个逻辑磁盘上相邻的一些块完成的，这些块被分为引导块(boot block)、超级块(superblock)、一系列的索引节点(index nodes — inodes)和文件数据块(file data blocks)。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字空间

UNIX文件系统命名图的实现

- 1) 引导块是一个特殊的块，当系统启动时该块中的数据和指令自动地被装入到内存中，通过该块中指令的执行将操作系统装入到内存中。
- 2) 超级块保存着整个文件系统的有关信息，这些信息包括文件系统的大小、磁盘中哪些块还未被分配、哪些索引节点未被使用等等。
- 3) 每个inode除了包含它所对应的文件的数据可以在磁盘中什么地方找到的确切信息外，还包含它的属主、产生的时间、最后一次修改的时间和保护等信息。每个目录都有一个带唯一索引号的inode，所以，命名图中的每个节点的标识符是它的inode索引号。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字解析

在名字空间中，根据节点的路径名，就可以寻找到这个节点所存储的任何信息，这个查找的过程就称为名字解析。

1) 名字解析过程：N:<label-1, label-2, ..., label-n>

这个路径名的解析是从命名图中节点N开始的，首先在节点N的目录表中查找名字label-1，得到label-1所代表的节点的标识符；然后在label-1所代表的节点的目录表中查找名字label-2，得到label-2所代表的节点的标识符；此过程一直进行下去，如果N:<label-1, label-2, ..., label-n>在命名图中是实际存在的，就能够得到label-n所代表的节点的标识符，从而得到该节点的内容。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字解析

2) 名字解析的每个过程都分为两步完成:

第一步是找到对应节点的标识符，第二步是通过该标识符找到对应节点的内容。例如一个UNIX文件系统，如果一个节点是一个目录，首先我们要找到节点的inode号，通过inode号我们知道实际的目录表存放在磁盘的什么位置；如果一个节点是一个文件，首先我们要找到节点的inode号，通过inode号我们知道实际的文件数据存放在磁盘的什么位置。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字解析

3) 名字空间进行合并的机制有两种：安装(mount)机制；设置一个新的根节点。

a) 安装机制

在分布计算系统中，要安装一个外部名字空间至少需要如下信息：

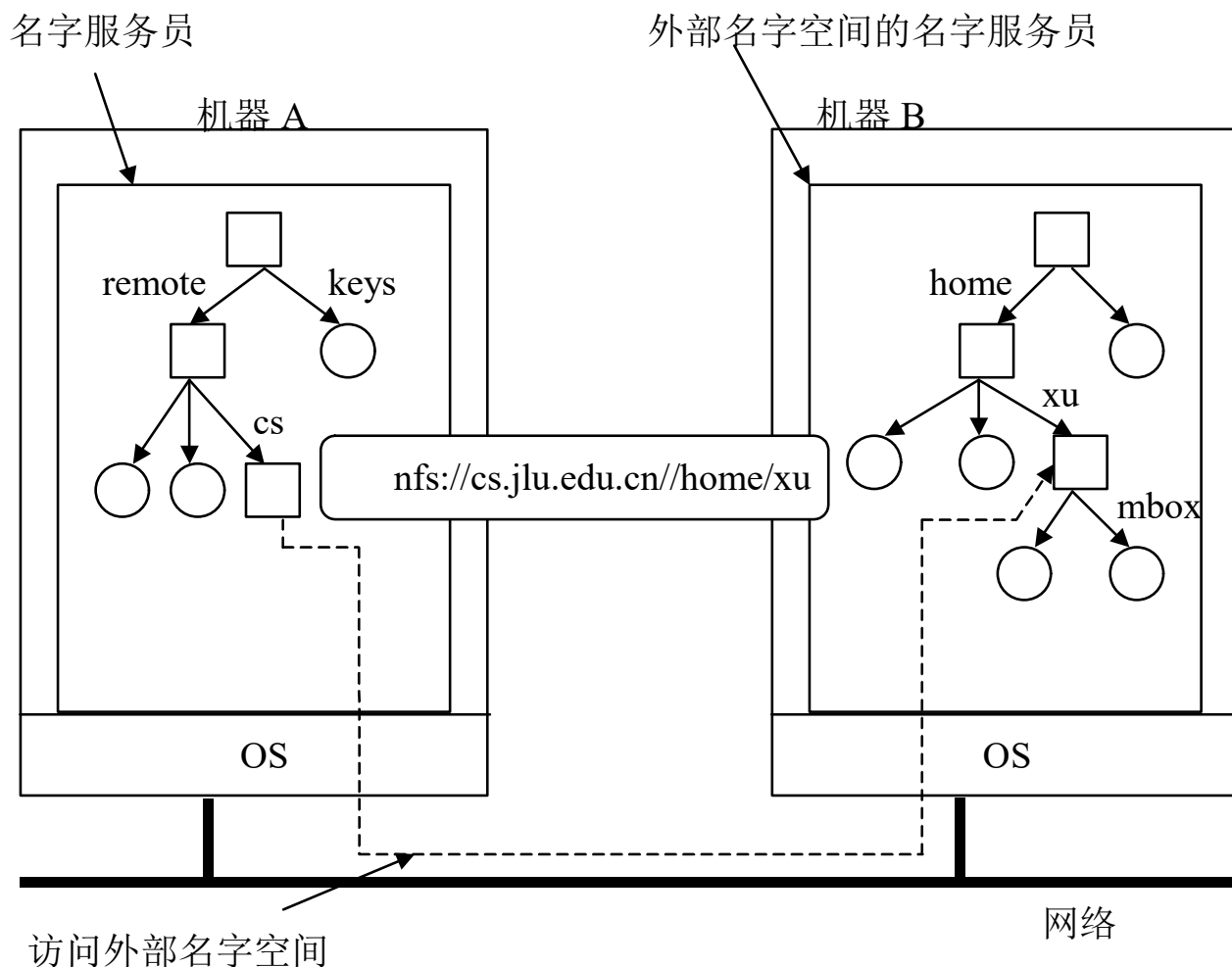
- (1) 访问协议名字；
- (2) 服务员名字；
- (3) 外部名字空间中被安装点的名字。

如nfs://cs.jlu.edu.cn//home/xu，nfs是访问协议名字，cs.jlu.edu.cn是服务员名字，/home/xu是外部名字空间中被安装点的名字。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字解析 安装机制



第四章 命名与保护

4.1 分布式系统中的命名

❖ 名字解析

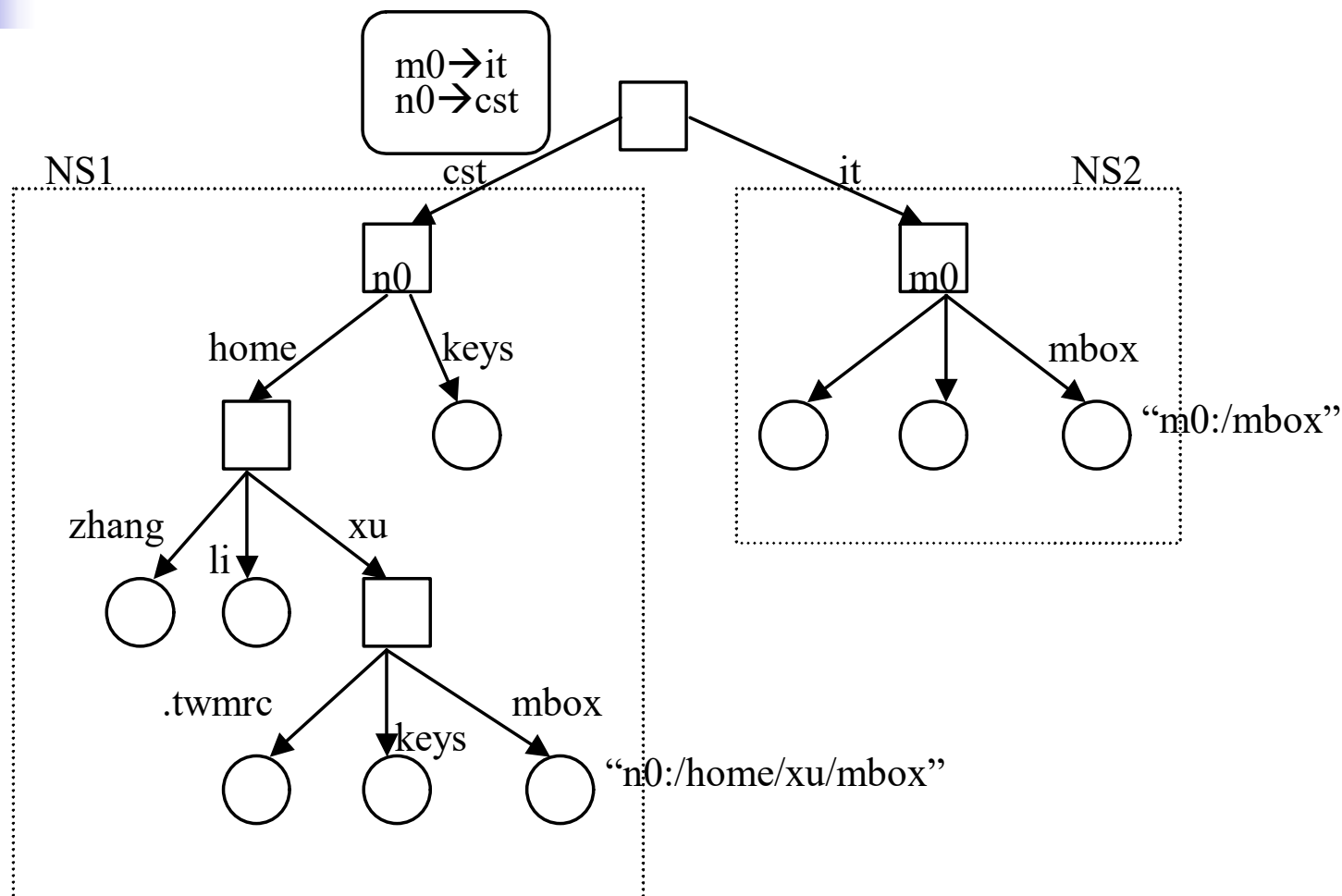
b) 设置新的根节点

- 1) 设置一个新的根节点，现有的各名字空间的根节点变成此新的根节点的子节点。
- 2) 原先的绝对路径名使用时改为相对路径名，每个机器包含一个隐含节点标示符，此节点 添加到绝对路径名前，变成相对路径名。如下图：

/home/xu/keys → n0: /home/xu/keys

第四章 命名与保护

4.1 分布式系统中的命名



❖ 名字解析设置新的根节点

b)

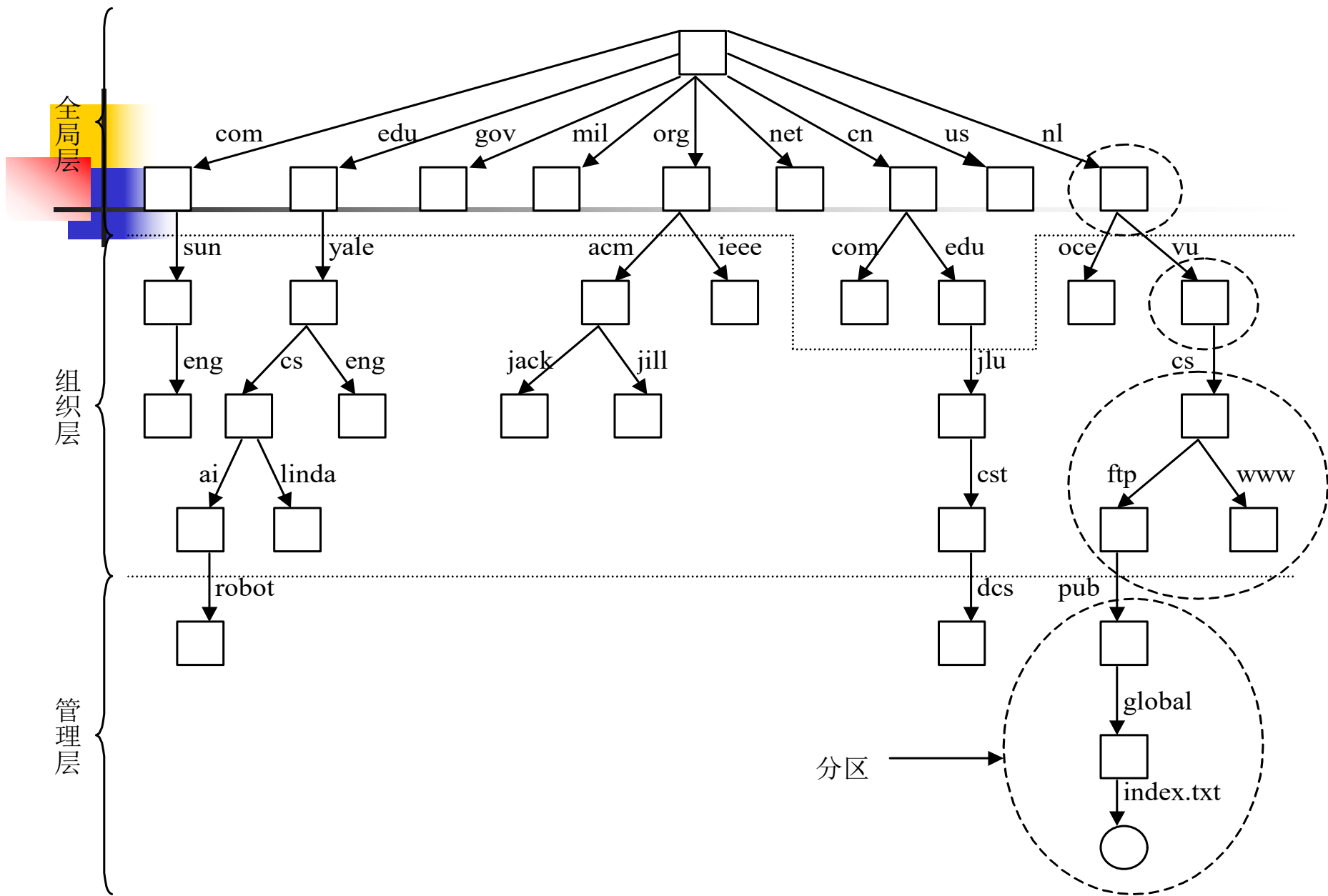
第四章 命名与保护

4.1 分布式系统中的命名

❖ 分布式系统中的名字空间的实现

➤ 大规模分布计算系统中名字空间的组织方式

- 1) 全局层由最上层的一些节点组成，这些节点包括根节点和它的一些子目录节点组成。用来代表一些机构，这些机构的名称保存在名字空间中。
- 2) 组织层由一个机构内的所有目录节点组成，这些目录节点被某个机构管理。
- 3) 管理层由那些可能经常变动的节点组成。



DNS 的部分名字空间

第四章 命名与保护

4.1 分布式系统中的命名

❖ 分布式系统中的名字空间的实现

➤ 对不同层次上的名字服务员的要求

要求	全局层	组织层	管理层
网络地理范围	全球	机构	部门
节点数目	少量	很多	大量
查询响应时间	几秒	毫秒	立即
更新传播周期	长	立即	立即
副本数目	很多	没有或很少	没有
顾客方缓存	需要	需要	有时需要

第四章 命名与保护

4.1 分布式系统中的命名

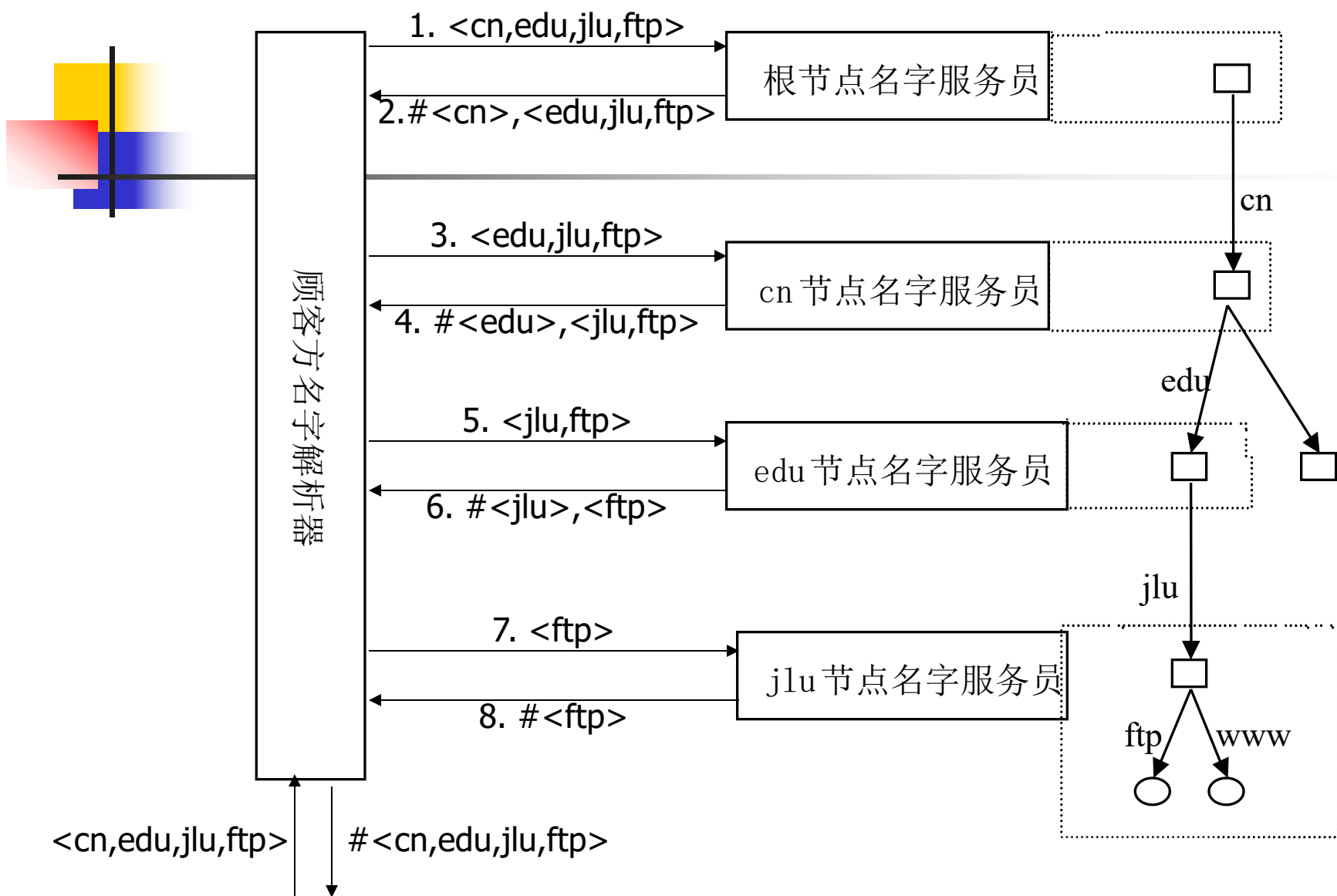
❖ 分布式系统中的名字空间的实现

➤ 大规模分布计算系统中名字解析

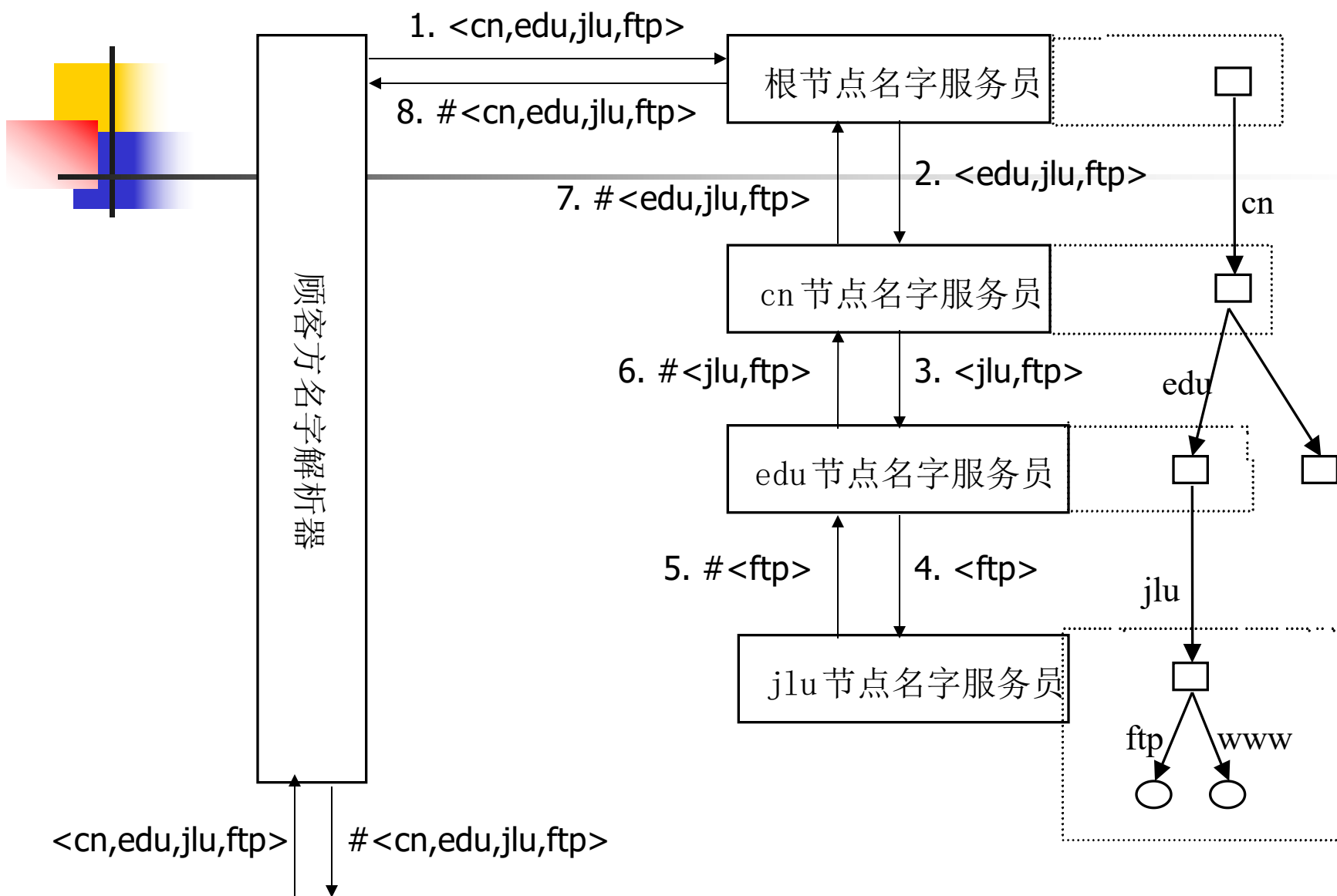
重复式名字解析；

递归式名字解析。

重复式名字解析



递归式名字解析



第四章 命名与保护

4.1 分布式系统中的命名

❖ 分布式系统中的名字空间的实现

➤ 大规模分布计算系统中名字解析

重复式名字解析和递归式名字解析的比较：

- 1) 递归式名字解析的主要缺点是要求每个名字服务员具有较高的性能。递归式名字解析要求名字服务员完整地解析它所得到的整个路径名，特别是对于全局层的名字服务员来说，情况更为严重。
- 2) 递归式名字解析有两个主要优点。第一个优点是如果采用缓存，那么递归式名字解析的缓存效果同重复式名字解析的缓存效果相比更为有效；第二个优点是可以减少通信代价。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 分布式系统中的名字空间的实现

➤ 大规模分布计算系统中名字解析

重复式名字解析和递归式名字解析的比较：

对名字 root:<cn,edu,jlu,ftp>解析完成后相关名字服务员的缓存内容

对应节点的名字服务员	需要解析的部分 路径	需要查询的地址	要求其他名字 服务员完成的部分	收到并缓存的信息	以后可直接返回给 请求者的信息
jlu	<ftp>	#<ftp>	--	--	#<ftp>
edu	<jlu,ftp>	#<jlu>	<ftp>	#<ftp>	#<jlu> #<jlu,ftp>
cn	<edu,jlu,ftp>	#<edu>	<jlu,ftp>	#<jlu> #<jlu,ftp>	#<edu> #<edu,jlu> #<edu,jlu,ftp>
root	<cn,edu,jlu,ftp>	#<cn>	<edu,jlu,ftp>	#<edu> #<edu,jlu> #<edu,jlu,ftp>	#<cn> #<cn,edu> #<cn,edu,jlu> #<cn,edu,jlu,ftp>

第四章 命名与保护

4.1 分布式系统中的命名

❖ 命名系统实例--DNS

- 1) DNS的名字空间：DNS的名字空间被一个树形图组织成分层结构。弧的标号是由数字和字母组成的字符串，标号的最大长度为63个字符。一个路径名的最大长度为255个字符，一个路径名是由一串标号组成的，起始的标号在路径名的最右边，标号之间由点（“.”）分隔，根节点也用点“.”表示。例如一个路径名root:<cn, edu, jlu, ftp>在DNS中表示为ftp. jlu. edu. cn. ，最右边的点“.”代表根节点，但是在使用中常将最右边的点“.”省略。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 命名系统实例--DNS

- 2) 域(domain) : DNS名字空间中的一个子树称为一个域(domain), 到达这个子树的根节点的路径名称称为这个域的域名(domain name)。正如路径名有绝对路径名和相对路径名一样, 域名也可以是绝对域名和相对域名。
- 3) DNS的名字空间被划分为不重叠的部分, 这些部分被称为分区(zone), 一个分区是名字空间的一部分, 由一个独立的名字服务员来实现。
- 4) 资源记录(resource records) 。DNS名字空间中节点所包含的内容由一些资源记录所组成, 资源记录有不同的类型。

第四章 命名与保护

4.1 分布式系统中的命名

❖ 命名系统实例--DNS

DNS 中最重要的一些资源记录的类型

记录类型	相关实体	描述
SOA	分区	保持对应分区的相关信息
A	主机	包含该节点所代表主机的 IP 地址
MX	域	发向该域的邮件由 MX 指定的邮件服务器处理
SRV	域	指向该域中处理特定服务的服务器
NS	分区	指明该分区中一个名字服务员的名字
CNAME	节点	用在别名节点中，到正式名字节点的一个符号连接
PTR	主机	用于 IP 地址命名的节点中，包含主机的正式名字
HINFO	主机	包含该节点所代表的主机的相关信息
TXT	任何实体	包含与该实体有关的任何有用信息

分区 cs.vu.nl 的 DNS 数据库的部分内容

名字	记录类型	记录值
cs.vu.nl	SOA	star(1999121502,7200,3600,2419200,86400)
cs.vu.nl	NS	star.cs.vu.nl
cs.vu.nl	NS	top.cs.vu.nl
cs.vu.nl	NS	solo.cs.vu.nl
cs.vu.nl	TXT	“Vrije Universiteit – Math. & Comp. Sc.”
cs.vu.nl	MX	1 zephyr.cs.vu.nl
cs.vu.nl	MX	2 tornado.cs.vu.nl
cs.vu.nl	MX	3 star.cs.vu.nl
star.cs.vu.nl	HINFO	Sun Unix
star.cs.vu.nl	MX	1 star.cs.vu.nl
star.cs.vu.nl	MX	10 zephyr.cs.vu.nl
star.cs.vu.nl	A	130.37.24.6
star.cs.vu.nl	A	192.31.231.42
zephyr.cs.vu.nl	HINFO	Sun Unix
zephyr.cs.vu.nl	MX	1 zephyr.cs.vu.nl
zephyr.cs.vu.nl	MX	2 tornado.cs.vu.nl
zephyr.cs.vu.nl	A	192.31.231.66
www.cs.vu.nl	CNAME	soling.cs.vu.nl
ftp.cs.vu.nl	CNAME	soling.cs.vu.nl
soling.cs.vu.nl	HINFO	Sun Unix
soling.cs.vu.nl	MX	1 soling.cs.vu.nl
soling.cs.vu.nl	MX	10 zephyr.cs.vu.nl
soling.cs.vu.nl	A	130.37.24.11
laser.cs.vu.nl	HINFO	PC MS-DOS
laser.cs.vu.nl	A	130.37.30.32
vucs-das.cs.vu.nl	PTR	0.26.37.130.in-addr.arpa
vucs-das.cs.vu.nl	A	130.37.26.0

第四章 命名与保护

4.1 分布式系统中的命名

❖ 命名系统实例—DNS

分区之间的关联：

域 `vu.nl` 到子域 `cs.vu.nl` 的关联

名字	记录类型	记录值
<code>cs.vu.nl</code>	NS	<code>solo.cs.vu.nl</code>
<code>solo.cs.vu.nl</code>	A	130.37.24.1

第四章 命名与保护

4.2 加密技术

❖ 保护和安全的三个方面：

- 1) 数据加密：主要研究对数据加密的算法、算法实现的效率等问题。
- 2) 计算机网络的安全保密：计算机网络的目的是资源共享，同时也是分布计算系统、网格系统的基础平台，但网络容易产生不安全和失密问题。方便并且行之有效的方法是采用加密技术，实现用户到用户之间的加密，确保数据在网络传输过程中(特别是通信系统中)不丢失和被篡改。
- 3) 访问控制：像多用户单机系统那样，规定什么人可以访问系统的什么数据，得到何种服务，并对用户进行身份鉴别，对他们的访问权限加以限制(授权)。

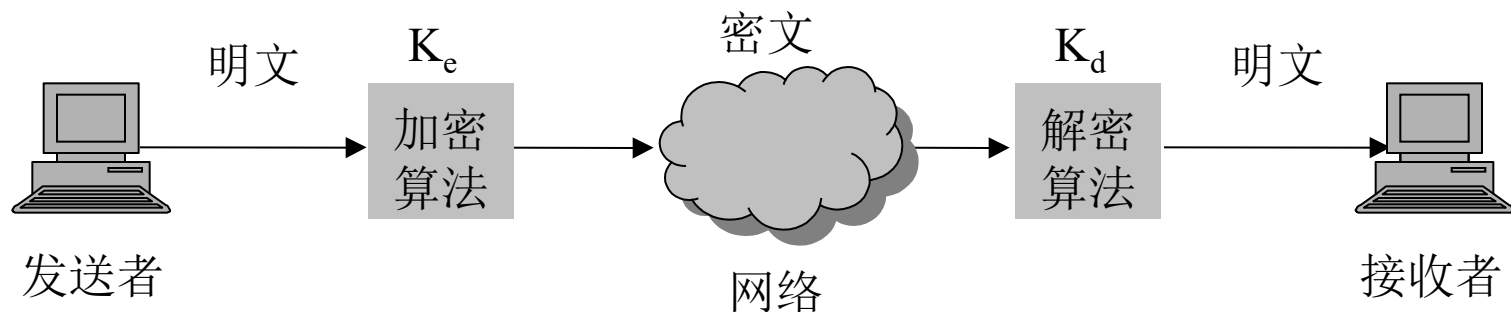
上述三个问题归并成两个：加密技术，访问控制。

第四章 命名与保护

4.2 加密技术

❖ 加密概念：

- 1) 加密和解密：加密意味着发送者将信息从最初的格式改变为另一种格式，将最终不可阅读的消息通过网络发送出去。解密是加密的相反过程，它将消息变换回原来的格式。
- 2) 加密和解密的过程：



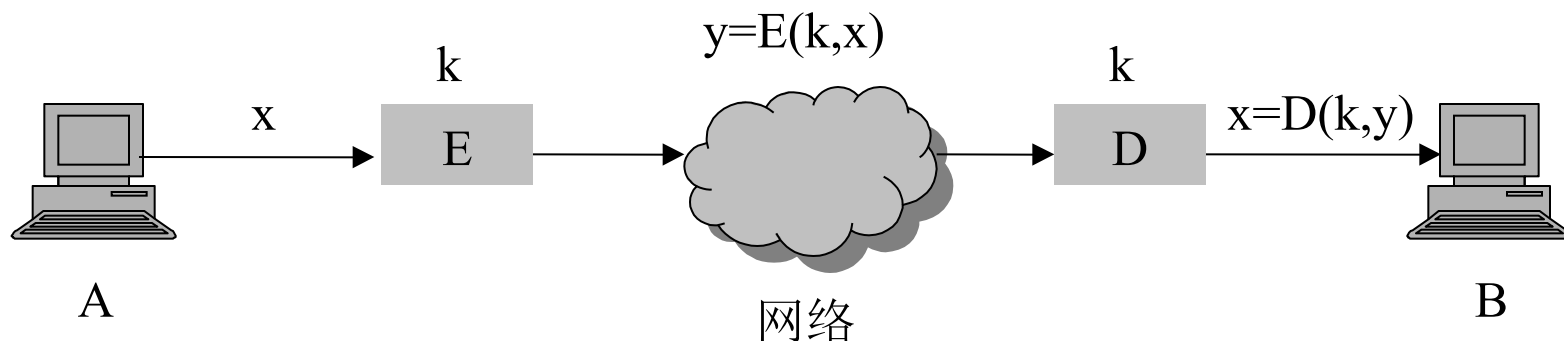
- 3) 加密和解密的方法分为两种类型：传统方法和公开密钥方法。

第四章 命名与保护

4.2 加密技术

❖ 传统加密方法:

1) 单密钥系统加密模型



- 2) 密钥: 理论上, 加密和解密算法应该经常更换, 可是, 能够经得起攻击的过硬算法并不是很容易寻找和设计的, 经常更换加密和解密算法是不现实的, 因此才使用密钥。加密和解密算法可以长时间使用, 但是密钥应该经常更换。

第四章 命名与保护

4.2 加密技术

❖ 传统加密方法：

- 3) 替换法：在这种加密方法中，每个字符都被另一个字符所代替。这种加密方法称为恺撒密码，因为它最初被恺撒使用。
- 4) 位置交换法：在这种方法中，字符将保持它们在原文中的格式，但是它们的位置将被改变来创建密文。

第四章 命名与保护

4.2 加密技术

❖ 传统加密方法:

位置交换法实例：密钥MEGABUCK，

明文：please give me the books, when you come up next.

M	E	G	A	B	U	C	K
7	4	5	1	2	8	3	6
p	l	e	a	s	e		g
i	v	e		m	e		t
h	e		b	o	o	k	s
,		w	h	e	n		y
o	u		c	o	m	e	
u	p		n	e	x	t	.

密文：a bhcnsmoeoe k etlve upee w gtsy .pih,oueeonmx

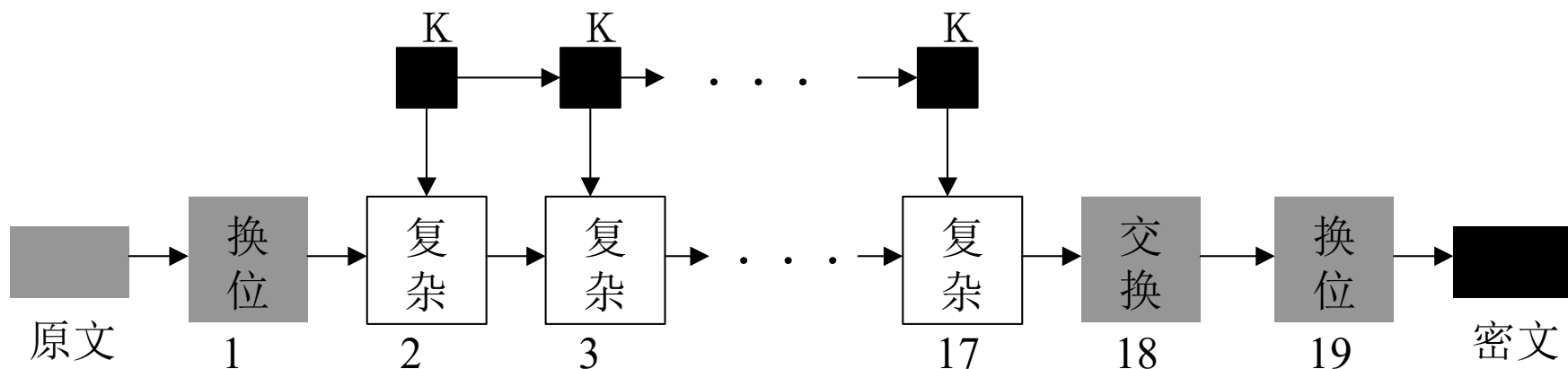
第四章 命名与保护

4.2 加密技术

❖ 传统加密方法:

5) DES加密

算法使用64比特的原文和56比特的密钥，原文经过19个不同而复杂的过程来产生一个64比特的密文。



第四章 命名与保护

4.2 加密技术

❖ 传统加密方法：

6) 密钥的分配

解决如何通过系统本身保密传送密钥的问题？

两个用户要进行对话，对话之前要商定一个对话密钥，由于它们之间没有一个对话密钥，密钥的传送是不能用明文的形式传送的，所以需要有一个双方都信任的第三者帮助他们完成密钥的分配，将这个第三者叫做网络安全中心(NSC)。

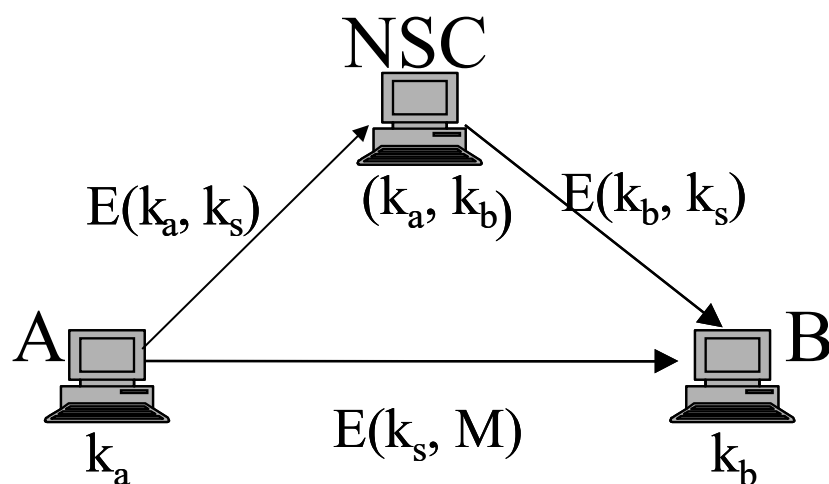
第四章 命名与保护

4.2 加密技术

❖ 传统加密方法:

6) 密钥的分配

密钥分配两方案:



(a)接收者从NSC得到密钥

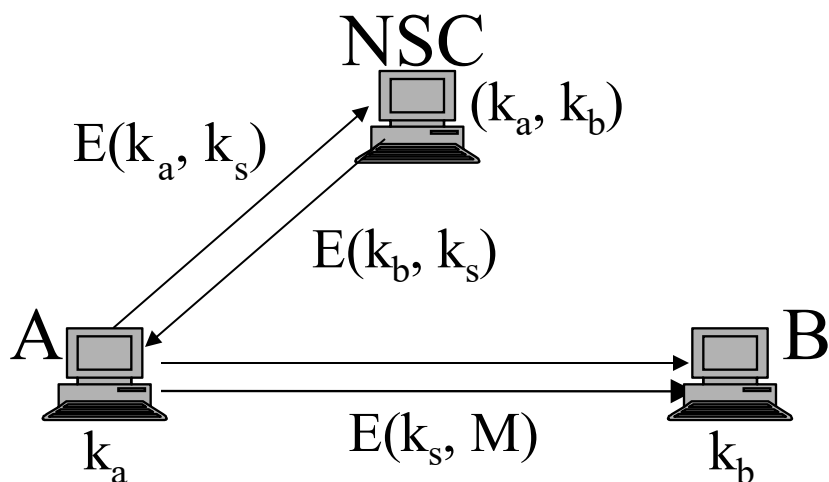
第四章 命名与保护

4.2 加密技术

❖ 传统加密方法:

6) 密钥的分配

密钥分配两方案:



(b)接收者从发起者得到密钥

第四章 命名与保护

4.2 加密技术

❖ 公开密钥加密方法：

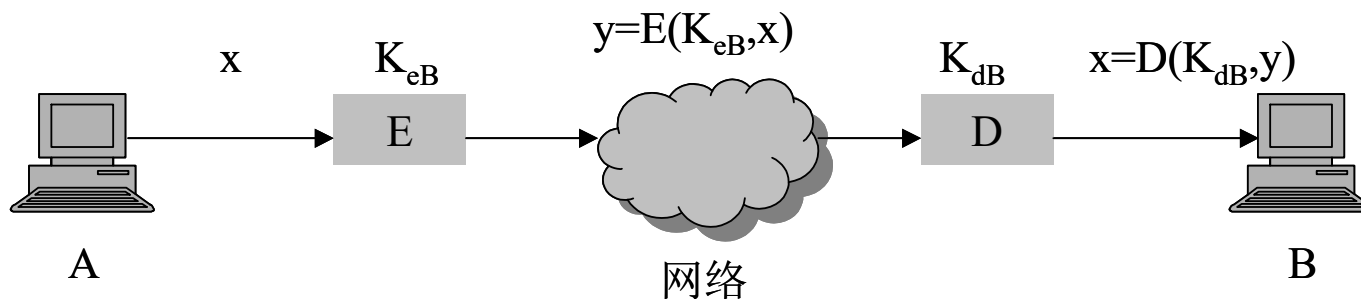
- 1) **公开密钥加密方法的思想：** 加密算法E和解密算法D无法保持秘密，不如干脆公开，但是使用两个密钥：加密密钥 K_e 和解密密钥 K_d 。加密密钥是不保密的，谁都可以使用，所以叫做公开密钥；解密密钥是保密的，只有接收密文的一方才知道，所以叫做专用密钥或保密密钥。选择某种类型的算法E和算法D，使得局外人即使知道了加密密钥 K_e ，也推算不出来解密密钥 K_d 。

第四章 命名与保护

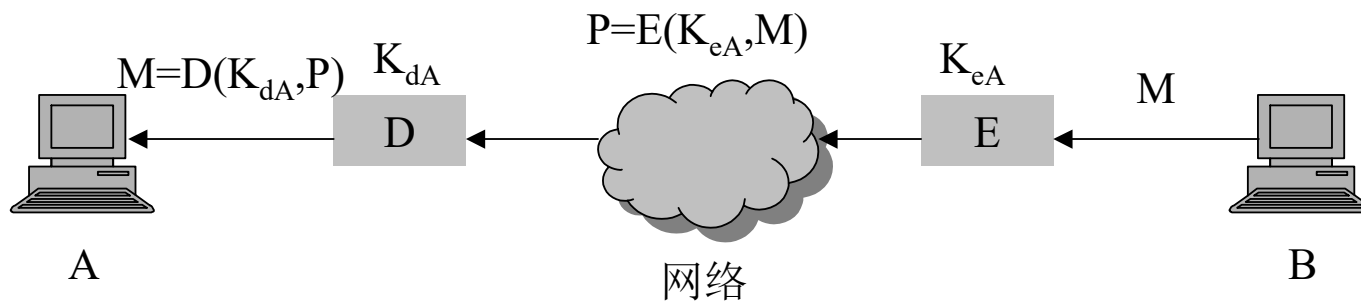
4.2 加密技术

❖ 公开密钥加密方法:

2) 公开密钥加密方法的加密模型:



(a) A向B发送保密信息 x



(b) B向A发送保密信息 M

第四章 命名与保护

4.2 加密技术

❖ 公开密钥加密方法:

3) RSA 加密:

a) 确定密钥的算法:

- (1) 选择两个素数, p 和 q 。
- (2) 计算 $n=p \times q$ 和 $z=(p-1) \times (q-1)$ 。
- (3) 选择一个与 z 互质的数 d 。
- (4) 找出 e , 使得 $e \times d = 1 \pmod{z}$ 。

确定公开密钥和保密密钥: 公开密钥由 (e, n) 构成, 保密密钥由 (d, n) 构成。

b) 加密的过程: 设 X 是明文, 计算 $Y = X^e \pmod{n}$, 则 Y 就是密文。

c) 解密的过程: 设 Y 是密文, 计算 $X = Y^d \pmod{n}$, 则 X 就是明文。

第四章 命名与保护

4.2 加密技术

❖ 公开密钥加密方法:

RSA 加密实例：对明文“SUZANNE”进行加密。在此例子中我们只考虑英文大写字母的加密，我们可以对英文字母A~Z按顺序编码为1~26。在该例子中，我们选择 $p=3$ ， $q=11$ ，得到 $n=33$ ， $z=(p-1) \times (q-1)=2 \times 10=20$ 。由于7和20互为质数，故可以设 $d=7$ 。对于所选的 $d=7$ ，解方程 $7e=1 \pmod{20}$ ，可以得到 $e=3$ 。

第四章 命名与保护

4.2 加密技术

❖ 公开密钥加密方法:

RSA 加密实例的结果:

		加密		解密		
明文(X)	X^3	密文(Y)	Y^7	$Y^7 \pmod{33}$	符号	
符号	数值	$X^3 \pmod{33}$				
S	19	6859	28	13492928512	19	S
U	21	9261	21	1801088541	21	U
Z	26	17576	20	1280000000	26	Z
A	1	1	1	1	1	A
N	14	2744	5	78125	14	N
N	14	2744	5	78125	14	N
E	5	125	26	8031810176	5	E

第四章 命名与保护

4.3 保护

❖ 保护的目标与要求

- 1) 保护的目标：保护机构阻止非法用户偷用磁盘空间、文件系统、处理机，读其他人的文件，修改别人的数据库及干扰别人的计算。保护机构还有其他作用，帮助检测程序中的差错，防止用户错误操作以及用于项目管理等。
- 2) 保护机构要解决以下几个基本问题：
 - a) 保密性：用户必须能秘密地保存数据，不被其他用户看见；
 - b) 专用性：必须保证用户给出的信息仅用于达到预想的目的；
 - c) 真实性：提供给用户的数据必须是真实的，也就是说，如果某些数据声称来自X，则该用户必须能够验证此数据确实是由X送来的；
 - d) 完整性：存放在系统中的数据不会被系统或未被授权的用户破坏。

第四章 命名与保护

4.3 保护

❖ 公开密钥加密技术实现数字签名

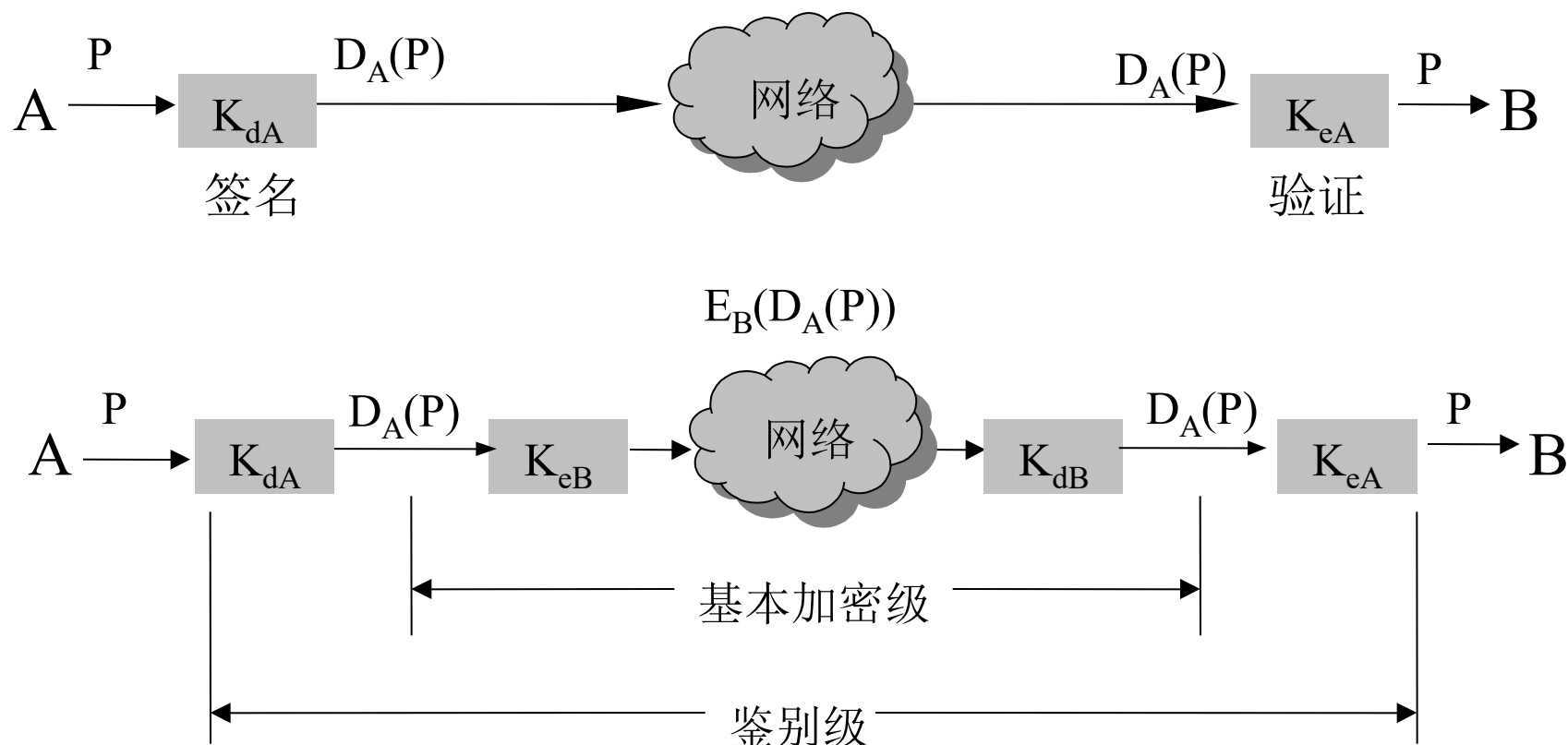
- 1) 实现数字签名，要解决两个问题：第一，接收者能验证所要求的发送者的身份；第二，发送者在发送已经签名的报文后不能否认。
- 2) 公开密钥加密技术实现数字签名要求加密函数E和解密函数D满足下列条件--就是E和D可以互换：
 $E(D(P))=P$ ，当然同时还有 $D(E(P))=P$

第四章 命名与保护

4.3 保护

❖ 公开密钥加密技术实现数字签名

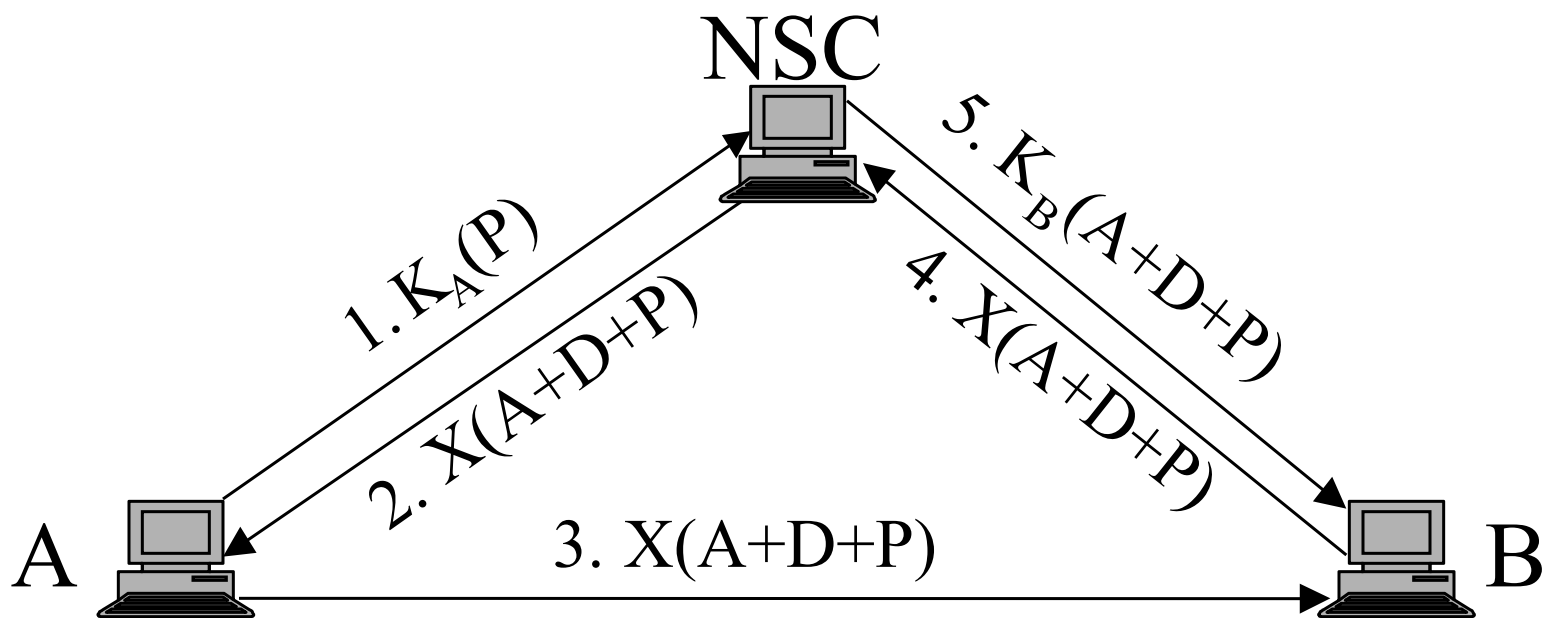
3) 公开密钥加密技术实现数字签名的过程：



第四章 命名与保护

4.3 保护

❖ 单密钥加密技术实现数字签名



第四章 命名与保护

4.3 保护

❖ 使用报文摘要实现数字签名

- 1) 报文摘要：使用单向散列(hash)函数可以从一段很长的明文中计算出固定长度的比特串。这个比特串通常被称为该报文的摘要(message digest)。
- 2) 报文摘要具有三个重要的属性：
 - a) 给出报文P就很容易计算出其报文摘要MD(P)。
 - b) 只给出MD(P)，几乎无法推导出P。
 - c) 无法生成这样的两条报文，它们具有同样的报文摘要。

第四章 命名与保护

4.3 保护

❖ 使用报文摘要实现数字签名

3) 报文摘要实现数字签名的过程:

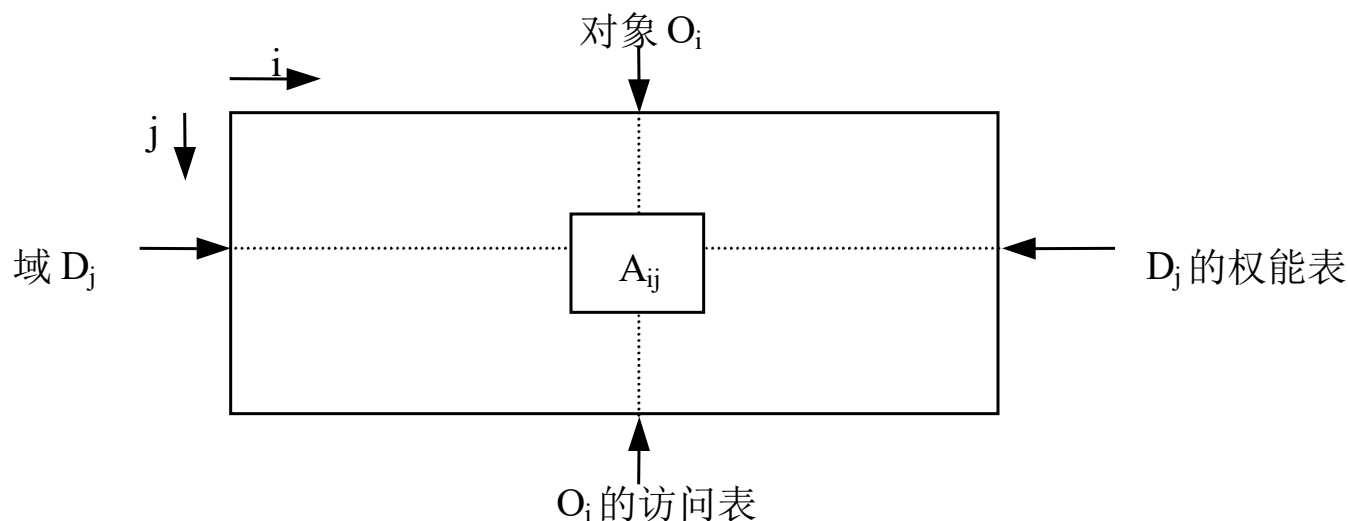
- a) A首先计算P的报文摘要 $MD(P)$;
- b) A用自己的保密密钥对 $MD(P)$ 进行加密以达到签名的目的;
- c) A将报文摘要的签名形式 $D_A(MD(P))$ 连同明文P一起发送给B;
- d) B用A的公开密钥解密 $D_A(MD(P))$ ，从而得到 $MD(P)$ ，验证是否是A发送的;
- e) B通过明文P重新计算 $MD(P)$ ，发现是否有非法用户修改报文P。

第四章 命名与保护

4.3 保护

❖ 权能的保护

- 1) 访问矩阵：矩阵中的项 A_{ij} 含有用户 j 对对象 i 的一些访问权限，它由一些数值代表对一个文件的读、写，或一个程序的执行，或对一个终端的连接，或改变权限等。



第四章 命名与保护

4.3 保护

❖ 权能的保护

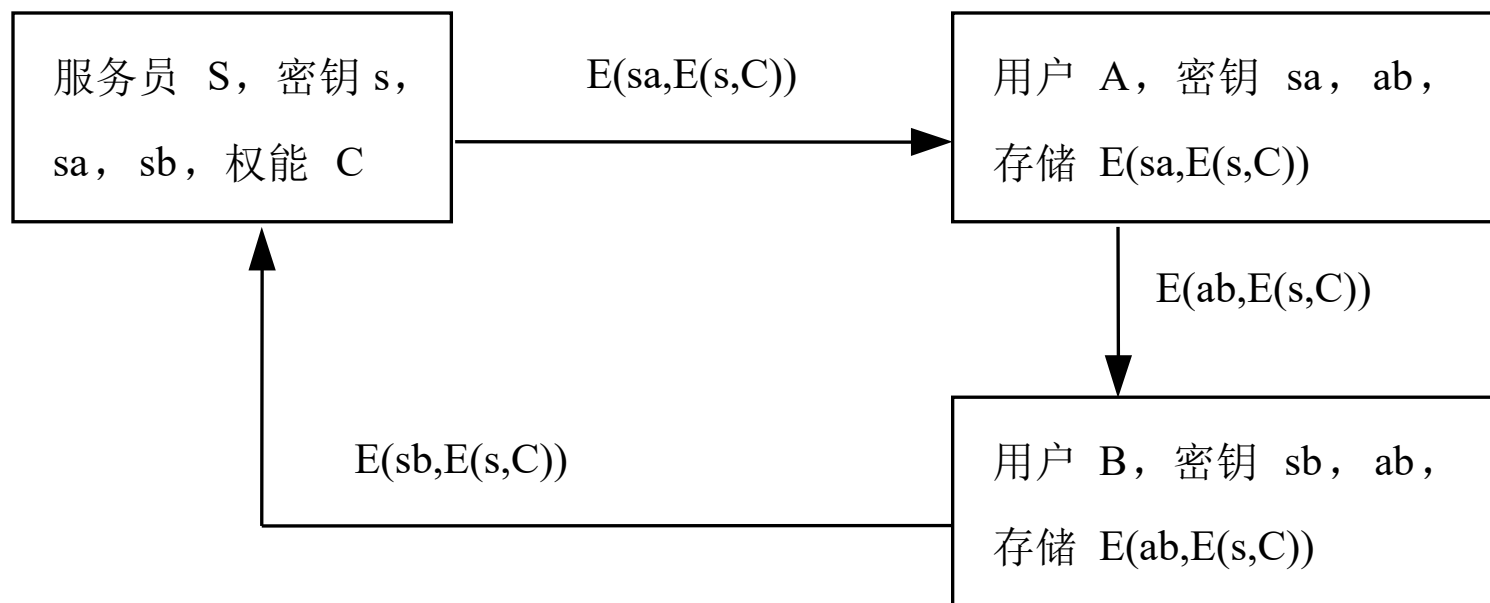
- 2) 权能：因为这个矩阵的很多项都是空的。有两种主要方法在实际系统中使用，实现访问控制：按行或按列列表。访问矩阵中对应 O_i 那一列说明可以访问对象 O_i 的所有的域，叫做关于 O_i 的访问控制表。另一种存储访问矩阵的方法是取一行列表，叫做权能(capability)表，它对应于一个域 D_j ，而不是对应某个对象。权能表的每一项叫做一个权能，由对象名和一套对此对象操作的权利组成。

第四章 命名与保护

4.3 保护

❖ 权能的保护

3) 使用单密钥加密技术保护权能：

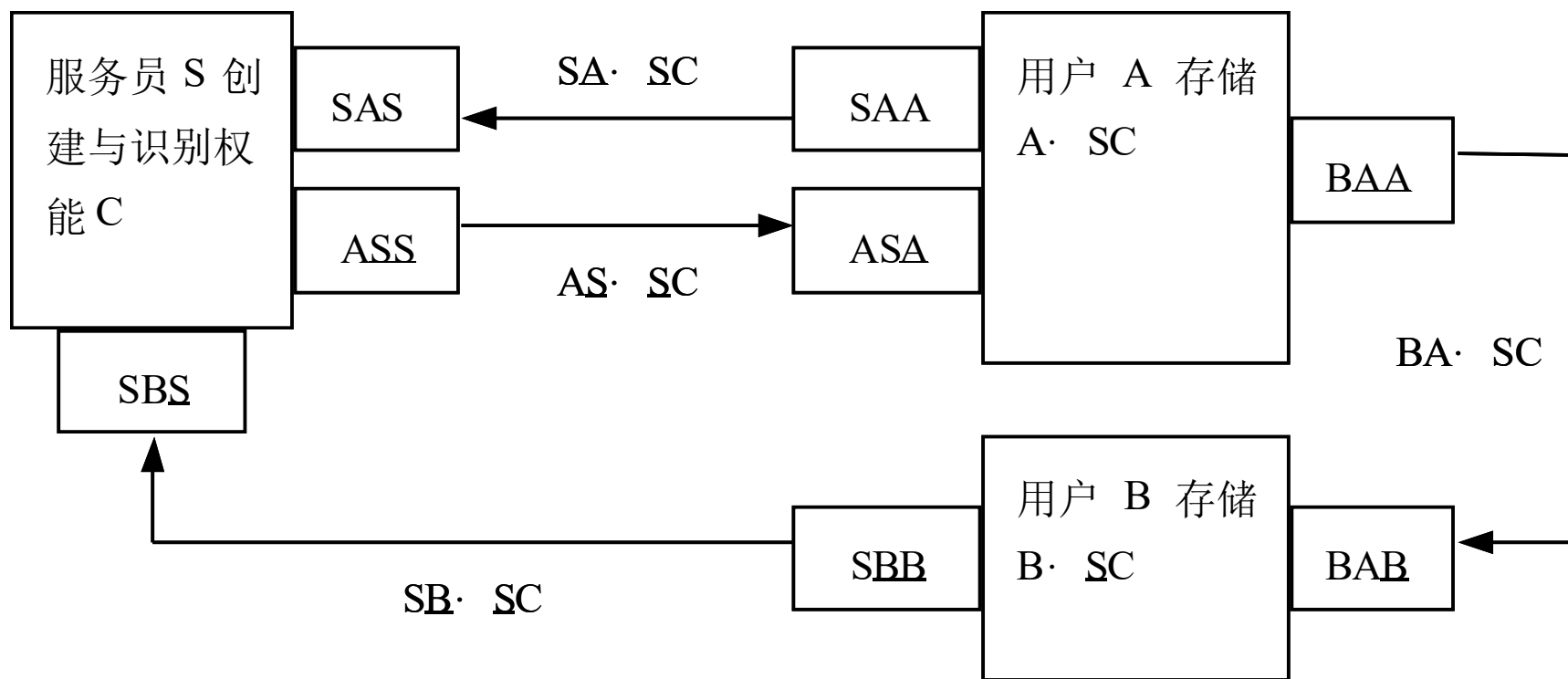


第四章 命名与保护

4.3 保护

❖ 权能的保护

4) 使用公开密钥加密技术保护权能：

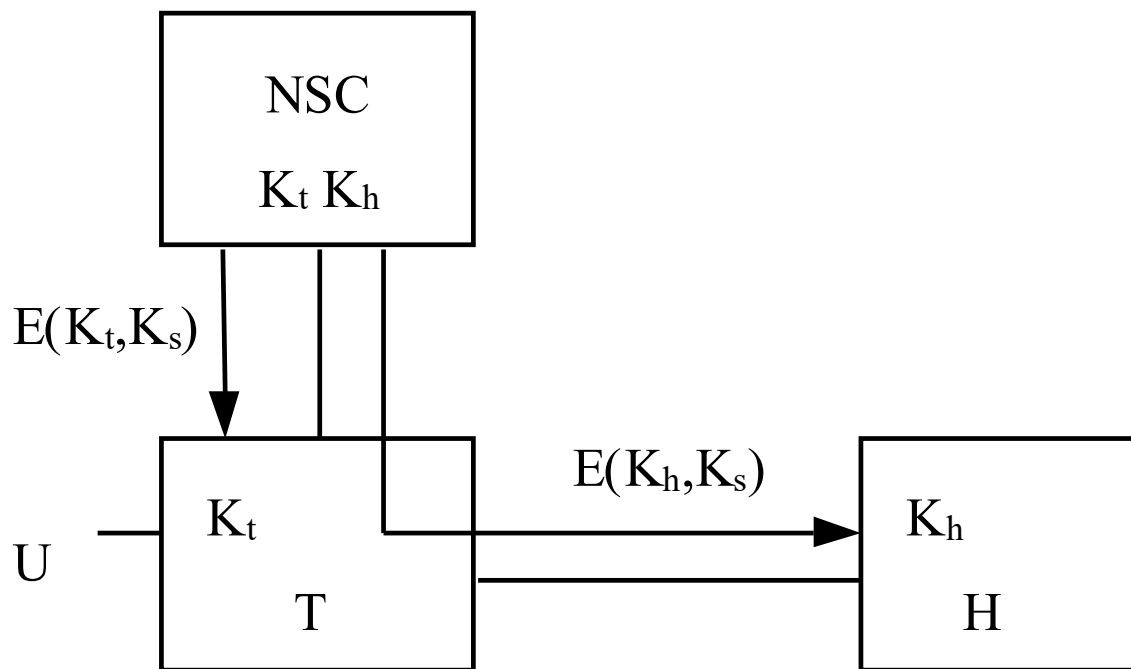


第四章 命名与保护

4.3 保护

❖ 分布系统中访问位置的控制

1) 向终端和主机传送对话密钥：

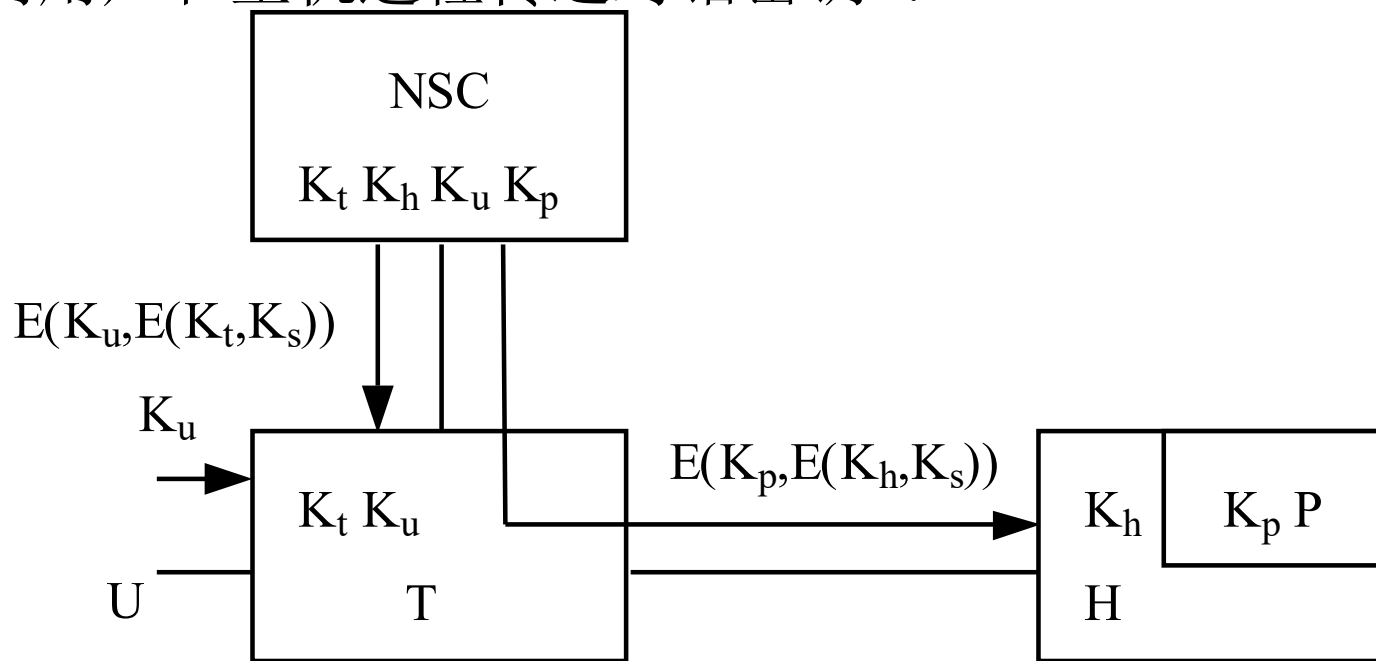


第四章 命名与保护

4.3 保护

❖ 分布系统中访问位置的控制

2) 向用户和主机进程传送对话密钥：



其他具有访问终端的密钥的用户不能伪装成用户U，主机上知道主机密钥 K_h 的其他进程也不能伪装成目的进程P

第四章 命名与保护

4.4 保护的例子：Amoeba

❖ 信口

网络上的任何顾客和服务员都有两个信口：G和P，G是保密的，P是公开的。 $P=F(G)$ ，由P不能推出G。

1) 通信规则：

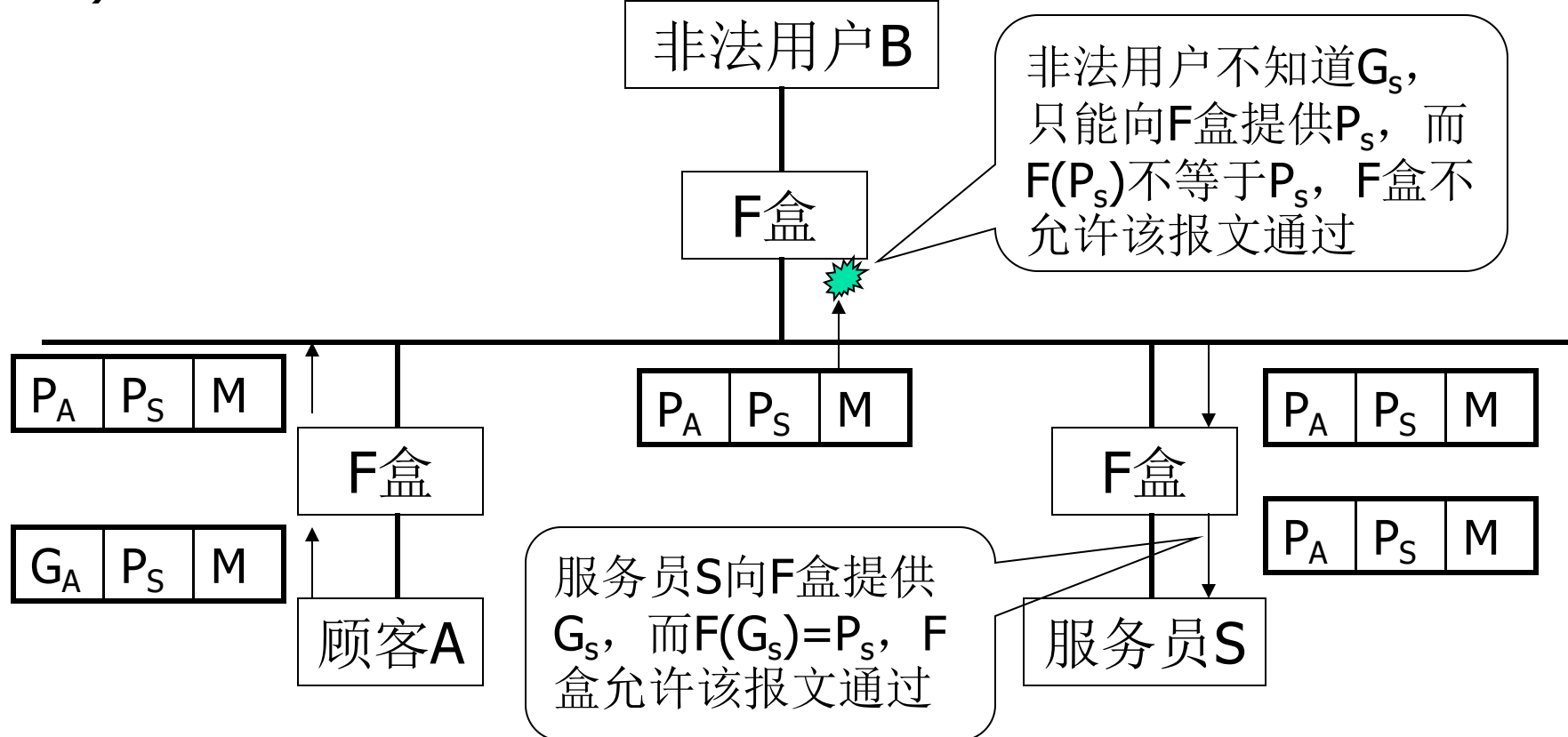
- (1) 任何发送出去的报文必须先经过F盒，才能进入网络；
- (2) 任何到达的报文必须经过F盒才能被用户接收；
- (3) 发送者发送报文时，报文的源信口是发送者的G信口，报文的目的信口是接收者的P信口；
- (4) 报文经过发送者的F盒时，F盒将源信口由G信口变换成P信口，目的信口不变换。
- (5) 接收者要接收报文，必须向F盒提交自己的G信口，如果报文的目的是信口 $P=F(G)$ ，则F盒将报文提交给接收者，否则不提交给接收者。

第四章 命名与保护

4.4 保护的例子: Amoeba

❖ 信口

2) 防止非法用户冒充服务员接收顾客请求:

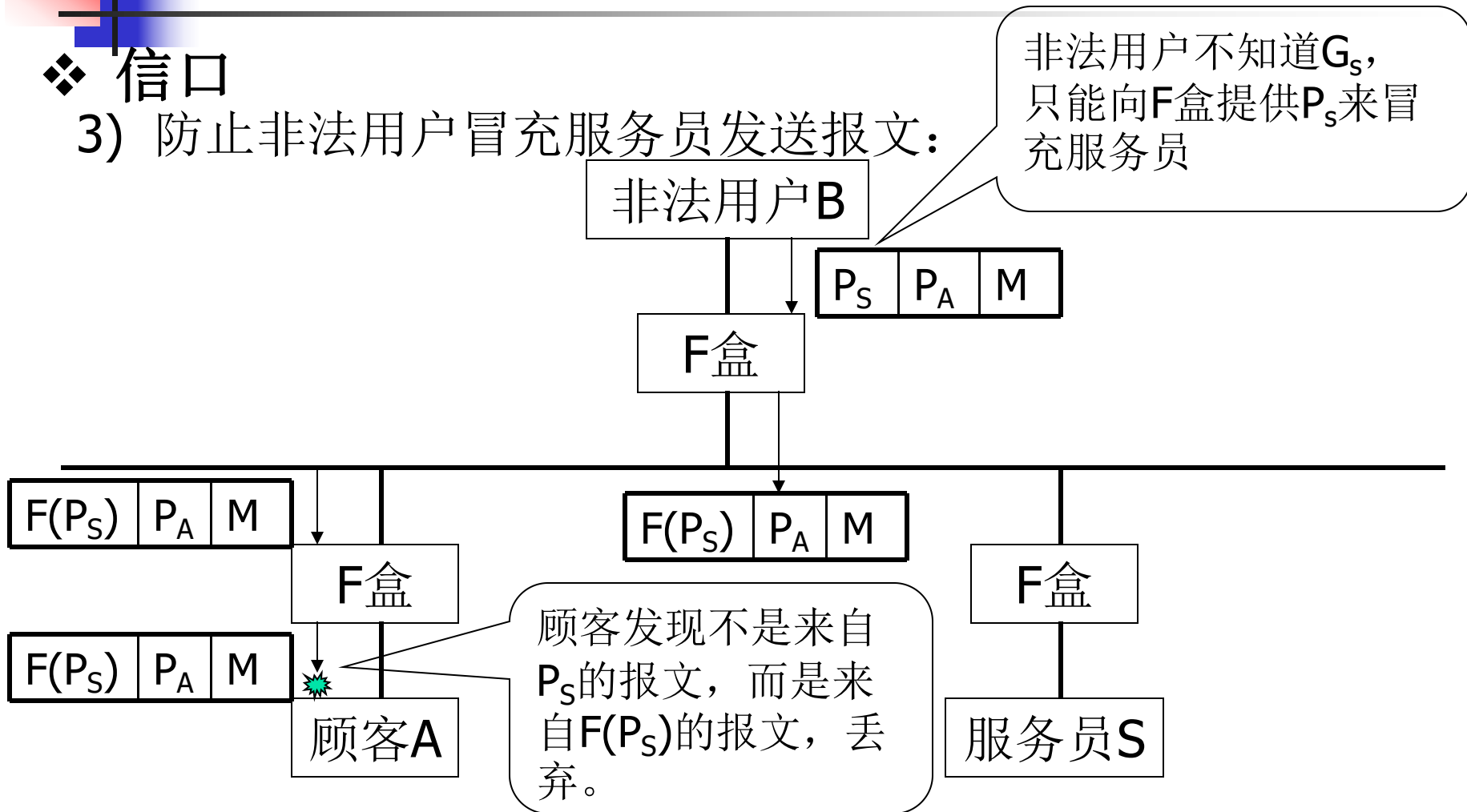


第四章 命名与保护

4.4 保护的例子: Amoeba

❖ 信口

3) 防止非法用户冒充服务员发送报文:



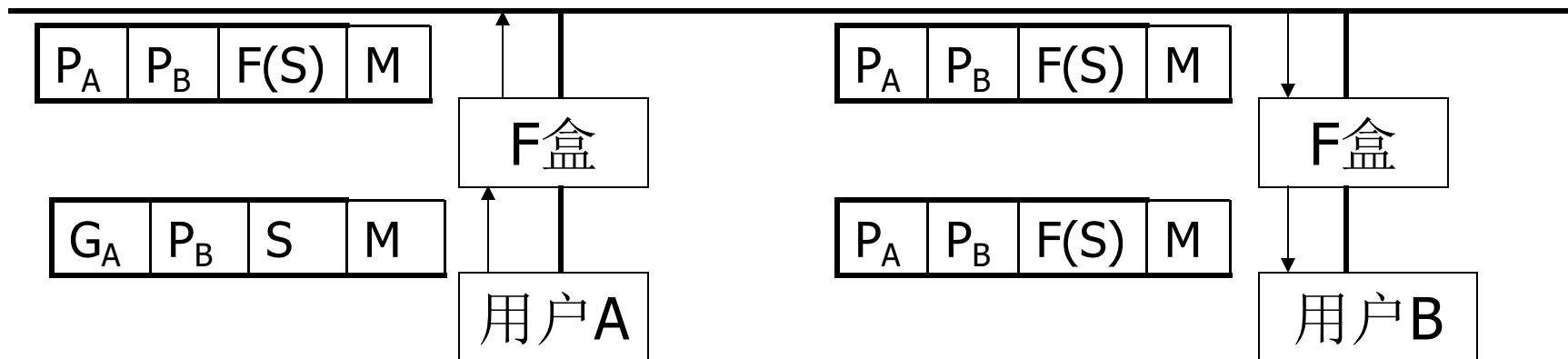
第四章 命名与保护

4.4 保护的例子：Amoeba

❖ 信口

4) 使用F盒实现数字签名：

顾客A选择一个随机签名 S 并公布 $F(S)$ 。A交给F盒用来发送的报文报头中包含三个特别的段：目的地(P_B)，源地址(G_A)和签名(S)。F盒对 G_A 和 S 进行单向函数变换，变换成 P_A 和 $F(S)$ 。B知道确实是A发送的，因为只有A才知道在第三段中放一个什么数，只有此数才能产生众所周知的 $F(S)$ 。



第四章 命名与保护

4.4 保护的例子：Amoeba

❖ 权能

1) 权能由四段组成：

服务员地址	对象标识符	权利码	检验码
64位	32位	32位	32位

服务员地址是拥有此对象的服务员地址put_port，对象标识符是一个内部标识符，只有服务员用它可以得知它所代表的对象，很像UNIX中的inode号。权利码中的每一位指出可以对此对象进行何种操作。检验码是一个大随机数，用来对该权能的真实性进行检验。

第四章 命名与保护

4.4 保护的例子：Amoeba

❖ 权能

2) 权能的保护：

- (1) 服务员创建对象A，产生一个随机数 R_A ；
- (2) 服务员给对象A创建权能，权利码为 P_A ，检验码为 C_A ；

服务员地址	A	P_A	C_A
-------	---	-------	-------

- (3) 用随机数 R_A 作为密钥对 P_A 和 C_A 加密；

服务员地址	A	$R_A(P_A)$	$R_A(C_A)$
-------	---	------------	------------

服务员保存A，随机数 R_A 和检验码为 C_A 。

- (4) 检验权能：用 R_A 对 $R_A(C_A)$ 解密，能得到 C_A ，则确认权能未被修改。

第四章 命名与保护

4.4 保护的例子：Amoeba

❖ 权能

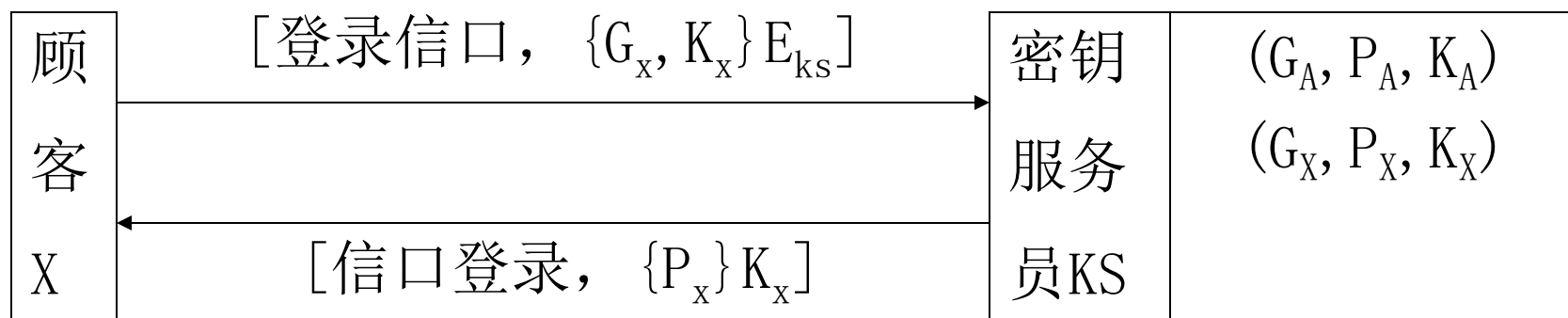
- 3) Amoeba权能保护方法的优点：
 - a) 服务员可在任何时候将某一对象的全部权能作废，这只需简单地改变检验码就可以了。
 - b) 尽管没有一个集中的机构来管理谁有什么权能，但是很容易改变现有的权能。对象的所有者希望这样做的时候只需简单地向服务员发一个请求报文，服务员简单地修改内部表中该项的随机数并返回一个新权能就可以了。
- 4) 缺点：缺点是如果顾客进程想把它所获得的权能传递给其他用户，但不想让此用户具有它的全部权限的操作能力时，必须再向服务员请求产生一个新的权能，顾客本身不能简单地修改其权能。它必须在发向服务员的报文中说明如何修改操作码。将权能和一个屏蔽码送给服务员，服务员将屏蔽码和权利码进行“与”操作就可以实现这一修改。

第四章 命名与保护

4.4 保护的例子: Amoeba

❖ 用软件F盒保护

1) 请求者X向密钥服务员登录:

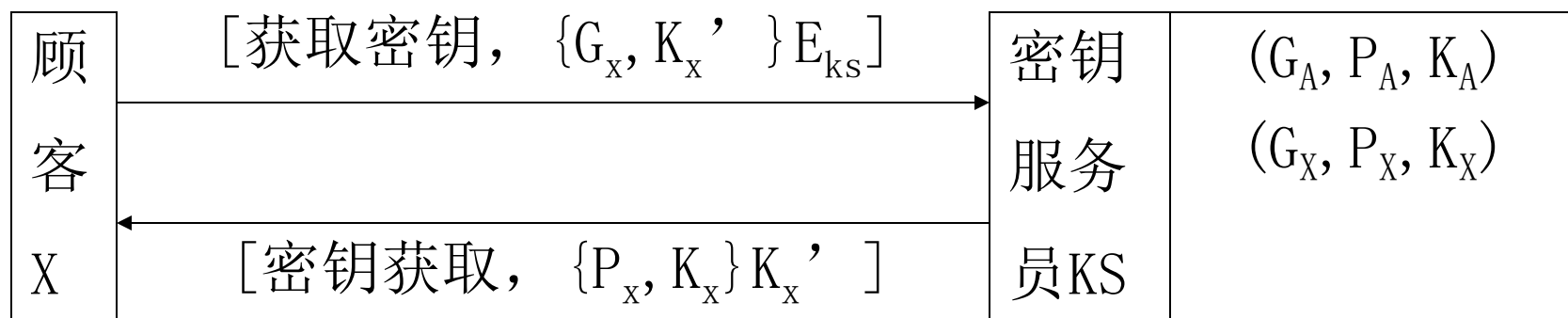


第四章 命名与保护

4.4 保护的例子: Amoeba

❖ 用软件F盒保护

2) 请求者X向密钥服务员获取密钥:

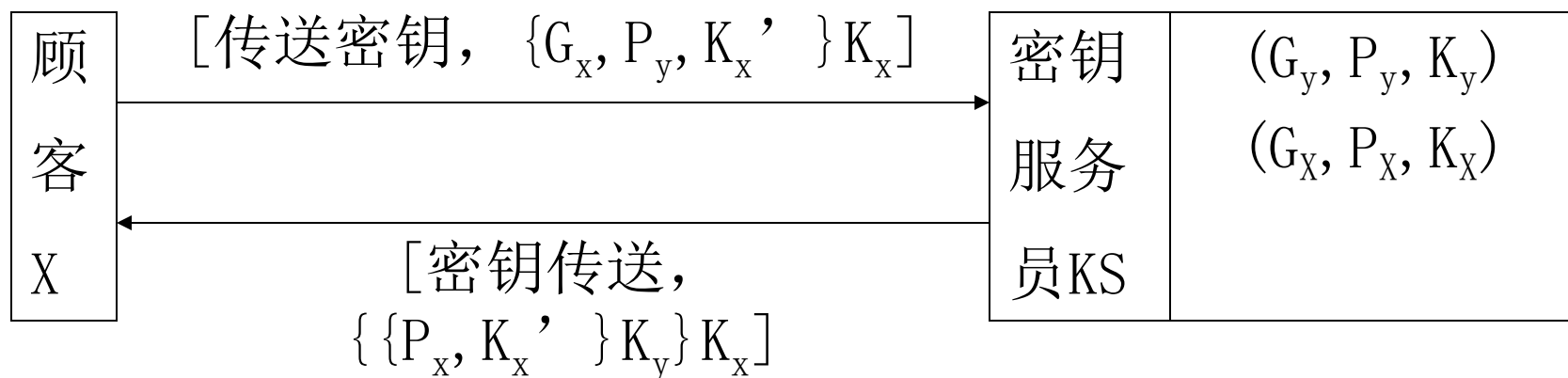


第四章 命名与保护

4.4 保护的例子: Amoeba

❖ 用软件F盒保护

3) 请求者X请求密钥服务员传送X和Y对话的密钥:

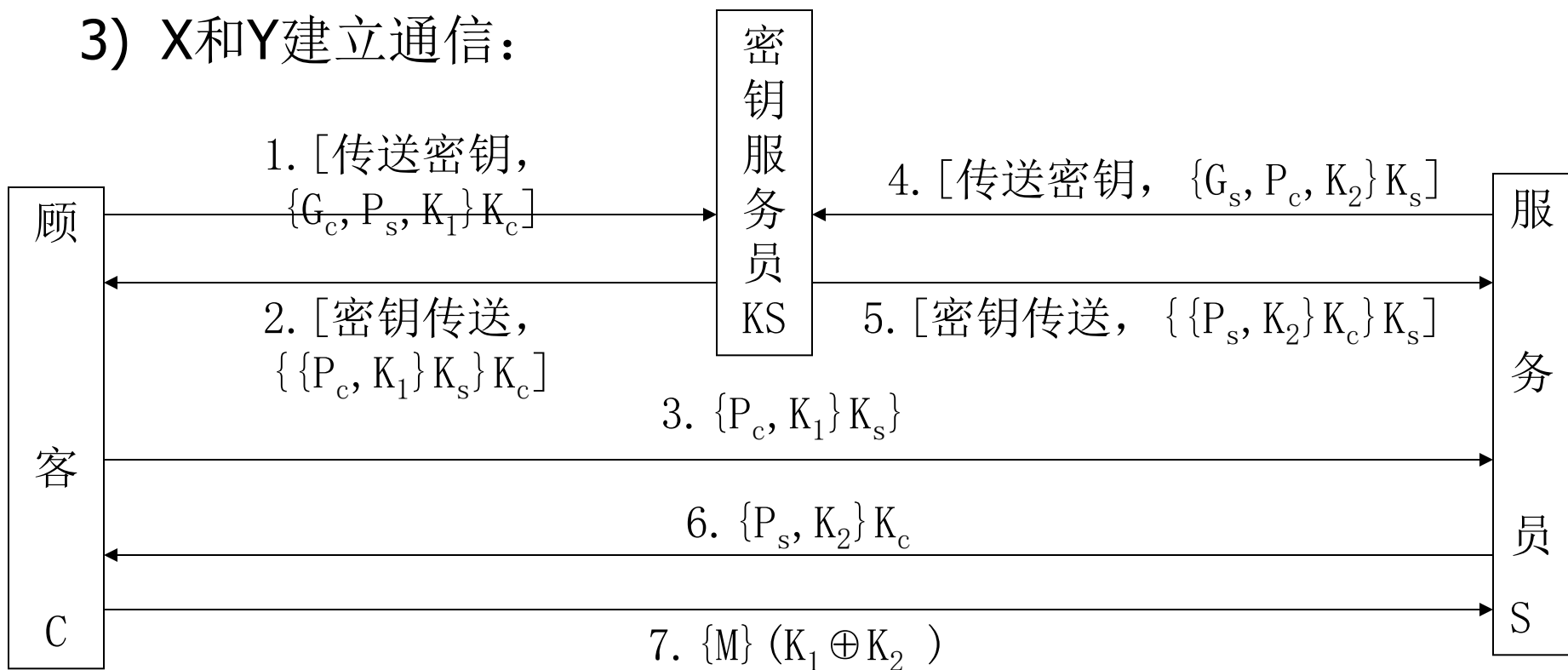


第四章 命名与保护

4.4 保护的例子: Amoeba

❖ 用软件F盒保护

3) X和Y建立通信:

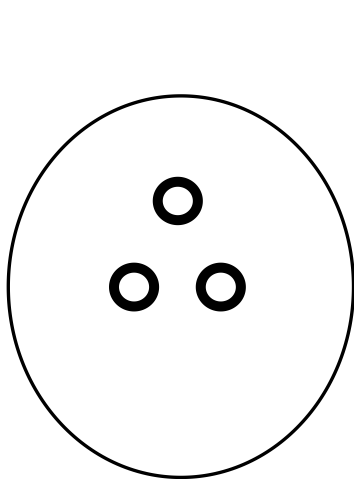


第五章 同步和互斥

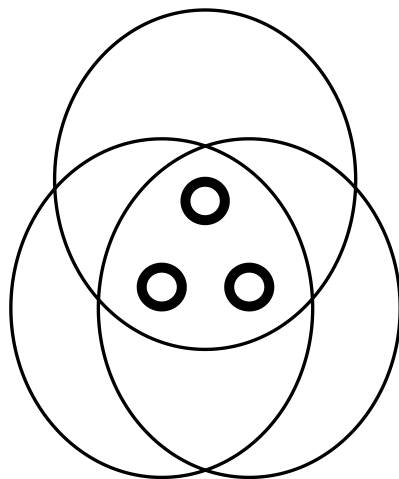
5.1 分布式系统中的资源管理

❖ 资源管理方式

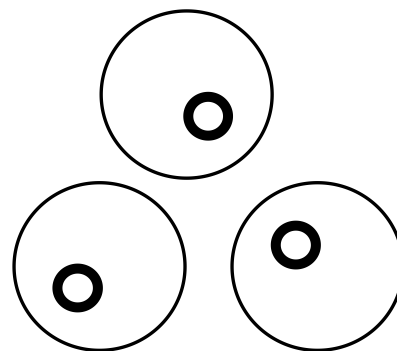
- 1) 全集中管理方式：所有资源都由一个服务员管理；
- 2) 集中分布管理方式：一个资源由一个服务员管理；
- 3) 全分布管理方式：一个资源是由多个服务员共同管理。



(a) 全集中式



(b) 全分布式



(c) 集中分布式

第五章 同步和互斥

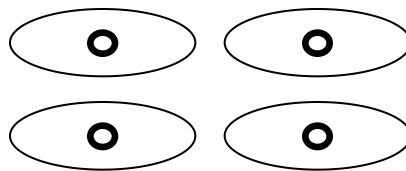
5.1 分布式系统中的资源管理

❖ 控制空间

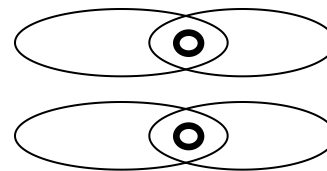
➤ 说明资源管理分散程度的参数：

- 1) 参加一个资源的多重管理的服务员数；
- 2) 被多个服务员管理的资源数。

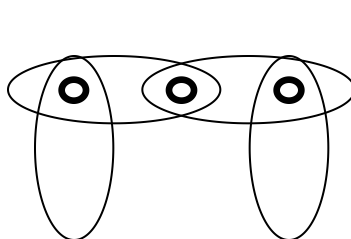
按参加多重管理的
服务员数排序：



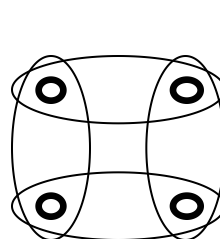
(a) 集中的控制



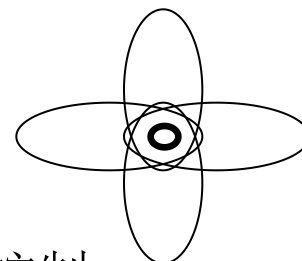
(b) 低度的分散控制



(c) 中度的分散控制



(d) 高度的分散控制

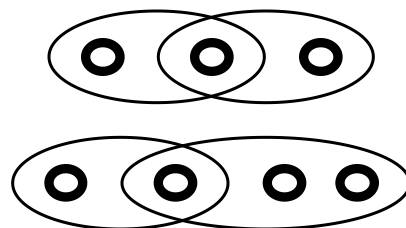
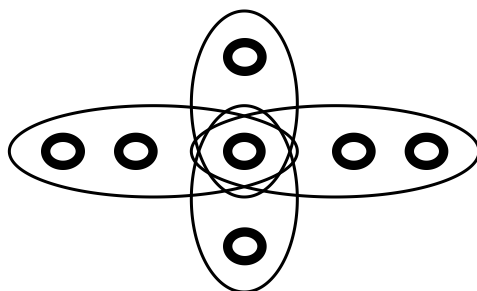
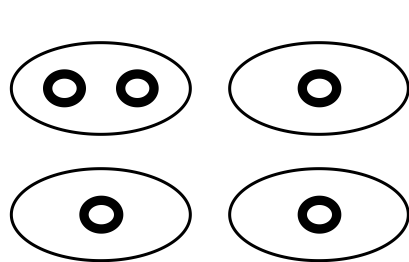


(e) 全分散控制

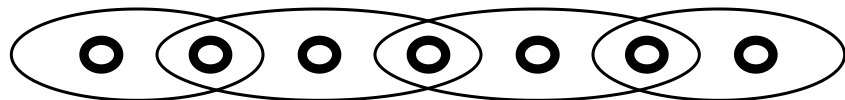
第五章 同步和互斥

5.1 分布式系统中的资源管理

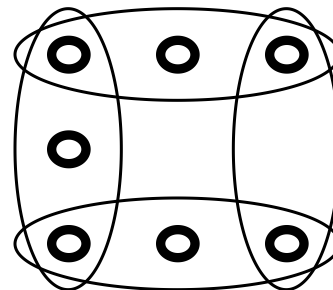
按由多个服务员管理的资源数目排序



(a) 没有资源被多重管理 (b) 一个资源被多重管理 (c) 两个资源被多重管理



(d) 三个资源被多重管理



(e) 四个资源被多重管理

第五章 同步和互斥

5.1 分布式系统中的资源管理

➤ 控制空间：



第五章 同步和互斥

5.1 分布式系统中的资源管理

➤多个服务员参加对同一资源进行控制的方式：

- 1) 顺序方式：按某种顺序，先由一个服务员控制一段时间，之后再由另一个服务员控制一段时间。
- 2) 分工方式：由不同的服务员并发或顺序地控制同一资源执行不同的活动。
- 3) 民主方式：所有服务员共同协商一致对同一资源执行每个管理活动。

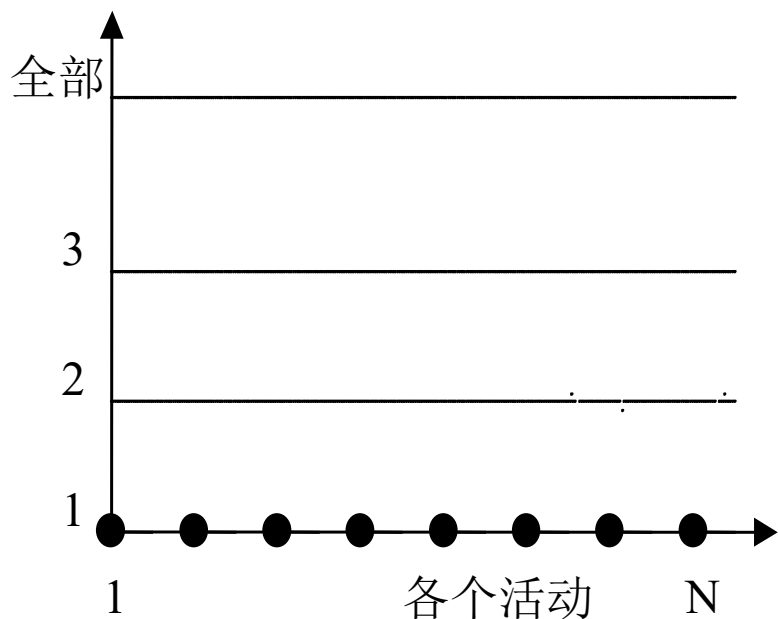
➤说明各种多重管理形式的分散性的参数：

- 1) 一致性是指由所有服务员共同完成的对同一个资源管理的活动数目。
- 2) 所谓均等性是各服务员对同一资源进行某一控制活动时被分配的管理权限和责任的平等程度。
- 3) 参加每个活动的服务员数目。

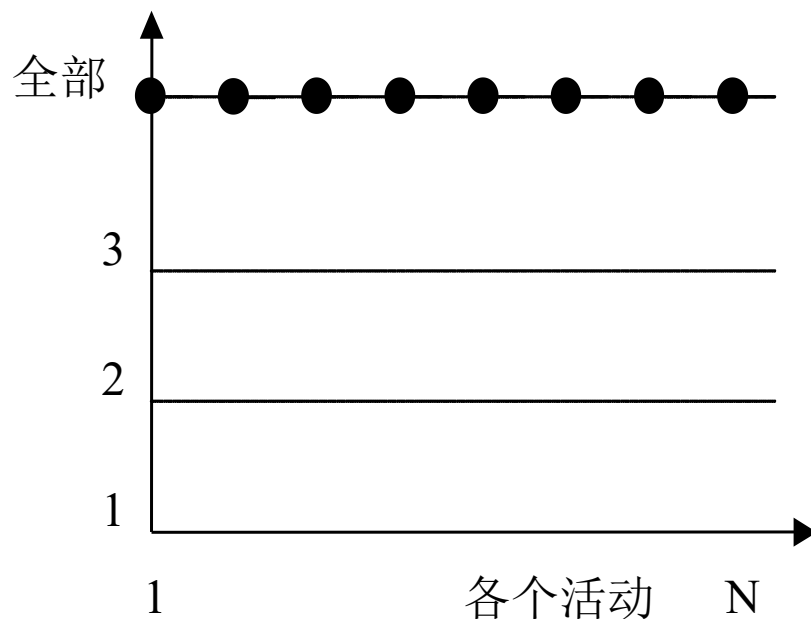
第五章 同步和互斥

5.1 分布式系统中的资源管理

一致性:



(a) 一致性的最大集中

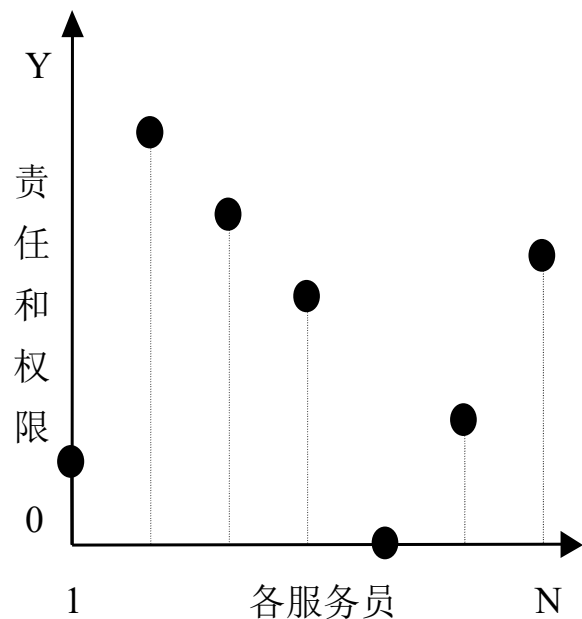


(b) 一致性的最大分散

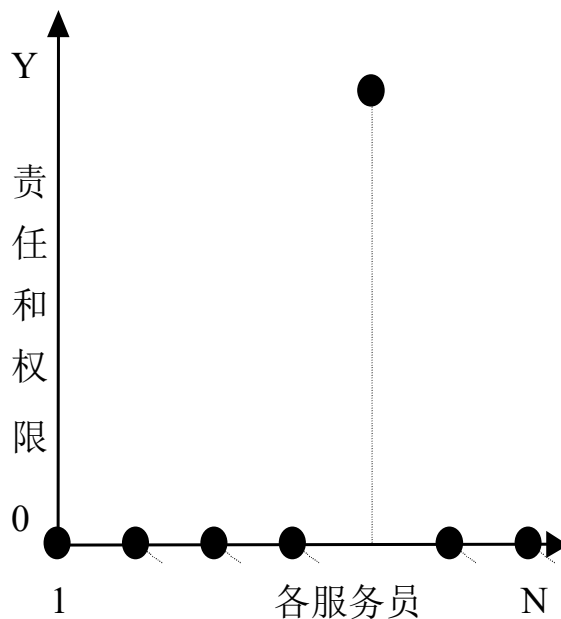
第五章 同步和互斥

5.1 分布式系统中的资源管理

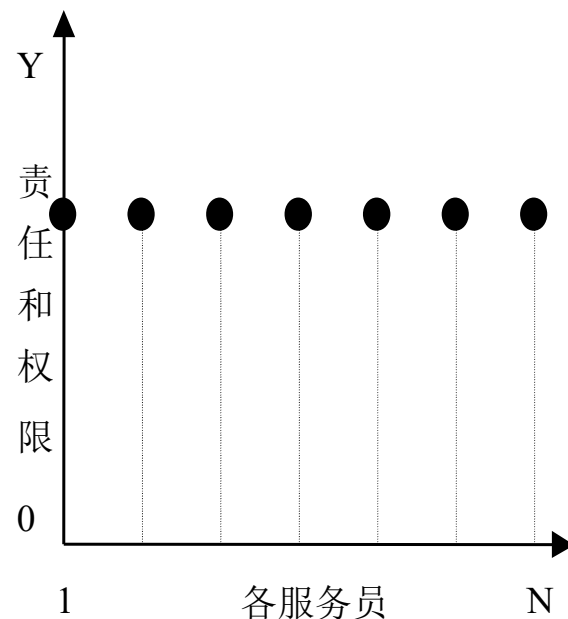
均等性:



(a) 各服务员的不同管理权限和责任



(b) 均等性中的最大集中

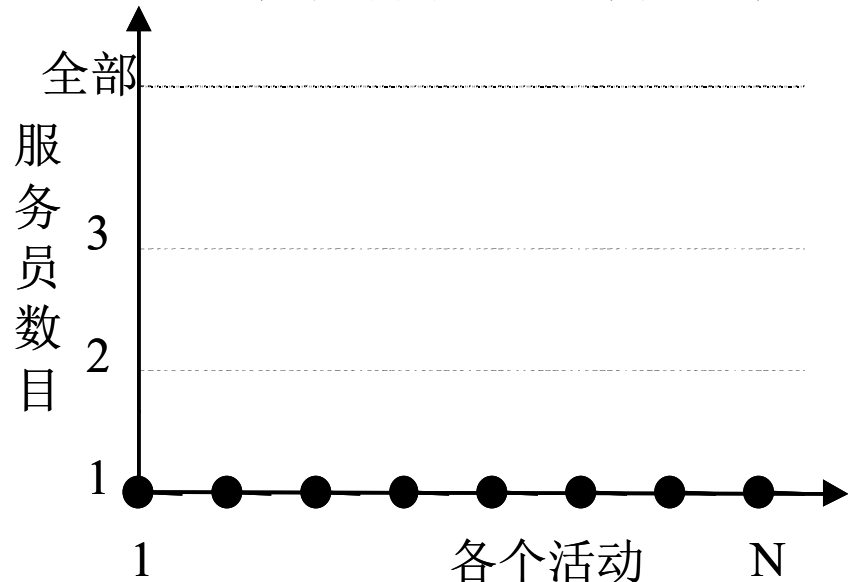


(c) 均等性中的最大分散

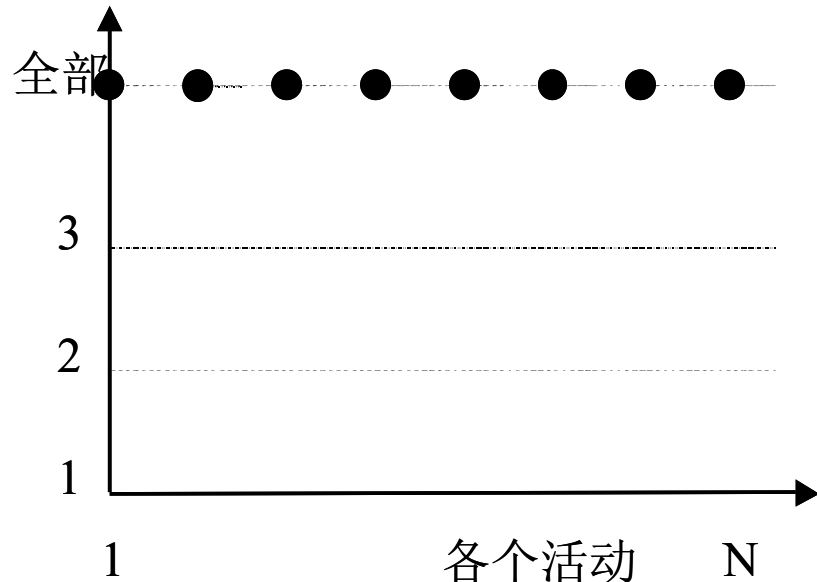
第五章 同步和互斥

5.1 分布式系统中的资源管理

参加每个活动的服务员数目：



(a) 最集中情况

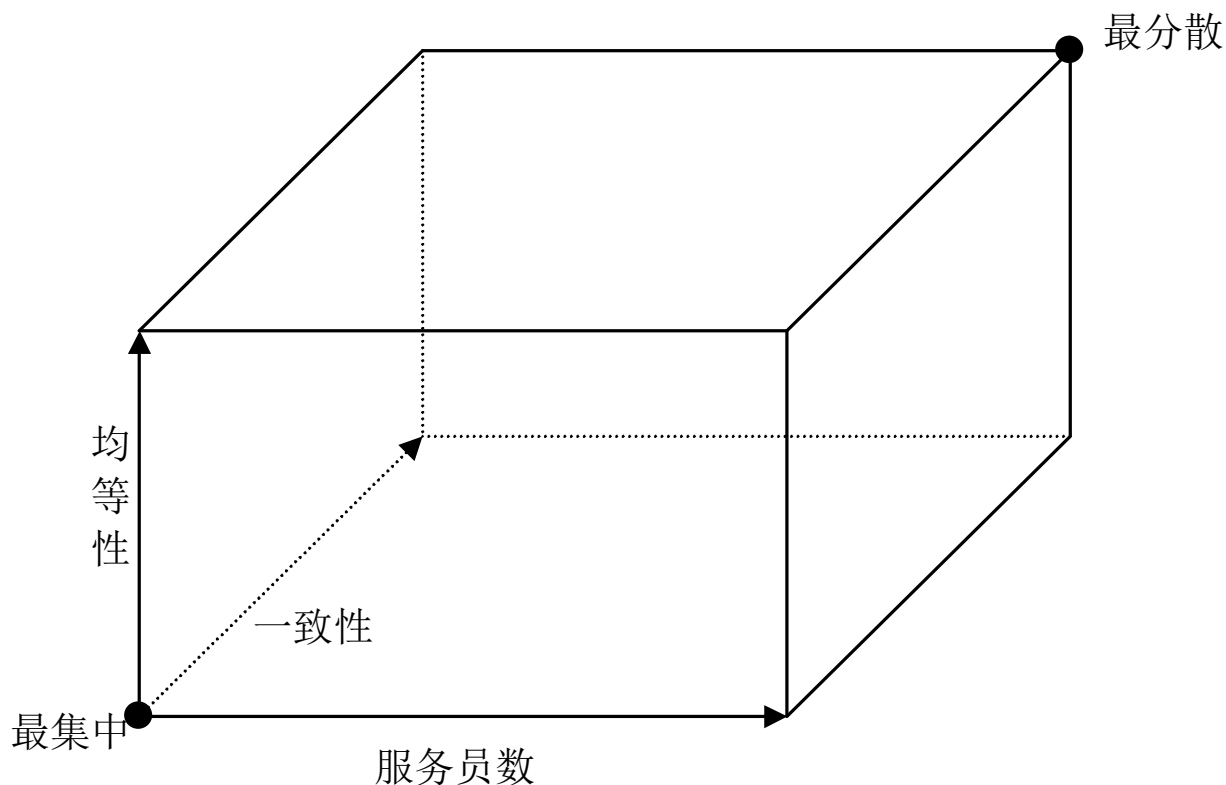


(b) 最分散情况

第五章 同步和互斥

5.1 分布式系统中的资源管理

单个资源的控制空间：



第五章 同步和互斥

5.2 同步机构

❖ 分布式系统中同步机构的作用

- 同步点：为了达到合作，各个进程在执行的过程中必须存在若干点，超过这些点时，一个进程在另一个进程执行完某一活动前不能继续执行，这些点称为这两个进程之间的同步点。
- 分布计算系统中共享资源的两类：一类是各进程可以同时访问的，如中央处理机(允许多个进程交叠使用一个处理机)、只读文件和不允许修改的存放子程序和数据的主存区域等；另一类是不允许多个进程同时访问的，每次只允许一个进程使用，如大多数外部设备(如打印机)、可写的文件以及主存中可修改的区域等。同步机构在互斥控制中的作用是对活动的执行进行排序。

第五章 同步和互斥

5.2 同步机构

❖ 分布式系统中同步机构的作用

➤ 一致状态：一个计算系统应该在所有时间内满足一定的外部规定或约束，如果一个计算系统确实在所有时间内满足了一定的外部规定或约束，这时我们称系统状态是一致的。同步机构的目的是给进程提供某种手段，使系统保持一致状态。

➤ 分布计算系统的计算方式分成三种：

- 1) 完全复制的计算。任何操作所激发的每个活动必须由所有的消费者共同处理，要求所有的活动均应完成。这时同步机构的目的是保证消费者处理活动的次序必须相同。

第五章 同步和互斥

5.2 同步机构

- 分布计算系统的计算方式分成三种：
- 2) 完全分割的计算。一个操作所激发的不同活动由不同的消费者分别独自处理。在这种情况下，同步机构的目的是保证所有相互干扰的活动成为有序的，使得该操作保持原子性(要么完成操作，要么干脆不发生)。
- 3) 分割和部分复制的计算。一个操作所激发的活动中，某些是由不同的消费者独自处理的，某些操作是由一些消费者共同处理。它兼有前面两种计算形式的特点。对于不同的计算方式，同步机构的目的是要求也是不同的。

第五章 同步和互斥

5.2 同步机构

➤ 评价同步机构的标准：

- 1) 响应时间和吞吐量。各种机构应尽量利用系统的并行性质，以提高吞吐量和缩短响应时间。
- 2) 恢复能力。同步机构应能使系统从故障中恢复过来。
- 3) 开销。指使用同步机构的代价，包括额外增加的报文长度、数量和对它们的处理时间，以及用于存放同步信息所需的额外存储空间。
- 4) 公平性。操作发生冲突时，同步机构应能避免生产者饿死，各生产者具有平等的权利。

第五章 同步和互斥

5.2 同步机构

➤ 评价同步机构的标准：

- 5) 可扩充性。系统扩充新的处理机时同步机构应不影响其正常运行。
- 6) 连接方式。使用某些同步机构要求生产者在逻辑上全部互连，这样所产生的开销可能很大；有些同步机构只要求一个生产者知道其邻居的情况，开销也较少。
- 7) 初始化。使用同步机构要求系统应容易进行初始化，知道进程何时可以进行生产和消费活动。
- 8) 排序方法。当生产者对一序列操作进行某种指定排序时，必须交换报文，各种同步机构实现效率可能大不相同。

第五章 同步和互斥

5.2 同步机构

➤ 分布式系统中同步机构：

同步实体：物理时钟、事件计数器、逻辑时钟、循环令牌和顺序器、进程等。

集中式同步机构和分布式同步机构：

- 1) 在集中式同步机构中，每个生产者每次发动一个操作时均要访问该同步实体。集中式同步实体有一个为所有必须相互同步的进程都知道的名字，任何时候这些进程的任何一个均可访问这一同步实体。执行每个功能如进程调度、数据访问控制等均要经过集中的同步实体进行控制。
- 2) 分布式同步机构不存在一个集中的同步实体，执行各种功能时是分散控制的。

第五章 同步和互斥

5.2 同步机构

➤ 分布式系统中同步机构：

集中式同步机构和分布式同步机构的优缺点：

- 1) 集中式同步机构最大的缺点是不可靠，一旦出故障就可能造成全局不工作；另外在性能方面也大大下降，因为集中会产生一个瓶颈。但实现简单。
- 2) 分布式同步机构在可靠性和性能方面优于集中式同步机构，也有很多种，主要有多重物理时钟、多重逻辑时钟、循环令牌等。但实现复杂。

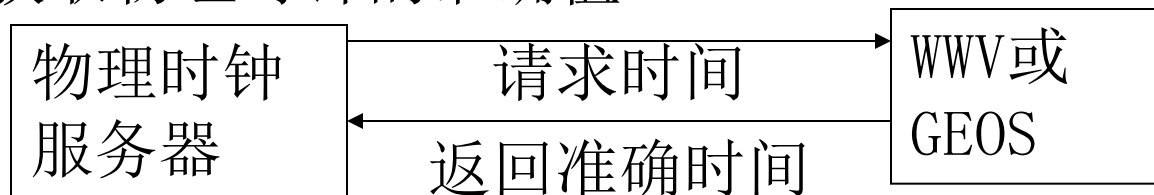
第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 准确时间值的获取和物理时钟的同步

获取物理时钟的准确值：



然后，根据通信延迟进行准确地校正。

第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 准确时间值的获取和物理时钟的同步

物理时钟的同步过程：

- 1) A通过网络向B发送请求；
- 2) B读取本地时钟值；
- 3) B的时钟值通过网络传递给A；
- 4) 按照网络所需的传输延迟对B的时钟值进行校正；
- 5) 比较A的时钟值和B的时钟值。

物理时钟的同步方式：集中式、分布式。

第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 集中式物理时钟

集中式物理时钟的实现方式：基于广播的方式、请求驱动的方式。

在基于广播的集中式物理时钟的实现方案中，集中的时钟服务员定期地向系统中的各个成员广播当前的时间。

基于广播的方式1：

顾客只是简单地将本地时间同所接收到的时间进行对比，当然这种对比考虑到了通常的网络传输延迟，然后校正自己的本地时钟。

第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 集中式物理时钟

基于广播的方式1:

时间校正方法:

- 1) 如果顾客的时钟值大于时钟服务员的时钟值，顾客将自己的时钟调慢，使之逐渐接近准确的时间。时钟值不能往回调，因为映像到此时钟的事件已经产生。
- 2) 如果顾客的时钟值落后于时钟服务员的时钟值，则顾客将时钟值向前拨，同时将时钟适当地调快。

第五章 同步和互斥

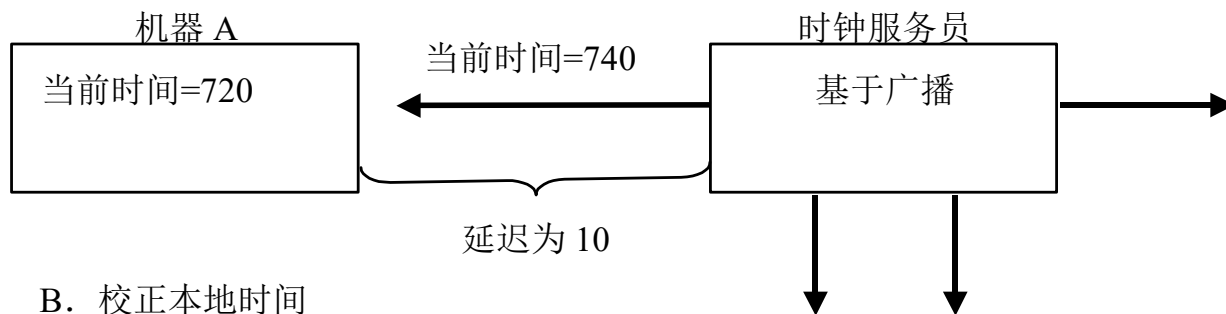
5.2 同步机构

❖ 物理时钟

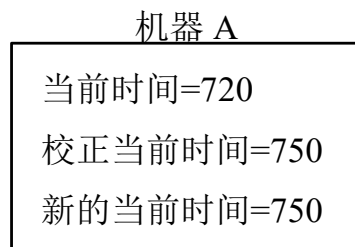
➤ 集中式物理时钟

时间校正方法：

A. 时钟服务员广播当前时间



B. 校正本地时间



第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 集中式物理时钟

基于广播的方式2（**Berkeley**算法）：

- 1) 顾客收到广播时间之后向集中的时间服务员发送它本地的当前时间值；
- 2) 每个顾客到时间服务员有不同的平均延迟，这些延迟时间预先存放在时间服务员处。时间服务员根据这些延迟对不同顾客传送来的当地时间进行校正；
- 3) 任何校正过的时间如果同时间服务员上的时间差值超过了对应节点到时间服务员的延迟时间常量，那么这个时间将不被列入考虑之列。因为这个时间可能是由于系统故障导致的，被认为是不准确的。

第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 集中式物理时钟

基于广播的方式2（**Berkeley**算法）：

- 4) 剩余被校正的时间值连同时间服务员上的时间值一起进行平均，这个平均值作为当前时间。
- 5) 时间服务员为每个顾客计算误差，然后将每个误差发送给对应的顾客。
- 6) 每个顾客校正自己的时钟。同前一个处理方式一样，时钟是不能往回拨的，但是可以按误差值将自己的始终慢下来。

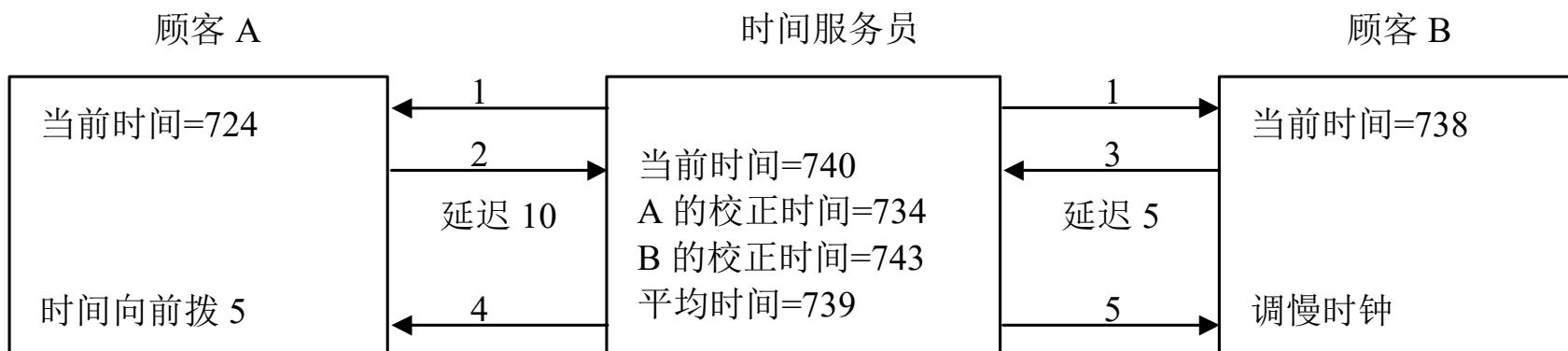
第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 集中式物理时钟

基于广播的方式2（Berkeley算法）：



1. 广播服务员时间 2. A 的当前时间 724 3. B 的当前时间 738 4. 向前拨 5 5. 放慢时钟 4

第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 集中式物理时钟

请求驱动方式：

- (1) 顾客向时间服务员发送请求，要求获得当前时间；
- (2) 时间服务员返回当前时间值；
- (3) 顾客计算本地的时间值和服务员返回的时间值之间的差值，这个差值用于时钟值的校正；它的实现不仅考虑了网络延迟，还包含了报文的响应和服务时间；
- (4) 如果校正值大于预先规定好的门限值，则被认为是不准确的，这可能是由于网络故障引起的，不准确的值被丢弃；
- (5) 如果服务员返回的时间值被认为是准确的，则对本地时钟进行校正，同样地，本地时钟不能往回拨，只能使本地时钟慢下来。

第五章 同步和互斥

5.2 同步机构

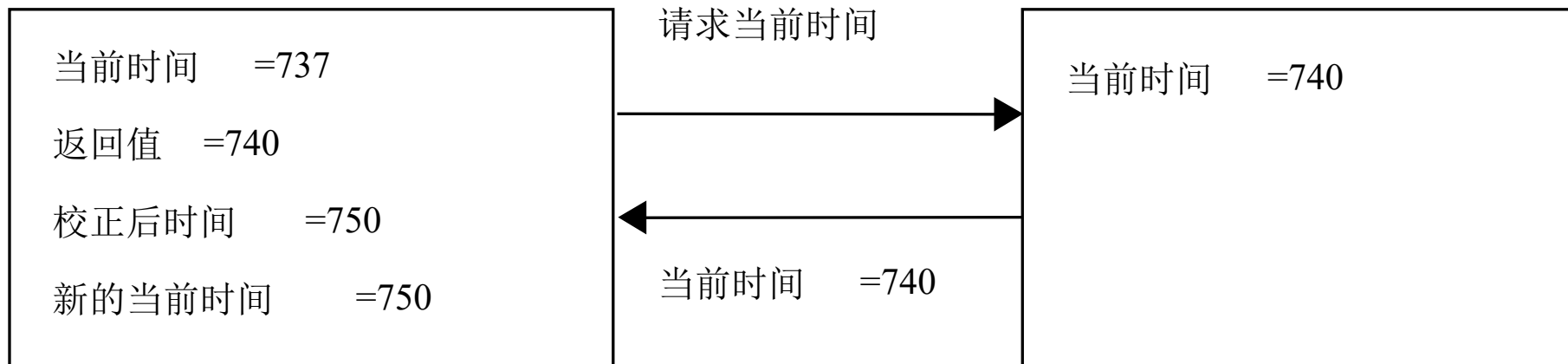
❖ 物理时钟

➤ 集中式物理时钟

请求驱动方式:

顾客

时间服务员



第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 分布式物理时钟

每个节点计算机以预先定义好的时间间隔定期地广播它的当前时间。由于时钟存在漂移，假定广播报文并不是很准确地在同一时刻发出。一旦一个节点广播了它的当前时间，就立即启动一个定时器，在定时期内接收其它节点的报文，每个报文标明了当地的当前时间，然后分别按对应的网络延迟对其它节点的时间值进行校正。

第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 分布式物理时钟

时间值校正方法：

- (1) 计算所有节点的平均值，把这个值作为当前时间。这种方法可能会产生不准确的结果，因为某些报文由于重发超出了通常的网络延迟。
- (2) 设定一个容错门限延迟，这个门限为单次发送的最大网络延迟，任何超过这个门限延迟的值被认为是错误的并被丢弃。其他未被丢弃的值进行平均，平均值作为当前时间。
- (3) 丢弃 m 个最大的时间值和 m 个最小的时间值，这些值被认为是不准确的。剩下的进行平均，平均值作为当前时间。

第五章 同步和互斥

5.2 同步机构

❖ 物理时钟

➤ 分布式物理时钟

时间值校正方法:

本地当前时间=740

收到时钟值

701	x
737	
742	
706	x
746	
742	
744	
750	
739	

x 表示超过延迟
门限的时间值
平均值即新的当前
时间=743

(a) 丢弃超过容错门限的时钟值

本地当前时间=740

收到时钟值

701	x
737	
742	
706	x
746	x
742	
744	
750	x
739	

m=2, x 表示丢弃时
间值
平均值即新的当前
时间=741

(b) 丢弃若干最大和最小的时钟值

第五章 同步和互斥

5.2 同步机构

❖ 逻辑时钟

逻辑时钟可以给分布计算系统中的事件一个唯一的排序。逻辑时钟的本质是基于Lamport定义的“在先发生关系”。

➤ 在先发生关系

- 1) 如果a和b均是同一进程中的两个事件，并且a在b之前出现，则 $a \rightarrow b$ ；
- 2) 若a代表“一个进程发送一个报文”这个事件，b代表“另一个进程接收这个报文”这个事件，则 $a \rightarrow b$ ；
- 3) 如果 $a \rightarrow b$ ，且 $b \rightarrow c$ ，则 $a \rightarrow c$ 。
两个不同的事件a和b，如果 $a \rightarrow b$ ，或 $b \rightarrow a$ ，则事件a和b是因果关联的。如果 $a \rightarrow b$ 和 $b \rightarrow a$ 均不成立，则称事件a和b是并发的。

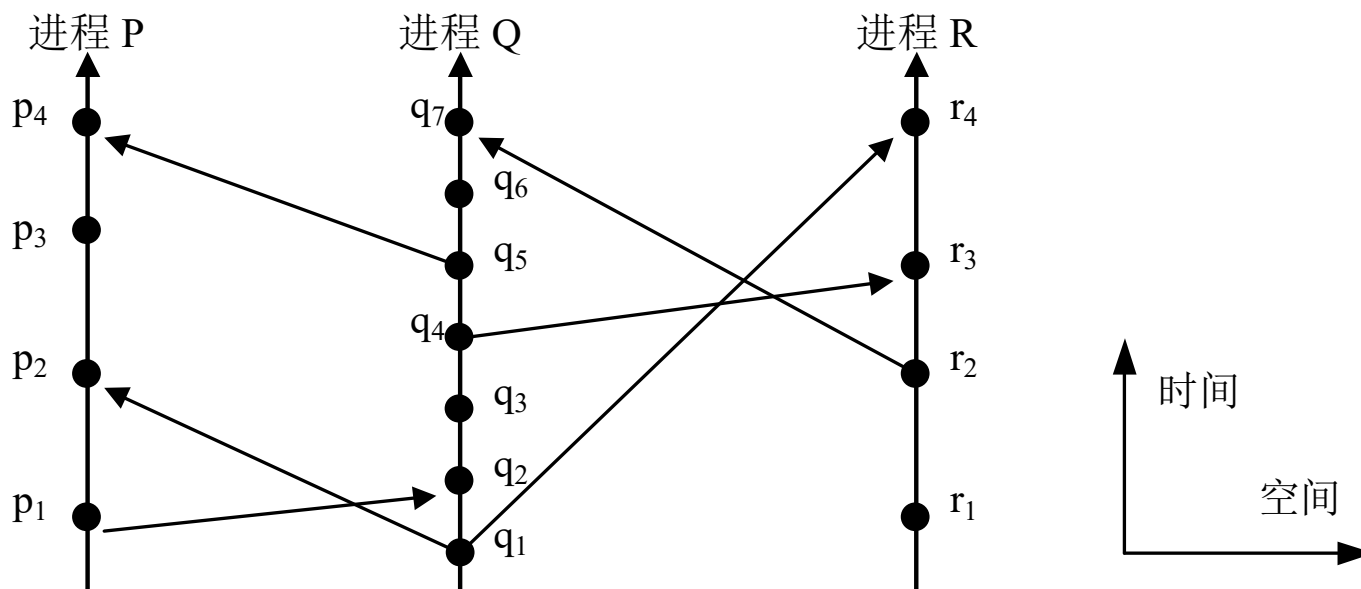
第五章 同步和互斥

5.2 同步机构

❖ 逻辑时钟

➤ 在先发生关系的时空图

水平方向代表空间，垂直方向代表时间，圆点代表事件，竖线代表进程，进程之间带箭头的线代表报文。



第五章 同步和互斥

5.2 同步机构

❖ 逻辑时钟

➤ 逻辑时钟

设 C_i 代表进程 i 的逻辑时钟，该逻辑时钟就是一个函数，它给进程 i 中的事件 a 分配一个正整数值 $C_i(a)$ 。

1) 时钟条件：

对任何事件 a 和 b ，如果 $a \rightarrow b$ ，则 $C(a) < C(b)$ 。但相反的结论不能成立。

a) 若 a 和 b 是同一进程 P_i 中的两个时间，并且 $a \rightarrow b$ ，则 $C_i(a) < C_i(b)$ ；

b) 若 a 代表“一个 P_i 进程发送一个报文”这个事件， b 代表“另一个进程 P_j 接收这个报文”这个事件， $C_i(a) < C_j(b)$ 。

第五章 同步和互斥

5.2 同步机构

❖ 逻辑时钟

➤ 逻辑时钟

2) 标量逻辑时钟

每个进程 P_i 有一个逻辑时钟 LC_i ， LC_i 被初始化为

$init (init \geq 0)$ 并且它是一个非减的整数序列。进程 P_i 发送的每个报文 m 都被标上 LC_i 的当前值和进程的标号 i ，从而形成一个三元组 (m, LC_i, i) 。任何一个逻辑时钟 LC_i 基于以下两条规则更新它的逻辑时钟值：

- a) 当发生一个事件(一个外部发送或内部事件)之前，我们更新 LC_i ： $LC_i := LC_i + d \quad (d > 0)$
- b) 当收到一个带时间戳的报文 (m, LC_j, j) 时，我们更新 LC_i ： $LC_i := \max(LC_i, LC_j) + d \quad (d > 0)$

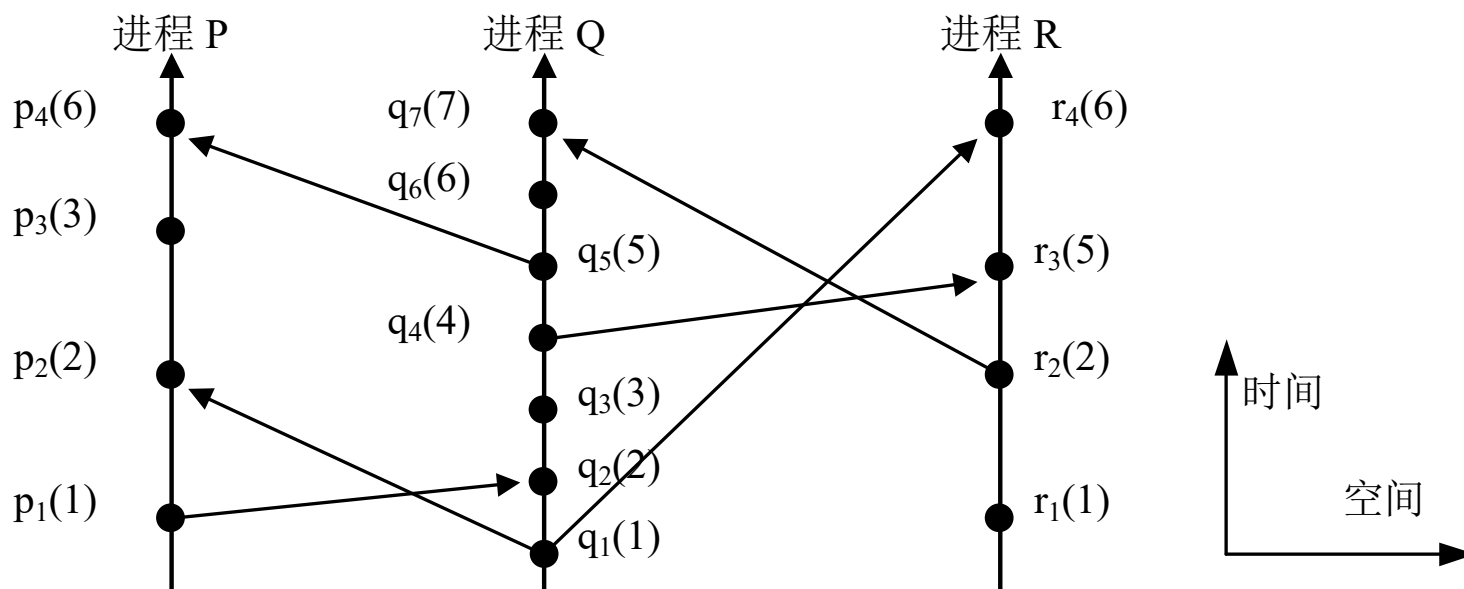
第五章 同步和互斥

5.2 同步机构

❖ 逻辑时钟

➤ 逻辑时钟

2) 标量逻辑时钟



第五章 同步和互斥

5.2 同步机构

❖ 逻辑时钟

➤ 逻辑时钟

2) 向量逻辑时钟

在向量逻辑时钟中，每个进程 P_i 和一个时间向量 $LC_i[1, \dots, n]$ 相关联，其中

- a) 向量元素 $LC_i[i]$ 描述了进程 P_i 的逻辑时间进展情况，即自身的逻辑时间进展情况；
- b) 向量元素 $LC_i[j]$ 表示进程 P_i 所知的关于进程 P_j 的逻辑时间进展情况；
- c) 向量 $LC_i[1, \dots, n]$ 组成进程 P_i 对于逻辑全局时间的局部视图。

第五章 同步和互斥

5.2 同步机构

❖ 逻辑时钟

➤ 逻辑时钟

2) 向量逻辑时钟

任何一个逻辑时钟 LC_i 基于以下两条规则更新它的逻辑时钟值:

- a) 当发生一个事件(一个外部发送或内部事件)之前, P_i 更新 $LC_i[i]$:

$$LC_i[i] := LC_i[i] + d \quad (d > 0)$$

- b) 每个报文捎带发送方在发送时的时钟向量, 当收到一个带时间戳的报文(m, LC_j, j)时, P_i 更新 LC_i :

$$LC_i[k] := \max(LC_i[k], LC_j[k]) \quad 1 \leq k \leq n$$

$$LC_i[i] := LC_i[i] + d \quad (d > 0)$$

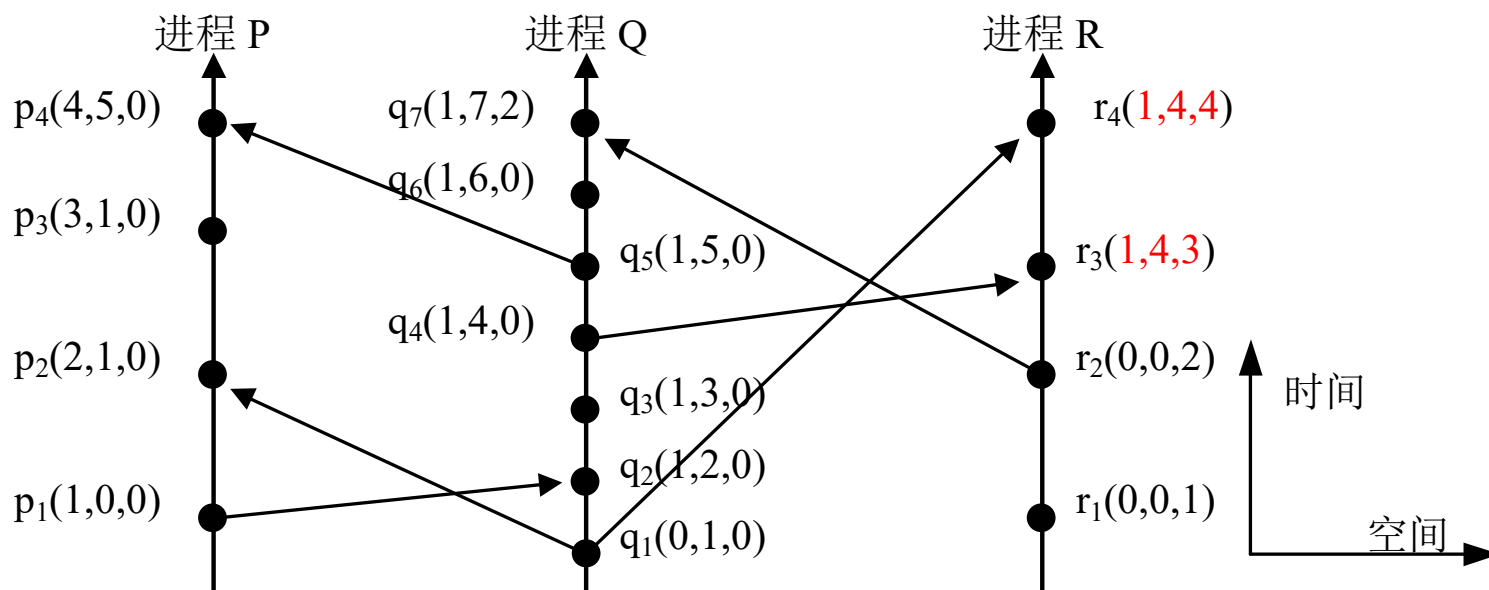
第五章 同步和互斥

5.2 同步机构

❖ 逻辑时钟

➤ 逻辑时钟

2) 向量逻辑时钟



第五章 同步和互斥

5.3 系统的全局状态

❖ 系统的全局状态的定义

令 LS_i 为进程 P_i 的局部状态，则全局状态 $GS=(LS_1, LS_2, \dots, LS_n)$ 。

1) 两个集合：

$$transit(LS_i, LS_j) = \{m \mid s(m) \in LS_i \wedge r(m) \notin LS_j\}$$

$$inconsistent(LS_i, LS_j) = \{m \mid s(m) \notin LS_i \wedge r(m) \in LS_j\}$$

集合 $transit$ 包括了进程 P_i 和进程 P_j 之间的通信通道上的所有报文，集合 $inconsistent$ 包括了所有接收事件记录在 P_j 而发送事件没有记录在 P_i 的报文。其中 m 是报文， $s(m)$ 是报文 m 的发送事件， $r(m)$ 是报文 m 的接收事件。

第五章 同步和互斥

5.3 系统的全局状态

❖ 系统的全局状态的定义

2) 一致的全局状态和非一致的全局状态:

a) 全局状态GS是一致的, 当且仅当

$$\forall i, \forall j, inconsistent(LS_i, LS_j) = \phi$$

b) 全局状态GS是非传送中的, 当且仅当

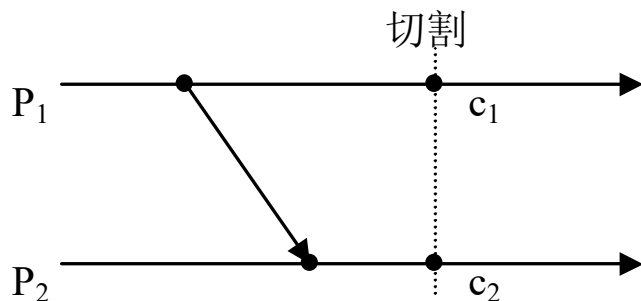
$$\forall i, \forall j, transit(LS_i, LS_j) = \phi$$

如果一个全局状态是一致的并且是非传送中的, 那么它就是强一致的, 也就是说, 局部状态集是一致的并且没有正在传送中的报文。

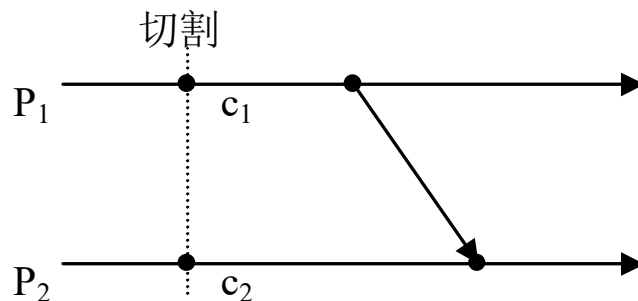
第五章 同步和互斥

5.3 系统的全局状态

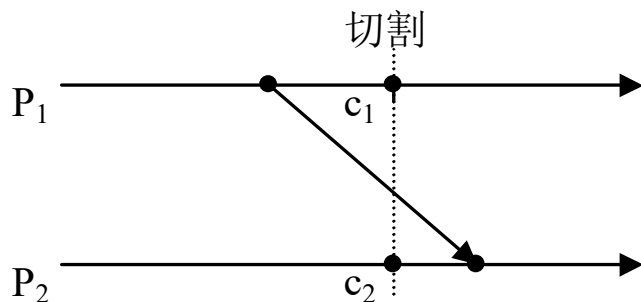
❖ 系统的全局状态的定义



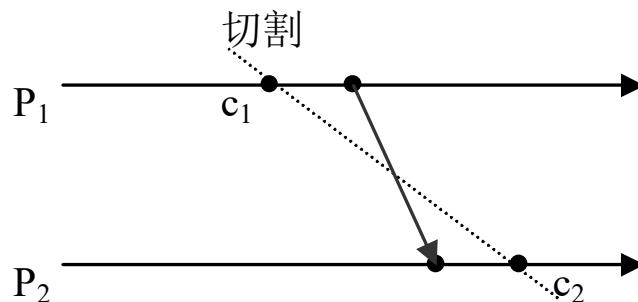
(a) 强一致全局状态



(b) 强一致全局状态



(c) 一致的全局状态



(d) 不一致的全局状态

第五章 同步和互斥

5.3 系统的全局状态

❖ 全局状态的获取（快照算法）：

- 1) 假如启动算法的进程为P，那么它首先记录自己的局部状态，然后它沿着它的输出通道发送一个标志(marker)，指示接收者应该参与记录一个全局状态的工作。
- 2) 当接收者Q通过它的输入通道C收到一个标志，它将依据不同条件执行以下不同操作：
 - a) 如果Q还没有记录自己的局部状态，它首先记录自己的局部状态，并记录通道C的状态为空报文序列，然后也沿着它自己的输出通道发送一个标志。
 - b) 如果Q已经记录了自己的局部状态，通过通道C收到的标志用来指示Q应该记录通道的状态。通道的状态是Q记录它的局部状态以来到收到这个标志前所收到的报文系列。

第五章 同步和互斥

5.3 系统的全局状态

❖ 全局状态的获取（快照算法）：

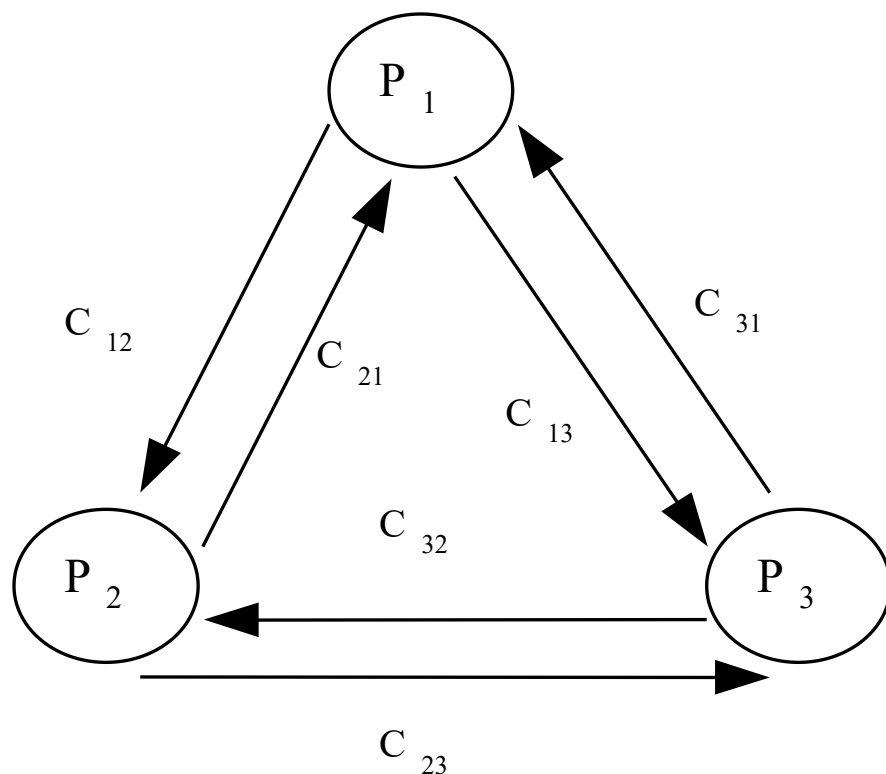
- 3) 如果一个进程已经沿它的每个输入通道接收到一个标志，并对每个标志进行了处理，就称它已经完成了它的那部分算法。
- 4) 一个进程的局部状态，连同它的所有输入通道的状态将被发送到这个快照的发起进程。

第五章 同步和互斥

5.3 系统的全局状态

❖ 全局状态的获取（快照算法）：

- 1) P_1 启动了快照算法，它同时执行三个动作：(a) 记录局部状态；(b) 发送一个标志到 C_{12} 和 C_{13} ；(c) 设置一个计数器对来自输入通道 C_{21} 和 C_{31} 的报文进行计数。

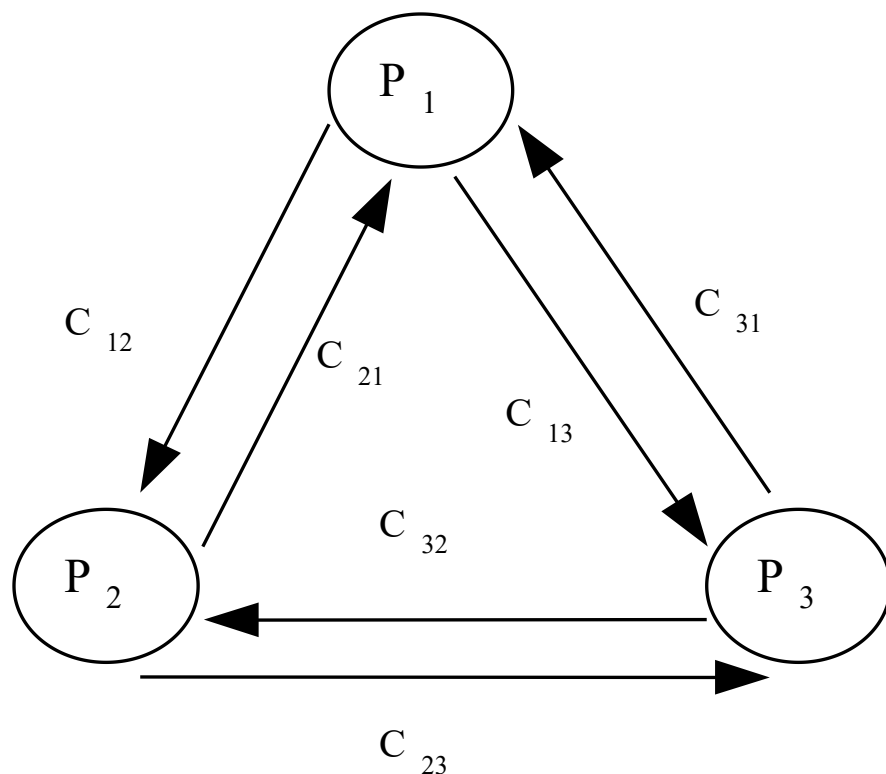


第五章 同步和互斥

5.3 系统的全局状态

❖ 全局状态的获取（快照算法）：

- 2) 一旦进程 P_2 从通道 C_{12} 接收到标志，它也执行三个动作：(a) 记录其局部状态并记录通道 C_{12} 的状态为空；(b) 发送一个标志到通道 C_{21} 和 C_{23} ；(c) 设置一个计数器对来自输入通道 C_{32} 的报文进行计数。



第五章 同步和互斥

5.3 系统的全局状态

❖ 全局状态的获取（快照算法）：

- 3) 类似地，进程 P_3 也执行三个动作。我们假定从进程 P_1 来的标志比从进程 P_3 来的标志早到达进程 P_2 。一旦从进程 P_3 来的标志到达进程 P_2 ， P_2 就记录通道 C_{32} 的状态为自设置计数器以来沿着这个通道接收到的报文的序列。于是进程 P_2 完成了自己的那部分算法，因为它已经从每个输入通道接收到一个标志并已经记录了自己的局部状态。类似地，进程 P_3 在接收到从 P_1 和 P_2 发来的标志后，属于它的那部分算法终止。进程 P_1 在接收到从 P_2 和 P_3 发来的标志后，属于它的那部分算法终止。

第五章 同步和互斥

5.3 系统的全局状态

❖ 一致全局状态的充要条件

Z字形路径：一条Z字形路径存在于进程 P_i 的检查点A到进程 P_j 的检查点B(P_i 和 P_j 可以是同一个进程)，当且仅当存在报文 m_1, m_2, \dots, m_n ($n \geq 1$) 满足：

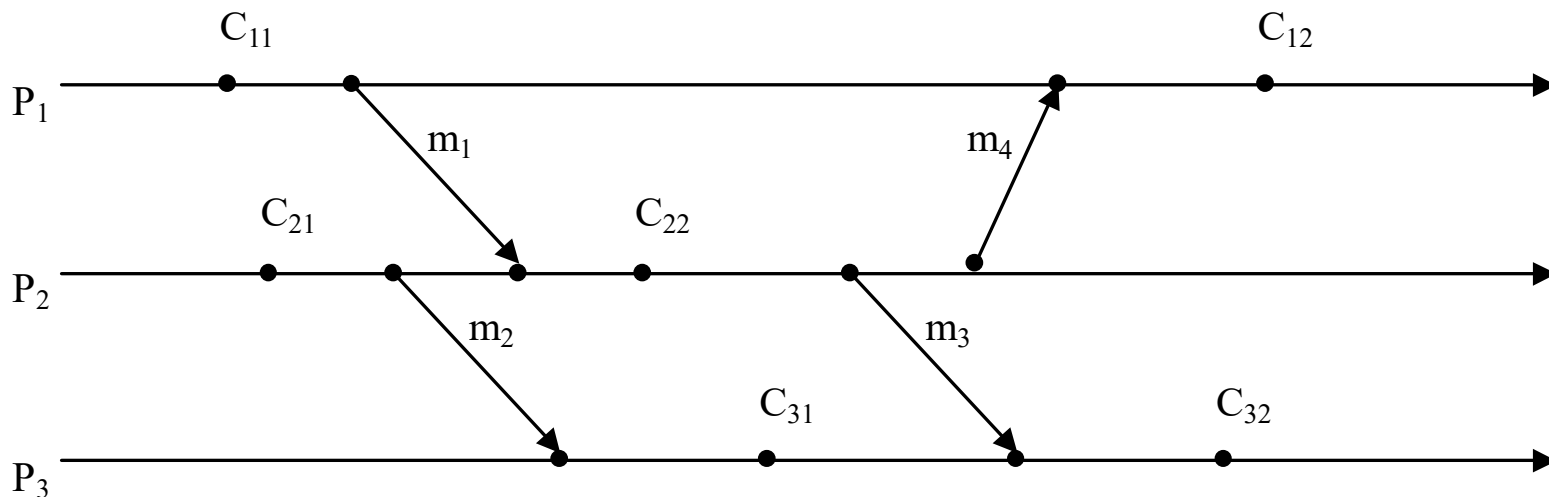
- a) m_1 是进程 P_i 在检查点A之后发送的；
- b) 如果 m_l ($1 \leq l \leq n$) 被进程 P_k 接收，则 m_{l+1} 在同一个或后面的检查点间隔被 P_k 发出。注意， m_{l+1} 可能在 m_l 被接收之前或之后发送；
- c) m_n 被进程 P_j 在检查点B之前接收。

在Z字形路径中，因果关系路径中那样的报文链是允许的，另外还允许这样的报文链，即链中的任何报文在前一个报文被接收之前发送，只要发送和接收在同一个检查点间隔里即可。

第五章 同步和互斥

5.3 系统的全局状态

❖ 一致全局状态的充要条件
Z字形路径实例：



从 C_{11} 到 C_{31} 的由报文 m_1 和 m_2 组成的路径是一条Z字形路径。从 C_{11} 到 C_{32} 的由报文 m_1 和 m_3 组成的路径是一条Z字形路径，同时也是一条因果关系路径。检查点 C_{22} 形成一条由报文 m_4 和 m_1 组成的Z字形循环。

第五章 同步和互斥

5.3 系统的全局状态

❖ 一致全局状态的充要条件

一致性定理：一个检查点集S，其中每个检查点属于不同的进程，它们属于同一个一致性的全局状态当且仅当S中不存在这样的检查点，它有一条到S中任何其他检查点(包括它自身)的Z字形路径。

推论：

- 1) 检查点C可以属于一个一致性全局状态当且仅当C不属于一个Z字形循环；
- 2) 属于不同进程的两个检查点A和B，它们可以属于同一个一致性全局状态当且仅当
 - a) 没有包括A或B的Z字形循环；
 - b) 在A和B之间不存在Z字形路径。

第五章 同步和互斥

5.4 互斥算法

❖ 互斥问题

互斥问题就是定义一些基本的操作来解决共享资源的多个并发进程的冲突问题。

互斥算法的主要目标是保证在任一个时刻只能有一个进程访问临界区。一个正确的互斥算法必须避免冲突(死锁和饿死)和保证公平性。为此，算法应满足以下三个条件：

- 1) 已获得资源的进程必须先释放资源之后，另一个进程才能得到资源；
- 2) 不同的请求应该按照这些请求的产生顺序获得满足，请求应该按照某种规则进行排序，例如使用逻辑时钟确定请求的顺序；
- 3) 若获得资源的每个进程最终都释放资源，则每个请求最终都能满足。

第五章 同步和互斥

5.4 互斥算法

❖ 互斥问题

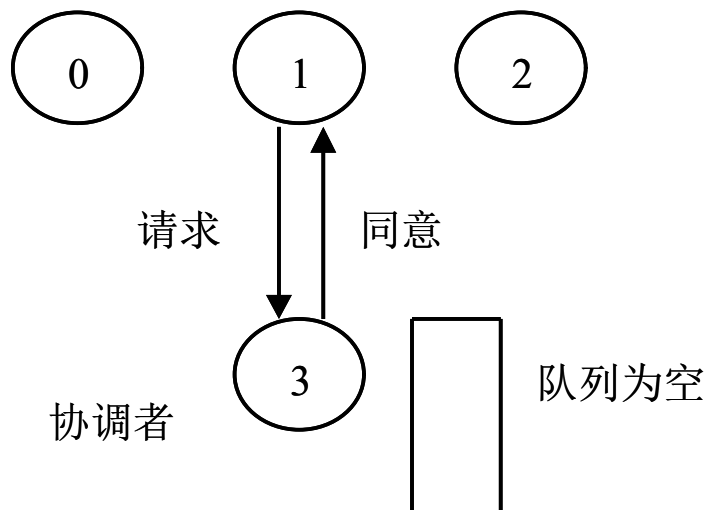
衡量互斥算法性能的参数：

- 1) 完成一次互斥操作所需的报文数目；
- 2) 同步延迟，即从一个进程离开临界区之后到下一个进程进入临界区之前的时间间隔；
- 3) 响应时间，即从一个进程发出请求到该进程离开该临界区之间的时间间隔。

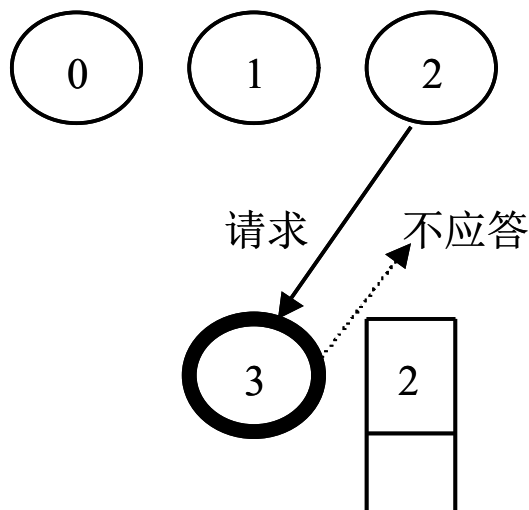
第五章 同步和互斥

5.4 互斥算法

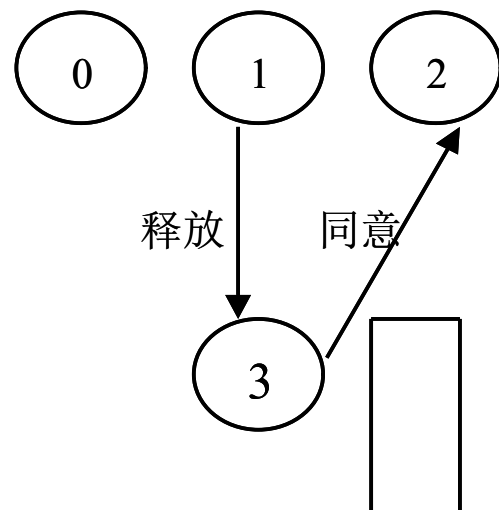
❖ 集中式互斥算法



(a) 临界区中无进程，
同意请求



(b) 临界区中有进程，
不同意请求



(c) 临界区被释放，同意缓冲
区中的第一个请求

第五章 同步和互斥

5.4 互斥算法

❖ Lamport时间戳互斥算法

Lamport时间戳互斥算法由以下5条规则组成：

- 1) 一个进程 P_i 如果为了申请资源，它向其它各个进程发送具有时间戳 $T_m:P_i$ 的申请资源的报文，并把此报文也放到自己的申请队列中；
- 2) 一个进程 P_j 如果收到具有时间戳 $T_m:P_i$ 的申请资源的报文，它把此报文放到自己的申请队列中，并将向 P_i 发送一个带有时间戳的承认报文。如果 P_j 正在临界区或正在发送自己的申请报文，则此承认报文要等到 P_j 从临界区中退出之后或 P_j 发送完自己的申请报文之后再发送，否则立即发送；
- 3) 一个进程 P_i 如果想释放资源，它先从自己的申请队列中删除对应的 $T_m:P_i$ 申请报文，并向所有其他进程发送具有时间戳的 P_i 释放资源的报文；

第五章 同步和互斥

5.4 互斥算法

❖ Lamport时间戳互斥算法

Lamport时间戳互斥算法由以下5条规则组成：

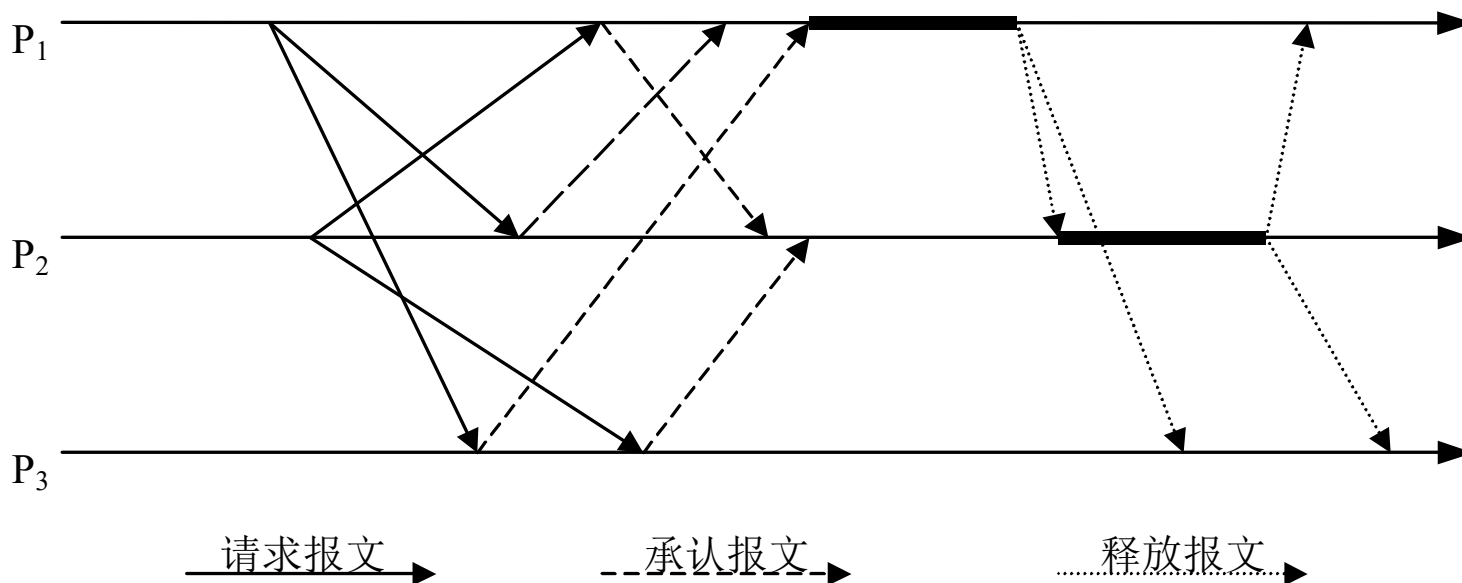
- 4) 一个进程 P_j 如果收到 P_i 释放资源的报文，它从自己的申请队列中删除 $T_m:P_i$ 申请报文；
- 5) 当满足下述两个条件时，申请资源的进程 P_i 获得资源：
 - a) P_i 的申请队列中有 $T_m:P_i$ 申请报文，并且根据时间戳它排在所有其它进程发来的申请报文前面；
 - b) P_i 收到所有其它进程的承认报文，其上面的时间戳值大于 T_m 。

第五章 同步和互斥

5.4 互斥算法

❖ Lamport时间戳互斥算法

Lamport时间戳互斥算法实例：



第五章 同步和互斥

5.4 互斥算法

❖ Ricart-Agrawala互斥算法

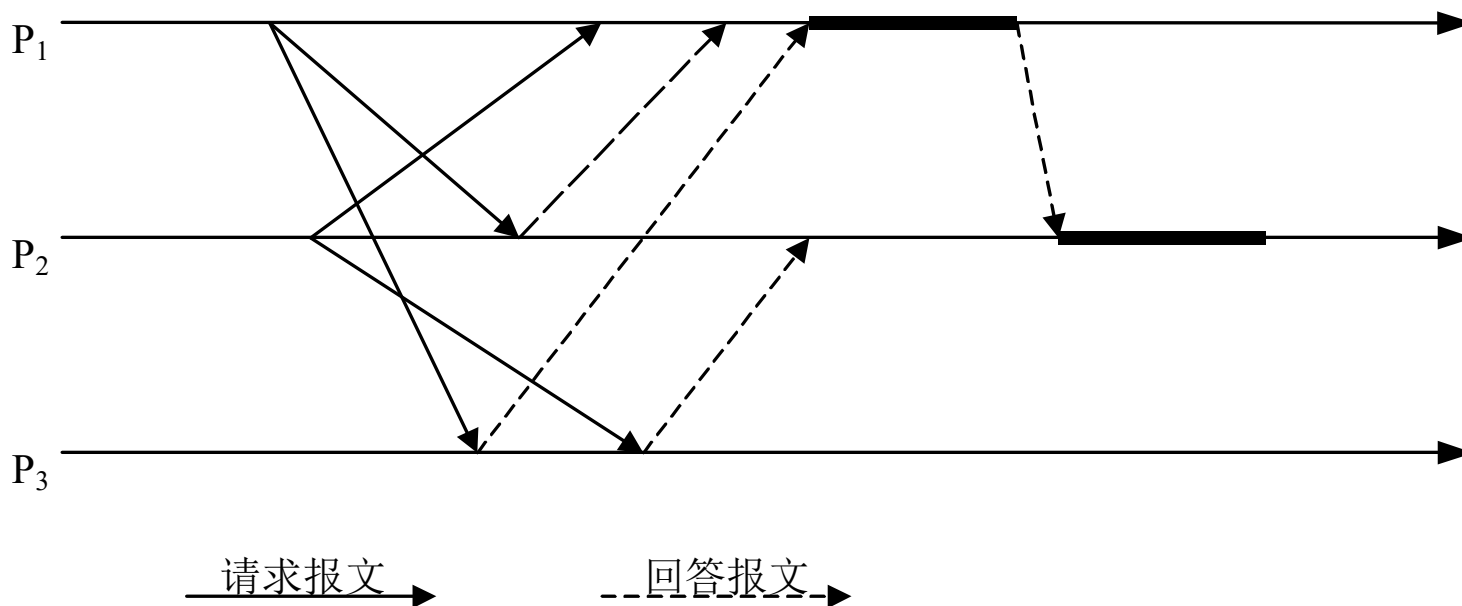
- 1) 一个进程申请资源时向所有其他进程发出申请报文;
- 2) 其它进程收到申请报文后若不在临界区并且自己未申请进入临界区, 或者自己虽然发出了申请报文, 但自己的报文排在收到的申请报文之后, 则回答表示同意;
- 3) 申请资源的进程仅在收到所有进程的回答报文后才进入临界区使用资源;
- 4) 一个进程使用完资源后, 它向所有未给回答的其它申请发送回答报文。

第五章 同步和互斥

5.4 互斥算法

❖ Ricart-Agrawala互斥算法

Ricart-Agrawala互斥算法实例：



第五章 同步和互斥

5.4 互斥算法

❖ Maekawa互斥算法

请求子集：在Maekawa互斥算法中，一个进程P在发出申请报文后，不用得到所有其他进程的回答，而只须得到一个进程子集S中的所有进程的回答即可进入临界区。称S是P的请求子集。假设 R_i 和 R_j 分别是进程 P_i 和 P_j 的请求子集，要求 $R_i \cap R_j \neq \text{NULL}$ 。

- 1) 当进程 P_i 请求进入临界区时，它只向 R_i 中的进程发送请求报文。
- 2) 当进程 P_j 收到一个请求报文时，如果它自上一次临界区释放后还没有发出过回答报文给任何进程，且自己的请求队列中无任何请求，它就给该请求报文一个回答。否则，请求报文被放入请求队列中。
- 3) 进程 P_i 只有收到 R_i 中的所有进程的回答后，才能进入临界区。
- 4) 在释放临界区时，进程 P_i 只给 R_i 中的所有进程发送释放报文。

第五章 同步和互斥

5.4 互斥算法

❖ Maekawa互斥算法

请求子集的例子：

考虑一个七个进程的例子，每个进程的请求子集如下：

$R_1: \{P_1, P_3, P_4\};$

$R_2: \{P_2, P_4, P_5\};$

$R_3: \{P_3, P_5, P_6\};$

$R_4: \{P_4, P_6, P_7\};$

$R_5: \{P_5, P_7, P_1\};$

$R_6: \{P_6, P_1, P_2\};$

$R_7: \{P_7, P_2, P_3\}。$

第五章 同步和互斥

5.4 互斥算法

❖ Maekawa互斥算法

Maekawa算法的两个极端情况：

(1) 退化为集中式互斥算法。 $P_c (c \in \{1, 2, \dots, n\})$ 作为协调者，对所有进程 P_i ，有：

$$R_i: \{P_c\}, 1 \leq i \leq n$$

(2) 完全分布式的互斥算法。对所有进程 P_i ，有：

$$R_i: \{P_1, P_2, \dots, P_n\}, 1 \leq i \leq n$$

第五章 同步和互斥

5.4 互斥算法

❖ 简单令牌环互斥算法

- 1) 在有 n 个进程的系统中，将这 n 个进程组成一个首尾相连的逻辑环。每个进程在环中有一个指定的位置，位置可以按网络地址进行排列，当然也可以采用任何其他可行的方式排列，但每个进程必须知道在环中哪个进程是它后面的进程。
- 2) 一个进程拥有令牌时就可以进入临界区，令牌可在所有的进程间传递。
- 3) 如果得到令牌的进程不打算进入临界区，它只是简单地将令牌传送给它后面的进程。

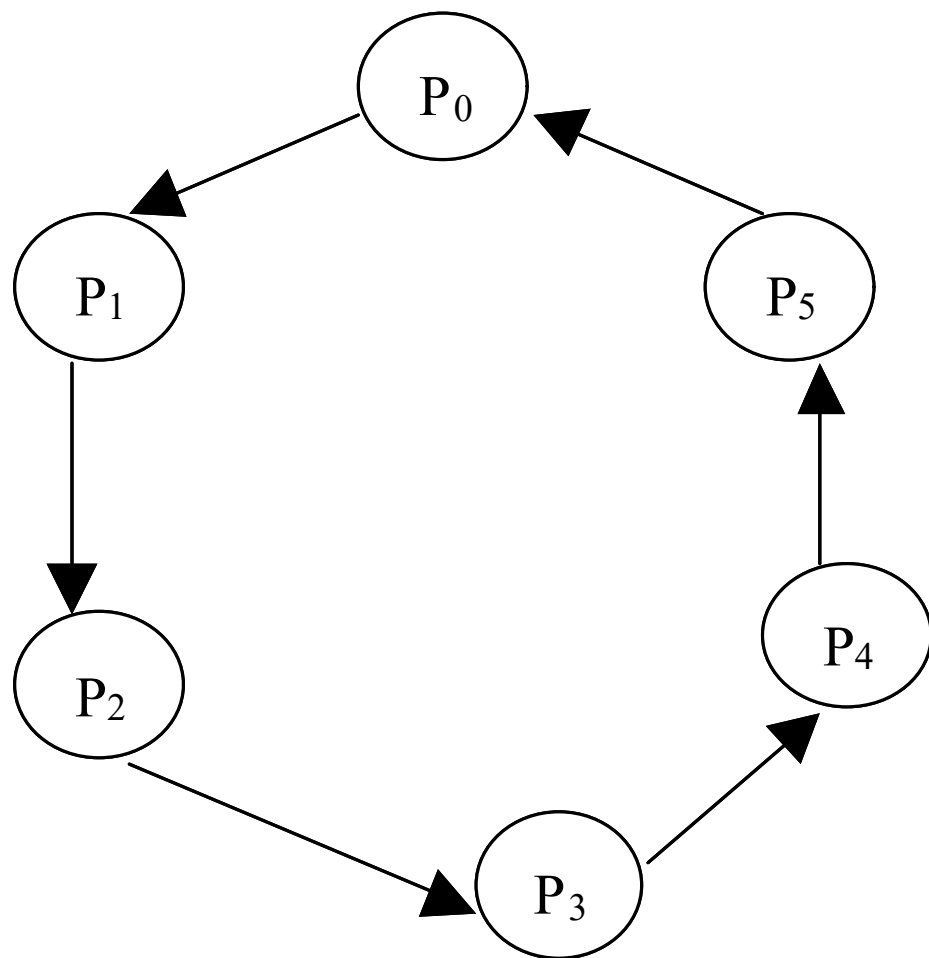
第五章 同步和互斥

5.4 互斥算法

❖ 简单令牌环互斥算法

问题：

- 如果令牌丢失，需要重新产生一个令牌，但检测令牌是否丢失是比较困难的。
- 另外一个问题是进程的崩溃，但进程的崩溃比较容易检测。
- 这个算法在高负载的情况下工作得很好。然而，它在轻负载的情况下工作得很差，出现很多不必要的报文传递。



第五章 同步和互斥

5.4 互斥算法

❖ Ricart-Agrawala令牌环互斥算法

- 1) 初始时，令牌被赋予给任何一个进程。
- 2) 请求进入临界区的进程 P_j 不知道哪个进程拥有令牌，所以它向所有其它进程发送一个带时间戳的请求报文，请求得到令牌。每个进程有一个请求队列记录有所有进程的请求，令牌中记录有每个进程最后一个持有令牌的时间。
- 3) 如果当前拥有令牌的进程 P_i 不再需要令牌，它就按照 $i+1, i+2, \dots, n, 1, 2, \dots, i-1$ 的顺序寻找第一个符合条件的进程 P_j ，并将令牌传递给进程 P_j 。

说明：虽然该算法不是按照每个请求的时间顺序来满足的，但是，由于令牌是按一个方向绕环传递的，所以不会有饿死现象发生。

第五章 同步和互斥

5.4 互斥算法

❖ 基于时间戳的令牌互斥算法

每个进程保持一张进程状态表，记录它所知的进程状态，进程状态包括该进程是否为请求进程，以及得到该状态的时间。令牌是一个特殊的报文，该报文中包含了发送该令牌的进程的进程状态表。

- 1) 初始化时，每个进程的状态表中各个进程均为非请求状态，时钟值为0，并任意指定一个进程为令牌的持有者。
- 2) 请求时，一个进程请求进入临界区时，如果它持有令牌，它不发送任何请求报文，将自己的进程状态表中相应于自己一栏的状态改为请求态，并记录该状态的时钟值，直接进入临界区。如果它不持有令牌，则它向所有其它进程发送带有时间戳的请求报文。发出请求报文后，将自己的进程状态表中相应于自己一栏的状态改为请求态，并记录该状态的时钟值。

第五章 同步和互斥

5.4 互斥算法

❖ 基于时间戳的令牌互斥算法

- 3) 收到请求时，当进程A收到进程B的请求报文时，A将B的请求报文中的时间戳同A的进程状态表中B的时间值进行比较。若B的请求报文中的时间戳大于A的进程状态表中B的时间值，则A修改自己的进程状态表。将A的进程状态表中对应于B的一栏改为请求状态，并记录此状态的时间。
- 4) 退出临界区时，进程A退出临界区后，将自己的进程状态表中关于自己的一栏改为非请求状态，时钟值加1，并将该时钟值作为该状态的时间。然后检查其进程状态表中是否记录有某个进程处于请求状态，若有，则从处于请求状态的进程中选取一个请求最早的进程B(具有最小的时间戳)，将令牌传送给它，并在令牌中附上A的进程状态表。

第五章 同步和互斥

5.4 互斥算法

❖ 基于时间戳的令牌互斥算法

- 5) 收到令牌时，收到令牌的进程把随令牌传来的进程状态表和自己的进程状态表进行比较。若随令牌传来的进程状态表中某进程的时间戳大于自己的进程状态表中相应进程的时间戳，则将自己的进程状态表中相应进程的状态和时间戳改成随令牌传来的进程状态表中相应的状态和时间戳。

说明：同Ricart-Agrawala令牌环互斥算法相比，具有更强的公平性，因为它是基于请求的先后顺序来满足的，而Ricart-Agrawala令牌环互斥算法是基于进程的逻辑环结构来满足的。

第五章 同步和互斥

5.4 互斥算法

❖ 选举算法

从进程集中选出一个进程执行特别的任务。大部分选举算法是基于全局优先级的，就是说给每个进程预先分配一个优先级，选举算法选择一个具有最高优先级的进程作为协调者。

➤ Bully选举算法

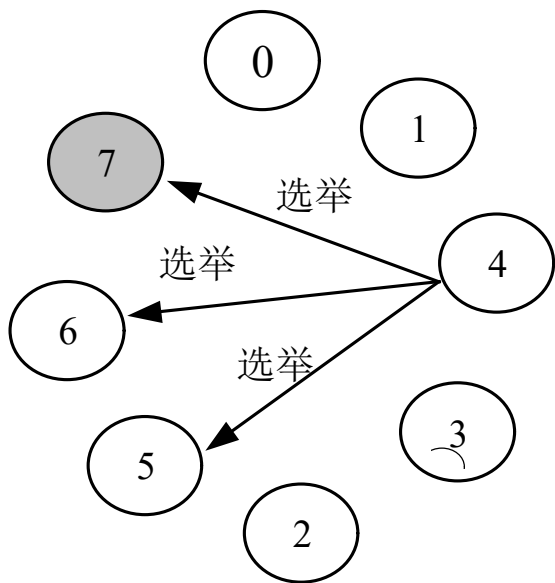
- 1) P发送选举报文到所有优先级比它高的进程。
- 2) 如果在一定时间内收不到任何响应报文，P赢得选举成为协调者。它向所有比它的优先级低的进程发送通知报文，宣布自己是协调者。
- 3) 如果收到一个优先级比它高的进程的回答，P的选举工作结束。同时启动一个计时器，等待接收谁是协调者的通知报文，如果在规定时间内得不到通知报文，则它重新启动选举算法。
- 4) 任何时候，一个进程可能从比它的优先级低的进程那儿接收到一个选举报文，它就给发送者回答一个响应报文，同时启动如上所述的相同的选举算法，如果选举算法已经启动，就不必重新启动。

第五章 同步和互斥

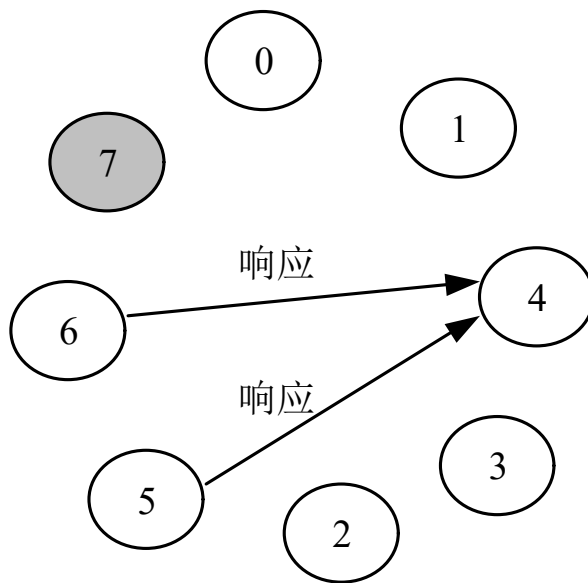
5.4 互斥算法

❖ 选举算法

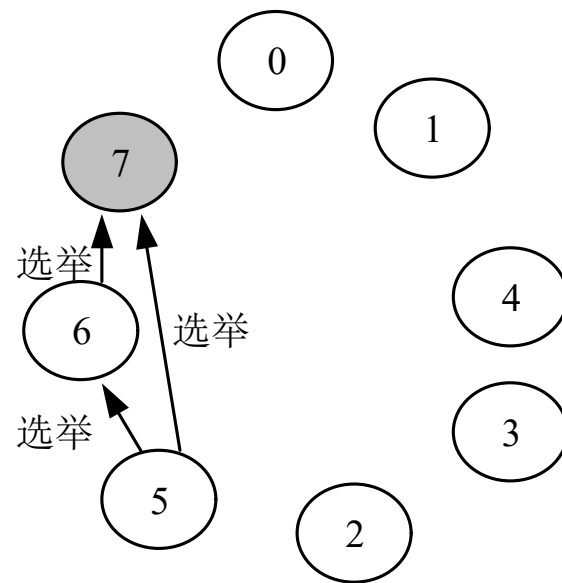
Bully选举算法实例：



(a) 进程 4 启动选举算法



(b) 进程 4 得到响应，等待通知报文



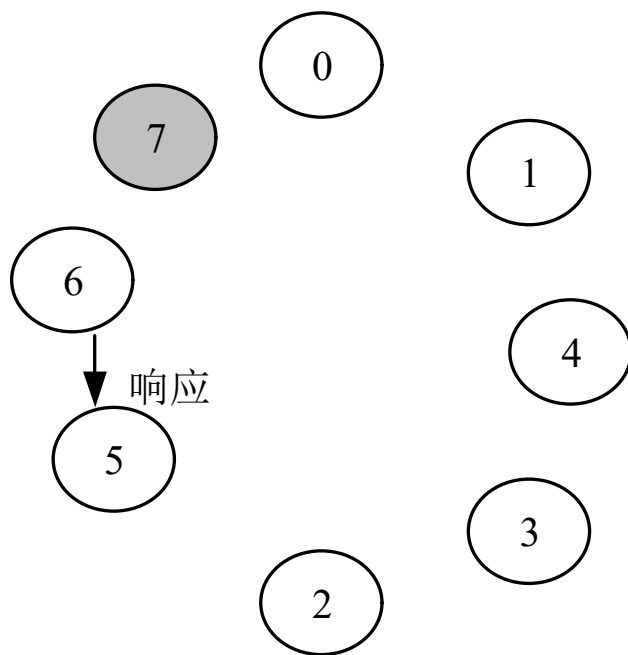
(c) 进程 5 和 6 启动选举算法

第五章 同步和互斥

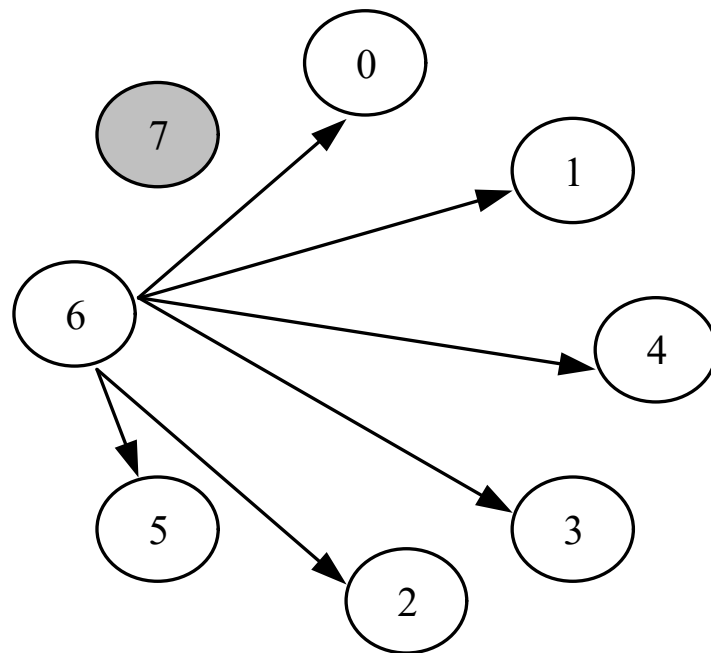
5.4 互斥算法

❖ 选举算法

Bully选举算法实例：



(d) 进程 5 得到响应，等待通知报文



(e) 进程 6 宣布自己是协调者

第五章 同步和互斥

5.4 互斥算法

❖ 选举算法

环选举算法：

- 1) 在环选举算法中，所有进程以任意的顺序排列在一个单向环上，每个进程知道环上的排列情况，任何进程在环上有一个后继进程。
- 2) 任何一个进程发现协调者失效时，它创建一个选举报文，将自己的进程号加入该报文中作为一个候选协调者，并把该选举报文传递到它的后继进程。
- 3) 收到该选举报文的后继进程，也将自己的进程号加入到该选举报文中作为一个候选协调者。如果发送者发现其后继者失效，它会将选举报文传送给后继者在环中的下一个进程，或沿环的方向可以寻找到的下一个运行的进程。

第五章 同步和互斥

5.4 互斥算法

❖ 选举算法

环选举算法：

- 4) 如果一个进程接收到自己所创建的选举报文，它将该报文的类型由选举报文改为协调者报文。
- 5) 这个协调者报文再绕环一周，这个报文用于通知每个进程协调者是谁，组成新环的成员有那些。如果进程号大的进程具有高的优先级，那么具有最大进程号的进程就是协调者；

第五章 同步和互斥

5.4 互斥算法

❖ 自稳定算法

自稳定系统：一个系统，如果无论它的初始状态是什么，总是能在有限的步骤内达到一个合法的状态，那么它是自稳定的。

一个系统S，如果它满足以下两个性质，那么它关于谓词P是自稳定的。

- (1) 终止性(closure)。P在S的执行下是终止的(closed)，也就是说，一旦P在S中被建立，它就不能被再被证明为假。
- (2) 收敛性(convergence)。从任意全局状态开始，S能保证在有限次状态转换内达到一个满足P的全局状态。

第五章 同步和互斥

5.4 互斥算法

❖ 自稳定算法

Dijkstra自稳定系统：有一个特别的进程，它对于状态转换使用独特的特权，其他进程使用另一个特权。给定 n 个 k 状态进程， $k > n$ ，标号分别为 P_0 到 P_{n-1} 。 P_0 是特别的进程，其他进程 P_i ($0 < i \leq n-1$) 是一样的。

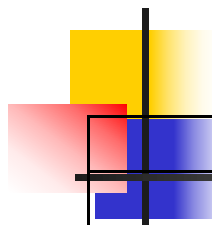
1) P_i 的状态转换如下：

$$P_i \neq P_{i-1} \rightarrow P_i := P_{i-1}, \quad 0 < i \leq n-1$$

2) P_0 的状态转换如下：

$$P_0 = P_{n-1} \rightarrow P_0 := (P_0 + 1) \bmod k$$

Di jkstra自稳定系统实例：n=3，k=4



Dijkstra 自稳定算法实例[wu,2001]

P ₀	P ₁	P ₂	特权进程	要移动的进程
2	1	2	P ₀ 、 P ₁ 、 P ₂	P ₀
3	1	2	P ₁ 、 P ₂	P ₁
3	3	2	P ₂	P ₃
3	3	3	P ₀	P ₀
0	3	3	P ₁	P ₁
0	0	3	P ₂	P ₂
0	0	0	P ₀	P ₀
1	0	0	P ₁	P ₁
1	1	0	P ₂	P ₂
1	1	1	P ₀	P ₀
2	1	1	P ₁	P ₁
2	2	1	P ₂	P ₂
2	2	2	P ₀	P ₀
3	2	2	P ₁	P ₁
3	3	2	P ₂	P ₂
3	3	3	P ₀	P ₀

第六章 分布式系统中的死锁

6.1 死锁问题

❖死锁发生的条件

- (1) 互斥。正如我们第五章所讨论的，互斥是一种资源分配方式，保证同一个资源在同一时刻最多只能被一个进程占用，它用于防止多个进程同时共享访问不可同时共享访问的资源。
- (2) 不可剥夺的资源分配。系统将一个资源的访问权分配给某一个进程后，系统不能强迫该进程放弃对该资源的控制权。
- (3) 占有并等待。必然有一个进程占用了至少一个资源，同时在等待获取被其他进程占用的资源。
- (4) 循环等待。在等待图中有一个循环路径。

第六章 分布式系统中的死锁

6.1 死锁问题

❖死锁的图论模型

可以用图模型来表示死锁，表示死锁的图模型有两种，一种是等待图，另一种是资源分配图。

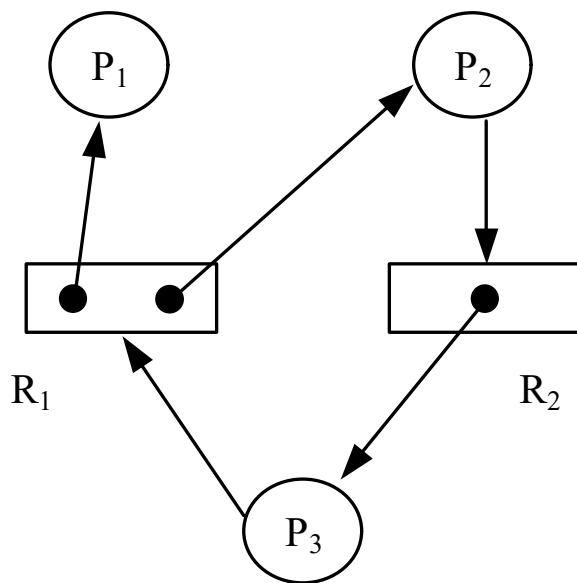
- 1) 在等待图中，节点代表进程，当且仅当进程 P_i 等待一个被进程 P_j 所占用的资源时，边 (P_i, P_j) 存在于等待图中，图中的边是有向的。
- 2) 资源分配图中的节点有两种：一种是进程节点，另一种是资源节点。每个边是一个有序对 (P_i, R_j) 或 (R_j, P_i) ，其中 P 代表进程， R 代表一个资源类型。边 (P_i, R_j) 表示进程 P_i 请求类型为 R_j 的一个资源，并且正在等待这个资源，一个资源类型中可能有多个资源。边 (R_j, P_i) 表示类型为 R_j 的一个资源已经分配给进程 P_i 。由于等待图假定一个资源类型中只有一个资源，所以资源分配图是一个比等待图更加有力的

第六章 分布式系统中的死锁

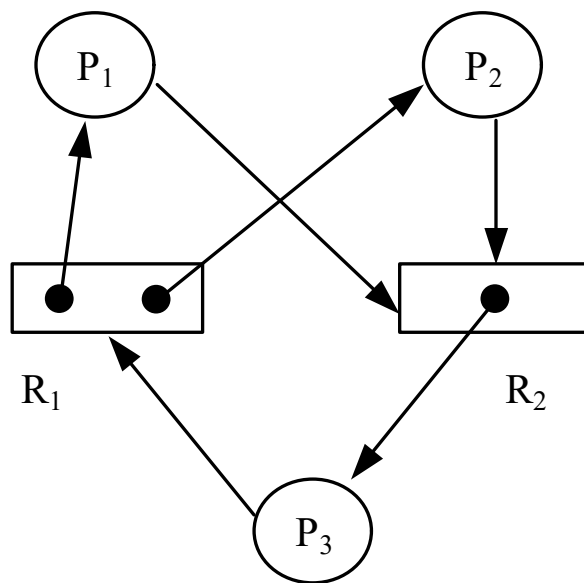
6.1 死锁问题

❖ 死锁的图论模型

资源分配图实例：



(a) 不处于死锁状态



(b) 处于死锁状态

第六章 分布式系统中的死锁

6.1 死锁问题

❖死锁的图论模型

资源分配图到等待图的转化：

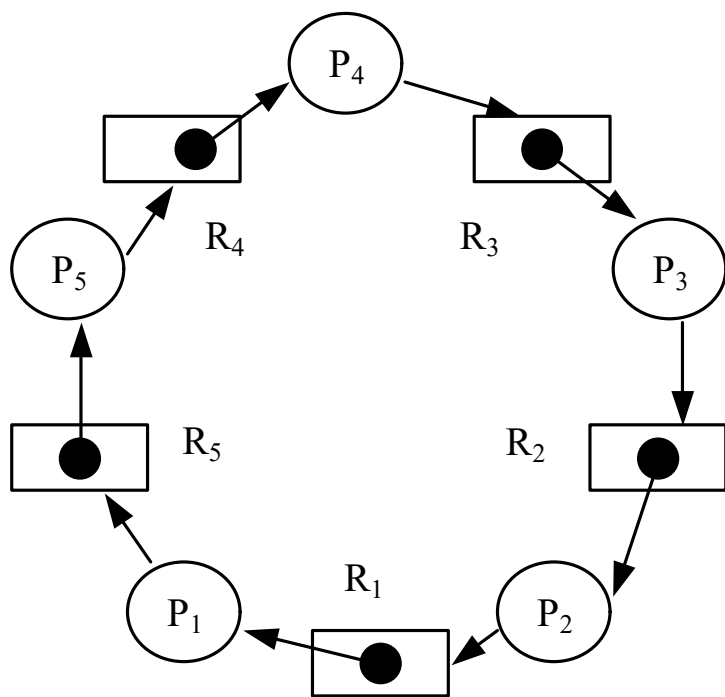
- (1) 在资源分配图中找到一个未被处理的资源R。如果所有的资源都已经处理，转向步骤3。
- (2) 从这个资源R的每个输入进程节点到每个输出进程节点之间加一条有向边。一个资源的输入进程节点是等待这个资源的进程节点，一个资源的输出进程节点是占有这个资源的进程节点。转向步骤1。
- (3) 删除所有的资源节点以及相应的边。

第六章 分布式系统中的死锁

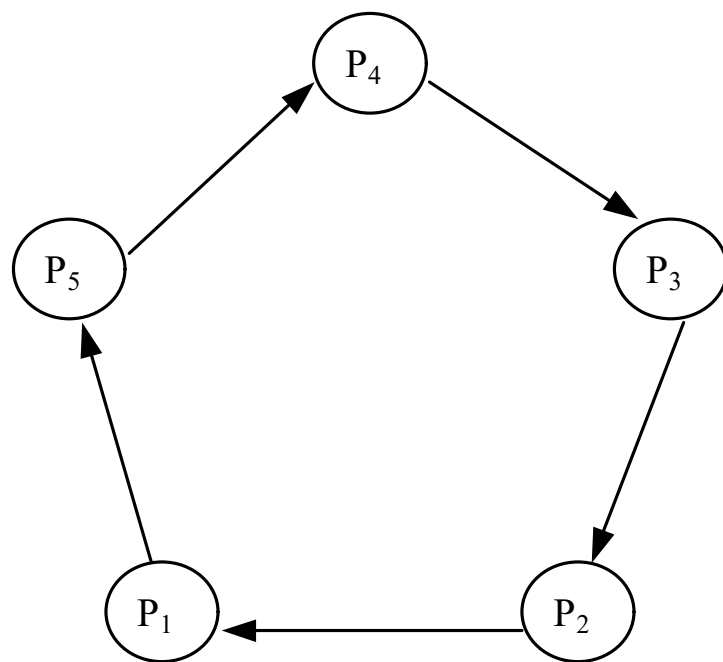
6.1 死锁问题

❖ 死锁的图论模型

资源分配图到等待图的转化实例：



(a) 资源分配图



(b) 等待图

第六章 分布式系统中的死锁

6.1 死锁问题

❖ 处理死锁的策略死锁

可以使用PAID来概括死锁处理的各种方法：预防(Prevent)、避免(Avoid)、忽略(Ignore)和检测(Detect)。

- 1) 预防死锁。通过限制请求，保证四个死锁条件中至少有一个不能发生，从而预防死锁。
- 2) 避免死锁。如果资源分配会导致一个安全的结果状态，就将资源动态地分配给进程。如果至少有一个执行序列使所有的进程都能完成运行，那么这个状态就是安全的。
- 3) 忽略死锁。忽略死锁是UNIX常采用的一种方法，这种方法只是简单地忽略死锁问题。
- 4) 检测死锁和从死锁中恢复。允许死锁发生，然后发现并解除死锁。

第六章 分布式系统中的死锁

6.1 死锁问题

❖死锁的AND条件和OR条件

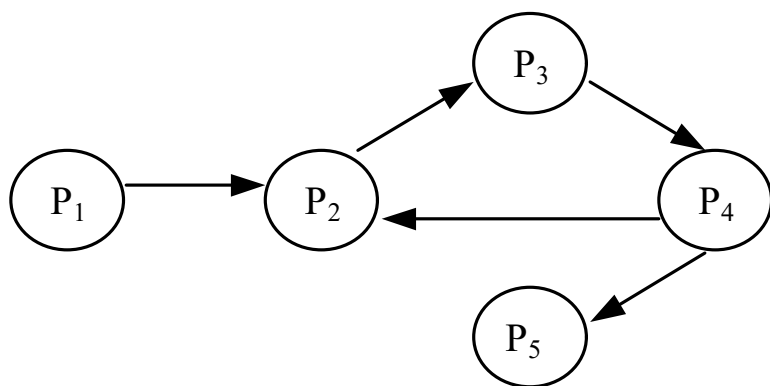
- 资源死锁和通信死锁：在通信死锁中，进程等待的资源就是报文。资源死锁和通信死锁的真正区别在于资源死锁通常使用AND条件，而通信死锁通常使用OR条件。
- 所谓AND条件就是当进程取得所有所需资源时，它才能继续执行；所谓OR条件就是当进程得到至少一个所需资源，它就能继续执行。
- 在使用AND条件的系统中，死锁条件是在等待图中存在回路。但是在使用OR条件的系统中，等待图中的回路未必会引发死锁。在使用OR条件的系统中，死锁条件是存在结(knot)。一个结K是一个节点集合，对于K中的任何节点a，a能到达K中的所有节点，并且只能到达K中的节点。

第六章 分布式系统中的死锁

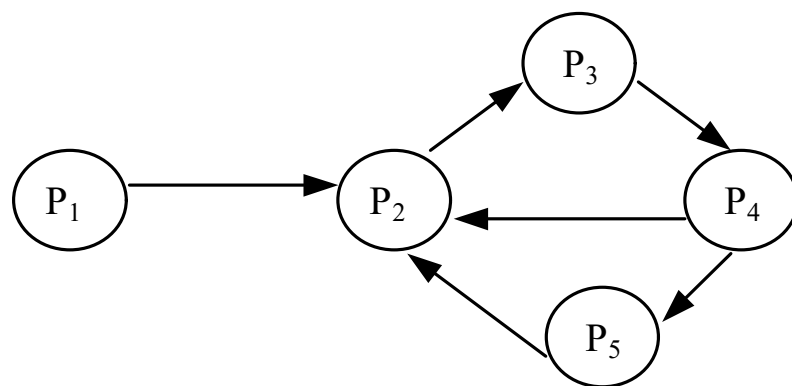
6.1 死锁问题

❖ 死锁的AND条件和OR条件

OR条件死锁图例：



(a) 无死锁



(b) 有死锁

第六章 分布式系统中的死锁

6.2 死锁的预防

❖ 预防死锁的一般方法

死锁预防算法是通过限制进程的资源请求来达到预防死锁的目的，都是通过打破四个死锁条件中的一个条件来实现的。

- 1) 进程在开始执行之前同时获得所有所需资源。这种方法打破了占有并等待的条件。
- 2) 所有的资源都要被赋予一个唯一的数字编号。一个进程可以请求一个有唯一编号 i 的资源，条件是该进程没有占用编号小于或等于 i 的资源。这样，就打破了循环等待的条件。
- 3) 每个进程被赋予一个唯一的优先级标识。优先级标识决定了进程 P_i 是否应该等待进程 P_j ，从而打破了不可剥夺的条件。

第六章 分布式系统中的死锁

6.2 死锁的预防

❖ 基于时间戳的预防死锁方法

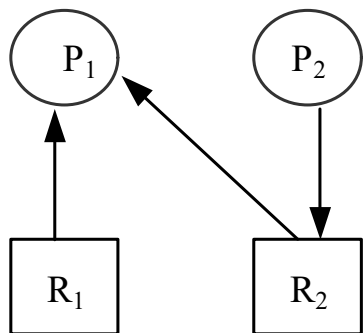
包括两种死锁预防方案。这两种方案相互补充，这种方法常用于分布式数据库系统中。

- 1) 等待—死亡方案(wait-die scheme)。该方案是基于非剥夺方法。当进程 P_i 请求的资源正被进程 P_j 占有时，只有当 P_i 的时间戳比进程 P_j 的时间戳小时，即 P_i 比 P_j 老时， P_i 才能等待。否则 P_i 被卷回(roll-back)，即死亡。
- 2) 伤害—等待方案(wound-wait scheme)。它是一种基于剥夺的方法。当进程 P_i 请求的资源正被进程 P_j 占有时，只有当进程 P_i 的时间戳比进程 P_j 的时间戳大时，即 P_i 比 P_j 年轻时， P_i 才能等待。否则 P_j 被卷回(roll-back)，即死亡。

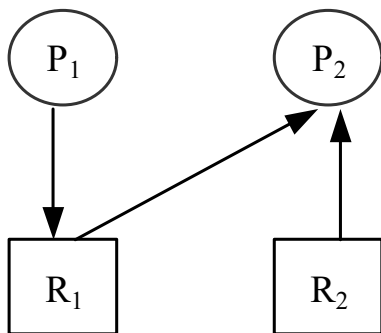
第六章 分布式系统中的死锁

6.2 死锁的预防

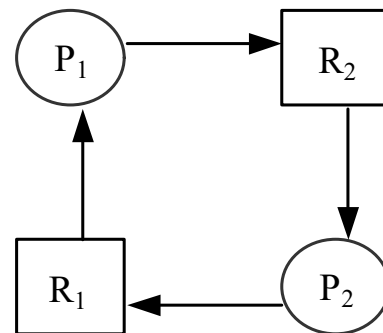
❖ 基于时间戳的预防死锁方法 图例说明：



(a) 年轻者请求年长者
占有的资源



(b) 年长者请求年轻者
占有的资源



(c) 年轻者请求年长者占有的资源, 同时,
年长者请求年轻者占有的资源

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 集中式死锁检测

在集中式死锁检测方法中，利用所有的局部资源分配图(或等待图)建立一个全局资源分配图(或等待图)。任何一个机器为它自己的进程和资源维持一个局部的资源分配图。整个系统只有一个协调者，它维持全局的资源分配图，全局的资源分配图是由局部资源分配图组合而成的。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 集中式死锁检测

全局资源分配图(或等待图)的获得方法:

- 1) 当在局部图中有边被加入或删除时, 向协调者发送一个报文, 协调者根据报文信息对全局图进行更新。
 - 2) 定期地更新, 每个机器定期地向协调者发送自上次更新以来所有添加的边和删除的边, 协调者根据报文信息对全局图进行更新。
 - 3) 当协调者认为需要运行回路检测算法时, 它要求所有的机器向它发送局部图的更新信息, 协调者对全局图进行更新。
- 当开始死锁检测时, 协调者便查找全局等待图。如果发现回路, 一个进程就会被卷回, 从而打破循环等待。

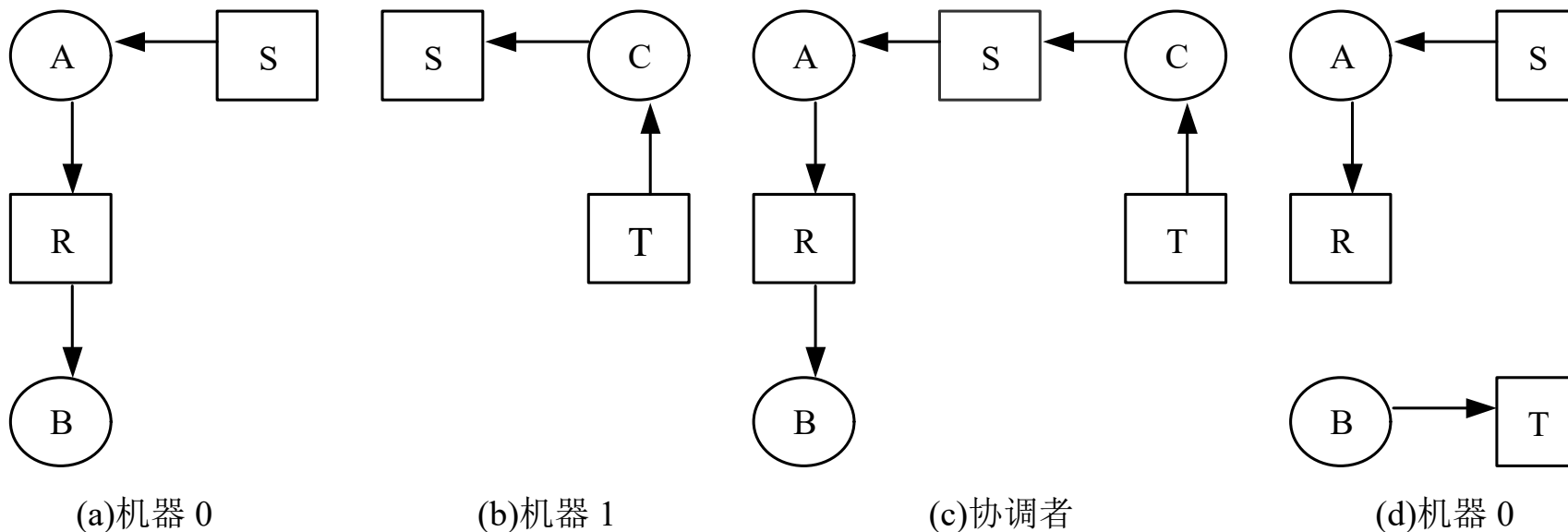
缺点: 容易产生假死锁的情况。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 集中式死锁检测

产生假死锁的图例说明：

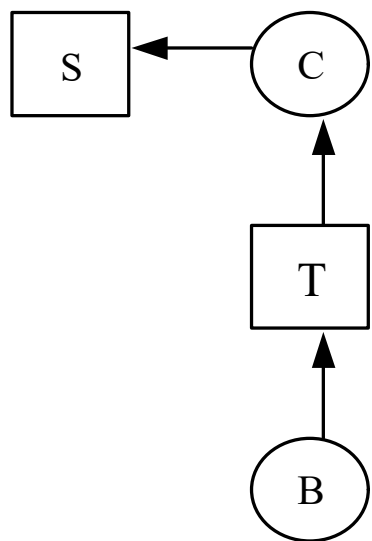


第六章 分布式系统中的死锁

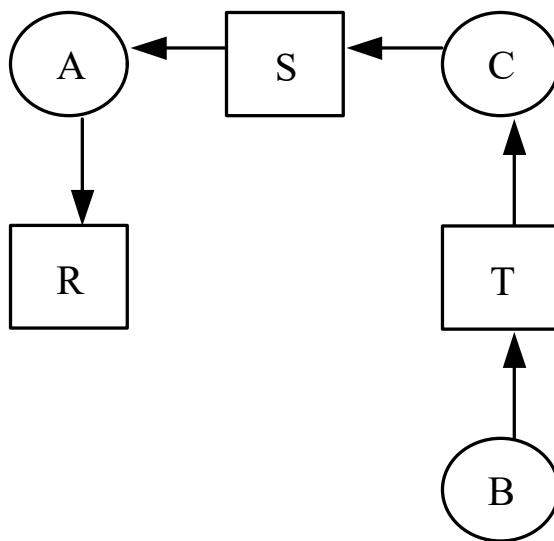
6.3 死锁的检测

❖ 集中式死锁检测

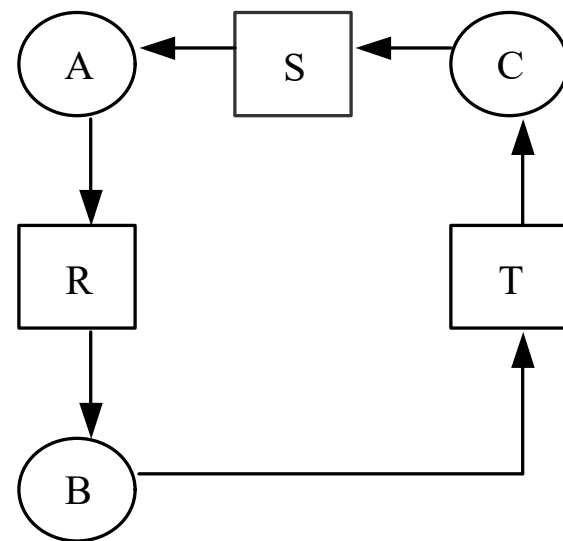
产生假死锁的图例说明：



(e) 机器 1



(f) 协调者



(g) 协调者：假死锁

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 分布式死锁检测

Knapp将分布式死锁检测算法分为以下四类：

- 1) 路径推动算法(path-pushing algorithm)。先在每个机器上建立形式简单的全局等待图。每当进行死锁检测时，各个机器就将等待图的拷贝送往一定数量的邻居节点。局部拷贝更新后又被传播下去。这一过程重复进行直到某个节点获得了足够的信息来构造一个等待图以便做出是否存在死锁的结论。
- 2) 边跟踪算法(edge-chasing algorithm)。分布式网络结构图中的回路可以通过沿图的边传播一种叫探测器的特殊信息来检测。当一个发起者得到一个与自己发送的探测器相匹配的探测器时，它就知道它在图中的一个回路里。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 分布式死锁检测

Knapp将分布式死锁检测算法分为以下四类：

- 3) 扩散计算(diffusing computation)。怀疑有死锁发生时，事务管理器通过向依赖于它的进程发送查询启动一个扩散进程。这里不会生成全局等待图。发送查询信息时，扩散计算就增长；接收回答后，扩散计算就缩减。根据所得信息，发起者会检测到死锁的发生。
- 4) 全局状态检测(global state detection)。这个方法基于Chandy和Lamport 的快照方法。可以通过建立一个一致的全局状态而无需暂停当前的计算来生成一个一致的全局等待图。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 层级式死锁检测

在层级式死锁检测算法中，站点被分层地放在一个树中。一个站点的死锁检测只涉及到它的下一级站点。例如，设A、B和C是控制器，而C是A和B的最低的公共祖先。假定节点 P_i 出现在控制器A和B的局部等待图中，那么节点 P_i 也必定出现在如下控制器的等待图中：

- (1) C控制器；
- (2) 所有位于从C到A的路径上的控制器；
- (3) 所有位于从C到B的路径上的控制器。

而且，如果 P_i 和 P_j 出现在控制器D的等待图中，并且在D的一个下一级控制器(子控制器)的等待图中有一条从 P_i 到 P_j 的路径，那么边 (P_i, P_j) 一定存在于D的等待图中。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖关于死锁检测和恢复的研究方向：

- 1) 算法正确性。严格证明死锁检测算法的正确性是困难的，由于报文的传输延迟是不可预料的，所以得到一致的全局状态是很困难的。
- 2) 算法性能。需要在信息流量(监测和恢复算法的复杂性)和死锁持续时间(监测和恢复的速度)之间达成妥协。
- 3) 死锁解决。一个好而快的死锁检测算法可能并不能提供足够的信息用于解决死锁。
- 4) 假死锁。一个检测程序不仅要满足前进要求，即必须在有限的时间内发现死锁，还要满足安全要求。如果一个死锁被发现，那么这个死锁应该是确实存在的。
- 5) 死锁概率。检测和恢复算法的设计依赖于给定系统中死锁发生的概率。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 死锁检测的实例

➤ AND模型下的Chandy-Misra-Hass算法

在Chandy-Misra-Hass算法中，分布式死锁检测算法使用一个特殊的报文，在等待图中该报文从一个进程传递到另一个进程，该报文称为探测报文(probe message)。如果报文回到发起者，那么就有死锁存在。探测报文包含一个三元组 (i, j, k) ，表示该报文是一个由进程 P_i 发起的死锁检测报文，现在由进程 P_j 所在的站点发往进程 P_k 所在的站点。

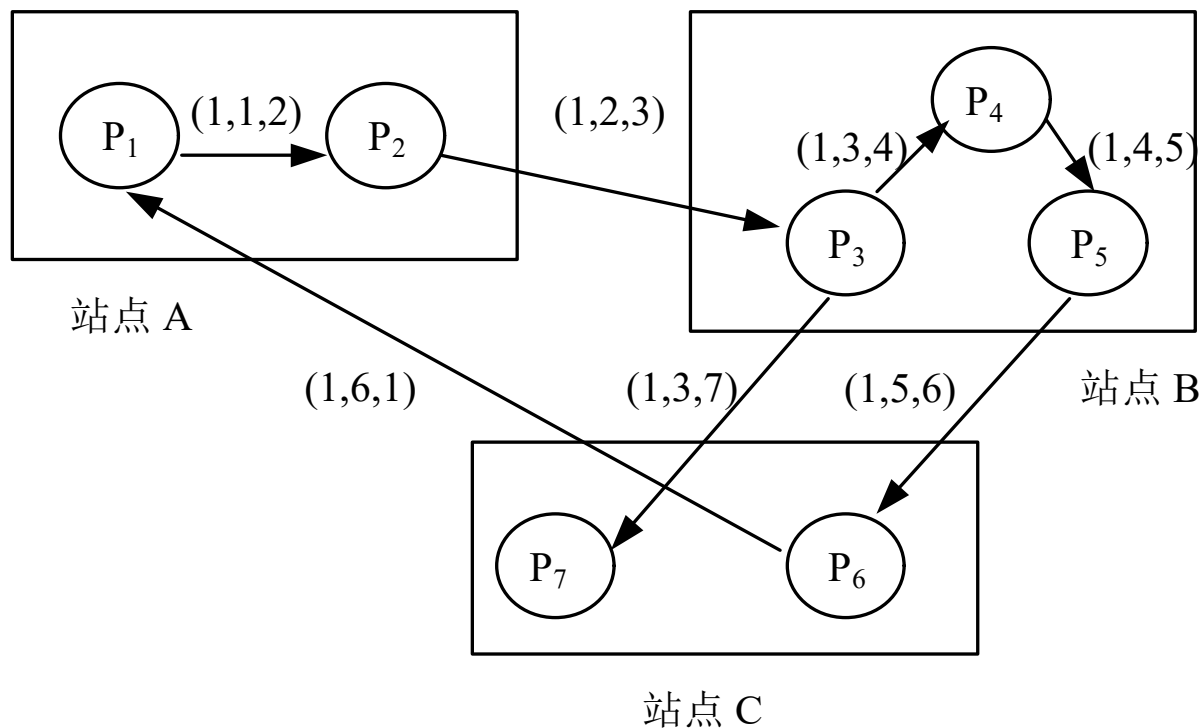
当一个进程接收到一个探测报文时，它首先检查自己是否等待某个(或某些)进程，如果它正在等待某个(或某些)进程，它将向所有它等待的进程转发这个探测报文。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 死锁检测的实例

AND模型下的Chandy-Misra-Hass算法图例说明：



第六章 分布式系统中的死锁

6.3 死锁的检测

❖死锁检测的实例

AND模型下的Chandy-Misra-Hass算法中打破死锁方法：

- 1) 一种方法是由探测报文的发起者杀死自己，但是，当有多个进程同时检测到同一个死锁时，与同一个死锁有关的多个进程会被杀死，这样做的效率会很低。
- 2) 另一种方法是每个收到探测报文的进程将自己的标识符附加到探测报文的后面，当探测报文回到发起者进程时，发起者进程选取一个具有最大标识符号码的进程杀死，即使有多个进程同时检测到同一个死锁，它们也会选择杀死同一个进程。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 死锁检测的实例

➤ AND模型下的Mitchell-Merritt算法

Mitchell-Merritt算法与Chandy-Misra-Hass算法的区别：

- 其限制是每个进程每次只能请求一个资源。
- 探测报文在等待图中，沿等待方向的相反方向传送，这样的图叫反向等待图(reversed wait-for graph)。
- 每当进程收到探测报文时，它将自己的标识符和探测报文发起者的标识符相比较，如果自己的标识符大于探测报文发起者的标识符，它就用自己的标识符取代探测报文发起者的标识符，自己变成探测报文的发起者。
- 当几个进程同时发起死锁检测时，只有一个进程能够成为唯一的探测者。

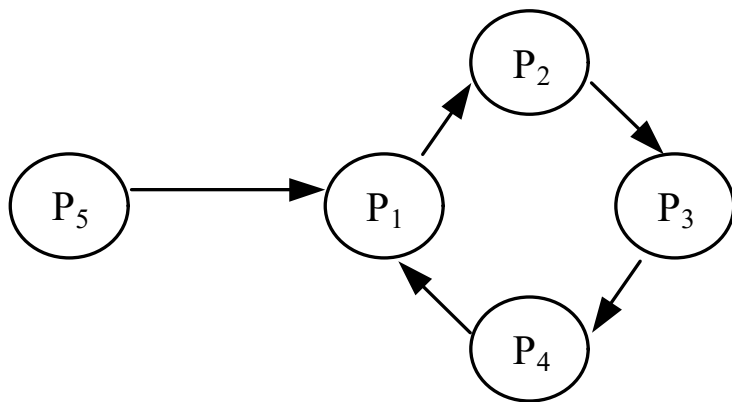
第六章 分布式系统中的死锁

6.3 死锁的检测

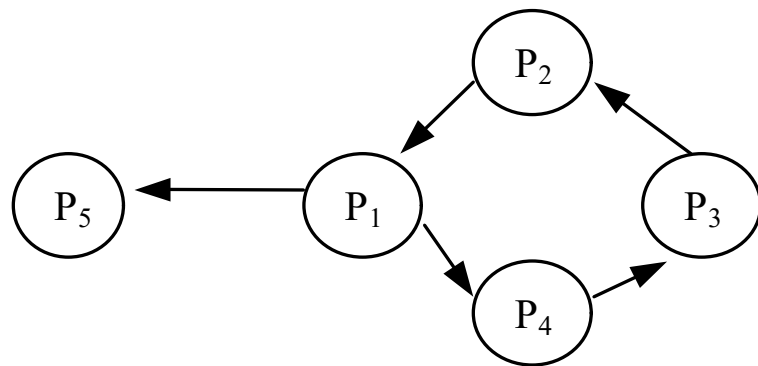
❖ 死锁检测的实例

➤ AND模型下的Mitchell-Merritt算法

Mitchell-Merritt算法图例说明：



(a) 等待图



(b) 反向等待图

第六章 分布式系统中的死锁

6.3 死锁的检测

❖ 死锁检测的实例

➤ AND模型下的Mitchell-Merritt算法

为什么每个进程每次只能请求一个资源？

在每个进程每次只能请求一个资源的情况下，等待图中的一个循环中的每个节点都等待环内的节点，所以只有进入环内的边。其反向等待图中则没有进入环内的边，就不会有大标识符的进程产生的探测报文进入回路，而不能检测出死锁。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖死锁检测的实例

➤OR模型下的Chandy-Misra-Hass算法

- 1) 使用两类报文: (query, i, j, k) 和 (reply, i, j, k) , 表示这些报文属于由进程 P_i 发起的并由 P_j 送往 P_k 的扩散计算。
- 2) 一个进程的依赖集合包括所有它在等待以便获得报文的进程。如果接收进程 P_k 是活动的, 它会忽略所有的查询和回答报文。如果它被阻塞, 它会向它的依赖集合中的进程发送查询。
- 3) 一旦收集到回答报文, 接收进程将向发起者发送一个回答报文。发起者以及每个中间进程用一个计数器记录查询和回答的数目。如果这两个数字相同, 即发起者的每个查询都得到了回答, 就表明发起者处于死锁状态。

第六章 分布式系统中的死锁

6.3 死锁的检测

❖死锁检测的实例

➤OR模型下的Chandy-Misra-Hass算法

当接收进程 P_k 处于阻塞状态时，会有几种可能：

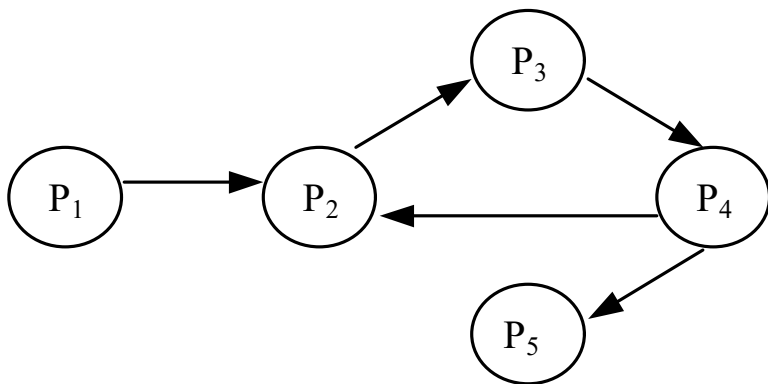
- 如果这是 P_i 发起的第一个来自 P_j 的报文(这个报文的发送者 P_j 叫做 P_k 关于 P_i 的结合者)，它将向它的依赖集合中的所有进程发送这个查询，并且将查询数目存储在一个局部变量 $num(i)$ 中。令局部变量 $wait(i)$ 表示这一进程从它接收到它的第一个由 P_i 发起的查询起一直被阻塞这一事实。
- 如果这个查询是 P_i 发起的但不是第一个来自 P_j 的报文，即当 $wait(i)$ 仍然成立时， P_k 将马上回答。
- 如果从 $wait(i)$ 变为假的那一时刻 P_k 运行过，那么这个查询就被丢弃。

第六章 分布式系统中的死锁

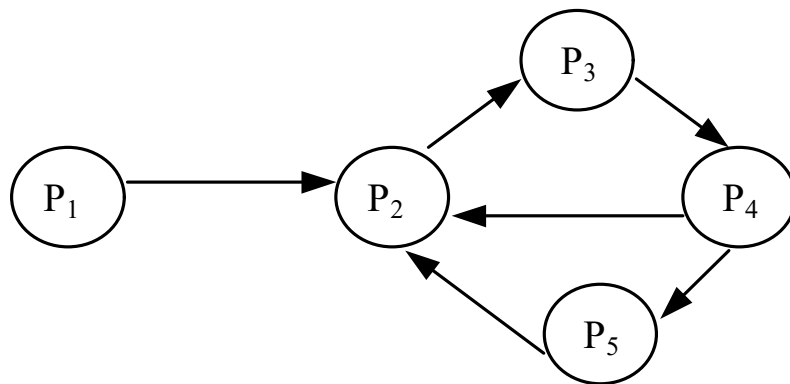
6.3 死锁的检测

❖ 死锁检测的实例

OR模型下的Chandy-Misra-Hass算法图例说明：



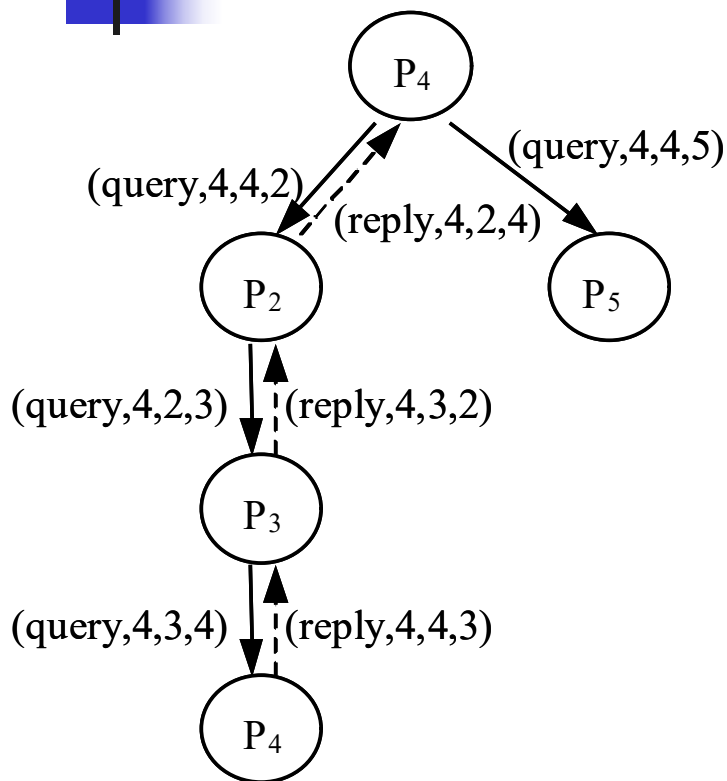
(a) 无死锁



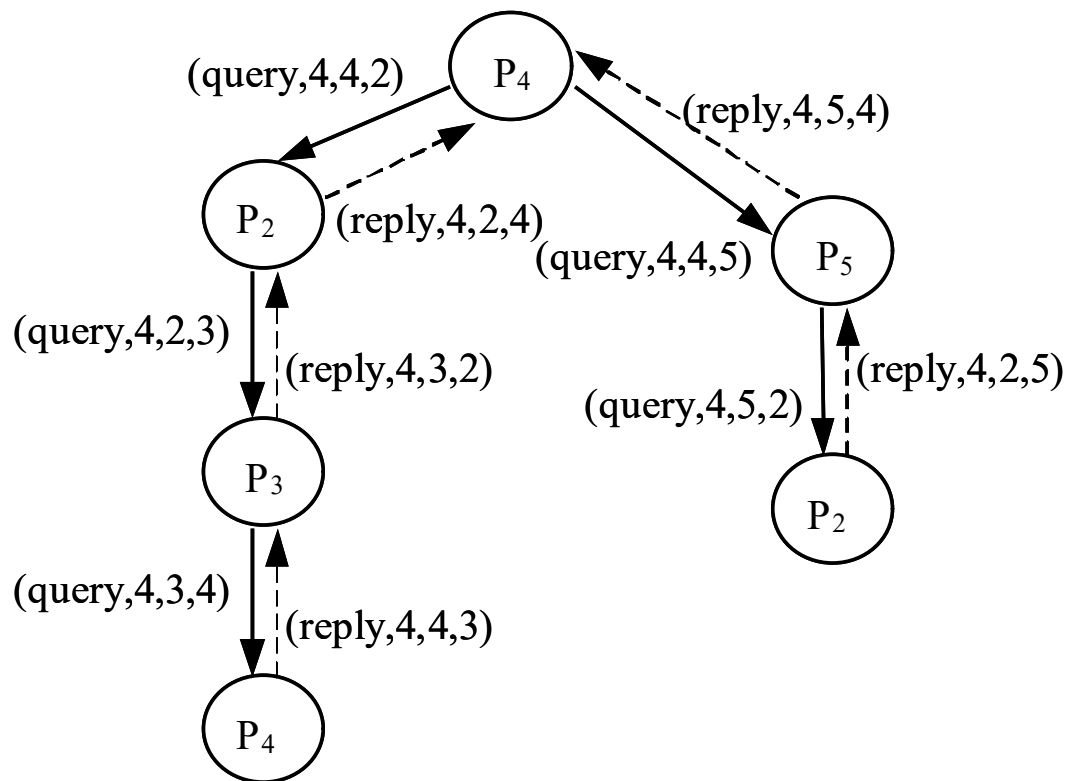
(b) 有死锁

第六章 分布式系统中的死锁

6.3 死锁的检测



(c) 图(a)的死锁检测过程



(d) 图(b)的死锁检测过程

第七章 分布式系统中容错技术

7.1 分布式系统中的故障模型

❖ 基本概念

- 1) 可用性：可用性反映的是系统随时可被用户使用的特性。
 - 2) 可靠性：可靠性指的是在错误存在的情况下，系统持续服务的能力。
 - 3) 安全性：安全性指的是在系统出现暂时错误的情况下，不出现灾难性后果的能力。
 - 4) 可维护性：可维护性指的是系统一旦出现故障，系统易于修复的能力。
 - 5) 保密性：保密性要求系统资源不被非法用户访问。
- 错误的时间特性来看，错误可分为：暂时性的(transient)、间歇性的(intermittent)和永久性的(permanent)。

第七章 分布式系统中容错技术

7.1 分布式系统中的故障模型

❖基本的故障模型

➤故障划分几种类型：

分布式系统中故障类型

故障类型	说明
崩溃性故障	服务员停机，但是在服务员停机之前工作是正常的
遗漏性故障	服务员对输入的请求没有响应
接收性遗漏	服务员未能接收到输入报文
发送性遗漏	服务员未能发送出输入报文
时序性故障	服务员对请求的响应不是按特定的时间间隔进行的
响应故障	服务员的响应是错误的
值错误	服务员给出了错误的响应值
状态转换错误	服务员背离了正确的控制流程
随意性故障	服务员在随意的时刻产生了一个随意的响应。

第七章 分布式系统中容错技术

7.1 分布式系统中的故障模型

❖ 基本的故障模型

➤ 容错是建立在冗余的基础上的，下面是四种冗余类型：

- (1) 硬件冗余。附加额外的处理器、I/O设备等。
- (2) 软件冗余。附加软件模块的额外版本等。
- (3) 信息冗余。如使用了额外位数的错误检测代码等。
- (4) 时间冗余。如用来完成系统功能的额外时间。

有些研究者将冗余分为三类，即物理冗余、信息冗余和时间冗余。物理冗余可以用硬件冗余的方式或软件冗余的方式来实现，因为硬件和软件在逻辑上是等同的。

第七章 分布式系统中容错技术

7.1 分布式系统中的故障模型

❖ 基本的故障模型

➤ 故障的基本处理方法：

- (1) 主动复制。所有的复制模块协同进行，并且它们的状态紧密同步。
- (2) 被动复制。只有一个模块处于动态，其他模块的交互状态由这一模块的检查点定期更新。
- (3) 半主动复制。是主动复制和被动复制的混合方法。此种方法所需的恢复开销相对较低。

➤ 失效的检测可分为外部检测和内部检测两类：

- 1) 外部检测是指将检测节点失效的职责赋予被检测节点的外部附件。
- 2) 内部检测将节点的失效检测机制置于该节点内部，通常检测部件被假定为一个可以完全信赖的“硬核”。

第七章 分布式系统中容错技术

7.2 容错系统的基本构件

❖ 坚固存储器

坚固存储器是对一个可以经受系统失效的特定存储器的逻辑抽象，也就是说，坚固存储器里的内容不会被一个失效所毁坏。

坚固存储器的实现方法：

- 磁盘镜像：坚固存储器可以用一对普通磁盘来实现。坚固存储器中的每一个块由两个独立的磁盘块组成，分别位于不同的驱动器上，使得它们同时由于硬件故障受到损坏的机会最小。
- RAID：另一种实现坚固存储器的方法是使用廉价磁盘冗余阵列(RAID)。RAID是通过运用位交错技术将数据分布到多个磁盘中，从而提供高I/O性能。可以用一个或几个磁盘来检测或屏蔽错误，RAID与传统磁盘相比有显著的优点，并可承受多个失效。

第七章 分布式系统中容错技术

7.2 容错系统的基本构件

❖故障—停止处理器

- 当一个处理器失效，最可能的是它不进行任何不正确的操作，并且简单地停止运行，这样的处理器被称为故障—停止处理器，一个故障—停止处理器由多个处理器组成。
- 失效的效果：当出现一个故障时，故障—停止处理器会有以下效果：(a) 处理器停止运行；(b) 易失性存储器的内容丢失，而坚固存储器不受影响；(c) 任何其他处理器均可以检测到故障—停止处理器的失效状态。

第七章 分布式系统中容错技术

7.2 容错系统的基本构件

❖故障—停止处理器

➤故障—停止处理器的实现：现有一个可靠的坚固存储器、一个可靠的存储处理器(控制存储媒介的处理器)以及 $k+1$ 个处理器，将它们转变成故障—停止处理器的方法如下： $k+1$ 个处理器中的每一个都运行同样的程序并通过存储处理器访问同一个坚固存储器。如果任何一个请求是不同的，或者任何一个请求没有在指定的期间到达，则意味着检测到一个失效事件，因而应该丢弃所有请求。因为系统表现为一个故障—停止处理器，这一方法产生一个 k —故障—停止处理器，除非系统中 $k+1$ 个或更多的部件失效。

第七章 分布式系统中容错技术

7.2 容错系统的基本构件

❖ 原子操作

一个原子操作就是由硬件独立执行的一系列动作。也就是说，每一个动作或者被完全彻底地执行，或者所有的动作根本没有执行，系统的状态保持不变。原子操作中的每一个动作都是孤立的，当执行这一动作时，在进程中感觉不到外界活动的存在，也意识不到外界状态的变化。与此相似，任何外界的进程均感觉不到一个孤立的原子操作的内在状态的变化。这就是所谓的原子操作的“全有或全无”的性质，即一个原子操作要么全部完成，要么在执行过程中出现错误的时候相当于根本没有执行。

原子操作失效时，可以通过简单地重做来恢复。

第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向前式恢复和向后式恢复：

- 1) 在向前式恢复中，假定可以完全准确地得到系统中的故障和损失的性质，这样就有可能去掉这些故障从而使得系统继续向前执行。
- 2) 向后式恢复适用于系统的故障无法预知和去掉的情况，在这种情况下，要定时地存储系统的状态，这样当失效导致系统处于不一致的状态时，系统可以恢复到从前没有发生故障的状态，在此状态下重新执行。

第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向后式恢复：

- 1) 检查点：在向后式恢复中进程被恢复到一个先前的正确的状态。进程执行中的一些点被称为“检查点”(checkpoint)，在以后发生错误的情况下，进程可以被恢复到这些点。在检查点的实现过程中，需要考虑两个主要问题：检查点的存储和检查点的更新。
- 2) 有两种方法来保存检查点：(1) 每一个检查点被组播到每一个备份模块；(2) 每个检查点被存储在它的本地坚固存储器中。当进程正确地从一个旧的检查点运行到一个新的检查点时，旧的检查点就要被新的检查点替换。当进程执行到两个检查点之间时发生错误，那么进程应该卷回到旧的检查点处重新执行。

第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向后式恢复:

3) 检查点的原子更新: 当使用新的检查点替换旧的检查点的过程中, 系统也会发生失效。这可以通过检查点的原子更新过程来解决, 也就是说, 在检查点的更新中, 要么旧的检查点被新的检查点替换, 要么旧的检查点被完整地保留。

4) 检查点原子更新的实现:

假设库A和库B现在保存的检查点是C1, 现在要用检查点C2取代库A和库B的内容。在取代前, 假设 $T_{a1}=T_{a2}=T_{b1}=T_{b2}=1$, 检查点的更新过程如下: (1) 为了更新库A, 先置 $T_{a1}=2$; (2) 将库A的内容用检查点C2取代; (3) 库A更新完毕, 置 $T_{a2}=2$; (4) 为了更新库B, 先置 $T_{b1}=2$; (5) 将库B的内容用检查点C2取代; (6) 库B更新完毕, 置 $T_{b2}=2$;

第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向后式恢复：

5) 识别在检查点更新过程中发生的失效：

条件	失效	措施
$T_{a1}=T_{a2}=T_{b1}=T_{b2}$	没有	没有
$T_{a1}>T_{a2}=T_{b1}=T_{b2}$	在刷新库 A 的过程中失效	将库 B 复制到库 A 中
$T_{a1}=T_{a2}>T_{b1}=T_{b2}$	库 A 刷新完成之后，库 B 开始刷新之前失效	将库 A 复制到库 B 中
$T_{a1}=T_{a2}=T_{b1}>T_{b2}$	在刷新库 B 的过程中失效	将库 A 复制到库 B 中

第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向后式恢复：

6) 基于影像页面技术的恢复方案：

在基于影像页面技术的方案中，当进程需要修改一个页面时，系统复制该页并保留在坚固存储器中。系统中每个页面都有两个拷贝，当进程在执行的过程中，只有其中的一个拷贝被进程修改，另一个拷贝就作为影像页面。如果进程失效，则丢弃被修改的拷贝，系统根据影像页面进行恢复。如果进程成功运行，则每一份影像页面被相应的修改后的页面替换。

第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向前式恢复:

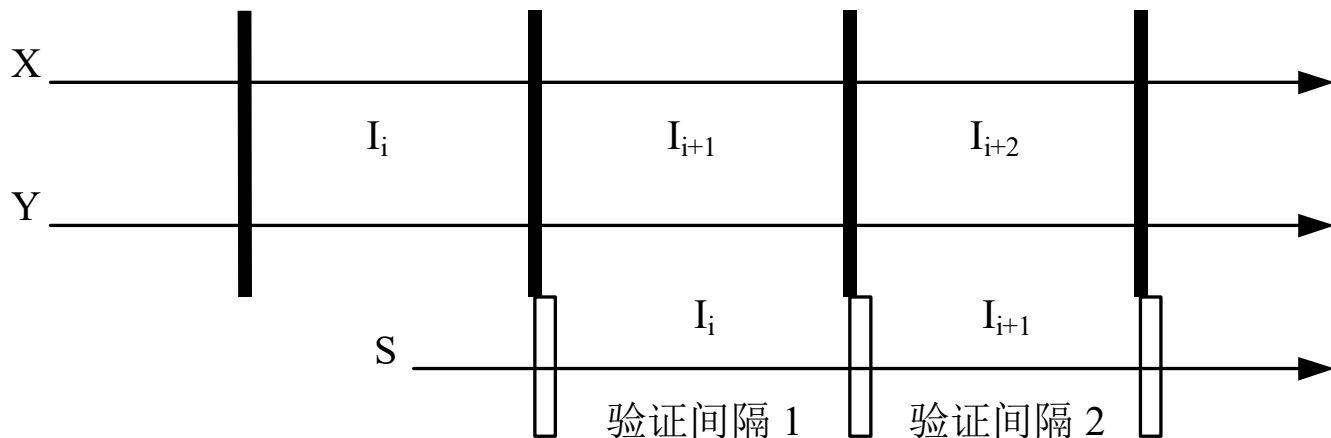
- 1) 一个进程或任务的初始拷贝由不同的处理器来运行。
- 2) 这些版本的结果在检查点进行表决或比较, 如果表决结果是成功的, 则可以获得一个储存在坚固存储器中的正确结果。
- 3) 如果表决结果是失败的, 对以前的任务进行一次回卷执行。也就是说, 在后备处理器上再运行以前的任务, 目的是获得正确的结果。
- 4) 尽管在所有版本都失效(所有结果都不正确)或者表决也不能获得正确结果的情况下, 回卷运行是不可避免的, 但由于利用了现存的正确结果而不必从头重新开始, 还是节省了回卷时间。

第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向前式恢复:

向前式恢复方案的实例：其中 I_i ， I_{i+1} 和 I_{i+2} 是检查点间隔。两个进程X和Y均运行一个进程的同一个版本。在每一个检查点之前，需要对它们的结果进行比较并确认是否正确。S是一个后备处理器，对两个间隔 I_i 和 I_{i+1} 进行验证。有以下四种可能的情况：

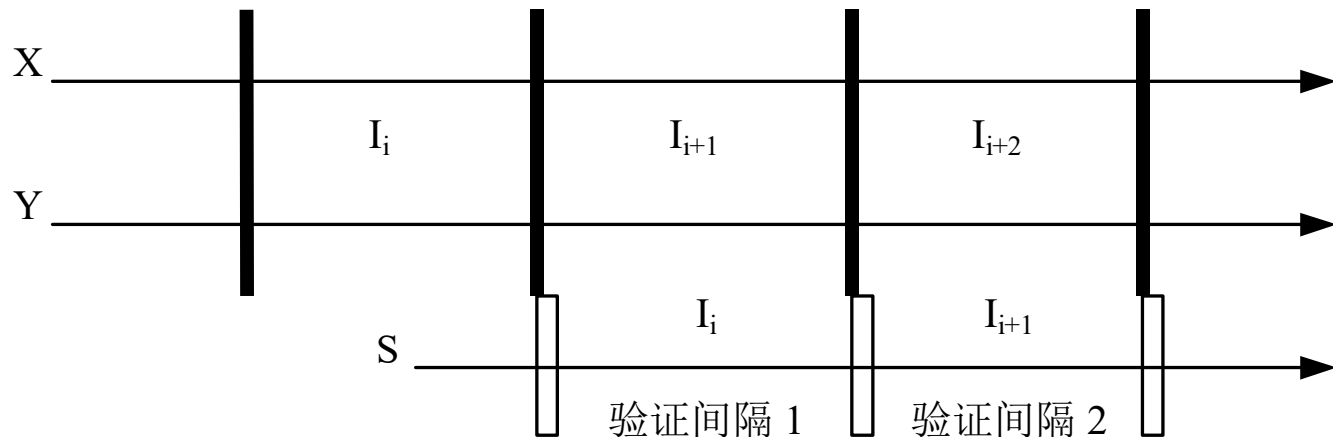


第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向前式恢复:

(1) 没有并发重试。如果X和Y都在间隔 I_i 正确运行，那么X和Y在间隔 I_i 所得的结果是相同的，S不进行并发重试。

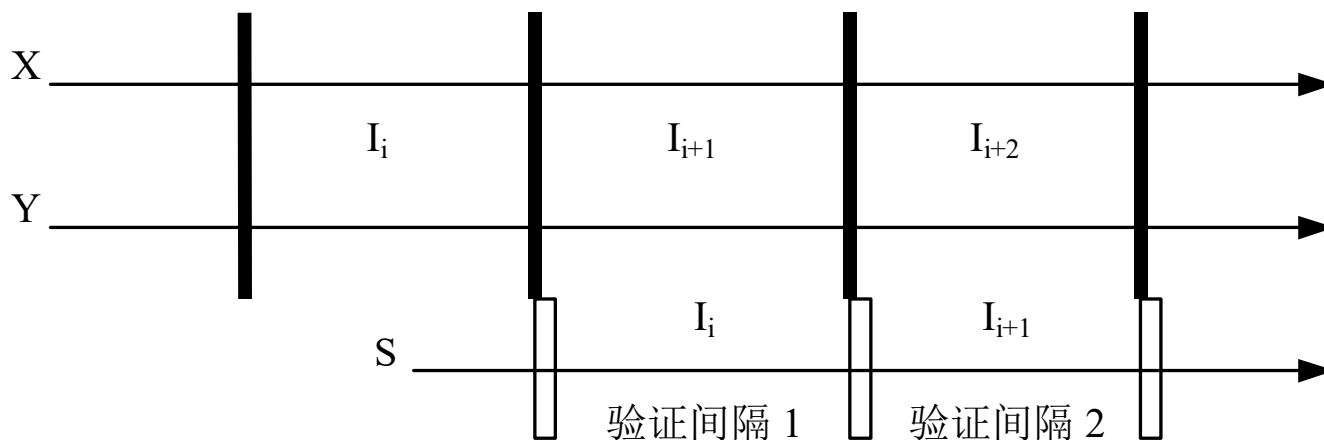


第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向前式恢复:

(2) 有非回卷的并发重试。在 I_i 中出现了错误，但在两个合法的检查点间隔 I_{i+1} 和 I_{i+2} 中间没有错误。如果我们用 (X_i, Y_i, S_i) 代表在间隔 I_i 之中 X 、 Y 和 S 的状态，并且用0代表错误，1代表正确，d代表没有关系(也就是说，或者错误或者正常)。 (X_i, Y_i, S_i) 就有两种情况： $(1, 0, 1)$ 和 $(0, 1, 1)$ ，无论哪种情况，系统都可以判断哪个进程是正确的，所以不用回卷。

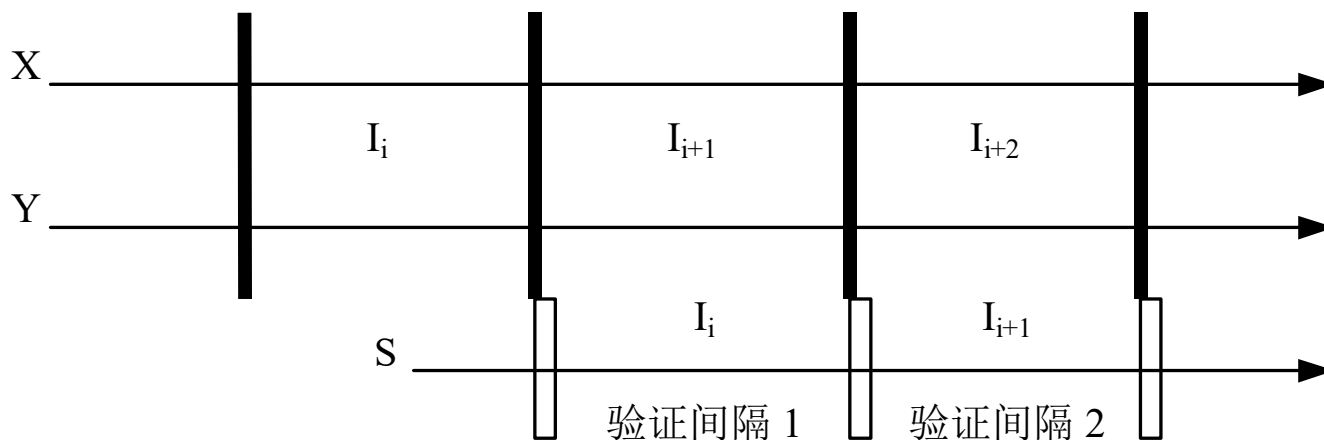


第七章 分布式系统中容错技术

7.3 节点故障的处理

❖ 向前式恢复:

(3) 在一次并发重试的间隔后进行回卷。这种情况对应于在 I_i 中有两个进程(X、Y和S中的两个)有错误的情况。如果我们用 (X_i, Y_i, S_i) 代表在间隔 I_i 之中X、Y和S的状态, 那我们就得到三种情况: $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, d)$ 。这三种情况是在一次并发重试的间隔后进行回卷。系统卷回到 I_i 的开始处。



第七章 分布式系统中容错技术

7.3 节点故障的处理

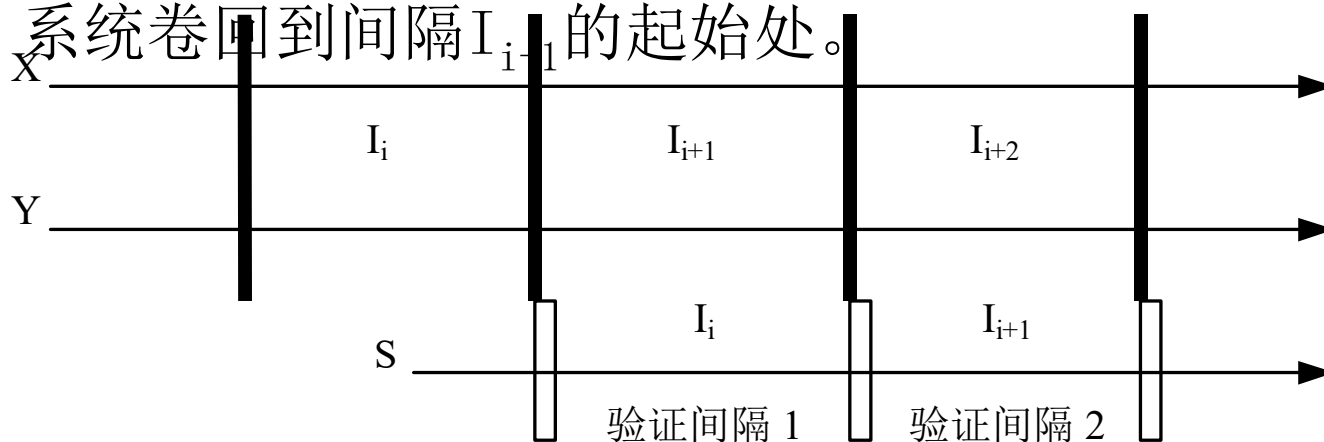
❖ 向前式恢复:

(4) 在并发重试的两次间隔之后回卷。该情况下在检查点间隔 I_{i+1} 处出现了一个额外的错误，如果我们用

$(X_i, Y_i, S_i, X_{i+1}, Y_{i+1}, S_{i+1})$ 代表在间隔 I_i 和 I_{i+1} 中 X 、 Y 和 S 的状态，

这种情况对应于以下描述的四种情形：(1, 0, 1, d, d, 0)，

(1, 0, 1, 0, d, d)，(0, 1, 1, d, d, 0)，(0, 1, 1, d, 0, d)。可以确定间隔 I_i 中哪个进程是正确的，但是不能确定间隔 I_{i+1} 中哪个进程是正确的。系统卷回到间隔 I_{i+1} 的起始处。



第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖一致性检查点

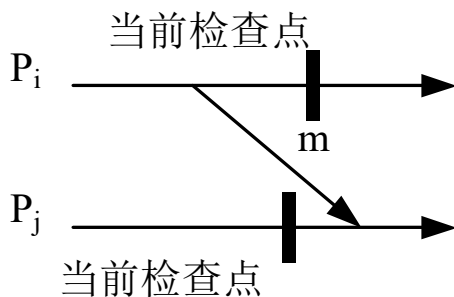
- 1) 全局状态：一种全局状态的定义是一系列局部状态的集合，这里的局部状态就是一个进程的检查点，每个局部进程有一个局部状态。
- 2) 局部检查点可能组成如下两种不一致的全局状态：
 - a) 丢失报文。进程 P_i 的检查点状态显示它给进程 P_j 发送了报文 m ，但是进程 P_j 并没有关于该报文的纪录。
 - b) 孤儿报文。进程 P_j 的检查点状态显示它收到了一个来自进程 P_i 的报文 m ，但是进程 P_i 的状态显示它没有向进程 P_j 发送过报文 m 。

第七章 分布式系统中容错技术

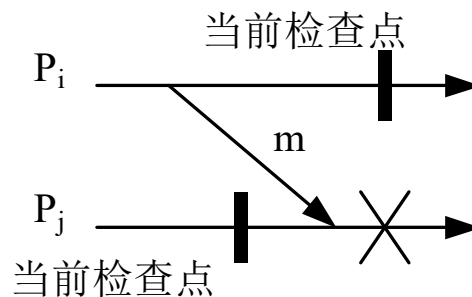
7.4 分布式检查点算法

❖ 一致性检查点

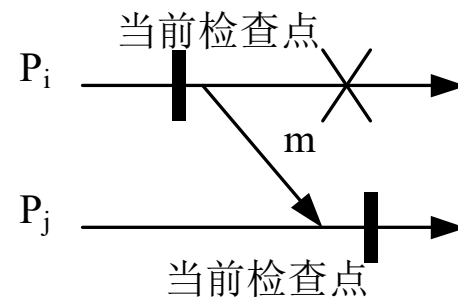
3) 不一致全局状态实例:



(a) 检查点设置不合理导致报文丢失



(b) 下一个检查点之前的崩溃导致报文丢失



(c) 失效导致孤儿报文

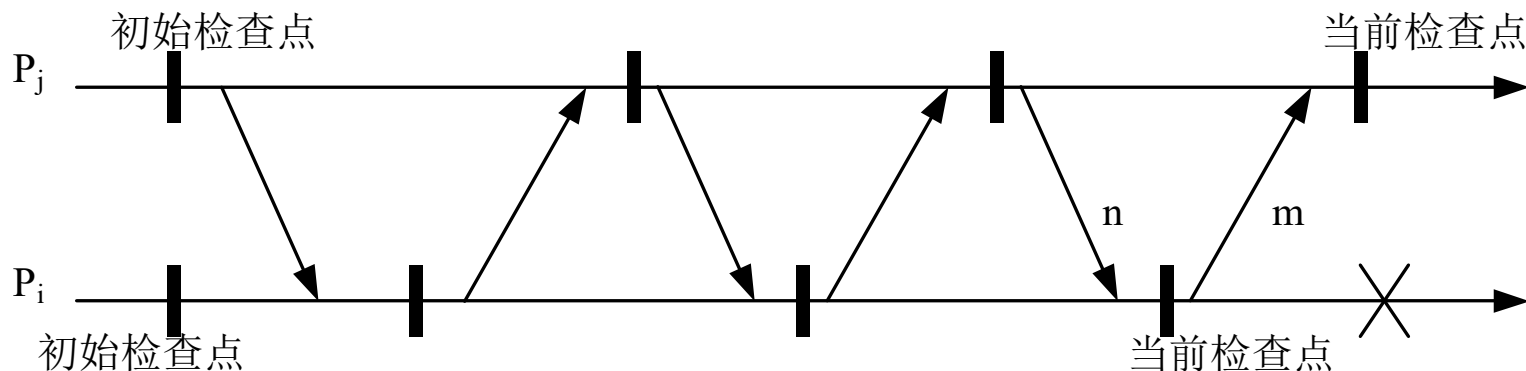
第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 一致性检查点

4) 多米诺效应(domino effect)

为了解决孤儿报文的问题，进程 P_j 回卷到上一个检查点时，要清除对孤儿报文的纪录。然而，这样一来有可能出现这样一种情况：在最近的检查点和上一个检查点之间， P_j 向 P_i 发送了一个报文 n ，假定 P_i 在当前检查点之前收到了报文 n ，现在这个报文 n 成了孤儿报文。这样， P_i 需要进一步回卷。这种由于一个进程的回卷导致另外一个或多个进程的回卷的效应叫做多米诺效应。



第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖一致性检查点

- 5) 一个强一致 (strongly consistent) 的检查点集合是由一系列的没有孤儿报文和没有丢失报文局部检查点组成。
- 6) 一个一致的检查点集合是由一系列没有孤儿报文的局部检查点组成。

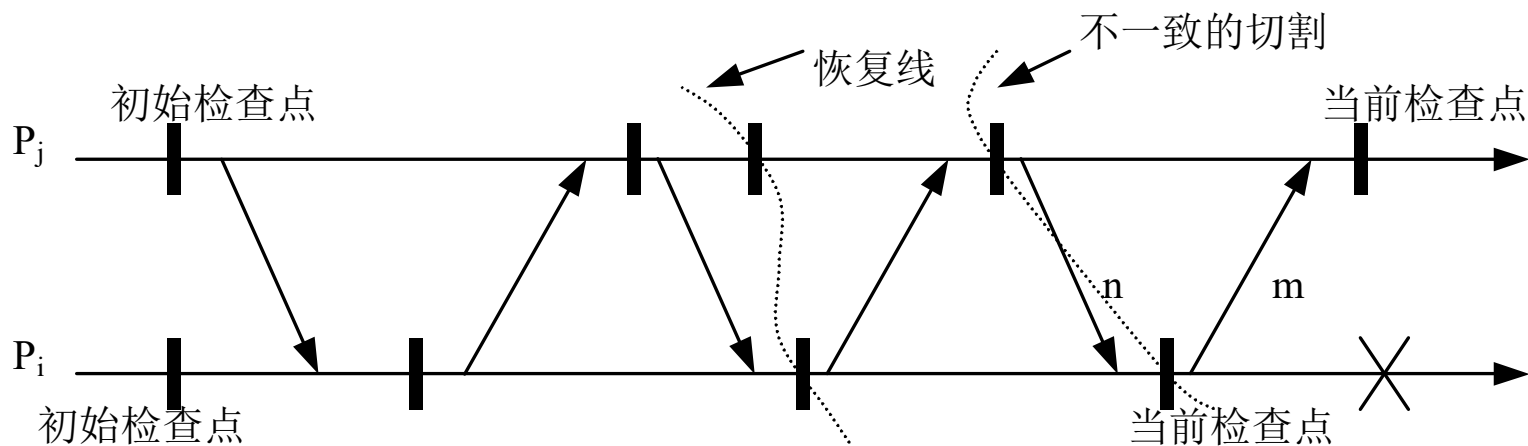
显然一个强一致的检查点集合包括一系列局部检查点，在这些检查点之间，进程之间没有报文传送。如果每个进程都在发送一个报文之后生成一个检查点，那么最近的检查点集合将永远是一致的。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 一致性检查点

- 7) 恢复线和切割线：当一个进程或系统失效的时候要求利用这些局部状态重新构造一个全局一致的状态。一个分布式快照对应一个全局一致的状态，这个分布式快照可以作为一个恢复线(recovery line)用于恢复。一个恢复线对应于最近的一个一致性切割线。



第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 异步检查点

- 1) 异步检查点算法中程序中检查点状态的保存过程较为简单，程序中各进程周期性地相互独立地保存自己的运行状态，程序各进程之间不需要相互协商。
- 2) 在恢复过程中，各进程之间则需要相互协商通过复杂的回卷算法各自回卷到合适的检查点时刻以使整个程序的各个进程恢复到最近的一个一致的全局状态。
- 3) 一致检查点的检测方法：
 - a) 比较发送的和接收的报文数量来检测孤儿报文的存在。如果接收到的报文数目和任何发送报文的进程发送的报文的数目是一致的，那么就可以认为找到了一个局部检查点的一致集合。
 - b) 使用间隔依赖图来进行检测。如果每个进程 i 的向量时钟是 LC_i ，一个检查点集合是一致的，当且仅当不存在 i 和 j 满足 $LC_i < LC_j$ 。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 异步检查点

- 4) 异步检查点算法的优缺点：优点是允许分布式程序的各个进程拥有最大程度的自治性，因而算法的延迟较小。缺点之一是由于每个进程需要保存若干时刻的检查点信息，空间开销较大；缺点之二是在恢复过程中可能会重复回卷，甚至出现多米诺效应，使程序一直回卷到初始状态。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 同步检查点

- 1) 在同步检查点算法中，各相关的进程协调它们的局部检查点的建立行为，以保证所有的最近的检查点都是一致的。在同步检查点中，只有最近的一致检查点集合才需要被维护和保存。
- 2) 同步检查点算法的优缺点：由于使用同步检查点算法，各进程的局部检查点组成的集合是一个全局一致的状态，所以在恢复时各个进程只需要简单地从检查点处重新开始执行。同步检查点算法的优点是每个进程只需保存最近时刻的检查点信息，空间开销较小，且在恢复的时候没有多米诺效应。其缺点是，在建立检查点时，各进程间的同步使程序运行中止时间较长，且牺牲了分布式程序的自治性。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 同步检查点

- 3) Sync-and-Stop (SNS) 算法中有一个进程pc负责管理全局检查点建立过程。各进程的检查点建立过程如下：
 - a) pc向所有进程广播检查点开始报文Mb(第一次同步开始);
 - b) 任一个进程接收到报文Mb后停止运行, 并在自己所发送的报文全部到达接收者后向pc进程发送报文Ms1;
 - c) pc接收到所有进程发送的报文Ms1后, 即意味着第一次同步结束。pc向各进程广播报文Mchk, 第二次同步开始;
 - d) 任一个进程接收到报文Mchk后, 立即作局部检查点, 检查点建立完成之后向pc发送报文Ms2;

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖同步检查点

- 3) Sync-and-Stop (SNS) 算法中有一个进程pc负责管理全局检查点建立过程。各进程的检查点建立过程如下：
 - e) pc接收到所有进程发送的报文Ms2后，意味着第二次同步结束。pc向所有进程广播报文Me；
 - f) 各进程接收到报文Me后，删除旧的检查点，仅保留新的检查点，然后继续执行。SNS算法的恢复过程十分简单，只需回卷到检查点处继续执行。

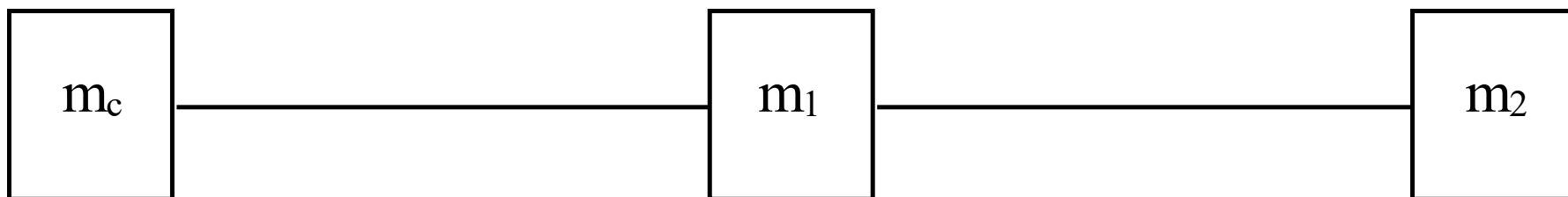
经过第一次同步之后，任何进程所发送的报文都已经被对应的接收进程接收到，任何进程之间不会存在孤儿报文，满足一致性的要求。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 同步检查点

4) Chandy-Lamport (CL) 算法:



机器 m_c 与 m_1 之间有通道直接相连，机器 m_c 与 m_1 之间可直接相互发送报文，机器 m_c 与 m_1 称为直接相连；机器 m_c 与 m_2 之间没有直接通道相连，机器 m_c 与 m_2 之间相互发送的报文要通过 m_1 转发，机器 m_c 与 m_2 称为间接相连。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 同步检查点

4) Chandy-Lamport (CL) 算法:

- a) 建立检查点的过程可由任一个进程pc发起，pc进程停止运行，并向与其所在机器直接相连的机器上的进程广播报文Mb，然后进程pc建立局部检查点；
- b) 进程p接收到报文Mb后，若进程p还未开始建立检查点，则进程p停止运行并立即向与其所在机器直接相连的机器上的进程广播报文Mb，然后进程p建立局部检查点；
- c) 进程p开始建立检查点后，若接收到其他进程发送的非检查点控制报文m，则保存报文m；

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 同步检查点

4) Chandy-Lamport (CL) 算法:

- d) 当进程p完成局部检查点的建立，并且接收到与其所在机器直接相连的机器上的所有进程发送的报文Mb后，进程p向pc进程发送报文Ms；
- e) 当进程pc接收到所有进程发送的报文Ms后，pc进程向所有进程发送报文Me，并删除本进程旧的检查点，进程pc继续执行；
- f) 其他进程p接收到报文Me后，删除本进程旧的检查点，继续执行。

在恢复过程中，CL算法在回卷到当前检查点重新执行的同时还必须重发过程(3)中保存的报文m。与SNS算法相比，CL算法减少了两次全局同步的开销。CL算法的缺点是其控制报文的数目与机器间的拓扑结构有关。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖混合检查点

➤其基本思想是在一个较长的时间段中使用同步检查点，而在较短的时间段内使用异步检查点。也就是说，在一个同步时间段里，会有若干个异步时间段。因此，我们可以有一个可以控制的回卷，从而保证不会在建立检查点的过程中引入过多的开销。

➤例如：准同步检查点。这个方法允许每个进程异步地设置检查点，从而保证了进程的独立性。同时，对恢复线的扩展采用发起通信的检查点协调方法，从而可以限制恢复过程中回卷的传播。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志

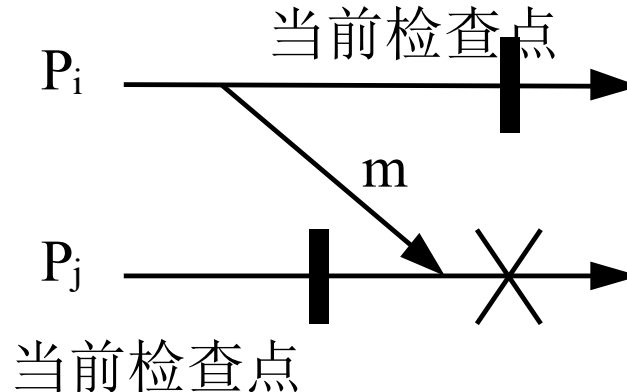
➤ 报文日志：为了减少回卷时撤销的计算工作量，所有接收的和发送的报文都可以记录下来。前者叫做接收者日志，后者叫做发送者日志。

➤ 当 P_j 的检查点被恢复到一个没有孤儿报文，而且所有要发送的报文都已经发送的一致状态的时候，可以用 P_j 的接收者日志减少回卷工作量，即只需要将 P_j 所收到的报文重新向 P_j 发送一遍即可。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志

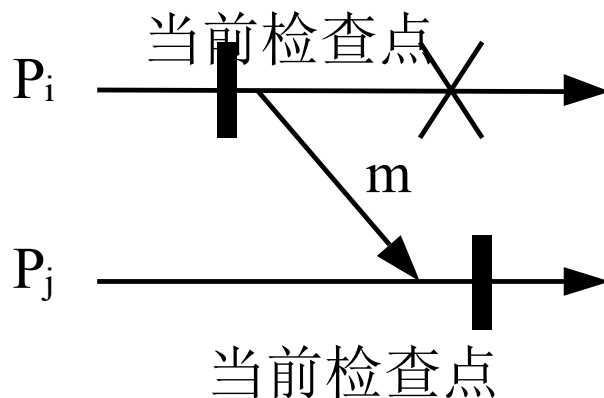


➤ 如果进程 P_j 记录了报文 m 的接收者日志，那么 P_i 和 P_j 的当前检查点集合就可以看作是一致的。一旦由于 P_j 由于失效回卷到当前检查点重新执行的时候，报文 m 就可以通过 P_j 的接收者日志重新发送给进程 P_j ，不会引起进程 P_i 的任何回卷。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志



➤如果 P_i 在发送完报文 m 后失效，那么当进程 P_i 恢复到当前检查点后，它会根据发送者日志的纪录知道曾经发送过报文 m ，这样就没有必要再发送一次了。如果接收者 P_j 失效，而且没有接收者日志，它仍然可以根据从发送者日志中得到的报文正确恢复。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志

➤ Alvisi和Marzullo的报文日志方案

- **报文格式：**每个报文 m 包含有一个报文头用于存放重发这个报文和正确处理这个报文所必需的一些信息，例如，该报文的发送者和接收者，用于识别报文重复的序列号，另外还有一个传输号用于决定何时该报文需要传递给接收进程。

- **坚固的报文：**如果一个报文不再会丢失，则称这个报文是坚固的，例如当一个报文已经被写入到坚固存储器中，就可以说这个报文是坚固的。坚固的报文可以重新发送给失效后重新恢复的进程。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志

➤ Alvisi和Marzullo的报文日志方案

• **进程集合 $DEP(m)$ 的组成：**一个报文 m 对应于一个进程集合 $DEP(m)$ ，该集合包含了与报文 m 传输有关的所有进程。

- 1) $DEP(m)$ 包含了所有接收该报文 m 的进程；
- 2) 如果另外一个报文 m' 和报文 m 有因果依赖性关系，并且 m' 是传递给进程 Q 的，那么 $DEP(m)$ 集合中应该包含进程 Q 。

因果依赖性关系：如果 m' 和 m 是由同一个进程发送的，并且 m 的发送先于 m' 的发送，则说 m' 和报文 m 的传输有因果依赖性关系。同样地，如果 m'' 和 m' 有因果依赖性关系， m' 和 m 有因果依赖性关系，则说 m'' 和 m 有因果依赖性关系。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志

➤ Alvisi和Marzullo的报文日志方案

- **进程集合COPY(m) 的组成：** 如果一个进程有报文m的一个拷贝，但是这个拷贝还没有写入到它的局部坚固存储器中的话，该进程就属于集合COPY(m)。当一个进程发送了报文m，它也属于集合COPY(m)。值得注意的是COPY(m)集合中所包含的进程是那些拥有报文m的拷贝，并且在出现失效的时候，能够重新传输报文m的进程。当COPY(m)中的所有进程都失效时，显然报文m就不能被重新传输。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志

➤ Alvisi和Marzullo的报文日志方案

- **孤儿进程**：假设在一个分布式系统中，某个或某些进程失效了， Q 是一个存活下来的进程。有一个报文 m ，如果进程 Q 是集合 $DEP(m)$ 中的一个元素，而集合 $COPY(m)$ 中的所有进程都失效了，那么 Q 就是一个孤儿进程。也就是说，当一个进程依赖于报文 m ，但是无法向该进程重发报文 m ，该进程就是一个孤儿进程。

避免孤儿进程的出现：需要确保当集合 $COPY(m)$ 中的进程都失效的时候， $DEP(m)$ 中没有存活的进程。也就是说， $DEP(m)$ 中的进程也必须全部失效。这可以通过如下的强制方式得以实现，当一个进程成为 $DEP(m)$ 中的一个成员的时候，我们强制它也成为 $COPY(m)$ 的一个成员。也就是说，当一个进程依赖于报文 m 的传输的时候，它将保持报文 m 的一个副本。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志

➤ Alvisi和Marzullo的报文日志方案

- **悲观的日志协议：**每一个非坚固的报文 m ，确保最多只有一个进程依赖于报文 m 。也就是说，对于每一个非坚定的报文 m ，悲观的日志协议确保该报文最多只传给了一个进程。值得注意的是，一旦一个非坚定的报文 m 传递给了进程 P ， P 就成为集合 $COPY(m)$ 的一个成员。

最坏的情况是在报文 m 写入到坚固存储器之前，进程 P 失效了。因为在悲观的日志协议下，在报文 m 写入到坚固存储器之前，不允许 P 发送任何报文，所以不会有其他进程依赖于报文 m ，也就不会有重发报文 m 的可能性。所以，使用悲观的日志协议避免了孤儿进程的问题。

第七章 分布式系统中容错技术

7.4 分布式检查点算法

❖ 报文日志

➤ Alvisi和Marzullo的报文日志方案

- **乐观的日志协议：**在乐观的日志协议下，实际的工作是在失效发生之后进行的。假定对某个报文 m 来说，如果集合 $COPY(m)$ 中的每个进程都失效了， $DEP(m)$ 中的每个孤儿进程一直要回卷到以前的某个状态，在这个状态下，该进程不再是集合 $DEP(m)$ 中的一个成员。很明显，乐观的日志协议需要保持追踪依赖性关系，从而使得它的实现变得复杂。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖故障-停止型故障与拜占庭式故障：在故障-停止模型中，我们假定一个处理器将停止工作并且不再恢复运转。在其他情况下，一个故障可能做出破坏性的行为。例如，一个有故障的处理器可能会向不同的处理器发送不同的令它们费解的报文，这种故障叫做随意性故障(arbitrary fault)，或拜占庭式故障(Byzantine fault)。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖恢复中的设计问题

➤进程组的内部结构

- 1) 平面结构：所有的进程是平等的，没有任何一个进程处于领导地位，任何决定都是共同做出的。
- 2) 层次结构：在最简单的层次结构的进程组中，一个进程是协调者，其他所有的进程为工作者。无论是进程组外的一个顾客，还是进程组中的一个工作者向进程组发出一个工作请求，由协调者决定哪一个工作者最适合这项工作，并且将此工作请求转交给它。

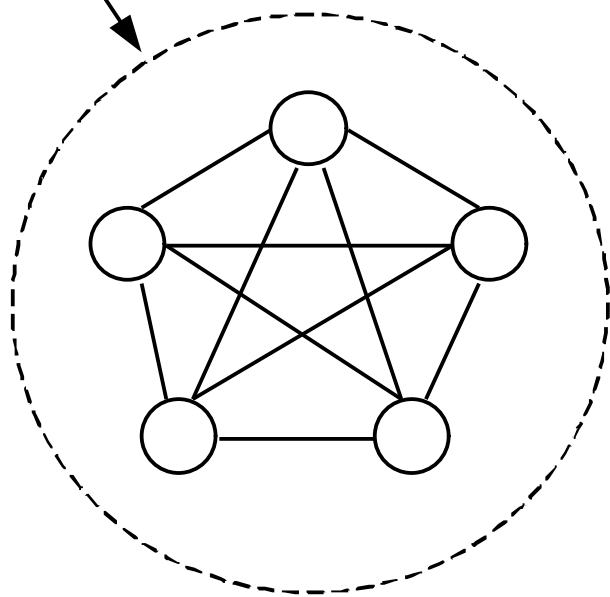
第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖ 恢复中的设计问题

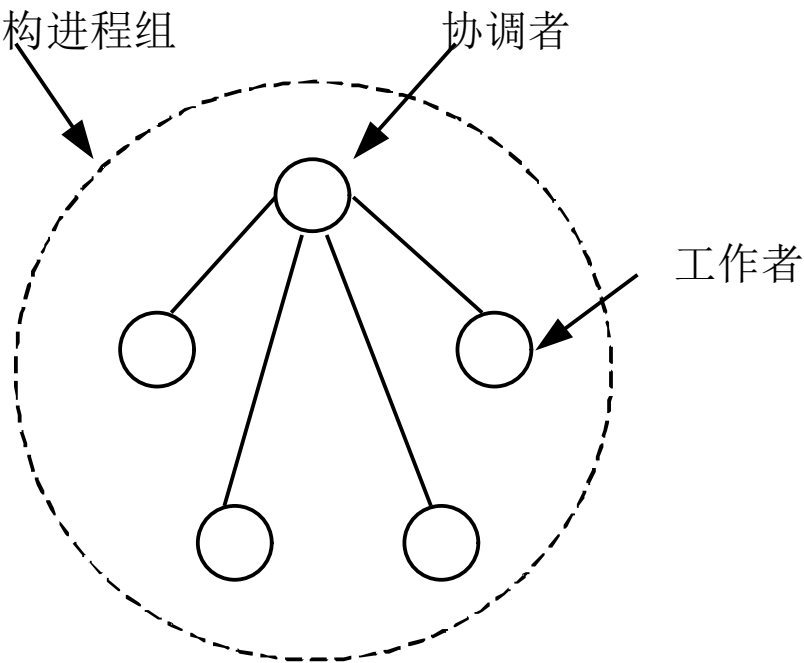
➤ 进程组的内部结构

平面结构进程组



(a) 平面进程组中的通信模式

分层结构进程组



(b) 分层进程组中的通信模式

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖恢复中的设计问题

➤**进程组管理**：对于进程组通信来说，需要一定的办法进行进程组的创建和删除，同时还要允许进程能加入到一个进程组中去以及允许一个进程离开一个进程组。

- 1) **集中式管理**：设置一个进程组服务员，所有的服务请求都发送给进程组服务员。进程组服务员使用一个数据库来保存所有进程组的信息和一个进程组中所有成员的有关信息。
- 2) **分布式管理**：另外一个进程组管理的办法是采用分布式的方式管理进程组。例如当一个组外的进程要加入到这个进程组时，它向这个进程组中的所有成员发送一个报文，声明自己要加入到这个进程组。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖ 错误屏蔽和进程复制

➤ **利用进程组屏蔽错误：**进程组是构造容错系统问题的一部分。特别是，如果用相同的进程来构成一个进程组，那么就能够屏蔽进程组中一个或多个出错的进程。也就是说，我们可以用多个相同的进程构造一个进程组，用这个容错的进程组取代相对脆弱的单个进程。

➤ **利用进程组进行容错需要多少个进程副本：**

- 1) **故障—停止型故障：**如果进程组中有 $k+1$ 个进程，那么就足以提供 k 故障容错。
- 2) **拜占庭式故障：**那么为了取得 k 故障容错，那么至少需要 $2k+1$ 个进程。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤分布式一致算法基本目标：是使得对于某个问题来说，所有非出错的进程能够达成一致，并且能够在有限的步骤内达成一致。

		接收者			
		P_1^*	P_2	P_3	P_4
发送者	P_1^*	—	—	—	—
	P_2	T	T	T	T
	P_3	F	F	F	F
	P_4	T	T	T	T
决策		—	T	T	T

故障—停止型故障

		接收者			
		P_1^*	P_2	P_3	P_4
发送者	P_1^*	T	F	F	T
	P_2	T	T	T	T
	P_3	F	F	F	F
	P_4	T	T	T	T
决策		—	T	F	T

拜占庭式故障

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤分布式一致算法的正确性条件：

- (1) 一致性。所有正确的进程取得一致的结果，而且是最后的结果；
- (2) 合法性。所有进程同意的结果必须来自某个正确的进程的输入；
- (3) 有限性。每个进程在有限的步骤内取得一致的结果。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖ 容错系统中的一致性算法

➤ 交互一致性算法

交互一致性：系统中的每个非出错进程都使用来自进程 P_i 的同样的值来进行决策。这样，一般的一致性问题的系统变为系统中的进程都同意一个特殊的进程(比如 P_0)的值。确切地说：

- (1) 所有非出错进程都使用进程 P_0 的同样的值 v_0 。
- (2) 如果发送进程 P_0 是非出错的，那么所有非出错进程都使用 P_0 发送的值。

结论：在有 k 个出错节点的情况下，只有进程的总数至少为 $3k+1$ 时才能获得一致。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤LAMPART交互一致性算法IC:

IC(m), $m < k$, 开始时 $m=0$, $S=\{\}$ 。

- 1) 发送者将它的值和发送者列表 S 发送给其他的进程, 共 $(n-1-m)$ 个 ;
- 2) 设 v_i 是进程 P_i 从发送者接收到的值或者是如果没有收到值时使用的缺省值。在IC($m+1 \neq k$)时进程 P_i 作为发送者, 将结果 v_i 和发送者列表 $S \cup \{P_i\}$ 发送给其他不在发送者列表中的 $n-2-m$ 个进程。如果 $m+1=k$, 则调用IC(k) ;
- 3) 对每个进程 P_i , 设 v_j 是进程 P_j 接收到的值(但是由 P_j 转发给 P_i)。节点使用值 $\text{majority}(v_j)$, $j \in S$;

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤LAMPART交互一致性算法IC:

其中, $IC(k)$:

- 1) 发送者将它的值发送给其他 $n-k-1$ 个进程;
- 2) 每个进程使用从接收者收到的值, 或者, 如果它没有收到任何值, 就使用缺省值。

结论: 上述算法中, 被交换的报文总数为 $(n-1)(n-2)\dots(n-k-1)$, 其复杂度为 $O(n^k)$, k 可以是 $(n-1)/3$ 。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤LAMPART交互一致性算法IC:

实例：具有7个进程的例子， P_i ， $0 \leq i \leq 6$ 。假定 $k=2$ ，即最多有2个故障， $n=7$ 。设 P_0 是初始发送者，发送值为1。不妨设 P_5 和 P_6 是出错进程，它向其他进程发送不确定的值。在这个例子中共需要 $k+1=3$ 轮信息交换：

- 1) P_0 将 $\{V_0, S\} = \{1, \{P_0\}\}$ 发送给进程 P_1, P_2, \dots, P_6 。
- 2) P_1 至 P_6 中每个进程都接收到报文 $\{1, \{P_0\}\}$ ，它们将所收到的报文转发给除了开始的发送者和它本身之外的所有其他的进程。例如 P_1 向 P_2 至 P_6 发送报文 $\{1, \{P_0, P_1\}\}$ 。它分别从 P_2 至 P_6 处接收到报文 $\{1, \{P_0, P_2\}\}$ 、 $\{1, \{P_0, P_3\}\}$ 、 $\{1, \{P_0, P_4\}\}$ 、 $\{d, \{P_0, P_5\}\}$ 、 $\{d, \{P_0, P_6\}\}$ 。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤LAMPORT交互一致性算法IC:

- 3) 在第三轮中, P_1 将报文 $\{1, \{P_0, P_2\}\}$ 以 $\{1, \{P_0, P_2, P_1\}\}$ 的形式分别发送给进程 P_3 、 P_4 、 P_5 、 P_6 , 将报文 $\{1, \{P_0, P_3\}\}$ 以 $\{1, \{P_0, P_3, P_1\}\}$ 的形式分别发送给进程 P_2 、 P_4 、 P_5 、 P_6 , 将报文 $\{1, \{P_0, P_4\}\}$ 以 $\{1, \{P_0, P_4, P_1\}\}$ 的形式分别发送给进程 P_2 、 P_3 、 P_5 、 P_6 , 将报文 $\{d, \{P_0, P_5\}\}$ 以 $\{d, \{P_0, P_5, P_1\}\}$ 的形式分别发送给进程 P_2 、 P_3 、 P_4 、 P_6 , 将报文 $\{d, \{P_0, P_6\}\}$ 以 $\{d, \{P_0, P_6, P_1\}\}$ 的形式分别发送给进程 P_2 、 P_3 、 P_4 、 P_5 。第三轮递归结束, 返回第二轮。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤LAMPART交互一致性算法IC:

- 4) 进程 P_1 收到报文 $\{1, \{P_0, P_2, P_3\}\}$ 、 $\{1, \{P_0, P_2, P_4\}\}$ 、 $\{d, \{P_0, P_2, P_5\}\}$ 、 $\{d, \{P_0, P_2, P_6\}\}$ ，连同在第二轮中收到的报文 $\{1, \{P_0, P_2\}\}$ ，将这5个报文中的值执行 $\text{majority}(1, 1, d, d, 1)=1$ ，得到的这个新值1作为报文 $\{1, \{P_0, P_2\}\}$ 的新值，记为新报文 $\{1, \{P_0, P_2\}\}'$ 。同样的道理，得到新报文 $\{1, \{P_0, P_3\}\}'$ 、 $\{1, \{P_0, P_4\}\}'$ 、 $\{1, \{P_0, P_5\}\}'$ 、 $\{1, \{P_0, P_6\}\}'$ 。
- 5) P_1 对6个报文 $\{1, \{P_0, P_2\}\}'$ 、 $\{1, \{P_0, P_3\}\}'$ 、 $\{1, \{P_0, P_4\}\}'$ 、 $\{1, \{P_0, P_5\}\}'$ 、 $\{1, \{P_0, P_6\}\}'$ 、 $\{1, \{P_0, P_2\}\}$ （第二轮递归结束）中的值执行 $\text{majority}(1, 1, 1, 1, 1, 1)=1$ ，这个新值是 P_1 所确信的最终值。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖ 容错系统中的一致性算法

➤ 交互一致性算法的扩展：

可以通过对每个发送者都重复同样的协议将交互一致性扩展到多个发送者的情况。扩展的方法：在第 k 轮收到的值将被发送到所有的其他节点，而不仅仅是那些没有被发送过的节点。

实例：假定有4位将军，其中有一个叛徒。将军之间相互通报自己的人数，忠诚的将军说实话，叛徒恶意地将不同的值通报给其他将军。不失一般性，设将军 P_1 是叛徒，它分别向其他3位将军谎报自己军队的人数是 x 、 y 、 z ，将军 P_2 向所有的将军通报自己的人数是4000，将军 P_3 向所有的将军通报自己的人数是5000，将军 P_4 向所有的将军通报自己的人数是6000。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤交互一致性算法的扩展：

一个取得一致的算法

	P_1^*	P_2	P_3	P_4
第 1 轮	(-,4k,5k,6k)	(x,-,5k,6k)	(y,4k,-,6k)	(z,4k,5k,-)
第 2 轮	(x -,5k,6k)	(-,a,b,c)	(-,d,e,f)	(-,g,h,i)
	(y,4k,-,6k)	(y,4k,-,6k)	(x,-,5k,6k)	(x,-,5k,6k)
	(z,4k,5k,-)	(z,4k,5k,-)	(z,4k,5k,-)	(y,4k,-,6k)
结果向 量	(未知,4k,5k,6k)	(未知,4k,5k,6k)	(未知,4k,5k,6k)	(未知,4k,5k,6k)

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖ 容错系统中的一致性算法

➤ 交互一致性算法的扩展：

$n=3, k=1$ 时不能取得一致

	P_1^*	P_2	P_3
第 1 轮	$(-, 4k, 5k)$	$(x, -, 5k)$	$(y, 4k, -)$
第 2 轮	$(x, -, 5k)$ $(y, 4k, -)$	$(-, a, b)$ $(y, 4k, -)$	$(-, d, e)$ $(x, -, 5k)$
结果向量	(未知, 4k, 5k)	(未知, 4k, 未知)	(未知, 未知, 5k)

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤一致性协议对系统的要求：

同步系统与异步系统：如果所有的处理器都是在可以预料的速度下运行，那么这个系统就是同步的。确切地说，处理器是同步的当且仅当存在一个常量 $s \geq 1$ ，每当任何一个处理器运行了 $s+1$ 步，其他的所有处理器都至少运行了1步；否则这个系统就是异步的。

用卡诺图描述一致性对系统的要求：

定义布尔变量

- (1) 系统是同步的($A=1$)还是异步的($A=0$)。
- (2) 通信延迟是有限的($B=1$)还是无限的($B=0$)。
- (3) 报文是有序的($C=1$)还是无序的($C=0$)。
- (4) 传输机制是广播的($D=1$)还是点到点的($D=0$)。

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤一致性协议对系统的要求：

		CD			
		00	01	11	10
AB	00	0	0	1	0
	01	0	0	1	0
	11	1	1	1	1
	10	0	0	1	1

第七章 分布式系统中容错技术

7.5 拜占庭故障的恢复

❖容错系统中的一致性算法

➤一致性协议对系统的要求：

可以得到一致性对系统的要求为：

$$AB+AC+CD=True$$

即在下面三种情况下，系统满足一致性协议的要求：

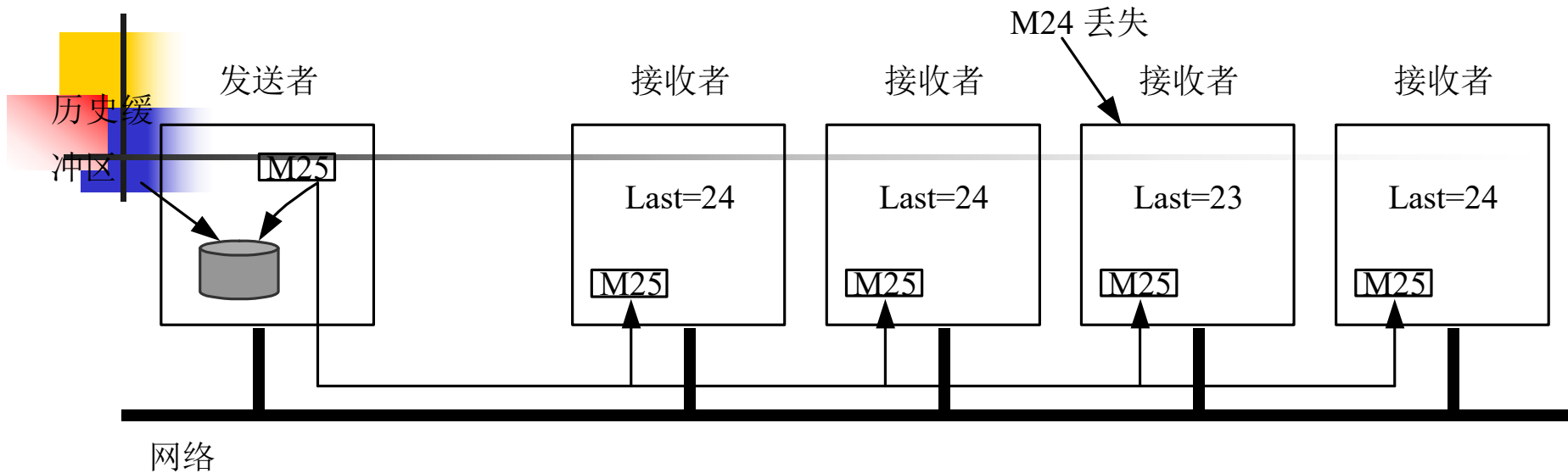
- (1) ($AB=1$)，处理器是同步的，通信延迟是有限的；
- (2) ($AC=1$)，处理器是同步的，报文是有序的；
- (3) ($CD=1$)，报文是有序的，传输机制是广播。

第七章 分布式系统中容错技术

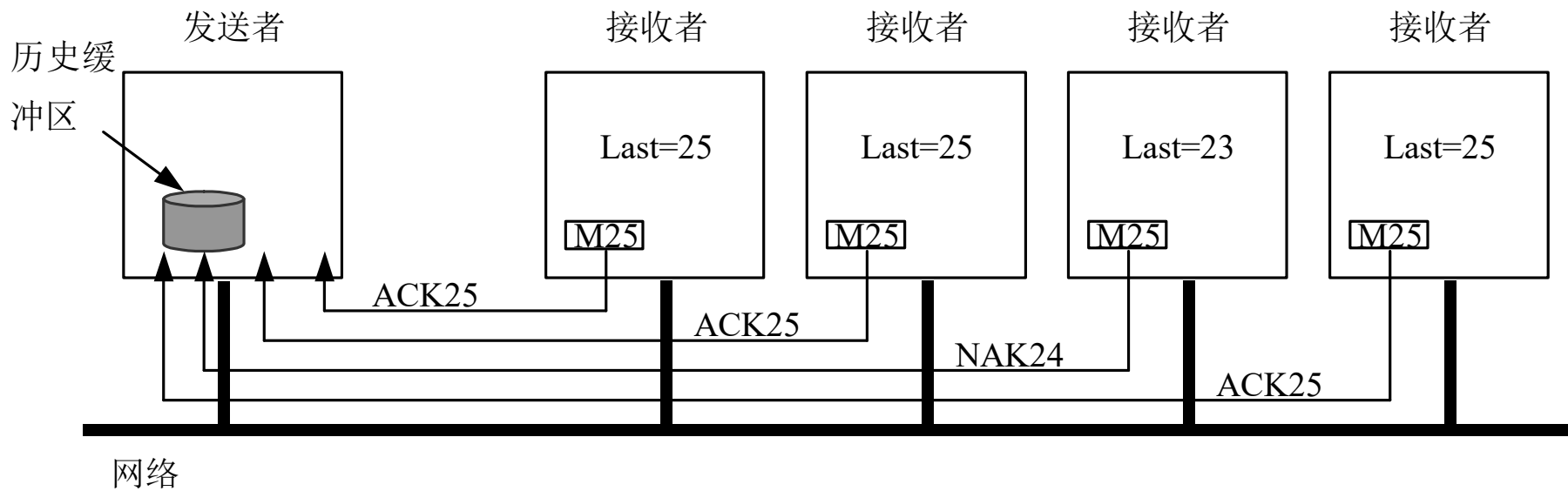
7.6 可靠的组通信

❖ 基本的可靠组播方案

所谓可靠的组播通信，就是要求发送给某个进程组的报文必须确保传送到进程组中的每一个成员。



(a)



(b)

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖可靠的组播通信中的可扩充性

➤**简单方案：**接收者收到组播报文后不向发送者发送应答报文，只有在接收者发现丢失一个报文后，才向发送者发送一个反馈报文，指出某个报文丢失了。

问题1：多个接收者丢失报文时，仍然会出现反馈报文拥塞的情况；

问题2：在理论上讲，发送者必须在其历史缓冲区中长时间保存它发送的每一个广播报文。

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖可靠的组播通信中的可扩充性

➤非层级式反馈控制

- 1) 成功接收到组播报文的接收者不向发送者发送反馈报文，只有在接收者发现有报文丢失的时候才会发送反馈报文，也就是说只有否定应答被当作反馈报文返回给发送者。
- 2) 如果一个接收者发现它丢失了报文 m ，它不仅将反馈报文发送给发送者，还将反馈报文组播给组内的所有其他进程。
- 3) 其他同样丢失了报文 m 的接收者收到这样的反馈报文后，就不会发送同样的反馈报文。这样就收者只会收到一个反馈报文，从而只重复组播报文 m 一次。

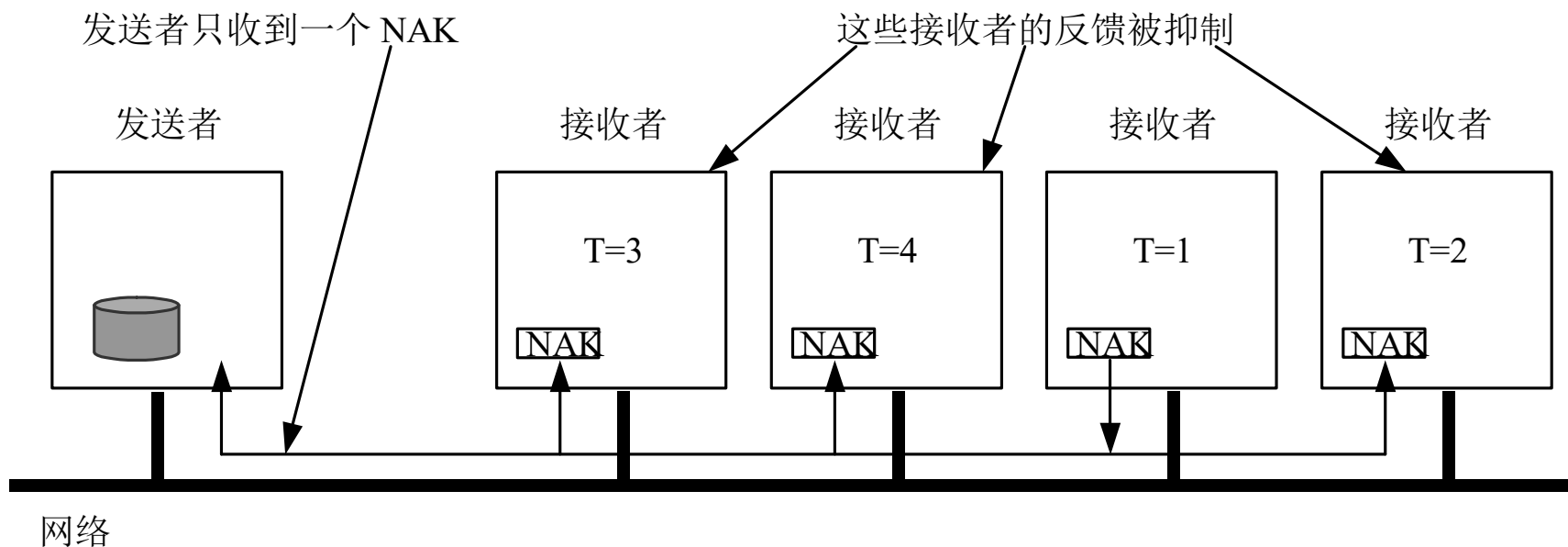
第七章 分布式系统中容错技术

7.6 可靠的组通信

❖可靠的组播通信中的可扩充性

➤非层级式反馈控制

实现：



第七章 分布式系统中容错技术

7.6 可靠的组通信

❖可靠的组播通信中的可扩充性

➤非层级式反馈控制

该实现方法的问题：

问题1：为了确保只有一个反馈报文返回给发送者，需要对所有接收者的反馈报文进行精确地调度，即给每个接收者精确地设置延迟时间，在一个广域网中，精确地设置延时是十分困难的。

问题2：对反馈报文进行组播的时候会对成功地接收到组播报文的接收进程产生干扰，也就是说，其他成功接收到组播报文的接收进程被迫接收和处理一些对他们来说毫无用处的报文。

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖可靠的组播通信中的可扩充性

➤层级式反馈控制

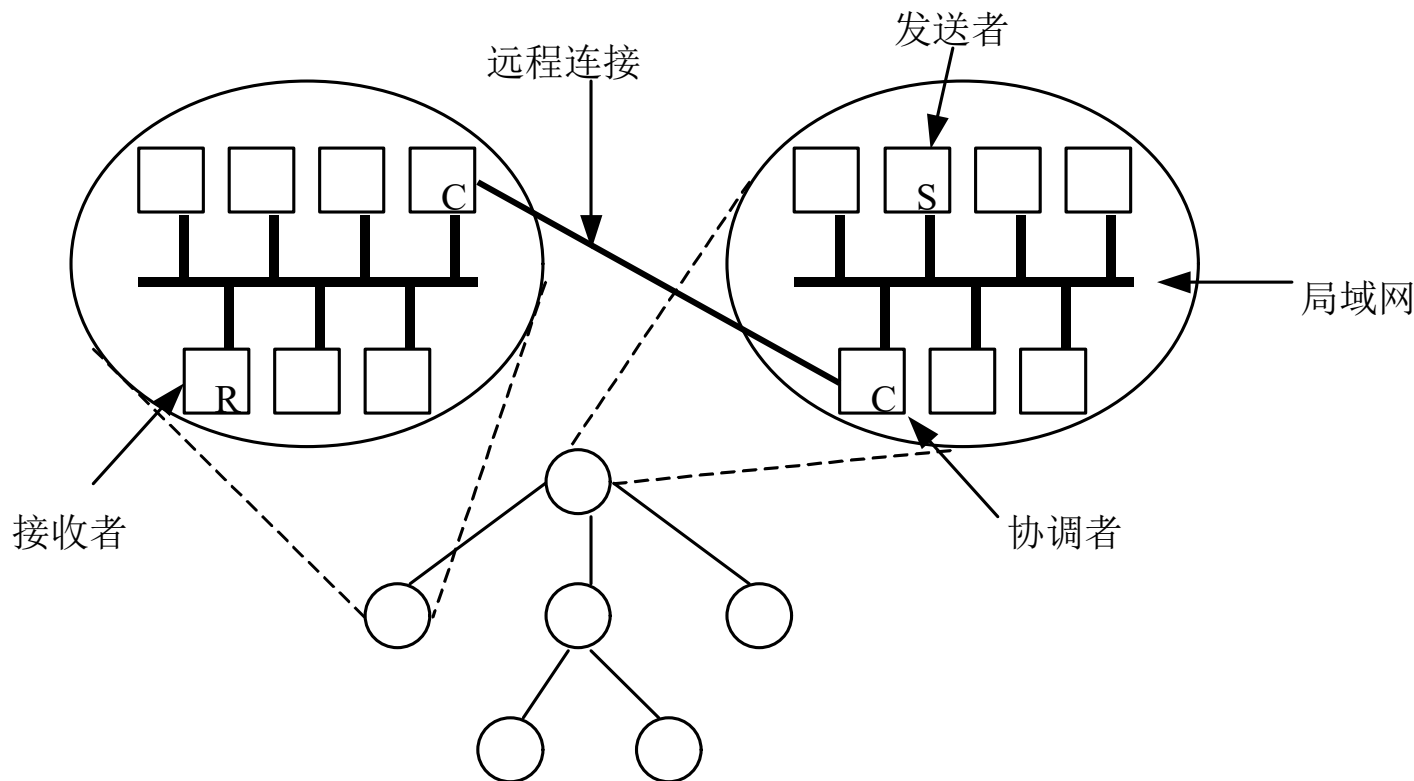
- 在层级式可靠的组播方案中，进程组被划分成多个子进程组，这些子进程组被组织成树型结构，包含有发送者的子进程组是树的根。在一个子进程组内，可以采用任何一种可靠的组播技术。
- 每个子进程组任命一个本地协调者，负责处理该子进程组内的接收者的重发请求。本地协调者有自己的历史缓冲区。如果协调者本身丢失了报文 m ，它会要求它的父亲子进程组的协调者重发报文 m 。

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖ 可靠的组播通信中的可扩充性

➤ 层级式反馈控制



第七章 分布式系统中容错技术

7.6 可靠的组通信

❖ 原子组播

原子组播：必须保证一个组播报文要么被进程组内所有的进程接收，要么没有一个进程接收该报文。另外，还要求所有的组播报文应该以同样的顺序被进程组中所有的进程接收。

➤ 虚同步(virtual synchrony)

- **组视图：**原子组播的实质是一个组播报文 m 唯一地和需要收到这个报文的一组进程相联系。需要收到这个报文的进程表是包含在这个进程组中所有进程的集合的一个视图，这个视图成为组视图(group view)。
- **虚同步：**一个进程在发送报文 m 的进程失效的时候，进程组的其他进程要么都收到报文 m ，要么所有的其他进程都忽略报文 m ，具有这种性质的可靠的组播被称为是虚同步的组播。

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖ 原子组播

➤ 虚同步(virtual synchrony)

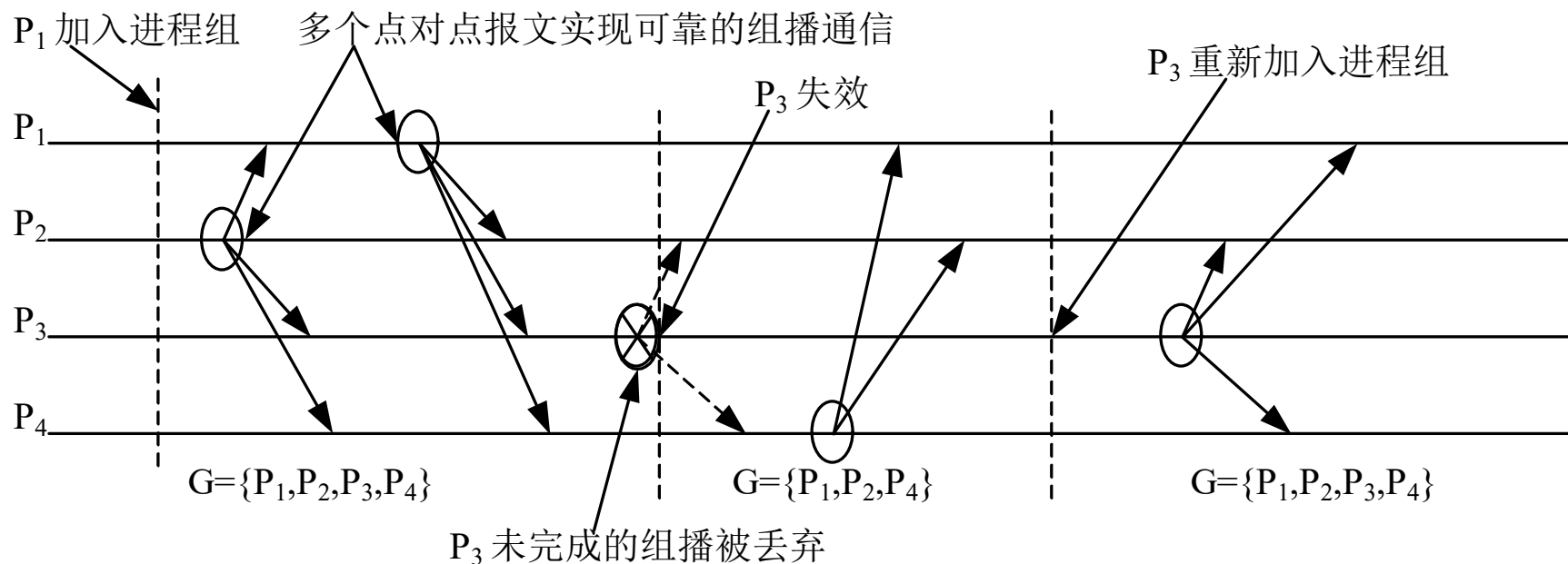
虚同步的原理其实就是每一个组播通信发生在视图改变之间。也就是说，一个视图改变扮演者一个栅栏的作用，组播不能跨越栅栏。任何一个组播通信都是发生在一个视图改变之后，并且在下一个视图改变起效之前完成。

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖ 原子组播

➤ 虚同步(virtual synchrony)



第七章 分布式系统中容错技术

7.6 可靠的组通信

❖ 原子组播

➤ 报文顺序

按照组播报文的排序规则，组播可以分为如下四类：

- 1) 无序组播：可靠的无序组播是一个虚同步组播，它不保证不同的接收进程对不同的组播报文的接收顺序。

进程 P_1	进程 P_2	进程 P_3
send m1	receive m1	receive m2
send m2	receive m2	receive m1

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖ 原子组播

➤ 报文顺序

按照组播报文的排序规则，组播可以分为如下四类：

- 2) FIFO顺序组播：通信层保证同一个进程发送的不同的组播报文，不同接收进程会按报文的发送顺序接收。但是不同的进程发送的不同组播报文，不同的接收进程可能会有不同的接收顺序。

进程 P ₁	进程 P ₂	进程 P ₃	进程 P ₄
send m1	receive m1	receive m3	send m3
send m2	receive m3	receive m1	send m4
	receive m2	receive m2	
	receive m4	receive m4	

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖ 原子组播

➤ 报文顺序

按照组播报文的排序规则，组播可以分为如下四类：

- 3) 因果关系顺序组播：所有接收进程必须按报文固有的因果关系顺序接收。例如报文m1按因果关系排在报文m2之前，那么所有的接收进程必须先接收报文m1后接收报文m2，不管m1和m2是否是由同一个发送者发送的。如果报文之间不具有因果关系的先后顺序，不同的接收进程可以有不同的接收顺序。
- 4) 在全局顺序组播中，不同的接收者必须按完全相同的顺序接收所有的报文。无论是无序组播、FIFO顺序组播还是因果关系顺序组播都可以附加一个全局顺序的约束条件。

第七章 分布式系统中容错技术

7.6 可靠的组通信

❖ 原子组播

➤ 虚同步的实现

要解决的主要问题：发送给视图G的所有报文在组成员发生改变之前都要到达G中所有非出错的进程。由于向G发送报文m的发送者可能在完成发送之前失效，所以有可能存在进程组中某些进程接收到报文m，而另一些进程没有接收到报文的情况。

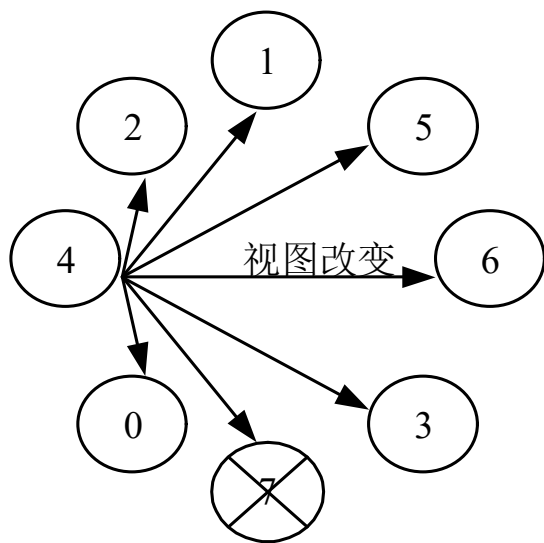
解决问题的办法：让G中每个获得报文m的进程保存m，直到该进程确信组中的其他进程都已经收到报文m。如果m已经被进程组中的所有进程接收到，则称m是坚定的(stable)。只有坚定的报文才能被进程实际接收。为了确保坚定性，选择G中的任何一个可操作的进程要求它将报文m发送给所有其他的进程。

第七章 分布式系统中容错技术

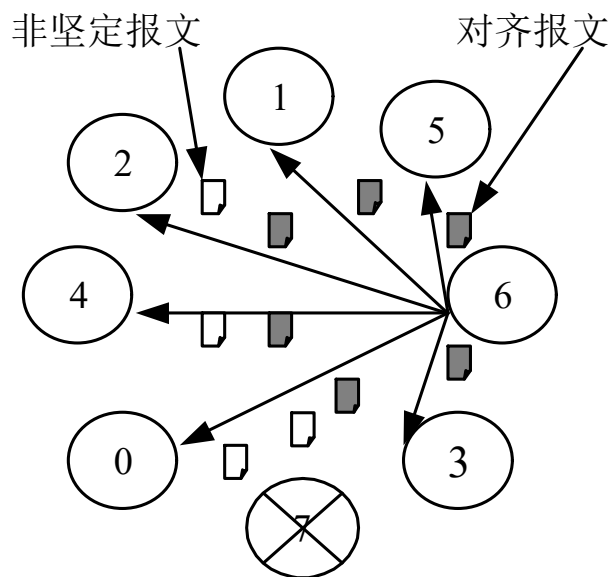
7.6 可靠的组通信

❖ 原子组播

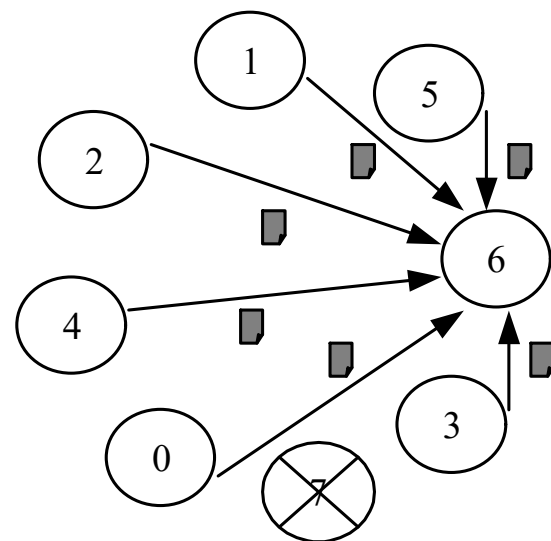
➤ 虚同步的实现 实例：



(a) 进程 4 向所有进程发送
一个改变视图的报文



(b) 收到改变视图报文的进程先发送
非坚定报文，再发送对齐报文



(c) 收到所有对齐报文的进程
可以安装新的视图

第八章 分布式数据管理

8.1 一致性模型

一致性模型是进程和数据存储之间的一个基本约定。也就是说，如果进程对数据的访问遵守特定的规则，那么数据存储就能够正确地进行。

❖ 严格一致性

对于严格一致性模型来说，对一个数据项所进行的任何读操作所返回的值总是对该数据项最近一次进行写操作的结果。

$P_1:$ $W_1(x)a$

$P_2:$ $R_2(x)a$

(a) 符合严格一致性

$P_1:$ $W_1(x)a$

$P_2:$ $R_2(x)NIL$ $R_2(x)a$

(b) 不符合严格一致性

第八章 分布式数据管理

8.1 一致性模型

❖ 顺序一致性和可线性化一致性

➤ 所有进程对数据项的所有操作可以认为是按照某个顺序进行的，任何进程对这个顺序的观点是一样的。当然这个顺序不一定是物理意义上顺序。

P₁: W₁(x)a

P₂: W₂(x)b

P₃: R₃(x)b R₃(x)a

P₄: R₄(x)b R₄(x)a

(a) 符合顺序一致性

P₁: W₁(x)a

P₂: W₂(x)b

P₃: R₃(x)b R₃(x)a

P₄: R₄(x)a R₄(x)b

(b) 不符合顺序一致性

第八章 分布式数据管理

8.1 一致性模型

❖ 相关一致性

具有潜在因果关系的所有写操作能够被所有的进程以相同的顺序看见，而并发的写操作可以被不同的进程以不同的顺序看见。

P ₁ :	W ₁ (x)a		W ₁ (x)c
P ₂ :		R ₂ (x)a	W ₂ (x)b
P ₃ :		R ₃ (x)a	R ₃ (x)c R ₃ (x)b
P ₄ :		R ₄ (x)a	R ₄ (x)b R ₄ (x)c

第八章 分布式数据管理

8.1 一致性模型

❖ 相关一致性

具有潜在因果关系的所有写操作能够被所有的进程以相同的顺序看见，而并发的写操作可以被不同的进程以不同的顺序看见。

P₁: W₁(x)a

P₂: R₂(x)a W₂(x)b

P₃: R₃(x)b R₃(x)a

P₄: R₄(x)a R₄(x)b

(a) 不符合相关一致性

P₁: W₁(x)a

P₂: W₂(x)b

P₃: R₃(x)b R₃(x)a

P₄: R₄(x)a R₄(x)b

(b) 符合相关一致性

第八章 分布式数据管理

8.1 一致性模型

❖ FIFO一致性

一个进程中的所有写操作能够以它在该进程中出现的顺序被所有其他进程看见。但是不同进程中的写操作在不同的进程看来具有不同的顺序。

P₁: W₁(x)a

P₂: R₂(x)a W₂(x)b W₂(x)c

P₃: R₃(x)b R₃(x)a R₃(x)c

P₄: R₄(x)a R₄(x)b R₄(x)c

第八章 分布式数据管理

8.1 一致性模型

❖ FIFO一致性

➤ 处理机一致性模型：在这种模型中，不仅要求一个进程中的所有写操作能够以它在该进程中出现的顺序被所有其他进程看见，还要求不同进程对同一个数据项的写操作，应该被所有进程以相同的顺序看见。

P₁: W₁(x)a

P₂: R₂(x)a W₂(x)b W₂(x)c

P₃: R₃(x)b R₃(x)c R₃(x)a

P₄: R₄(x)b R₄(x)c R₄(x)a

第八章 分布式数据管理

8.1 一致性模型

❖ 弱一致性

弱一致性模型是使用同步变量实现的一个一致性模型，他有如下三个特性：

- (1) 访问与数据存储有关的同步变量时，必须遵守顺序一致性模型；
- (2) 以前的所有写操作在每个副本上没有完成之前，对同步变量进行的操作不允许执行；
- (3) 以前的所有对同步变量的操作完成之前，对数据项进行任何读或写操作不允许执行。

第八章 分布式数据管理

8.1 一致性模型

❖ 弱一致性

$P_1: W_1(x)a \quad W_1(x)b \quad S$

$P_2: \quad \quad \quad R_2(x)a \quad R_2(x)b \quad S$

$P_3: \quad \quad \quad R_3(x)b \quad R_3(x)a \quad S$

(a) 符合弱一致性

$P_1: W_1(x)a \quad W_1(x)b \quad S$

$P_2: \quad \quad \quad S \quad R_2(x)a$

(b) 不符合弱一致性

第八章 分布式数据管理

8.1 一致性模型

❖ 释放一致性

释放一致性为用户提供了这两种同步操作：一种是acquire操作，它用于告诉系统调用进程要进入一个临界区，在这种情况下，其他进程对远程副本的修改应该传播到本地，并且要对本地的副本进行更新；另一种是release操作，用于告诉系统调用进程要离开一个临界区，在这种情况下，本地进程对本地数据副本的修改应该传播到所有的远程副本上。

释放一致性模型，必须遵守如下规则：

- (1) 前面的所有acquire操作完全成功完成以前，对共享数据的读写操作不能执行；
- (2) 在一个释放操作被允许执行之前，所有以前的读写操作必须完成；
- (3) 对同步变量的访问必须遵守FIFO一致性模型。

第八章 分布式数据管理

8.1 一致性模型

❖ 释放一致性

$P_1:$ $Acq_1(L)$ $W_1(x)a$ $W_1(x)b$ $Rel_1(L)$

$P_2:$ $Acq_2(L)$ $R_2(x)b$ $Rel_1(L)$

$P_3:$ $R_3(x)a$

第八章 分布式数据管理

8.1 一致性模型

❖进入一致性

进入一致性模型的数据存储必须符合以下条件：

- (1) 被保护数据的所有更新完成之前，一个进程对相应同步变量的acquire操作不能执行。
- (2) 如果有其他进程持有某个同步变量，即使是以非互斥的方式持有这个同步变量，一个进程以互斥的方式对这个同步变量的访问不能执行。
- (3) 当一个对同步变量的互斥方式访问完成以后，其他进程对这个同步变量的非互斥方式的访问也不能立即执行，除非在这个同步变量的所有者的访问完成之后。

第八章 分布式数据管理

8.1 一致性模型

❖ 进入一致性

$P_1:$ $Acq_1(Lx)$ $W_1(x)a$ $Acq_1(Ly)$ $W_1(y)b$ $Rel_1(Lx)$ $Rel_1(Ly)$

$P_2:$ $Acq_2(Lx)$ $R_2(x)a$ $R_2(y)NIL$

$P_3:$ $Acq_3(Ly)$ $R_3(y)b$

第八章 分布式数据管理

8.1 一致性模型

❖不同一致性模型的主要特性

一致性模型	特性
严格一致性	所有对共享数据的访问操作按绝对时间的顺序排序。
线性一致性	所有对共享数据的访问操作在每个进程看来是按照同样的顺序执行的，它们的是按照全局时间戳来排序的。
顺序一致性	所有对共享数据的访问操作在每个进程看来是按照同样的顺序执行的，但它们的排序不依赖于时间。
相关一致性	所有因果相关的对共享数据的访问操作在每个进程看来是按照同样的顺序执行的。
FIFO 一致性	一个进程内部的所有写操作以它们在该进程中执行顺序被所有其他进程看见。不同进程的写操作在不同进程看来具有不同的顺序。
弱一致性	只有在一个同步操作执行以后，才可以认为共享数据是一致的。
释放一致性	当从一个临界区中退出后，共享数据才获得一致。
进入一致性	当进入一个临界区时，与这个临界区对应的共享数据项获得一致

第八章 分布式数据管理

8.2 并发控制

❖ 并发控制的目标与事务处理

➤ **并发控制的目的**是在有多个用户的情况下允许每个用户像单个用户那样访问共享资源，多个用户同时访问时互相不干扰。并发控制要解决多个用户的活动之间的切换，保护一个用户的活动不受另一个用户的活动的影响，以及对相互依赖的若干活动进行同步等问题。

➤ **事务处理**：数据库中的事务处理(transaction)是施加在共享数据上的一组操作，这些操作是结合在一起的，被当作单个活动看待。

第八章 分布式数据管理

8.2 并发控制

❖ 并发控制的目标与事务处理

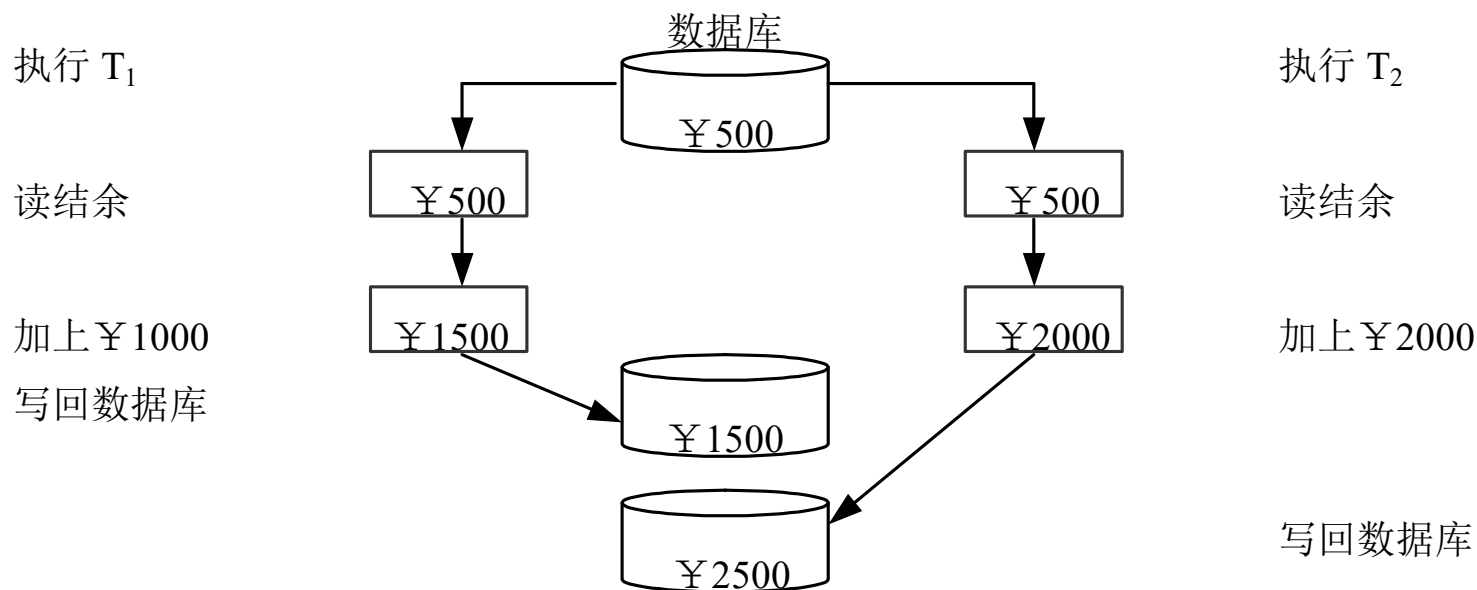
在无并发控制的情况下，两个并发的事务处理可能会相互干扰：

- (1) 丢失更新。多个事务处理同时对一个共同的数据对象进行写操作时，就会有丢失更新的现象发生，并且使数据库处于不一致状态。
- (2) 检索的不一致。检索的不一致发生在一个事务处理读取数据库中的某些数据对象，但是另一个事务处理对其中一些数据对象的修改还没有完成。

第八章 分布式数据管理

8.2 并发控制

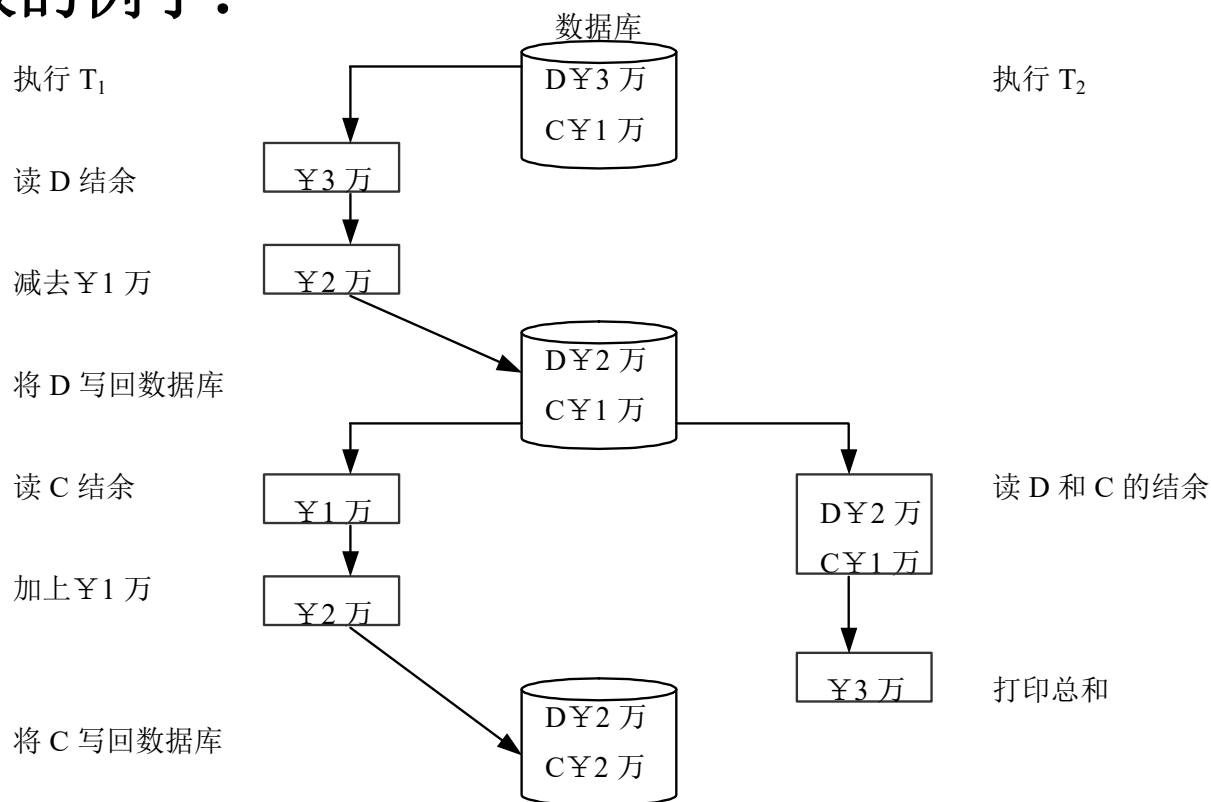
❖ 并发控制的目标与事务处理
丢失更新的例子：



第八章 分布式数据管理

8.2 并发控制

❖ 并发控制的目标与事务处理
检索不一致的例子：



第八章 分布式数据管理

8.2 并发控制

❖ 可串行化调度

➤ 并发控制正确性标准有两条：

- (1) 用户交给系统的每个事务处理最终将被执行，并最终得到完成；
- (2) 多个事务处理并发执行的结果和这多个事务处理串行执行的结果相同。

事务处理的例子：假设有三个账户(对象)A、B、C，最初各有存款¥200、¥100和¥50。现有两个事务处理 T_1 和 T_2 等待执行， T_1 是从A取出¥100存入B， T_2 是从B中取出¥50存入C。

第八章 分布式数据管理

8.2 并发控制

❖ 可串行化调度

T_1 :
begin

read(A)

得到账户A的存款数

read(B)

得到账户B的存款数

write(A)

(从A中减去¥100后)写入A

write(B)

(加入B中¥100后)写入B

end

T_2 :
begin

read(B)

得到账户B的存款数

read(C)

得到账户C的存款数

write(B)

(从B中减去¥50后)写入B

write(C)

(加入C中¥50后)写入C

end

第八章 分布式数据管理

8.2 并发控制

❖ 可串行化调度

➤ 并发事务处理执行中的两种现象：

- (1) 暂时不一致性。在 T_1 和 T_2 的第一次写操作之后而在第二次写操作之前，账户存款之和是不一致的。
- (2) 冲突。如果事务处理 T_2 被安排到 T_1 中的两次写操作之间运行，则账户存款总和为¥400，而不是¥350，它包含一个不一致状态。

暂时不一致性在所有顺序计算过程中是固有的，因此，在一个事务处理结束前一般不应该要求一致性。为了达到一致性，必须避免冲突。

第八章 分布式数据管理

8.2 并发控制

❖可串行化调度

➤可串行化调度：并发控制就是控制相互冲突操作的相对执行顺序，完成这种控制的算法也叫同步技术。我们希望使用这样的同步技术，它能使各个非冲突的操作交叠执行，以便进行各事务处理时具有最大的并发性。一组事务处理中各个操作的交叉次序叫做调度。如果一个调度给每个事务处理一个一致的状态观点，则这种调度叫做一致调度。一致调度的充分条件是这种调度执行的结果和所有事务处理串行执行的结果是一样的。满足这个条件的调度叫做可串行化调度或线性化调度。

第八章 分布式数据管理

8.2 并发控制

❖ 可串行化调度

➤ 调度的形式化描述:

$$T_1 = w_1(x), w_1(y), r_1(z)$$

$$T_2 = r_2(z), r_2(y), w_2(x)$$

$$T_3 = w_3(z), r_3(z), w_3(x)$$

$$L_1 = w_1(x), r_2(z), r_2(y), w_2(x), w_1(y), r_1(z), w_3(z), r_3(z), w_3(x)$$

$$L_2 = w_1(x), w_1(y), r_1(z), r_2(z), r_2(y), w_2(x), w_3(z), r_3(z), w_3(x)$$

$$L_3 = w_1(x), w_1(y), r_1(z), r_2(z), w_3(z), r_2(y), r_3(z), w_2(x), w_3(x)$$

第八章 分布式数据管理

8.2 并发控制

❖ 可串行化调度

➤ $r_j(x)$ 读自 $w_i(x)$:

如果调度L有两个操作 $w_i(x)$ 和 $r_j(x)$ ，我们说 $r_j(x)$ 读自 $w_i(x)$ 当且仅当

(1) $w_i(x) < r_j(x)$;

(2) 没有 $w_k(x)$ ，使得 $w_i(x) < w_k(x) < r_j(x)$ 。

第八章 分布式数据管理

8.2 并发控制

❖可串行化调度

$L_1 = w_1(x), r_2(z), r_2(y), w_2(x), w_1(y), r_1(z), w_3(z), r_3(z), w_3(x)$

$L_2 = w_1(x), w_1(y), r_1(z), r_2(z), r_2(y), w_2(x), w_3(z), r_3(z), w_3(x)$

$L_3 = w_1(x), w_1(y), r_1(z), r_2(z), w_3(z), r_2(y), r_3(z), w_2(x), w_3(x)$

➤两个调度等价：在事务处理系统上两个调度等价当且仅当

- (1) 两个调度中每个对应的读操作读自同一个写操作；
- (2) 两个调度中有同样的最终写。

在调度 L_2 还是 L_3 中， $r_1(z)$ 和 $r_2(z)$ 读自一个初始的 z ， $r_2(y)$ 读自 $w_1(y)$ ， $r_3(z)$ 读自 $w_3(z)$ 。另外 $w_3(x)$ 、 $w_1(y)$ 和 $w_3(z)$ 分别是对数据对象 x 、 y 和 z 的最终写。因此，调度 L_2 和 L_3 是等价的。

第八章 分布式数据管理

8.2 并发控制

❖可串行化调度

➤一组事务处理 $T = \{T_1, T_2, \dots, T_n\}$ ，对于一个给定调度 L ，串行化图定义如下：对某个 x ，如果 $r_i(x) < w_j(x)$ 、 $w_i(x) < r_j(x)$ ，或者 $w_i(x) < w_j(x)$ ，则有一条从 T_i 到 T_j 的边。如果从 T_i 到 T_j 有一条路径存在，则可以删除从 T_i 到 T_j 的边。

第八章 分布式数据管理

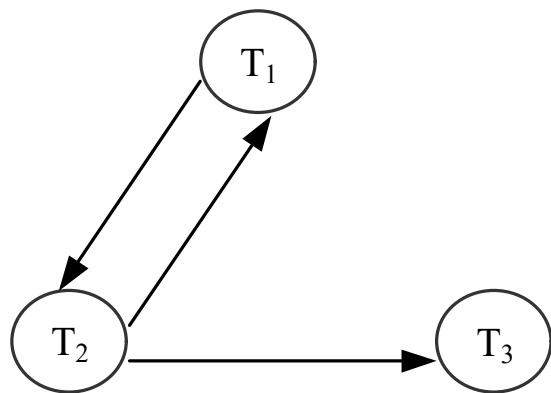
8.2 并发控制

❖可串行化调度

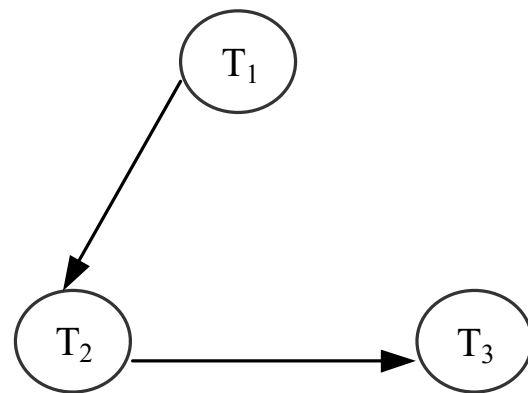
$L_1 = w_1(x), r_2(z), r_2(y), w_2(x), w_1(y), r_1(z), w_3(z), r_3(z), w_3(x)$

$L_2 = w_1(x), w_1(y), r_1(z), r_2(z), r_2(y), w_2(x), w_3(z), r_3(z), w_3(x)$

$L_3 = w_1(x), w_1(y), r_1(z), r_2(z), w_3(z), r_2(y), r_3(z), w_2(x), w_3(x)$



(a) L_1 的串行化图



(b) L_2 和 L_3 的串行化图

第八章 分布式数据管理

8.2 并发控制

❖ 基于锁的并发控制

➤ 静态封锁和动态封锁：

在静态封锁方式中，事务处理在执行操作前，对所有需要的数据对象封锁。这个方法相对简单，然而它限制了并发性，因为有冲突的事务处理必须串行执行。在动态封锁方式中，事务处理在执行的不同阶段对不同对象封锁。

➤ **封锁范围：**封锁范围是一个很重要的设计问题。显然，既可以对整个文件进行封锁，也可以对特定的记录进行封锁。封锁的范围越小，并发性越好。如果两个用户要修改同一个文件，并且它们是对这个文件的两个不同部分进行修改的话，则可以让它们同时进行。

第八章 分布式数据管理

8.2 并发控制

❖ 基于锁的并发控制

➤ **两阶段封锁：**两阶段封锁可以保证一致性，实现正确的并发控制。两阶段封锁的主要内容如下：访问一个对象前先封锁它，为此必须先获得锁；对所有要访问的对象封锁前不对任何对象进行解锁；不要封锁已经被封锁的对象，为此不同的事务处理不可同时获得冲突的锁；事务处理执行结束前，为每个被它封锁的对象解锁；一旦一个事务处理释放一个锁，该事务处理就不能再获得另外的锁。每个事务处理锁的过程可分成两个阶段：锁的增长阶段和锁的收缩阶段。增长阶段中事务处理获得所有的锁而不释放任何锁；收缩阶段释放所有的锁而不取得另外的锁。

第八章 分布式数据管理

8.2 并发控制

❖ 基于锁的并发控制

➤ 两阶段封锁的问题：

问题1：在锁的增长阶段会出现死锁的问题。发生死锁是因为在加锁机制中，锁是一种竞争的资源。死锁问题可以用第六章讨论的办法来解决。

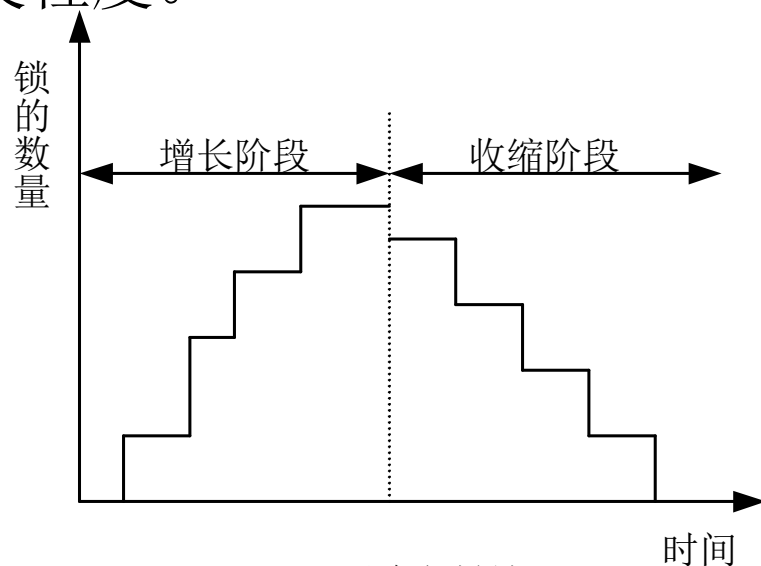
问题2：在锁的收缩阶段容易出现层叠回退(cascaded aborts)的问题。层叠回退的问题发生在一个事务处理操作失败后回退，在回退前由于该进程已经释放了某些数据对象上的锁，所以其他事务处理可能已经读取了被这个事务处理修改过的数据对象，这些事务处理也必须回退。

第八章 分布式数据管理

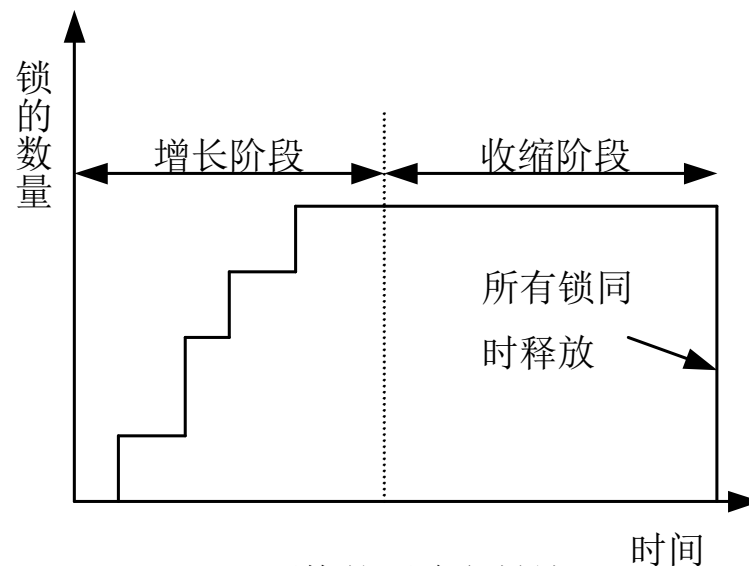
8.2 并发控制

❖ 基于锁的并发控制

➤ 层叠回退的问题可以用严格的两阶段封锁方案来解决：
在这种方案中，事务处理一直占有所有获得的锁直到操作完成，
然后在一个单一的原子操作中释放所有的锁，但是它降低了并
发程度。



(a) 两阶段封锁



(b) 严格的两阶段封锁

第八章 分布式数据管理

8.2 并发控制

❖ 基于锁的并发控制

➤ 加锁方案，共有三种实现方法：

- (1) 集中式加锁方法。在这种加锁方式中，锁管理是集中进行的。有一个集中的锁管理员，它负责锁的分配和释放。
- (2) 主站点加锁方法。每个数据对象有一个主站点，所有对数据对象的更新首先定向到主站点。对某个数据对象的加锁和释放锁的管理活动是由该数据对象的主站点上的锁管理员负责的。
- (3) 分散式加锁方法。锁的管理有所有站点共同完成。事务处理的执行包括许多站点的参与和协作。

第八章 分布式数据管理

8.2 并发控制

❖ 基于时间戳的并发控制

各种时间戳：

(1) $ts(T)$ ：每个事务处理 T 在它启动的时候被分配一个时间戳 $ts(T)$ ，一个事务处理 T 中的每个操作被加盖一个时间戳 $ts(T)$ ；

(2) $ts_{RD}(x)$ ：对数据对象最近一次读时的时间戳用 $ts_{RD}(x)$ 表示；

(3) $ts_{WR}(x)$ ：对数据对象最近一次写时的时间戳用 $ts_{WR}(x)$ 表示；

(4) $read(T, x, ts)$ ： $read(T, x, ts)$ 表示来自事务处理 T 对数据对象 x 读操作请求，该请求带有时间戳 ts ；

(5) $write(T, x, ts)$ ： $write(T, x, ts)$ 表示来自事务处理 T 对数据对象 x 写操作请求，该请求带有时间戳 ts 。

第八章 分布式数据管理

8.2 并发控制

❖ 基于时间戳的并发控制 读或写操作的处理方式:

(1) 读请求 $\text{read}(T, x, ts)$ 。如果 $ts < ts_{WR}(x)$, 说明有一个对 x 的写操作发生在事务处理 T 启动之后, 则读请求被拒绝, 事务处理 T 被取消。反之, 让 T 执行该读操作, 并把 $\max\{ts_{RD}(x), ts\}$ 赋给 $ts_{RD}(x)$ 。

(2) 写请求 $\text{write}(T, x, ts)$ 。如果 $ts < ts_{WR}(x)$, 或者 $ts < ts_{RD}(x)$, 说明有一个对 x 的写操作或者有一个对 x 的读操作发生在事务处理 T 启动之后, 则写请求被拒绝, 事务处理 T 被取消。反之, 让 T 执行该写操作, 并把 ts 赋给 $ts_{WR}(x)$ 。

第八章 分布式数据管理

8.2 并发控制

❖ 基于时间戳的并发控制

实例： 假设数据对象 x 的最近一次读和写的时间戳分别为 $ts_{RD}(x)=4$ 和 $ts_{WR}(x)=6$ ，对数据对象 x 有下列读写操作顺序：

$read(5, x, 5)$, $write(6, x, 7)$, $read(7, x, 9)$, $read(8, x, 8)$,
 $write(9, x, 8)$

(1) $read(5, x, 5)$ 操作请求被拒绝，对应的事务处理5被取消，因为 $ts=5 < ts_{WR}(x)=6$ ；

(2) $write(6, x, 7)$ 操作请求被接受，对应的事务处理6继续执行， $ts_{WR}(x)$ 更新为7；

(3) $read(7, x, 9)$ 被接受，对应的事务处理7继续执行， $ts_{RD}(x)$ 更新为9；

(4) $read(8, x, 8)$ 也被接受，对应的事务处理8继续执行，但是 $ts=8 < ts_{RD}(x)=9$ ，所以 $ts_{RD}(x)=9$ 保持不变；

(5) $write(9, x, 8)$ 操作请求被拒绝，对应的事务处理9被取消，因为

第八章 分布式数据管理

8.2 并发控制

❖ 基于时间戳的并发控制

保守时间戳顺序:为了减少或消除事务处理的取消和重启, 可使用下列保守时间戳顺序, 所有的站点严格地按照时间戳顺序执行请求。每个站点有一个写队列(W-queue)和一个读队列(R-queue)。并且按照请求的时间戳排序。

(1) 如果所有写队列非空, 而且每个队列的第一个写的时间戳大于 ts , 读请求 $read(T, x, ts)$ 可以被执行; 反之, 该读请求在读队列中缓存。

(2) 如果所有读队列和写队列非空, 而且每个队列的第一个请求的时间戳大于 ts , 写请求 $write(T, x, ts)$ 可以被执行; 反之写请求在写队列中缓存。

第八章 分布式数据管理

8.2 并发控制

❖ 乐观的并发控制

并发控制算法首先假设没有冲突，所以首先在本地尝试更新，如果没有一致性冲突就保留更新并扩散更新，这个方法有三个阶段：读、确认和写。

- 1) 读阶段，在临时存储空间中读入相关数据对象并进行更新；
- 2) 确认阶段，检查所有更新，确认是否违反数据库一致性。检查阶段要检查所有其他事务处理，确认自该事务处理启动以来是否有其他的事务处理对它所访问的数据对象进行了修改。如果有则该事务处理取消，否则检查通过，进入写阶段；
- 3) 写阶段，把所有更新写进数据库。

乐观并发控制算法的最大优点是没有死锁，并且可以最大限度地提供并行，因为没有有一个事务处理会等待。它的缺点是一旦在确认阶段不能通过检查，事务处理就要重新执行。

第八章 分布式数据管理

8.3 原子事务处理

❖ 原子事务处理的性质：

- 1) 原子性(Atomicity)。一个原子事务处理的执行必须确保是完全执行完毕或相当于完全没有执行。
- 2) 一致性(Consistency)。一个事务处理必须以一致性的状态开始，以一致性的状态结束。不能违背系统的恒定性。
- 3) 孤立性(Isolation)。原子事务处理所有的中间操作必须以孤立的形式执行。只允许在该事务处理操作的某个数据对象达到一致的状态时才能够被其他事务处理访问。也就是说，并发的事务处理之间不会发生相互干扰。孤立性也称为可串行性。
- 4) 持久性(Durability)。一旦事务处理已经被提交，即使在发生系统失效的情况下，结果将永不丢失。

第八章 分布式数据管理

8.3 原子事务处理

❖事务处理的分类：

- 1) 平面事务处理：事务处理都是由一系列基本的操作所组成；
- 2) 一个嵌套事务处理是由多个子事务处理组成的，顶层的事务处理产生多个子事务处理，这些子事务处理相互在不同的机器上并行执行。每个子事务处理也可以产生自己的子事务处理；
- 3) 分布式事务处理也是由一些子事务处理组成的，分布式事务处理在逻辑上是平面的，是施加在分布式数据对象上的一系列在逻辑上不可分的操作组成的；

第八章 分布式数据管理

8.3 原子事务处理

❖ 原子事务处理的实现：

实现原子事务处理必须提供以下几个组成部分：

- 1) 事务处理管理员：负责使该事务处理的各个参加者就该事务处理是否提交或夭折达成一致意见；
- 2) 恢复管理员：负责在事务处理失效后恢复状态；
- 3) 缓冲器管理员：负责在主存和磁盘间传送数据；
- 4) 运行记录(log)管理员：负责各种操作及状态的记录；
- 5) 锁管理员；负责并发控制；
- 6) 通信管理员：负责透明的跨网络的通信，在分布式事务处理中通知事务处理的管理部分。

第八章 分布式数据管理

8.3 原子事务处理

❖ 原子事务处理的局部恢复技术

➤ 意图表方法：

意图表方法按顺序完成以下操作：

- (1) 把事务处理要向各数据对象施加的所有修改操作都存放在一个表中，该表称为意图表；
- (2) 该表被写入坚固存储器中；
- (3) 事务处理管理员决定该事务处理是否提交；
- (4) 若该事务处理提交，则在表中设置完整标志，开始按照意图表修改在磁盘中的那些对象；
- (5) 删除意图表。

第八章 分布式数据管理

8.3 原子事务处理

❖ 原子事务处理的局部恢复技术

➤ 意图表方法：

失效后的处理方法：

- (1) 若意图表不存在，则夭折该事务处理。
- (2) 若意图表不完整，即在写意图表时节点崩溃，则夭折该事务处理并删除此表；
- (3) 若意图表完整，即在按坚固存储器中的意图表对数据对象进行实际的更新时节点崩溃，则系统按表中的说明进行更新，更新完毕后删除意图表。

第八章 分布式数据管理

8.3 原子事务处理

❖ 原子事务处理的局部恢复技术

➤ 先写运行记录方法：

- **执行过程：**先写运行记录的方法在事务处理的执行过程中，对数据对象进行了实际的修改。但是在数据对象被修改前，需要在运行记录中写一个记录项，用于告诉事务处理管理员对哪个数据对象进行修改，修改前的值和修改后的值是什么，以便用于失效后的恢复。只有在这个记录项正确地写到运行记录之后，才能对实际的数据对象进行修改。该运行记录是放在坚固存储器之中的。
- **恢复过程：**如果事务处理夭折，运行记录可以用来退回到事务处理执行前的起始状态。从运行记录的末尾往回退的过程中，读取每一个运行记录项，按记录项的内容执行恢复操作，最终会退回到事务处理执行前的状态，这个过程叫做卷回(rollback)。

第八章 分布式数据管理

8.3 原子事务处理

❖ 原子事务处理的局部恢复技术

➤ 先写运行记录方法:

• 实例:

x=0;			
y=0;	Log	Log	Log
Begin			
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			

(a) 一个简单的事务处理

(b) 第一条语句对应的运行记录

(c) 第二条语句对应的运行记录

(d) 第二条语句对应的运行记录

第八章 分布式数据管理

8.3 原子事务处理

❖ 原子事务处理的局部恢复技术

➤ 先写运行记录方法：

- 必须遵守的规则：

- 1) 在把数据对象的新值写入到运行记录之前，先把数据对象的旧值写入运行记录；
- 2) 在对数据对象进行实际的更新之前，先把该数据对象的旧值和新值写入到运行记录中。

第八章 分布式数据管理

8.3 原子事务处理

❖ 分布式提交协议

分布式提交用于保证一个进程组中的每一个成员要么都执行某一个操作，要么都不执行这个操作。分布式提交用于分布式事务处理的全局恢复。

➤ 两阶段提交协议(2PC)

• 协调者的准备提交阶段：

- (1) 向所有的参加者广播一个PREPARE(准备)报文；
- (2) 等待每一个参加者的回答。

每个参加者仅在完成此事务处理中属于它应该完成的那部分工作，同时写好运行记录之后才发出一个回答报文。

第八章 分布式数据管理

8.3 原子事务处理

➤ 两阶段提交协议 (2PC)

- 协调者的提交阶段:

如果有一个参加者回答一个ABORT(夭折), 或者有一个参加者超时未回答, 则将ABORT项写入运行记录中, 然后向所有参加者广播一个ABORT报文, 在收到参加者的承认后, 本次协议的执行结束。如果所有参加者发来的PREPARED(准备好了)报文, 则进行一下操作:

- (1) 在运行记录上写入一个COMMIT项;
- (2) 向所有的参加者广播一个COMMIT报文;
- (3) 等待每个参加者做出肯定回答ACK-COMMIT;
- (4) 将COMPLETE(完成)项记入运行记录, 本次协议的执行结束。

第八章 分布式数据管理

8.3 原子事务处理

➤ 两阶段提交协议 (2PC)

- 参加者的准备提交阶段：

- (1) 等待协调者的PREPARE报文；
- (2) 完成自己的工作，将运行记录项写入运行记录；
- (3) 如果成功，则将COMPLETE项记入运行记录，并向协调者发送PREPARED报文；否则发送ABORT报文。

- 参加者的提交阶段：

- (1) 等待协调者的裁决报文；
- (2) 向协调者发送回答。如果裁决是ABORT则将事务处理作废，同时将ABORT项写入运行记录中，释放锁，发送夭折承认报文ACK-ABORT；如果裁决是COMMIT，在运行记录上写入一个COMMIT项，释放锁，发送提交承认报文ACK-COMMIT。

第八章 分布式数据管理

8.3 原子事务处理

➤两阶段提交协议(2PC)

- 恢复管理员必须遵守以下规则：

- (1) 如果协调者在第二阶段的步骤(1)以前崩溃，此时协调者的运行记录中未包含COMMIT项，协调者重新启动后向所有参加者发送ABORT报文；

- (2) 如果协调者在第二阶段的步骤(1)之后，但在第二阶段的步骤(4)以前崩溃，此时协调者的运行记录中包含COMMIT项，但是无COMPLETE，协调者重新启动后向所有参加者发送COMMIT报文；

- (3) 如果协调者在第二阶段的步骤(4)之后崩溃，此时协调者的运行记录中包含COMPLETE项，则此事务处理已经提交，协调者的工作已经完成，对重新启动可以不予理睬。

第八章 分布式数据管理

8.3 原子事务处理

➤用于嵌套事务处理的提交协议

用于嵌套事务处理的提交协议也分为两个阶段：准备提交阶段和提交阶段。在这种提交协议中，每个子事务处理分别提交或夭折。只有在所有的子事务处理都提交以后，父事务处理才能够进入提交阶段。子事务处理必须保持事务处理的ACID特性，子事务处理提交的结果只有其父事务处理才能看见。最后，如果父事务处理确定它不能进入提交阶段，那么它及其所有的后代事务处理都必须卷回。

第八章 分布式数据管理

8.3 原子事务处理

➤用于嵌套事务处理的提交协议

准备提交阶段：

```
commit flag=1;                                //将提交标志flag置为1。
flag PREPARE-TO-COMMIT(transaction t)         //transaction 是一个结构变量用于保存
                                                //其用于保存事务处理t的相关信息。该
                                                //函数返回事务处理t是否准备好提交。

{
    IF childless(transaction t)                //如果事务处理t没有子事务处理
    THEN
        flag = flag & Request commitment(transaction t)
                                                //请求对事物处理t进行提交，返回值和
                                                //flag进行与操作。只有所有服务员都同
//意提交，flag的值才能保持为1。
    ELSE                                        //如果有子事务处理，需要递归
    {
        FOR ALL children(transaction t) DO
        PREPARE-TO-COMMIT(transaction a(current child));
    }
}
```

第八章 分布式数据管理

8.3 原子事务处理

➤用于嵌套事务处理的提交协议

提交阶段:

```
COMMIT-PHASE(transaction t)
```

```
{
```

```
    IF(flag=1)
```

```
//准备提交阶段获得成功，可以进入提交阶
```

```
段。
```

```
    THEN IF childless(transaction t)
```

```
//如果事务处理t无子事务处理
```

```
        THEN {
```

```
            Record in stable storage (commit, transaction
```

```
a);
```

```
//将提交记录项写入坚固存储器
```

```
Notify commitment(transaction);
```

```
//通知顾客提交阶段已经开始
```

```
        }
```

```
    ELSE {
```

```
        FOR ALL children(transaction a) DO
```

```
        COMMIT-PHASE(transaction a(current child));
```

```
    }
```

```
}
```

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 分布式系统中的系统数据库

➤ 系统数据库的分布有三种可能的方式：

完全分割方式：数据库的各部分分散到各地点，相互都是不重复的；

完全冗余方式：整个数据库的各个副本分散各处；

部分冗余方式：数据库分成若干部分，其中有些部分有副本存放在其他地点。

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 分布式系统中的系统数据库

➤ 数据库一致性包含两个方面：冗余副本的相互一致性和每个副本的内部一致性。

- 相互一致性是指整个数据库或其一部分的各副本是相同的。
- 分布式数据库的内部一致性是指该数据的信息内容而言，包含两个概念：

(1) 语义完整性：所存储的数据精确地反映该数据库所模仿的客观世界。这要求检验对数据库内容进行修改操作的每个事务处理在提交任何修改前不违背语义完整性的约束。这实际上就是我们前面所提到的一致性约束条件。

(2) 原子事务处理：作为一个事务处理的结果，要求在数据库中和提供给外界的信息中反映出该事务处理产生的活动，要么全部产生，要么一个也不产生。

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 兼容可串行化

➤ **兼容串行化**：设并发事务处理 T_1 和 T_2 分别使存储处理机P和Q产生活动 $A_1(P)$ 、 $A_1(Q)$ 和 $A_2(P)$ 、 $A_2(Q)$ ，如果 $A_1(P) \rightarrow A_2(P)$ 在P处是正确的串行顺序，则在Q处的可兼容串行化应为 $A_1(Q) \rightarrow A_2(Q)$ 。

➤ **复制控制算法**用来保证兼容可串行化，保持数据库多副本间的相互一致性。复制控制算法分为两大类：表决法和非表决法。

➤ **表决法**是在控制者之间交换报文，对分布式数据库要进行的各事务处理的整个次序达成一致意见。

- 在同步表决方案中，各控制者对一个给定事务处理进行表决，一旦取得一致意见就共同执行此事务处理的各步骤。
- 在异步表决方案中，允许各控制者和存储处理机并发地对不同事务处理的请求和执行进行表决。

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 兼容可串行化

➤ 非表决议法使用控制优先权，一种方法是在各控制者之间建立主从关系。

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖主站点方法

一个节点被指定是主站点，其他是备份的。所有请求直接发送到主站点。

- (1) 读请求。主站点只是简单地读取并返回结果。没有备份站点参与。如果主站点失效，经选举在备份站点中产生一个新的主站点。
- (2) 写请求。当主站点完成更新前收到一个写请求，它就向至少 k 个备份站点发送更新请求。一旦所有站点已经收到请求，主站点完成操作再送回结果。

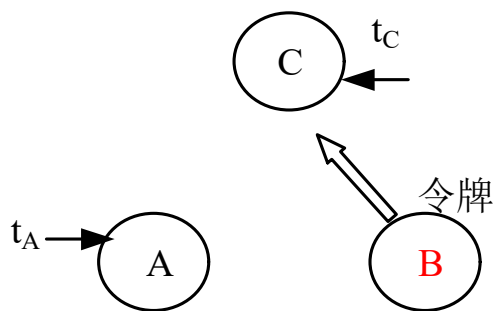
第八章 分布式数据管理

8.4 多副本更新和一致性管理

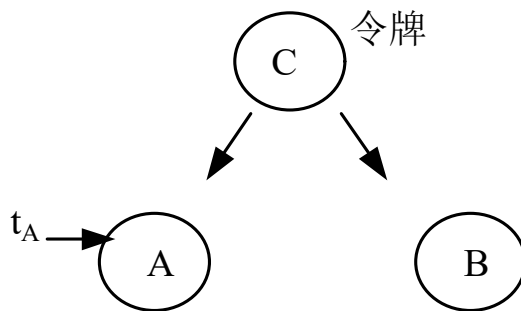
❖ 循环令牌方法

循环令牌法把全部控制者排成一个虚拟的单向环，给一个控制者分配一个暂时的唯一的控制优先权。使用循环令牌方法可以实现两种不同的多副本更新协议：

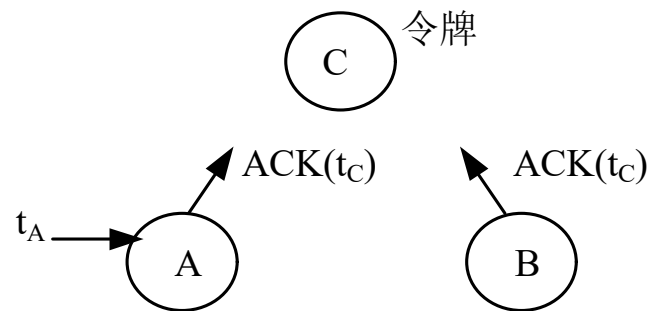
(1) 为了在全部控制者/存储处理机上发动执行事务处理，控制者必须等待收到控制令牌。



(a) B 没有事务处理，
把令牌传给 C



(b) C 收到令牌，就向 A
和 B 广播事务处理 t_c



(c) A 和 B 承认并执行
事务处理 t_c

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 循环令牌方法

(2) 令牌携带顺序号，用来给事务处理请求发“票”，每张票上都有顺序号，每发出一张票，就自动增加顺序号。这样，这些票给这些请求建立一个全排序。

一旦一个请求在队列中已经移到第一个位置，在所有的存储处理机上发动局部处理。

第八章 分布式数据管理

8.4 多副本更新和一致性管理

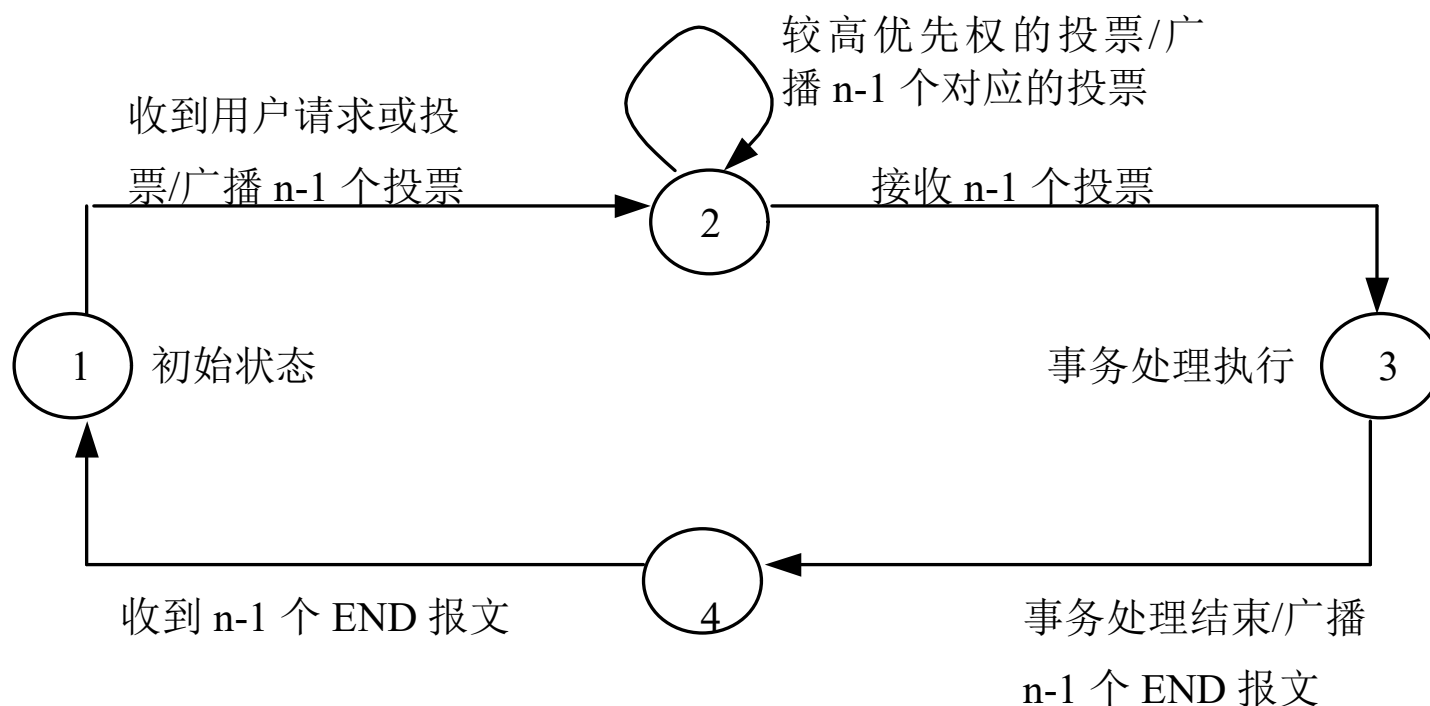
❖ 同步表决方法

- (1) 开始，控制者处于初始状态。接到请求后它对事务处理进行表决，并向所有其他控制者广播投票。
 - (2) 当收到对某一事务处理的投票后，它 also 对该事务处理进行表决，并向所有其他控制者广播投票。
 - (3) 当收到所有其他控制者对某一个事务处理的投票，并且自己也对该事务处理投票，则开始执行该事务处理。
 - (4) 事务处理执行完毕后，向所有其他控制者发出END报文。
 - (5) 收到所有其他控制者的END报文后返回到初始状态。
- 若有更高优先级的请求，则放弃当前的事务处理，对新的事务处理进行表决。

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 同步表决方法

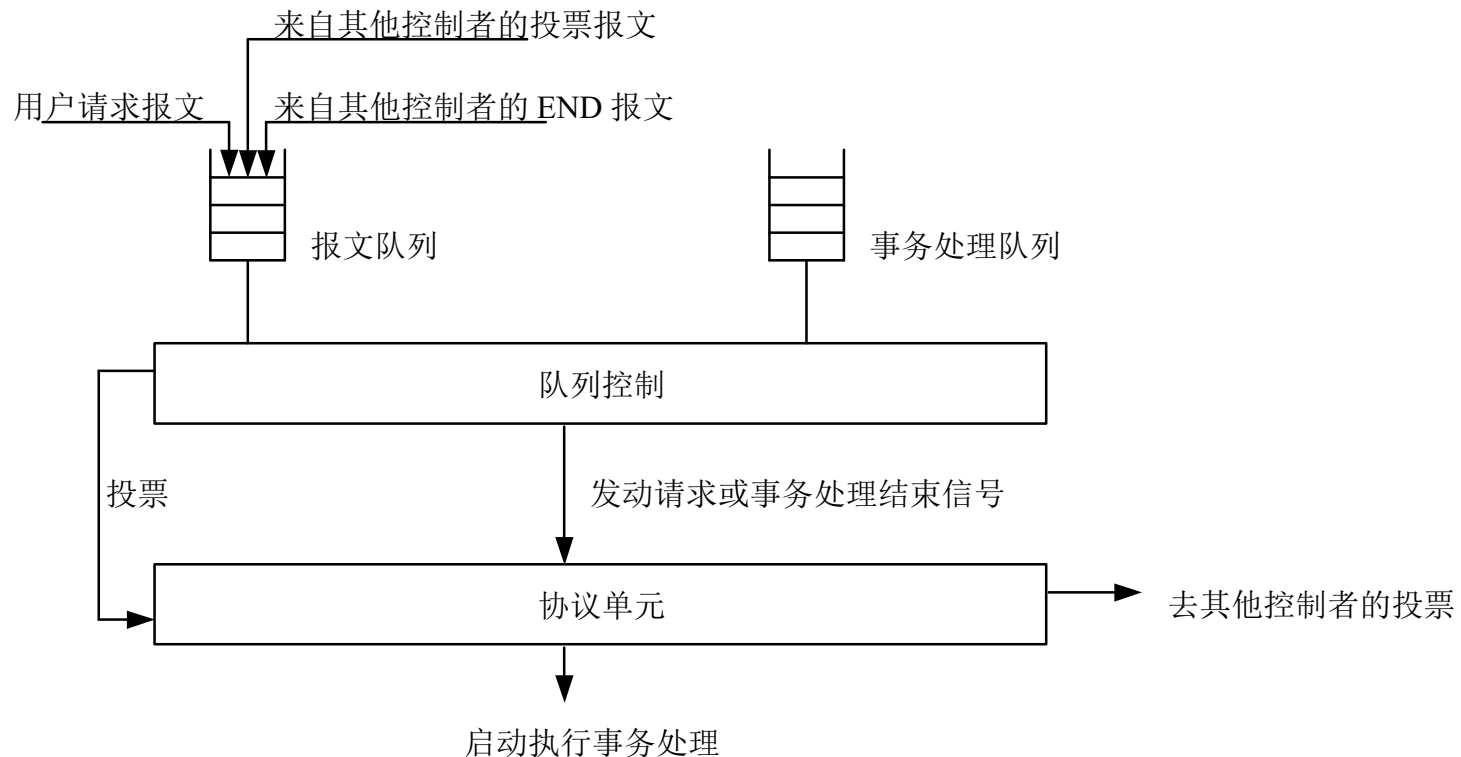


第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 同步表决方法

可进行同步表决排队的控制者的结构：



第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖活动复制控制方法

在每个时刻维护相互一致性是不可能做到的，一般的复制控制算法只要求顺序一致性，在这里顺序一致性表现为：每个控制者经历同样顺序的修改操作，同时，如果没有新操作时它们的值应该相等。活动复制方法类似于Lamport时间戳互斥算法。每个本地更新加上时间戳后广播到所有其他控制者。每个控制者 i 有一个队列 $Q_i[j]$ ，包含有从控制者 j 发送来的报文。因为每个队列中的报文是按照它们的发送顺序排列的，所以按顺序查看报文时只需要检查队列头。

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖法定数方法

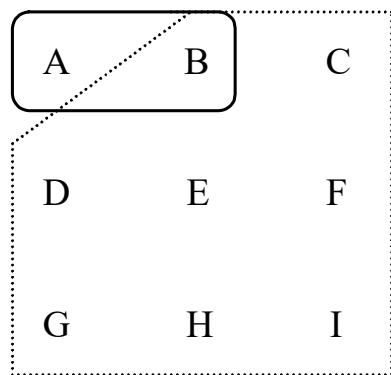
设具有某文件副本的服务员数目为 N 。要读取一个文件，必须要得到 N_R (读定额) 个服务员同意。要修改一个文件，需要得到 N_W (写定额) 个服务员同意。 N_R 和 N_W 应满足： $N_R + N_W > N$ 。只有当通过询问知道了有足够多的服务员愿意参加并同意之后，才能进行某种操作。

一般说来， N_R 取值小一些可使读操作易于得到满足，但是写操作的执行就难一些，反之亦然。 N_R ， N_W 的最佳选择取决于读、写操作哪个更频繁。

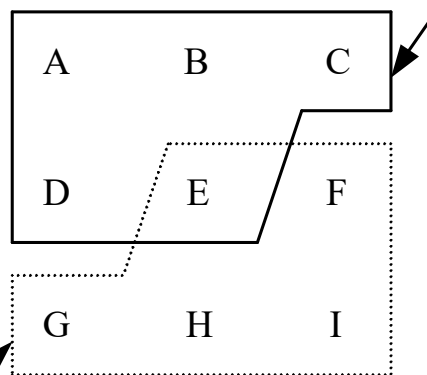
第八章 分布式数据管理

8.4 多副本更新和一致性管理

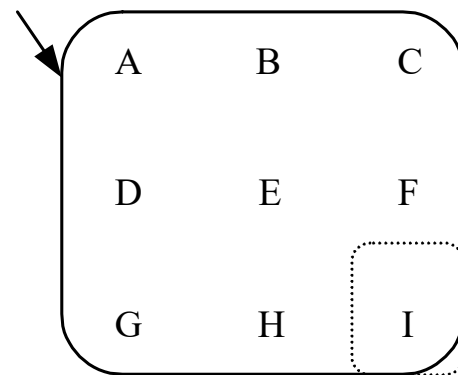
❖ 法定数方法



(a) $N_R=2$, $N_W=8$



(b) $N_R=5$, $N_W=5$

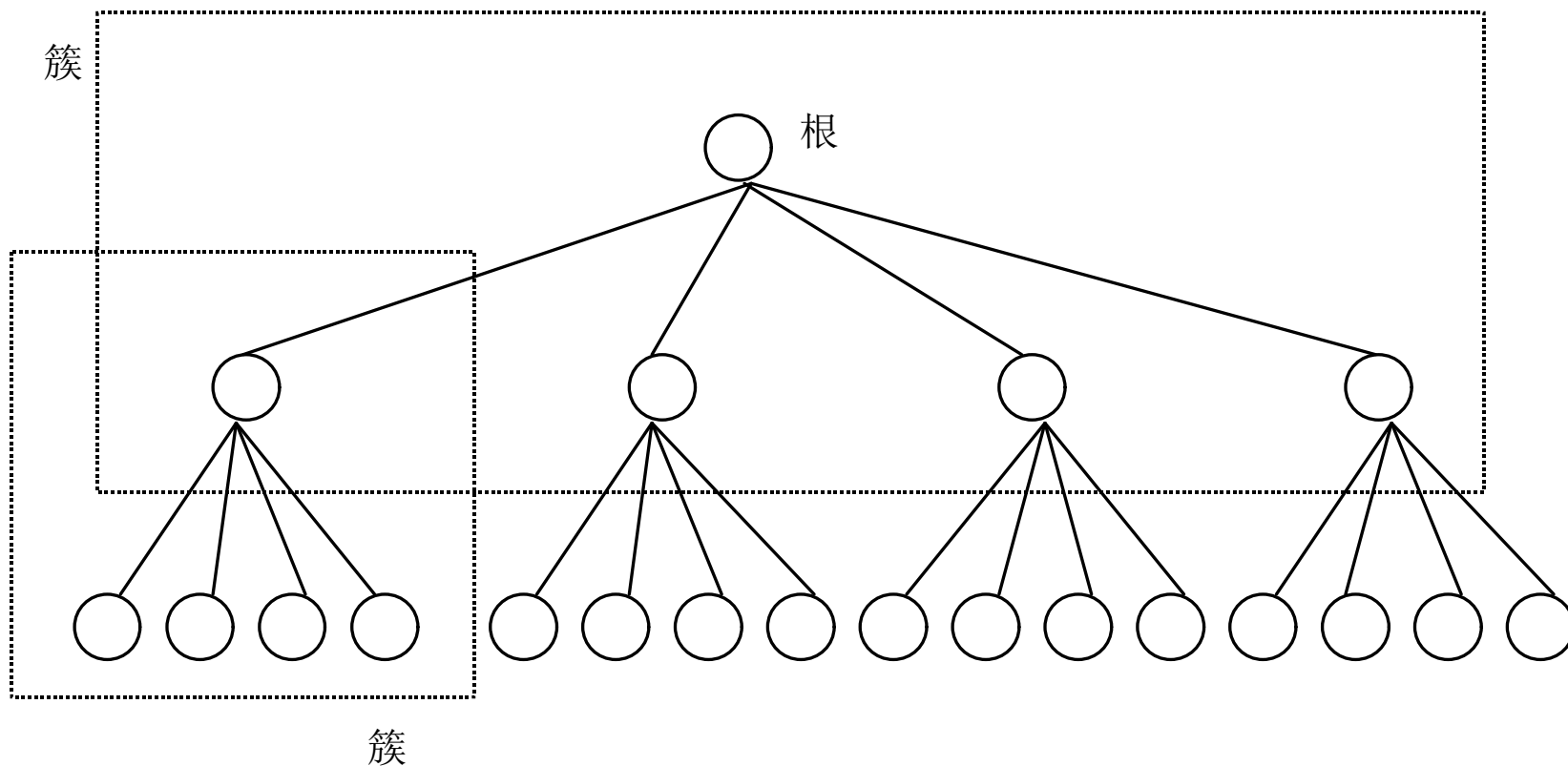


(c) $N_R=9$, $N_W=1$

第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 分层表决方法



第八章 分布式数据管理

8.4 多副本更新和一致性管理

❖ 分层表决方法

对于表决方法，使用分层的树型结构，还可以减少表决所需的报文数目。例如前面的Holler同步表决方案，完成一次同步表决至少需要 $2N(N-1)$ 个报文，其中每个控制者需要向所有其他 $N-1$ 个控制者发送投票报文，每个控制者完成本地副本更新后需要向所有其他 $N-1$ 个控制者发送END报文。图8.4.5的例子中，如果16个副本被线性组织，那么完成一次表决至少需要 $2 \times 16 \times 15 = 480$ 个报文。如果采用图8.4.5的树型结构，那么完成一次表决只需要 $5 \times 2 \times 4 \times 3 = 120$ 个报文。

第九章 分布式文件系统

9.1 分布式文件系统的特点和基本要求

❖ 分布式文件系统的特点

➤ 数据文件系统的四个关键问题。

- (1) 命名。用户怎样命名文件？命名空间是平面的还是分层的？
- (2) 编程接口。应用程序怎样访问文件系统？
- (3) 物理存储。文件系统抽象怎样映射到物理存储介质？编程接口是否独立于存储介质？
- (4) 完整性。在能源、硬件、介质和软件失效后文件如何保持一致性？

第九章 分布式文件系统

9.1 分布式文件系统的特点和基本要求

❖ 分布式文件系统的特点

➤ 分布式文件系统有如下特点：

- 1) 分布式文件系统的顾客、服务员和存储设备分散在各机器上，因此服务活动必须跨越网络完成；
- 2) 存储设备不是单一的集中的数据存储器，而是有多个独立的存储设备；
- 3) 分布式文件系统的具体配置和实现可以有很大不同，有的服务员运行在专用的机器上，有的机器既是服务员又是顾客。

分布式文件系统可以作为分布式操作系统的一部分实现，也可以作为一个独立的软件层实现，此软件层管理常规操作系统之间和文件系统之间的通信。总之，分布式文件系统的特点是系统中的顾客和服务员具有自治性和多重性。

第九章 分布式文件系统

9.1 分布式文件系统的特点和基本要求

❖ 分布式文件系统的基本要求

- 1) **透明性**: 对于顾客来说, 分布式文件系统应表现为常规的集中式的文件系统, 即服务员和存储器的多重性和分散性对顾客应该是透明的。透明性的另一个方面是用户的可移动性, 即用户可以在系统中的任何机器上登录。
 - 2) **性能**: 分布式文件的性能和常规文件的性能差不多。
 - 3) **容错**: 在发生各种故障时分布式文件系统应该能正常工作, 尽管其性能可能有所降低。
 - 4) **可扩充性**: 系统适应增加服务负载的能力叫做可扩充能力。
- 十个期望属性**: 透明性、用户灵活性、高性能、简单易用性、可扩充性、高可用性、高可靠性、数据完整性、安全性和异构性。

第九章 分布式文件系统

9.2 分布式文件系统的命名

❖命名方案

分布式文件系统中名字透明和位置透明：

- (1) 数据和位置分离为文件提供一个较好的抽象。
- (2) 名字透明给用户共享数据提供了一个方便的方法。
- (3) 位置透明把命名结构从存储器结构和服务员间的结构分开。

第九章 分布式文件系统

9.2 分布式文件系统的命名

❖命名方案

分布式文件系统的三种命名方法：

- (1) 把主机名和文件的本地名字结合起来给文件命名。这是一种最简单的命名方法，它能保证唯一的系统范围内的名字。
- (2) 把远程文件目录附加到本地名字空间中。SUN工作站的NFS是这种方法的典型代表，它使用安装(mount)机制把远程文件目录附加到本地名字空间中。
- (3) 把所有各部分文件系统全部集成组成单一的全局名字空间结构。同一名字空间可被所有顾客见到。理想上，合成的文件系统结构应和常规文件系统是同型的。

第九章 分布式文件系统

9.2 分布式文件系统的命名

❖命名的实现技术

组织名字服务员的方法有三种：

- (1) 使用集中式的名字服务员。集中的名字服务员维护一张表或者数据库，提供必要的名字到对象的映射关系。
- (2) 将文件系统分割成域，每个域有自己的名字服务员。构建一个全局名字树，当本地名字服务员查找一个名字a/b/c时，它可以生成一个指向另一个域(即另一个名字服务员)的指针，把名字剩余部分(b/c)发送给它。
- (3) 每个处理器管理自己的名字。查找名字时，首先广播到网络中。在每个处理机上，到来的请求被转发给本地名字服务员，只有当名字服务员找到相应的匹配纪录时才做出回答。

第九章 分布式文件系统

9.2 分布式文件系统的命名

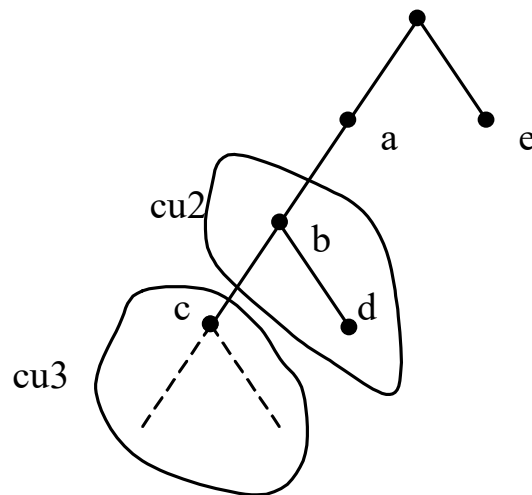
❖命名的实现技术

常用的命名实现技术有：

(1) 路径名翻译。正文名字到底层标识符的变换典型地都使用常规UNIX中的方法，即递归查找(recursive lookup)程序。

位置表

组成单元	服务员
cu1	机器 1
cu2	机器 2
cu3	机器 3



第九章 分布式文件系统

9.2 分布式文件系统的命名

❖命名的实现技术

常用的命名实现技术有：

(2) 有结构的标识符。实现透明的命名需要提供文件名到其位置的变换机制。

(3) 提示(hint)。这是用于位置变换的另一个方法。一个提示是一份信息，能提高性能，如果它不正确也不会引起任何语义上的副作用。

(4) 安装机制。为了创建一个全局名字结构而把远程文件系统连接起来，通常使用安装机制。

第九章 分布式文件系统

9.3 共享语义

共享语义是评价分布式文件系统允许多个顾客共享文件的重要标准。它说明分布式文件系统的这样一个特性，即多个顾客同时访问共享文件的效果。特别是，这些语义应当说明被顾客修改的数据何时可以被远程顾客看到。

(1) UNIX语义。在UNIX语义中，对分布式文件系统中的文件进行每个read(读)时能看到以前所有对该文件执行write(写)的效果。特别是，顾客对打开的文件进行的写可立即被其他同时打开此文件的顾客看到，即使该顾客在远程也能够看到。

(2) 对话语义。对话语义符合以下两个规则：第一，对于打开文件的write可立即被本地顾客看见，远程的顾客虽然也同时打开该文件，但却看不见。第二，一旦文件关闭，对此文件所作的修改仅在以后开始的对话中才能看见，该文件已经打开的各副本不表现这些修改。

第九章 分布式文件系统

9.3 共享语义

(3) 事务处理语义。具有事务处理的文件对话产生下述熟悉的语义：几个文件对话对一个文件的作用及其输出等效于以某个串行次序执行这些对话的作用及其输出。

(4) 不可改变的共享文件的语义。在这种语义中，一旦文件被其创建者说明为共享的，它就不能再被修改。不可改变的文件有两个重要性质：第一其名字不可再用；第二其内容不可改变。

第九章 分布式文件系统

9.4 缓存

❖文件的远程访问方法

文件的远程访问方法有两种：

- (1) 远程服务。在远程服务方法中，顾客把访问请求传送给服务员，服务员执行访问，结果回送给顾客。访问请求被变换成对服务员的报文，服务员的回答也打包成报文回送给顾客。
- (2) 缓存。如果请求的数据不在本地，则从服务员处取来那些数据的复制件给顾客。通常取来的数据量比实际请求的要多得多，例如整个文件或几个页面，所以随后的访问可在顾客所在地的本地副本中进行。

第九章 分布式文件系统

9.4 缓存

❖文件的远程访问方法

缓存方案的设计问题：

- (1) 被缓存的数据的粒度；
- (2) 顾客缓存器的地点，即使用主存进行缓存，还是使用磁盘进行缓存；
- (3) 如何传播被缓存的副本的修改；
- (4) 如何决定各个顾客缓存中的数据是否一致。

第九章 分布式文件系统

9.4 缓存

❖缓存的粒度和地点

缓存的粒度:缓存单位愈大，则下次访问的数据在顾客方的本地找到的可能性愈大，即命中率愈高，但传送数据的时间和一致性问题也增加了。大缓存单位的一个优点是减少网络开销。

从缓存的地点:使用磁盘缓存具有可靠性的优点。但是使用主存作缓存器也有若干优点。首先，它可支持无盘工作站；其次，从主存缓存器中访问数据要比从磁盘缓冲器中访问数据要快；第三，服务员缓存器(用于加速磁盘输入输出操作)是设在主存中，如果顾客缓存器也使用主存，就可以构造一个单一的缓存机构，服务员和顾客均可使用。两种缓存地点强调的功能不一样，主存缓存器主要减少访问时间，磁盘缓存器主要提高可靠性和单个机器的自治性。

第九章 分布式文件系统

9.4 缓存

❖更新策略、缓存有效性检验和一致性

立即写:最简单的策略是一旦数据写到缓存器中就把此数据写到服务员磁盘上, 这种策略叫做立即写(write-through)。它的优点是可靠性, 当顾客崩溃时丢失的信息很少。但是write性能很差。这种缓存方法等效于使用远程服务进行write访问, 仅对read访问使用缓存。

延迟写:另一个写策略是延迟对原本的更新。延迟写策略把修改先写到缓存器中, 早些时候再写到服务员磁盘上。这个方法的优点有两个: 第一, 因为写是对缓存进行的, 所以write访问完成比较快; 第二, 对一个数据的多次重复写, 只有最后一次有效并被写到服务员的原本上。但是, 延迟写的方法会产生可靠性的问题, 因为顾客崩溃时尚未写到服务员原本上的数据将丢失。

第九章 分布式文件系统

9.4 缓存

❖更新策略、缓存有效性检验和一致性

几种延迟写策略：

- 驱逐时写**：当被修改过的数据块要被从缓存器中驱逐出去时，该数据块被发送到原本；
- 周期性写**：周期地扫描缓存器，把从上次扫描以来已被修改过的块发送给原本；
- 关闭时写**：当文件关闭时把数据写回到服务员。在文件打开很短时间或很少修改的情况下，这种方法不会很大地减少网络通信。

“关闭时写”适合“对话语义”；“立即写”方法比较适合UNIX语义。

第九章 分布式文件系统

9.4 缓存

❖更新策略、缓存有效性检验和一致性

判定本地缓存的数据副本是否与原本一致，有两个基本方法验证其有效性：

- (1) **顾客发动的方法。**顾客与服务员联系，检查本地数据与原本是否一致。这个方法的关键是有效性检验的频度。
- (2) **服务员发动的方法。**服务员为每个顾客登记被该顾客缓存的文件或文件的某个部分。当服务员检测出可能不一致时，必须做出反应。服务员发动方法的一个问题是违背顾客/服务员模型。

第九章 分布式文件系统

9.4 缓存

❖缓存和远程服务的比较

两种远程访问方法和前面介绍的共享语义的关系：

- 1) 对话语义和缓存整个文件非常匹配。对话其内的读和写访问可以完全由被缓存的副本提供，因为可以把此文件按照语义和几个不同的映像联系起来。这个模型是很吸引人的，因为它实现简单。
- 2) 使用缓存对UNIX语义进行分布式实现有很严重的负面影响，它使得UNIX语义的分布式实现变得很困难。所有请求都由一个服务员指引服务的远程服务方法和UNIX语义符合得很好。
- 3) 不可改变的共享文件语义是为了缓存整个文件而创建的，使用这种语义，缓存一致性问题完全消失。
- 4) 当由同一机器上的同一服务员为所有的对同一文件的请求服务(如在远程服务中那样)时，事务处理语义可通过加锁的方法直截了当地实现。

第九章 分布式文件系统

9.4 缓存

❖缓存和远程服务的比较

两种方法的优缺点：选择缓存还是远程服务的问题本质上是选择改进性能潜力还是选择简单性的问题。

- 使用缓存时，大量的远程访问可由本地的缓存器有效地处理，大多数远程访问获得的服务速度和本地的一样快。
- 使用缓存时，服务员负载和网络通信量都减少了，扩充能力加强了。而在使用远程服务方法时，每次远程访问都是跨过网络处理的，明显增加了网络通信量和服务员负载，引起性能下降。
- 缓存时，传输大批数据的全部网络开销低于远程服务时传输一系列对具体请求的短的回答的网络开销。

第九章 分布式文件系统

9.4 缓存

❖缓存和远程服务的比较

两种方法的优缺点：

- 缓存方案的主要缺点是一致性问题。在不经常写的访问模式中，缓存方法是优越的；但在有经常写的情况下，用于解决一致性问题的机制在性能、网络通信量和服务员负载方面产生重大开销。
- 在用缓存作为远程访问方法的系统中，仿真集中式系统的共享语义是很困难的。使用远程服务时，服务员将所有访问串行化，因此能够实现任何集中的共享语义。
- 远程服务风范仅仅是本地文件系统接口在网络上的扩充。这样，机间接口是本地顾客和文件系统之间的接口的映射。

第九章 分布式文件系统

9.5 容错和可扩充性

❖ 无状态服务和有状态服务

概念： 当一个服务员在为其顾客请求进行服务的时候，如果它保存其顾客的有关信息时，称该服务员是有状态的(stateful)。相反，如果服务员在为其顾客请求进行服务的时候不保存该顾客的任何信息，就称该服务员是无状态的(stateless)。

崩溃对两种服务的影响： 如果服务员在服务活动中间发生崩溃时，有状态和无状态服务受到的影响有明显的不同。

- **有状态服务**需要有恢复协议，有状态服务员丢失内存中所有状态信息。较好的恢复通常使用基于和顾客对话的恢复协议恢复这个状态。稍差的恢复是当崩溃发生时将正在进行的操作夭折。
- **无状态服务员**避免了上述问题，因为可再产生一个服务员无任何困难地响应一个具有完整信息的请求。

第九章 分布式文件系统

9.5 容错和可扩充性

❖ 无状态服务和有状态服务

对无状态服务的约束：使用坚定的无状态服务的代价是需要较长的请求报文，并使得处理请求的速度较慢，此外，无状态服务在设计分布式文件系统时要加上一些约束。

- 首先，因为每个请求要能识别目标文件，所以需要有一个唯一的、全系统范围的、低层的命名。对每个请求，将远程名翻译成本地名更使处理该请求的速度降低；
- 第二，由于顾客会重发对文件操作的请求，所以这些操作必须是幂等的(idempotent)，一个幂等操作即使连续执行了多次，其效果仍相同。

第九章 分布式文件系统

9.5 容错和可扩充性

❖ 无状态服务和有状态服务
需要有状态服务的场合：

- 如果使用远程网和网际网，收到报文的次序可能与其发送的次序不同，此时有状态的面向虚拟电路的服务更为可取，因为使用所维持的状态可以把报文正确地排序。
- 另一种场合，如果服务员使用服务员发动的方法进行缓存一致性检验，则不能提供无状态服务，因为它维持一个哪些文件被哪些顾客缓存的纪录。

第九章 分布式文件系统

9.5 容错和可扩充性

❖ 可用性与文件复制

概念：

可恢复性： 当对某个文件的操作失败，或由顾客夭折此操作时，如果文件能转换到原来的一致状态，则说此文件是可恢复的。

坚定性： 如果当某个存储器崩溃和存储介质损坏时某个文件能保证完好，则说此文件是坚定的。

可用性： 如果无论何时一旦需要就可访问，甚至在某个机器和存储器崩溃，或者在发生通信失效的情况下，某个文件仍然可被访问，则这种文件叫做是可用的。

第九章 分布式文件系统

9.5 容错和可扩充性

❖ 可用性与文件复制

文件复制：文件复制是一个冗余措施。这里指的是不同机器上的文件复制，而不是同一机器上不同介质上的文件复制(如镜像磁盘)。复制文件可以增强其可用性。多机复制也能改进性能，因为可以选择最近的复制件给访问请求提供服务，从而获得短的服务时间。

文件复制的要求：

- 复制方案的基本要求是同一文件的不同复制件应该位于不同的机器上，这些机器中某一台发生故障不影响其他机器。这也就是说，一个复制件的可用性不受其他复制件可用性的影响。
- 应对用户隐匿复制细节。把一份复制件名字变换成指定的复制件是命名方案的任务。

第九章 分布式文件系统

9.5 容错和可扩充性

❖ 可用性与文件复制

文件复制的要求：

- 与复制件有关的主要问题是它们的更新。从用户观点看，文件的所有复制品代表同一逻辑实体，所以对任何复制件的更新必须反映到所有其他复制件。
- 大多数情况下，不能放弃文件数据的一致性，因此使用复制来增加可用性时还要使用复杂的更新传播协议。

第九章 分布式文件系统

9.5 容错和可扩充性

❖可扩充性

设计大规模系统要考虑的问题：

- 首先是有界资源(bounded resources)原理：“从系统的任何部分来的服务要求应该限于一个常数，此常数和系统中的节点数无关”。负载和系统规模成比例的任何服务员，一旦系统扩充到超过某一范围则必然阻塞，再增加资源也缓解不了这个问题，此服务员的能力限制了系统的扩充。
- 广播是一种使网络中的每个机器都参加的活动。在广播基础上建立的机构对超大型系统很明显不实际。

第九章 分布式文件系统

9.5 容错和可扩充性

❖可扩充性

设计大规模系统要考虑的问题：

- 网络拥挤和延迟是大规模系统的主要障碍。一个办法是使用缓存、提示和实施放松的共享语义等措施，使跨越机器的交互作用最少。
- 不应当使用集中控制方案和集中的资源建立可扩充的和容错的系统。
- 分散化的一个重要方面是系统的管理。分配管理职责时，应有利于自治性和对称性，不干扰分布式系统的连贯性和一致性。
- 将系统划分为若干半自治的小组。每个小组包括一些机器和一个专用的小组服务员。为了尽量减少跨越小组的文件访问，大多数时间，每个机器的请求应由其小组服务员满足。

第九章 分布式文件系统

9.5 容错和可扩充性

❖ 用线程实现文件服务员

- 单个进程服务员当然不好，因为一旦有一个请求需要磁盘输入输出时，整个服务将被推迟到输入输出完成为止。
- 每个顾客分配一个服务进程，但是这样要求CPU完成进程切换付出很大代价。
- 使用线程服务员，同一个进程里的多个线程之间只有很少的非共享状态。线程的切换和线程的创建比起进程的切换和进程的创建要便宜得多。这样阻塞一个线程并切换到另一个线程是解决一个服务员为多个请求服务问题的一个合理方法。
- 使用线程方案实现服务的优点有二：一是一个输入输出请求仅推迟一个线程而不是整个服务；二是容易实现诸线程共享数据结构。

第九章 分布式文件系统

9.6 安全性

分布式文件系统的安全性还包括保护和加密两个方面：

❖在保护方面，文件系统中的每个文件有一张小表格和文件相联系，在UNIX文件系统中，这个小表格称为inode，记录文件的所有者和分配的磁盘块。每个用户有一个用户标识符(UID)来访问文件。为了访问远程文件，有四种方法：

- (1) 所有远程用户想要访问处理机A上的文件，应该首先用A的本地用户名登录到A。
- (2) 任何用户可以不登录而访问任何处理机上的文件，除了具有对应于来宾(guest)或者其他公共访问名的UID的远程用户。
- (3) 操作系统提供UID间的映射，使本地处理机上UID为a的用户可以用它在远程处理机上的UID b来访问远程处理机。
- (4) 分布式系统设计中，每个用户应当有一个确定且唯一的UID，这个UID在任何处理机上有效而且不需要映射。

第九章 分布式文件系统

9.6 安全性

分布式文件系统的安全性还包括保护和加密两个方面：

❖加密对于在分布式系统中实施安全性是不可缺少的部分，它的主要作用是防止未经授权的数据发布和数据修改。这一机制的关键是协议的握手类型，每一方都要证明自己的身份。当前，多数系统使用下列两种不同的方法之一：

- (1) 一个认证服务器，它物理上安全地维护一个用户口令列表。
- (2) SUN公司的网络文件系统(NFS)使用的公共密钥机制，维护一个以用户口令加密后的确认密钥的公共可读数据库。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖NFS概述

NFS其实是一组协议，该组协议为顾客提供了一个分布式文件系统的模型。所以，运行在不同操作系统和不同机器上的NFS有多种不同的实现，但是这些不同的实现之间具有相互操作的能力，从而NFS可以运行在由异构的计算机连接起来的环境中。

➤NFS体系结构

顾客方：如果安装上NFS，顾客所使用的本地UNIX文件系统接口被虚拟文件系统VFS接口所代替，如果加在VFS接口上的操作请求是一个访问本地文件的操作请求，则该请求被传递到本地文件系统；如果该操作请求是一个访问远程文件的操作请求，NFS顾客把对NFS文件系统的访问操作变换成对文件服务员的远程过程调用。VFS的目的是隐藏不同文件系统之间的差别。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

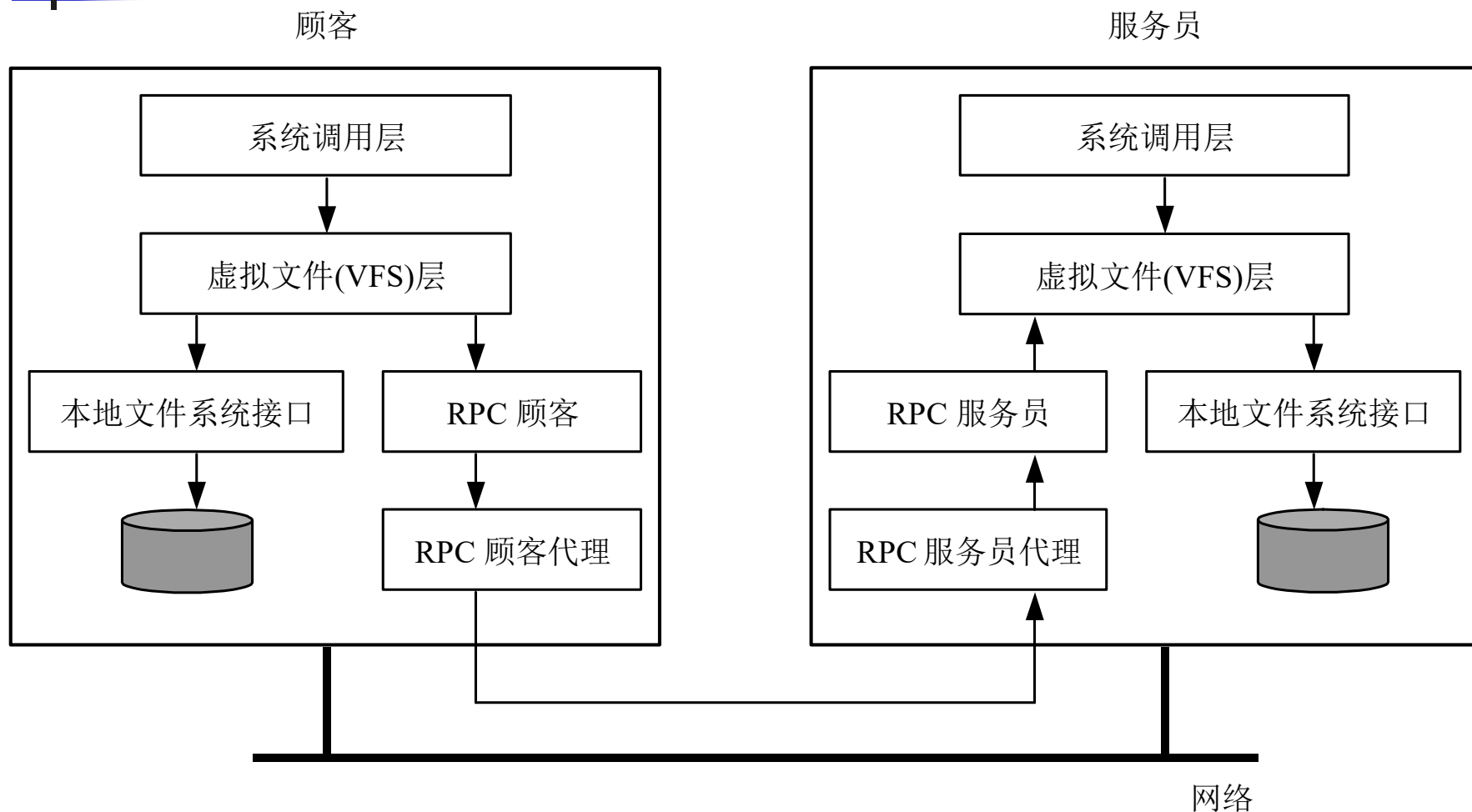
❖NFS概述

➤NFS体系结构

服务员方：服务员一方的结构和顾客方的结构类似，NFS服务员负责处理来自NFS顾客的请求。服务员方的RPC代理(RPC stub)对请求进行拆包，NFS服务员将请求转换成通常的VFS操作，然后传递给VFS层。最后，服务员方的VFS调用本地文件系统接口，在服务员方进行实际的文件操作。

第九章 分布式文件系统

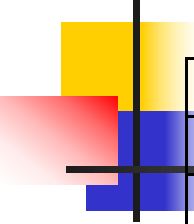
9.7 SUN网络文件系统(NFS)



第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

➤NFS文件系统模型：NFS提供的文件系统模型和UNIX提供的文件系统模型几乎完全相同。



NFS 所支持的主要文件操作

操作	v3	v4	功能描述
create	有	无	创建一个常规文件
create	无	有	创建一个非常规文件
link	有	有	创建一个文件的硬连接
symlink	有	无	创建一个文件的符号连接
mkdir	有	无	在一个给定的目录里创建一个子目录
mknod	有	无	创建一个特殊文件
rename	有	有	改变一个文件的名字
remove	有	有	将一个文件从一个文件系统中删除
rmdir	有	无	从一个目录中删除一个空的子目录
open	无	有	打开一个文件
close	无	有	关闭一个文件
lookup	有	有	通过文件名查找文件
readdir	有	有	读一个目录中的项
readlink	有	有	读一个符号连接中保存的路径名
getattr	有	有	获取一个文件的参数值
setattr	有	有	为一个文件设置参数
read	有	有	从一个文件中读取数据
write	有	有	向一个文件写入数据

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS中的通信

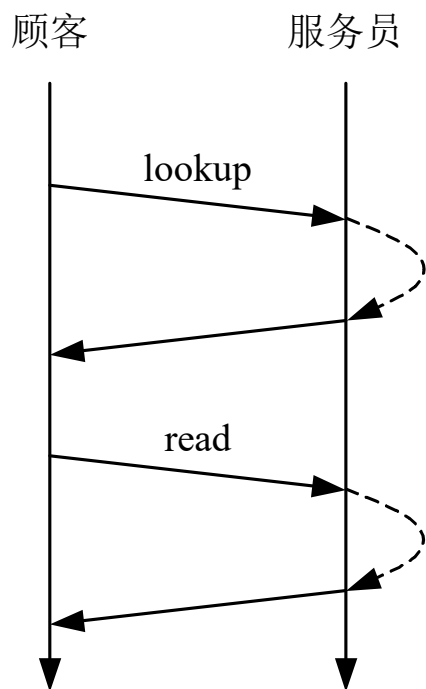
NFS协议建立于RPC层之上，RPC层隐藏了不同操作系统及不同网络之间的差别。NFS v3一个请求对应一个RPC，而NFS v4多个请求可以对应一个RPC。

- 前一种方案中，如果将NFS运行在广域网中，两个以上远程过程调用产生的延迟会导致性能的下降。
- 后一种方案中，将多个远程过程调用组织到一个单一的请求中，成为一个复合过程。NFS v4的复合过程很明显可以减小延迟并且减少网络中的信息传输。

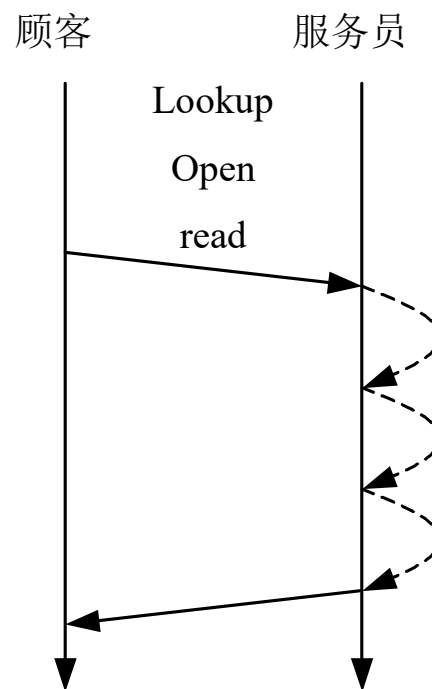
第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS中的通信



(a) NFS v3 中从一个文件中读取数据



(b) NFS v4 中从一个文件中读取数据

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖NFS服务员

NFS v4以前的版本同许多其他分布式文件系统的区别是，其文件服务员是无状态的。但是，NFS v4放弃了这种方案，其服务员是有状态的。

NFS v4服务员采用有状态方案的原因：

- 对文件进行加锁和需要进行访问认证；
- 希望NFS v4运行在广域网上，由于性能上的需要，要求顾客方有效地利用缓存技术，从而需要一个有效的缓存一致性协议。
- NFS v4支持回叫过程(callback procedure)，即服务员可以向顾客发送一个远程过程调用，很明显地，这要求服务员保持对其顾客状态的追踪。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

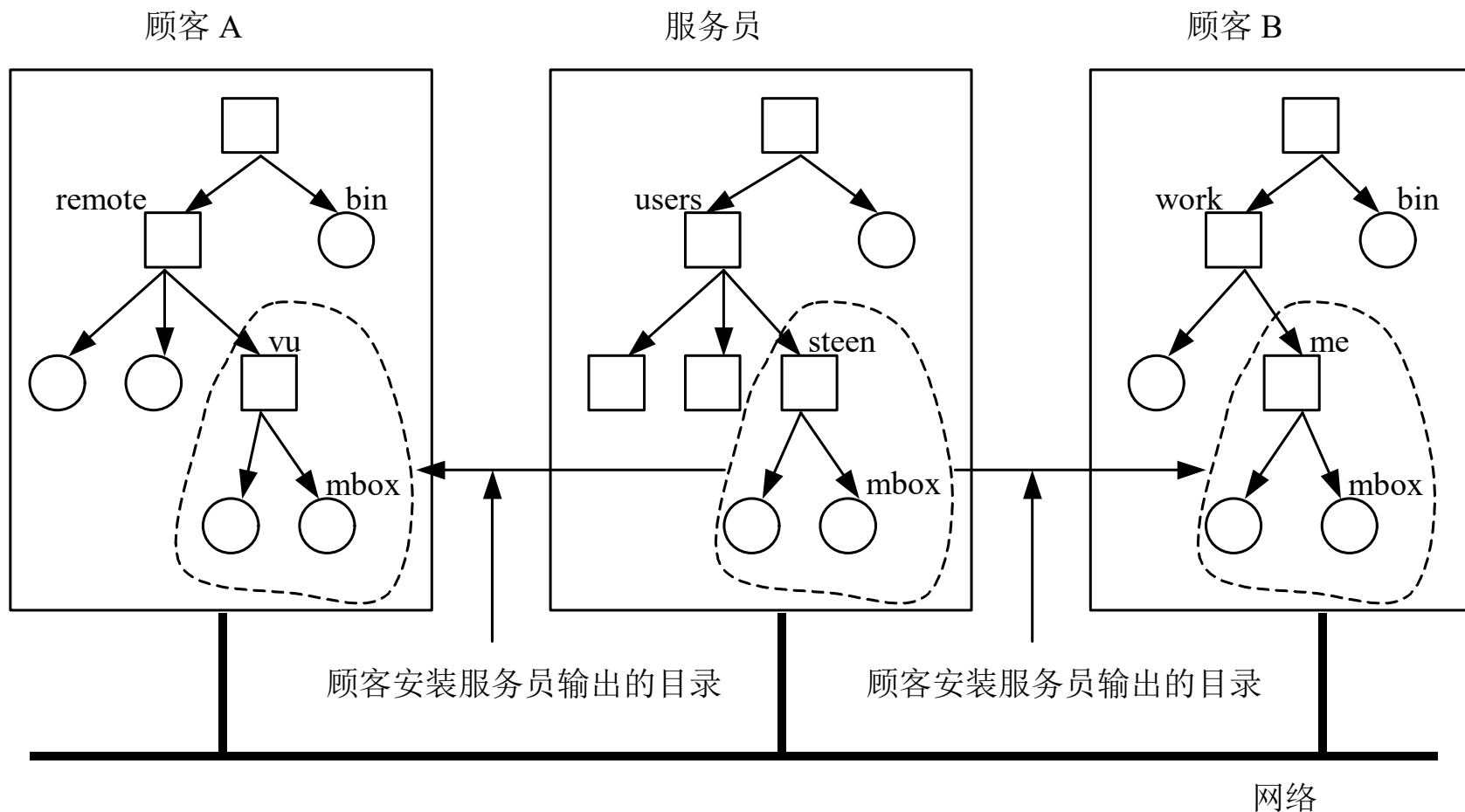
❖ NFS中的命名

➤ NFS的名字空间：

顾客能够将远程的文件系统或其一部分安装到本地文件系统上，通过这种方法可以取得透明性。

第九章 分布式文件系统

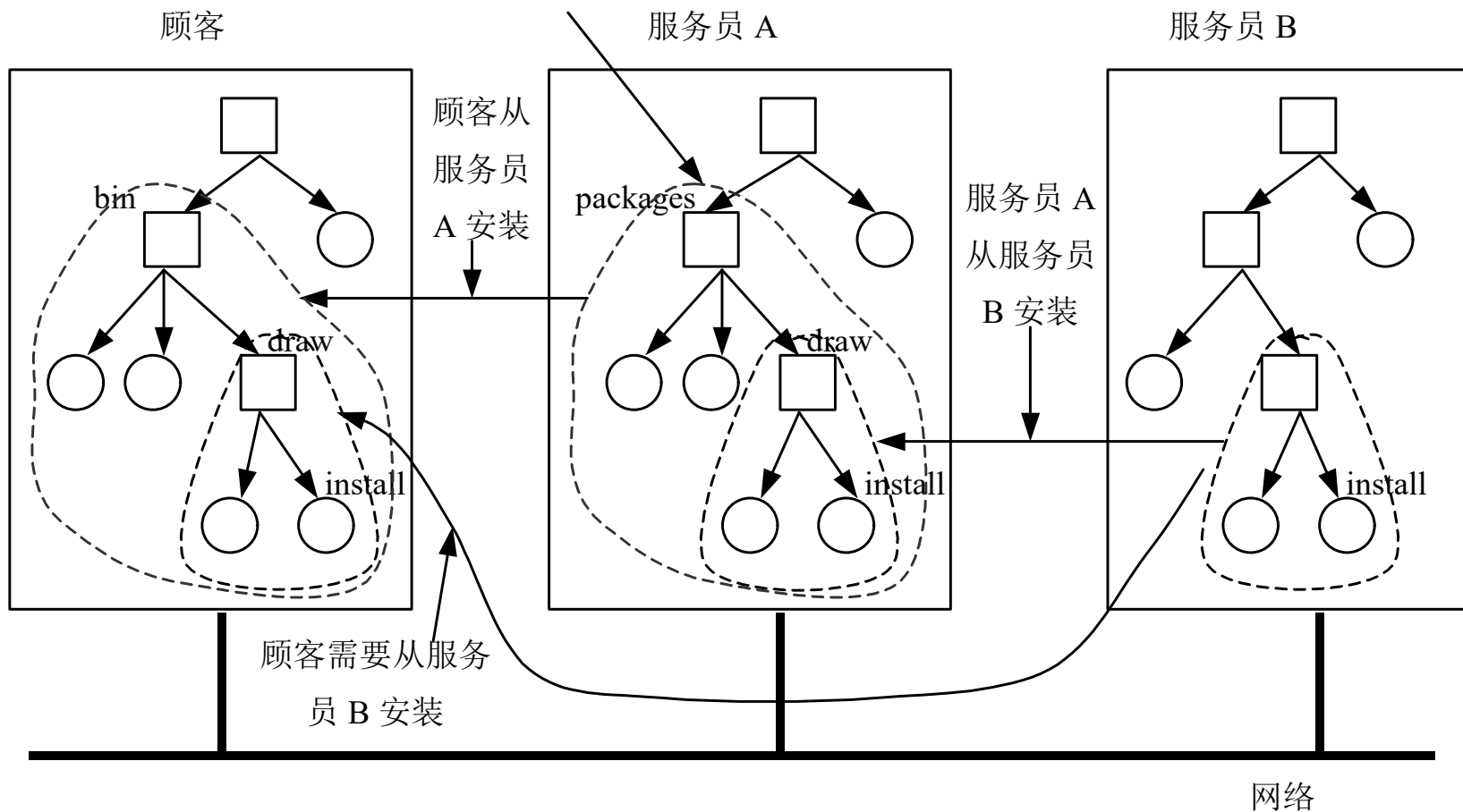
9.7 SUN网络文件系统(NFS)



第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

向顾客输出的目录中包含一个
从服务器 B 安装过来的目录



第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS中的命名

➤ 文件句柄：

- **文件句柄：**文件句柄是文件系统本身用来标识一个文件的，一个文件只有一个文件句柄。只要这个文件存在，它就一直存在。文件句柄的不变特性使得顾客能够使用名字查找该文件并在本地保存文件的文件句柄。

- **顾客在本地保存所访问文件的文件句柄好处：**第一个好处是能够改善性能，对文件进行一系列操作时，可以通过文件句柄对文件进行操作而不是通过名字，这样避免了对文件的每一个访问操作都要通过名字查找文件；另一个好处是对一个文件的访问可以不依赖于它当前的名字。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS中的命名

➤ 文件句柄：

- NFS v3中，顾客可以实际地安装一个远程文件系统，安装完成后，被安装的远程文件系统的根文件句柄被传递给顾客，这个根文件句柄就作为查找远程文件的起点。
- 在NFS v4中，通过一个特殊的操作`putrootfh`告诉服务员对某个文件系统的根进行解析获得其文件句柄，从这个根文件句柄开始，就可以得到对应文件系统中的所有文件的文件句柄。这种方案的好处是不需要一个独立的安装协议，当一个顾客要求服务员对其上的某个文件系统的根对应的名字进行解析时，该文件系统就在顾客方安装了。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖NFS中的命名

➤文件参数:

一个NFS文件有一系列参数与其相对应。在NFS v3中，参数数目是固定的，每个NFS都期望支持这些参数。在NFS v4中，文件参数被分成三组：

- 第一组是强制性的参数，所有的NFS实现必须支持这些参数；
- 第二组是推荐参数，这组参数应该尽量得到支持；
- 第三组参数是只被命名的参数，只被命名的参数随文件一起存储，NFS提供一些操作对这些参数进行读和写，然而这些参数和其对应的值的意义完全留给应用程序去解释。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖NFS中的命名

➤文件参数:

NFS 中一些强制性文件参数

参数	功能描述
TYPE	文件的类型(常规文件、目录、符号连接等)
SIZE	文件的字节数
CHANGE	指出文件是否修改过以及最后一次被修改的时间
FSID	为该文件所属文件系统提供服务的文件服务员的标识符

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖NFS中的命名

➤文件参数:

NFS 中一些推荐的文件参数

参数	功能描述
ACL	文件的访问控制表
FILEHANDLE	服务员提供的该文件的文件句柄
FILEID	该文件的在文件系统中唯一标识符
FS_LOCATIONS	文件所对应的文件系统在网络中的位置
OWNER	文件的属主的名字
TIME_ACCESS	最后一次访问该文件的时间
TIME_MODIFY	最后一次修改该文件的时间
TIME_CREATE	创建该文件的时间

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS中的文件封锁

分布式文件系统中的文件一般来说需要被多个顾客共享，文件的共享需要有一致性的同步机构。

- 在NFS v4以前的版本中，文件封锁是由一个附加的独立协议处理的，在NFS协议之外提供了一个有状态的锁管理员。
- 在NFS v4中，文件封锁被集成到NFS的文件访问协议中，顾客使用这种封锁机制要比使用NFS v3中的封锁机制要简单。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖NFS中的文件封锁

NFS v4 中文件封锁的操作类型

操作	功能描述
lock	为文件的某一部分创建一个锁
lockt	检测是否存在冲突的锁
locku	释放一个锁
renew	对一个特定的锁进行更新，延长加锁的时间

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

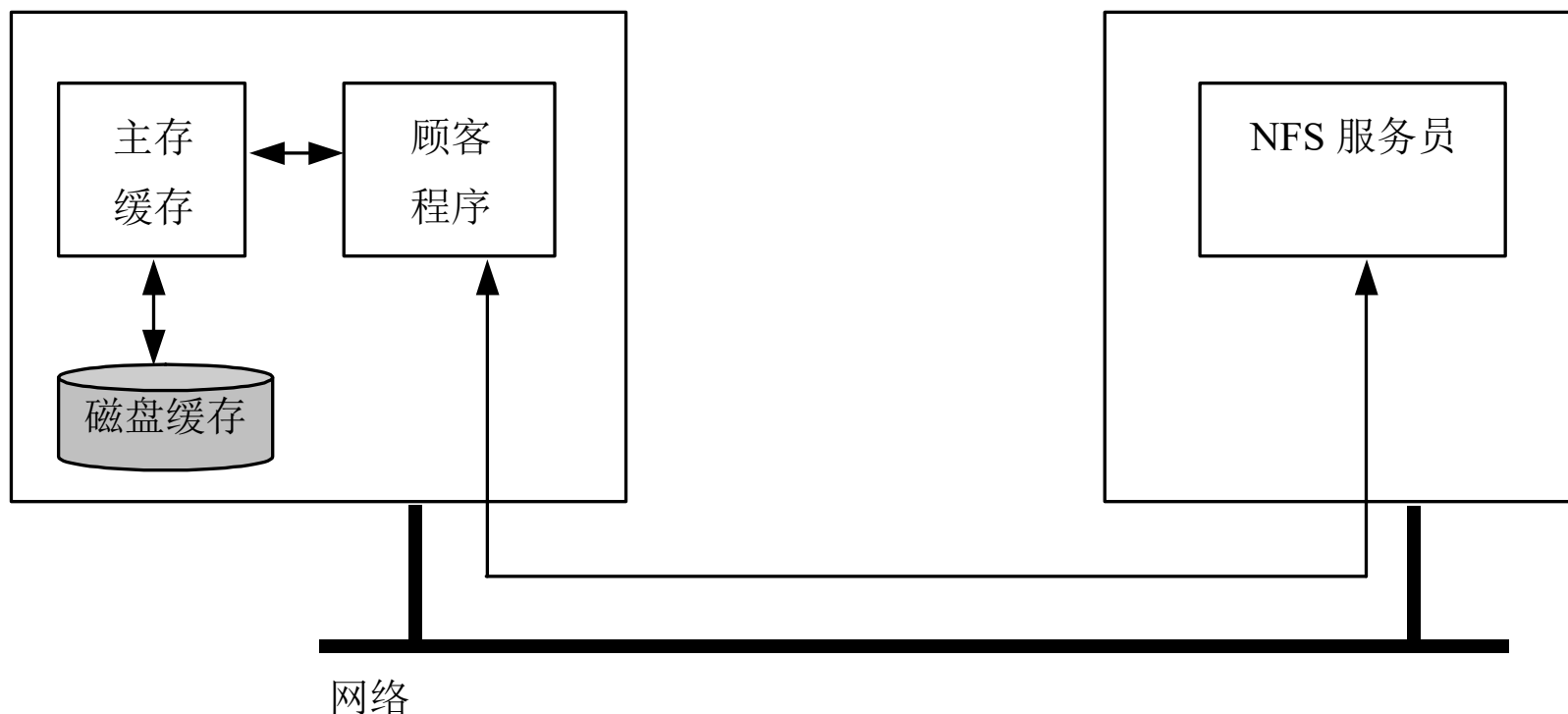
❖缓存和复制

- NFS v3中的缓存主要留在NFS协议之外实现，这种实现方式会导致使用不同的缓存策略，这些策略大部分不保证缓存一致性。
- NFS v4某种程度上解决了一致性问题。每个顾客有一个主存缓存，用来保存以前从服务员处读得的数据，另外顾客方还有一个磁盘缓存用于对主存缓存进行扩充。
- 缓存的内容：一般来说，顾客方会对文件的数据、参数、文件句柄等进行缓存。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖缓存和复制



第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖缓存和复制

NFS v4的两种缓存方案：简单方案和委派方案

➤**简单方案：**当顾客打开一个文件时，就将该文件的数据取到本地的缓存中，然后对该文件的读和写操作就在该缓存中进行。当关闭该文件时，缓存中被修改过的数据要被传回给文件服务员，对文件服务员上文件的原本进行刷新。NFS要求无论何时当一个顾客打开以前关闭的文件，而且这个文件在顾客方被缓存，那么顾客需要马上激活缓存中的数据。在激活缓存中的数据时，需要检查缓存中的数据是否是最新的，当缓存中的数据是过时的就使其无效。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖缓存和复制

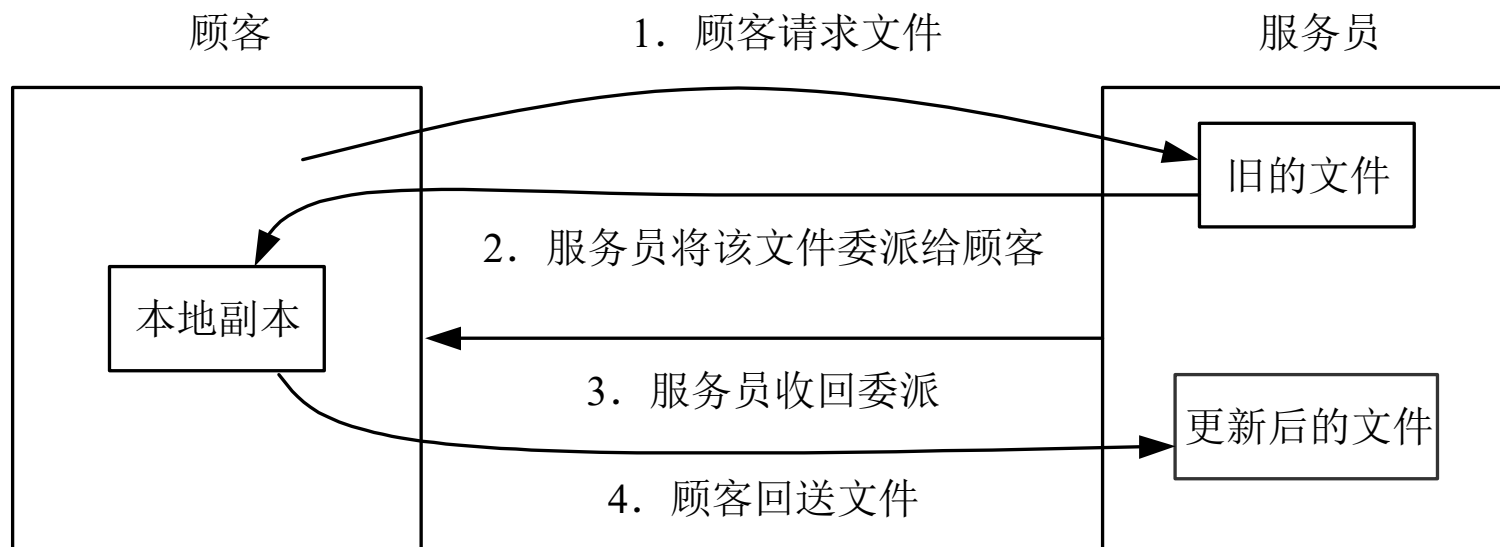
NFS v4的两种缓存方案：简单方案和委派方案

➤**委派方案：**允许NFS服务员将它的一部分权力委派给顾客。如果一个服务员将一个打开文件的权利委派给某个机器上的一个顾客，并且这个顾客得到了写权限，那么来自同一个机器上的其他顾客的文件加锁请求可以在本地得到处理。很明显服务员应该在随后能够对其委派出的权利进行收回。收回委派的权利要求服务员能够向顾客发送反向的调用。由于反向调用要求服务员能够追踪那些它向其委派过文件的顾客，所以在这里NFS服务员不再是无状态的。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ 缓存和复制



第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖缓存和复制

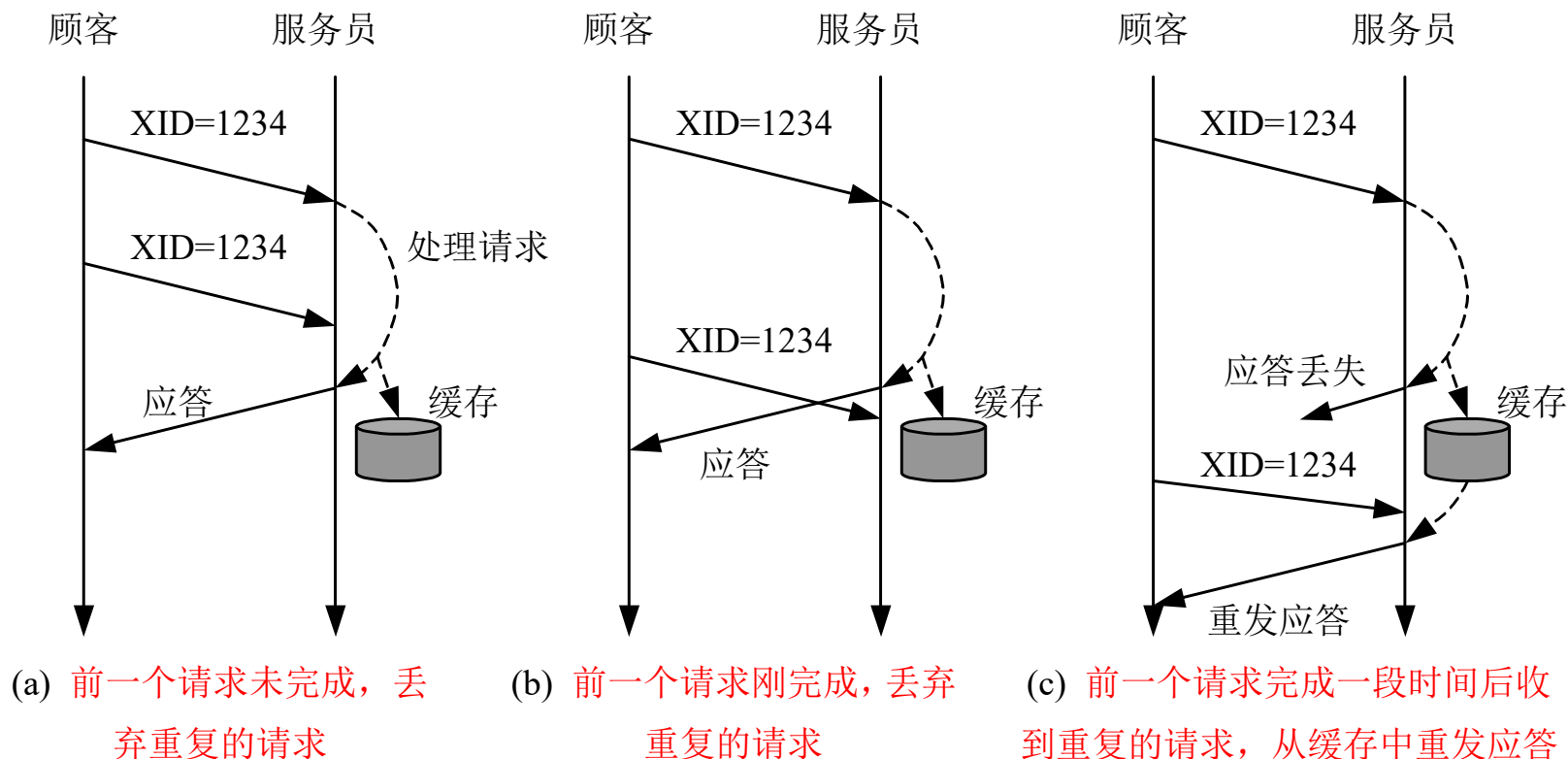
复制：NFS v4最低限度地支持了文件复制，在NFS v4中，复制的单位是整个文件系统。文件复制是以参数FS_LOCATIONS的形式得到支持的，每个文件有一个参数FS_LOCATIONS，这个参数指出了包含此文件的文件系统在哪些地方有副本，该参数的值是一个位置表，表中的每个位置是一个DNS名字或者是一个IP地址。文件复制由特定的NFS实现提供实际的支持，NFS v4并未规定怎样实现文件复制。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS中的容错

➤ RPC失效



第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS中的容错

➤ 存在错误情况下的文件封锁

解决顾客失效的问题：服务员对它分配给顾客的每个锁设定了一个有效的期限，当超过这个期限时，服务员会删除这个锁并释放对应的资源。顾客为了保证服务员不删除它的锁，顾客需要在期限到达之前执行renew操作，通知服务员继续为该顾客保持锁。如果顾客失效了，一段时间之后，服务员就会删除顾客的锁并释放对应的资源。

解决服务员失效的问题：在NFS v4中，当NFS服务员恢复后，设定了一个宽限期，在这个宽限期内，每个顾客会向服务员重新声明它以前所获得的所有锁，通过这种方式，服务员会重新建立起以前的关于锁的状态。在这个宽限期内，服务员只接受来自顾客的重新声明的请求，会拒绝顾客的普通的申请锁的请求。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS中的容错

➤ 存在错误情况下的打开委派

- 当这个顾客失效时，顾客可能还没有来得及将它对文件的更新传播给服务员。在这种情况下，如果顾客没有将更新存放在本地的坚固存储器中，文件的全部恢复是不可能的，所以顾客只部分地负责文件恢复。
- 当服务员失效恢复后，顾客需要向服务员重新申明所委派给它的文件。不过这时服务员先强迫顾客向服务员传送对文件的最新更新，刷新服务员上文件的原本，即先收回对文件的委派，然后再向顾客委派该文件。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖NFS的安全性

NFS的安全性主要表现在两个方面，一个方面是通信的安全性，另一个方面是服务员需要验证顾客的访问权限。

➤安全的RPC

NFS的RPC仅提供验证机制，实现验证机制的方法有三种：

- 第一种是最广泛使用的一种方法，但是它的安全性很有限。顾客仅仅简单地将其有效用户ID和用户组ID以明文的形式传送给服务员。很显然，服务员并不验证这些信息是否确实和实际的请求者对应。
- 第二种是使用Diffie-Hellman密钥交换来建立一个对话密钥，是一种公开密钥加密系统，这种方法使用在NFS v4以前的版本中，从而导致出现了安全的NFS。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS的安全性

➤ 安全的RPC

NFS的RPC仅提供验证机制，实现验证机制的方法有三种：

- 第三种是NFS v4中使用的方法，在NFS v4中安全性是由RPCSEC_GSS支持的。RPCSEC_GSS是一种通用的安全性框架，它可以支持使用多种安全机制来建立安全通信通道。所以，RPCSEC_GSS不仅可以用来提供多种不同的认证系统，还可以支持报文的完整性和机密性，这些特点在旧的NFS版本中是不支持的。

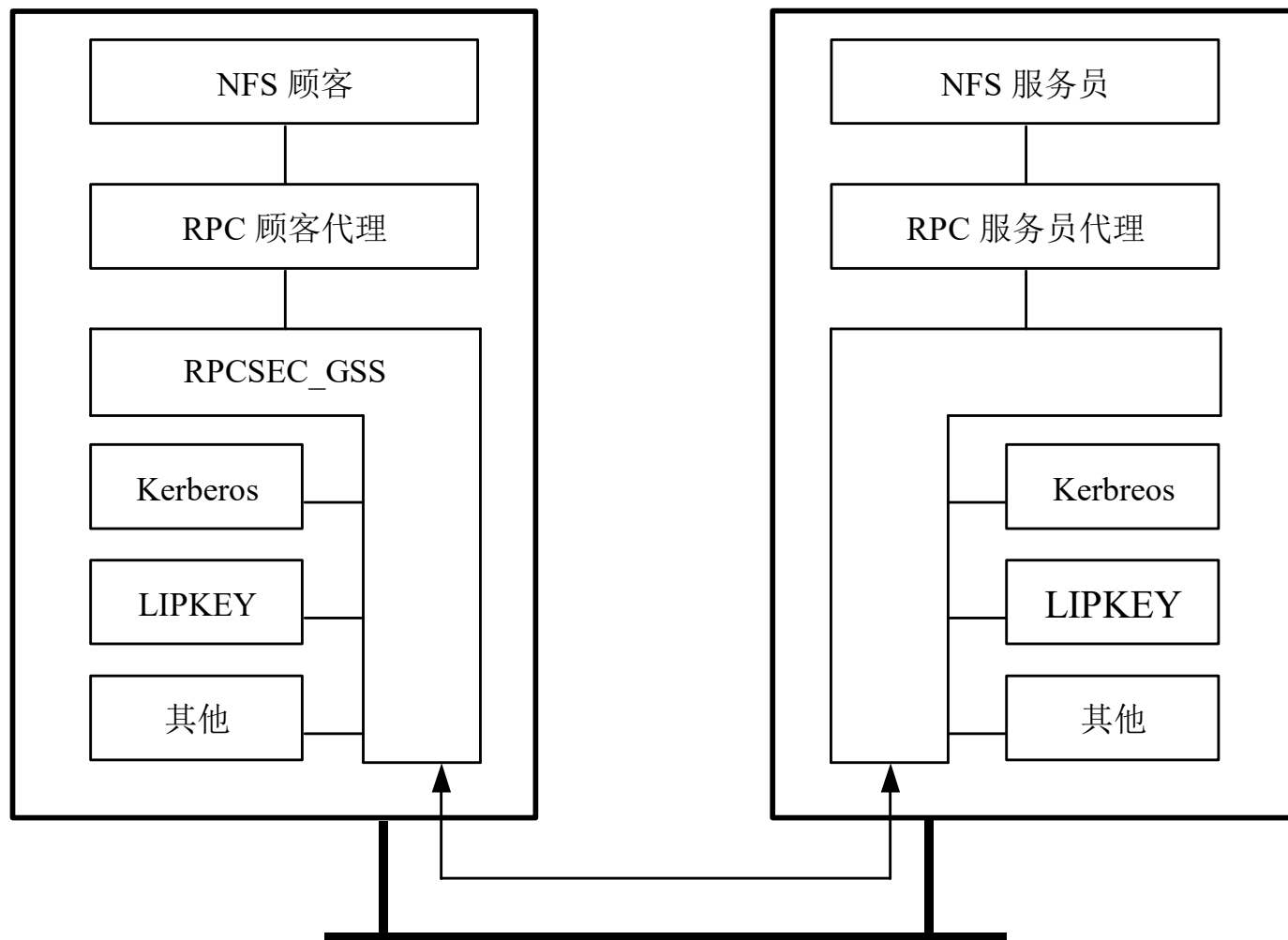
第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

顾客机

服务器机

❖NFS的安全性



第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

❖ NFS的安全性

➤ 访问控制

- 对访问控制的支持是由文件的ACL参数来支持的。访问控制参数说明了特定的用户或用户组对一个文件具有哪些访问权限。
- NFS定义了许多类型的用户或进程，NFS对不同的用户会给出不同的访问控制权限。

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

NFS 中的访问控制

操作	功能描述
Read_data	允许读文件的数据
Write_data	允许修改文件的数据
Append_data	允许向文件尾追加数据
Excute	允许执行文件
List_directory	允许列出目录的内容
Add_file	允许在一个目录中增加一个新文件
Add_subdirectory	允许在一个目录中创建一个子目录
Delete	允许删除一个文件
Delete_child	允许在一个目录中删除一个文件或者一个目录
Read_acl	允许读一个文件的访问控制表
Write_acl	允许修改一个文件的访问控制表
Read_attributes	允许读文件的其他参数
Write_attributes	允许修改文件的其他参数
Read_named_attrs	允许读文件的只被命名的参数
Write_named_attrs	允许写文件的只被命名的参数
Write_owner	允许改变文件的属主
Synchronize	允许在文件服务员上以同步的读或者写直接访问文件

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

NFS 中的访问控制

操作	功能描述
Read_data	允许读文件的数据
Write_data	允许修改文件的数据
Append_data	允许向文件尾追加数据
Excute	允许执行文件
List_directory	允许列出目录的内容
Add_file	允许在一个目录中增加一个新文件
Add_subdirectory	允许在一个目录中创建一个子目录
Delete	允许删除一个文件
Delete_child	允许在一个目录中删除一个文件或者一个目录
Read_acl	允许读一个文件的访问控制表
Write_acl	允许修改一个文件的访问控制表
Read_attributes	允许读文件的其他参数
Write_attributes	允许修改文件的其他参数
Read_named_attrs	允许读文件的只被命名的参数
Write_named_attrs	允许写文件的只被命名的参数
Write_owner	允许改变文件的属主
Synchronize	允许在文件服务员上以同步的读或者写直接访问文件

第九章 分布式文件系统

9.7 SUN网络文件系统(NFS)

NFS 所定义的用户类型

用户类型	功能描述
Owner	文件的属主
Group	与文件对应的用户组
Everyone	任何用户或进程
Interactive	来自于一个交互终端上的访问用户或进程
Network	通过网络访问文件的进程或用户
Dialup	通过拨号连接到服务器上访问文件的进程或用户
Batch	批处理作业中的一个访问文件的进程
Anonymous	无需通过认证访问文件的用户
Authenticated	任何通过认证的用户或进程
Service	系统定义的服务进程

第九章 分布式文件系统

9.8 其他的分布式文件系统及其比较

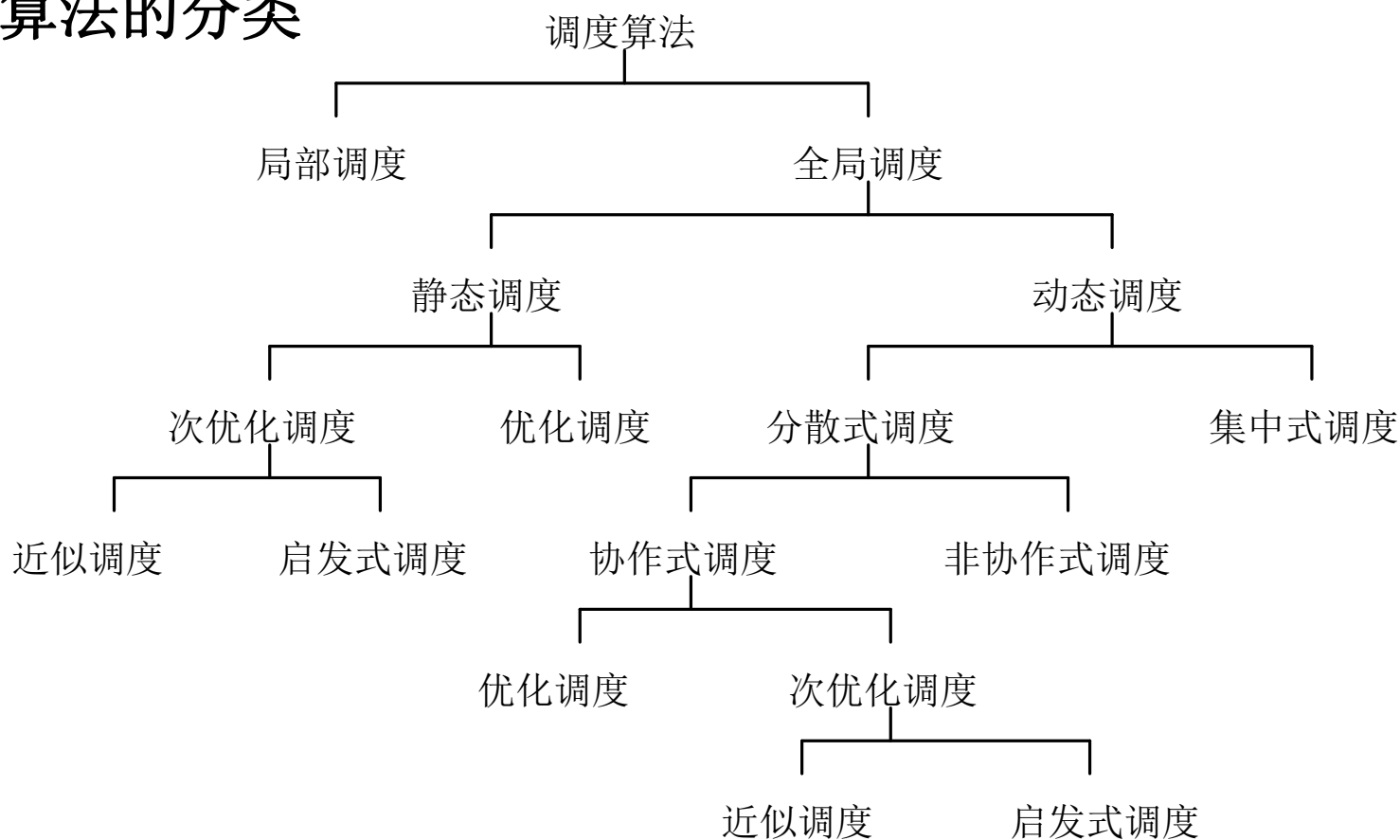
不同的分布式文件的比较[TANENBAUM,2002]

比较内容	NFS	Coda	Plan 9	XFS	SFS
设计目标	访问透明性	高可用性	资源访问统一性	无 服 务 员 系 统	可扩充的安全性
访问模式	远程访问	上载/下载	远程访问	写运行记录	远程访问
通信机制	RPC	RPC	定制的通信协议	主动报文	RPC
顾客进程	瘦/胖	胖	瘦	胖	中等的
服务员组	无	有	无	有	无
安装粒度	目录	文件系统	文件系统	文件系统	目录
名字空间	每个用户	全局	每个进程	全局	全局
文件标识符	服务员 范 围 唯 一	全局唯一	服务员范围唯一	全局唯一	文件系统范围唯 一
共享语义	对话语义	事务处理语义	UNIX 语义	UNIX 语义	未规定
缓存单位	文件(NFSv4)	文件	文件	文件块	未规定
缓存一致性	写回	写回	立即写	写回	写回
文件复制	最小限度	ROWA	无	服务员组支 持	无
容错	可靠通信	复制和缓存	可靠通信	服务员组支 持	可靠通信
恢复	基于顾客	服务员组重新整 合	未规定	检 查 点 和 写 运行记录	未规定
安全通信通 道	使用 现 有 任 何 机制	Needham- Schroeder	Needham- Schroeder	无	自认证的路径名
访问控制	提供许多操作	针对文件操作	基于 UNIX 的	基于 UNIX 的	基于 NFS 的

第十章 分布式调度

10.1 调度算法概述

❖ 调度算法的分类



第十章 分布式调度

10.1 调度算法概述

❖ 调度算法的分类

其他一些分类方法：

(1) 非抢先式的(non-preemptive)和抢先式的(preemptive)。对非抢先式的调度算法，一个进程开始执行后就不能中断。在抢先式调度算法中，进程可以中断，从一个处理机上移走，到另一个处理机上继续执行。

(2) 适应性(adaptive)和非适应性的(non-adaptive)。非适应性调度算法只使用一种负载分配策略，不会根据系统的反馈而改变自己的行为。适应性调度算法能够根据系统的反馈调整自己的行为，采用不同的负载分配策略。典型地，一个适应性调度算法是许多种调度算法的集合，根据系统的各种参数来选择一种合适的算法。

第十章 分布式调度

10.1 调度算法概述

❖ 调度算法的目标和有效性评价

分布式调度的基本目标是尽快得到计算结果和有效地利用资源。具体来说，调度算法的目标有两个：

一个目标是负载平衡(load balancing)，它的努力目标是维持整个分布式系统中各个资源上的负载大致相同。另一种目标是负载共享(load sharing)，它的目标仅仅是防止某个处理机上的负载过重。相对来说负载共享的目标要比负载平衡的目标容易达到。负载平衡的主要目的是提高整个系统的流量，而负载共享的主要目标是缩短特定程序的执行时间。

第十章 分布式调度

10.1 调度算法概述

❖ 调度算法的目标和有效性评价

从调度算法的有效性来看，调度算法分为最优调度算法和次优调度算法。为了实现最优调度算法，调度者必须获得所有进程的状态信息和系统中所有相关的可用信息。最优性常用执行时间、资源利用率、系统流量以及这些参数的某种综合来进行评价。一般来说最优调度是一个NP完全性问题。所以在实际的系统中，常采用次优的调度算法。

第十章 分布式调度

10.1 调度算法概述

❖调度算法的目标和有效性评价

有许多参数用于确定或测量一个调度算法的有效性：

- **通信代价：**使用这个参数的调度算法可能要考虑到向一个给定的节点传送或者从一个给定节点接收一个报文花费的时间，更为重要的是必须考虑到为一个进程分配一个执行地点而引起的通信代价。
- **执行代价：**这个参数反映的是将一个进程分配到一个指定的执行节点，在这个节点的执行环境下，执行这个程序所需的额外开销。
- **资源利用率：**常用来表明基于分布式系统当前各个节点的负载情况，给一个进程分配的执行节点是否是合适的。资源利用率参数常用负载状态来表示，常用的负载参数有资源的队列长度、内存的使用等等。

第十章 分布式调度

10.1 调度算法概述

❖ 调度算法的目标和有效性评价

次优的调度算法分为两类：近似的次优调度算法和启发式的次优调度算法：

- 近似的次优调度常和最优调度使用相同的算法，但是近似的次优调度不搜索这个算法的所有解空间，而是在这个算法的解空间中的一个子集中搜索，目的是尽快地找到一个较好的解。而最优调度则是搜索这个算法的整个解空间，目的是获得最好的解。使用近似的次优调度算法必须能够判定所得到的解是否可以被接受的，也就是说，必须能够确定最优解和次优解之间的近似程度。

第十章 分布式调度

10.1 调度算法概述

❖ 调度算法的目标和有效性评价

次优的调度算法分为两类：近似的次优调度算法和启发式的次优调度算法：

- 启发式的次优调度算法常使用比较简明的规则和一些直觉的规则来进行调度。这些启发式的规则往往是不能证明其正确性，在特定情况下可能还是错误的，但是在绝大多数情况下是能够被接受的。

第十章 分布式调度

10.1 调度算法概述

❖ 调度算法的目标和有效性评价

启发式调度算法中常采用的一些启发式规则：

- 相互依赖性较大的进程，由于它们之间常有比较多的进程通信应该分配到比较接近的执行节点上，可能的话，应该在同一个节点上。
- 访问共享文件的进程应该分配到比较接近的执行节点上，可能的话，应该分配在文件服务员节点上。
- 很少有内在关系的进程可以分布在不同的机器上。
- 如果一个节点已经是重负载的，不应该向该节点分配另外一个进程。

第十章 分布式调度

10.2 静态调度

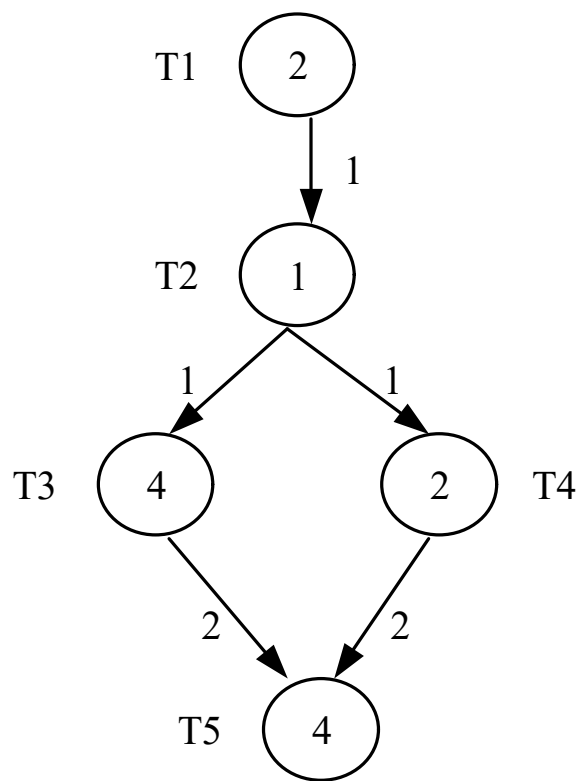
➤设计调度策略时要考虑的三个主要因素：静态调度算法的目标是调度一个任务集合，使它们在各个目标节点上有最短的执行时间。总体上来说，设计调度策略时要考虑的三个主要因素是处理机的互连、任务的划分和任务的分配。通常用图模型表示任务和处理机的结构。我们可以用任务优先图和任务交互作用图对任务集合建模。

➤任务优先图是一个有向无环图(DAG)，图中每个链接定义了任务间的优先关系，节点和链接上的标记表示任务的执行时间和任务完成后启动后续任务所需的时间间隔。

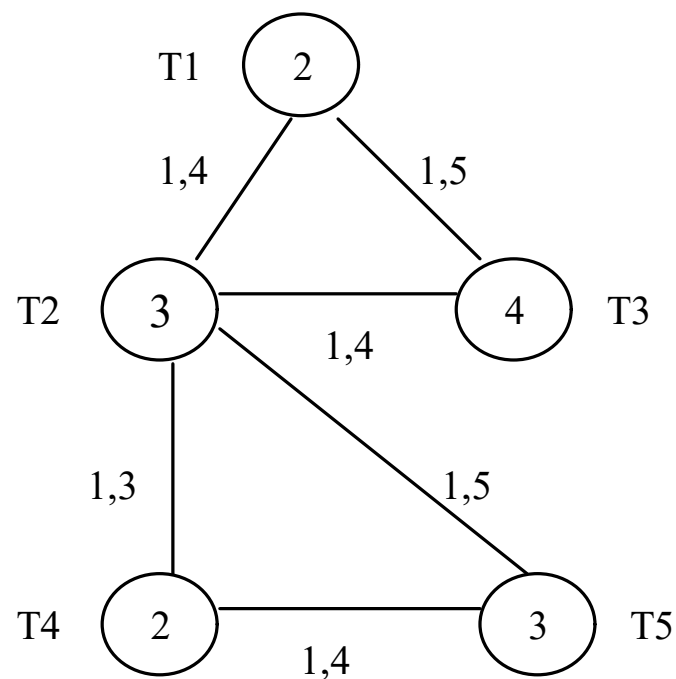
➤任务交互作用图中，链接定义了两个任务间的相互关系，每个链接赋予一对数，分别表示这两个任务在同一个处理机上时的通信开销和在不同处理机上时的通信开销。

第十章 分布式调度

10.2 静态调度



(a) 任务优先图



(b) 任务相互作用图

第十章 分布式调度

10.2 静态调度

❖任务划分与分配

➤**任务划分的粒度：**一个给定任务划分的粒度被定义为任务的计算量与通信量的比值。如果粒度太大，就会限制并行性，因为潜在的并行任务可能被划分进同一个任务而分配给一个处理器。粒度太小，进程切换和通信的开销就会增加，从而降低性能。

➤**任务聚类：**在图模型中，任务的划分被称作任务聚类，即在给定的图模型中对小任务进行分类。任务划分把任务图当作一个整体，将图中的小任务(节点)划分成不同的聚类，聚类中的小任务串行执行，不同的聚类之间并行执行。任务聚类中可以使用两种策略：

- (1) 将不相关的任务映射到一个聚类中；
- (2) 将DAG中一条优先路径上的任务映射到一个聚类中。

第十章 分布式调度

10.2 静态调度

❖任务划分与分配

一些划分算法：

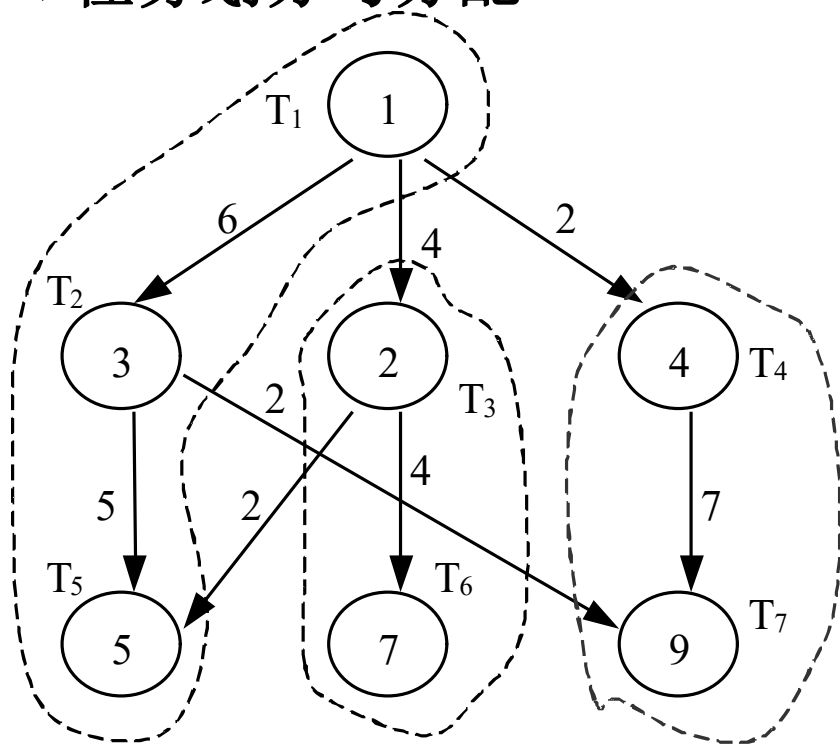
(1) **关键路径划分**。关键路径(最长路径)的概念常常在垂直划分中使用，即用在线性聚类中。应该清楚的是，依赖于任务优先图中关键路径的细粒度任务必须串行执行。

(2) **消除通信延迟的划分**。这个方法的关键之处在于消除通信的额外开销，所以要把通信频繁的节点聚集成一类。通常的方法是将一个节点的后继节点与节点自身聚集成一类，只要总的执行时间不会被延长。

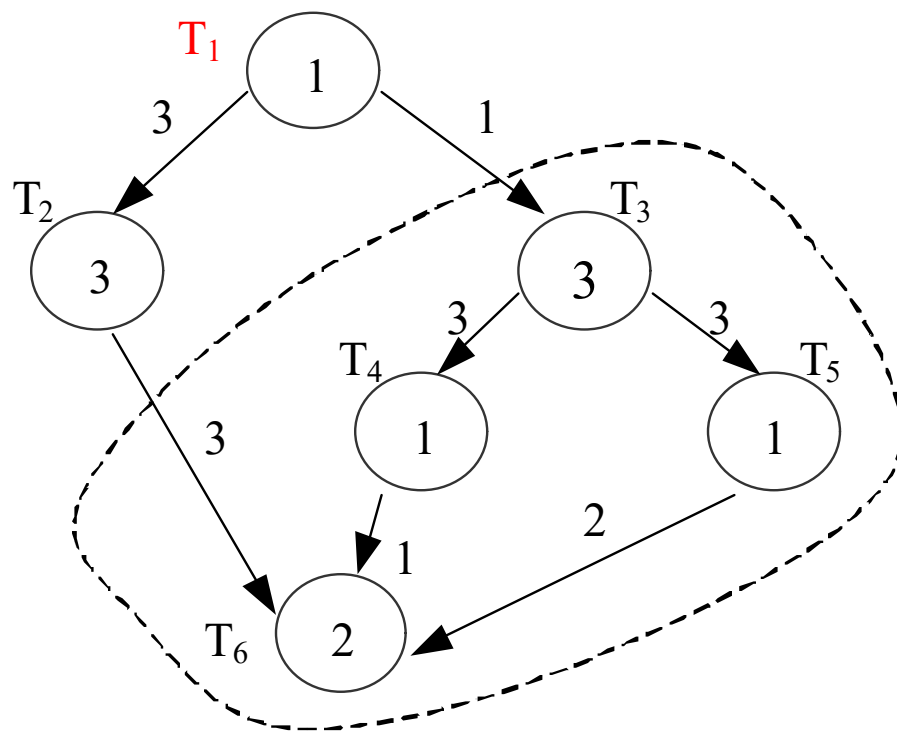
第十章 分布式调度

10.2 静态调度

❖ 任务划分与分配



关键路径划分的例子



消除通信延迟的划分

第十章 分布式调度

10.2 静态调度

❖任务划分与分配

一些划分算法：

(3) **任务复制**。为了消除任务间的通信开销，将任务在处理器上进行复制有时是最有效的方法。它是任务划分的一个可选方法。任务复制不仅能保留程序最初的并行性，同时也能减少通信开销。

(4) **其他技术**。Kim和Browne的线性聚类技术，在每一步，计算量和通信量最大的有向路径上的节点聚集成一个单独的线性聚类，并且这些节点被从图中除去。对图中剩余的节点迭代执行这个过程，直到整个任务图已经全部被划分成一些聚类。Sarkar的内在化聚类方案，将每个节点最初放在一个单独的聚类中，并且以弧上通信开销的下降顺序考虑将图中的节点划分成一些聚类。这个算法不断地将两个聚类合并成一个更大的聚类，如果在合并过程中生成的更大聚类不会增加这个图的估计并行执行时间，那么这个合并过程就被接受。这个过程一直进行下去，直到不再需要合并为止。

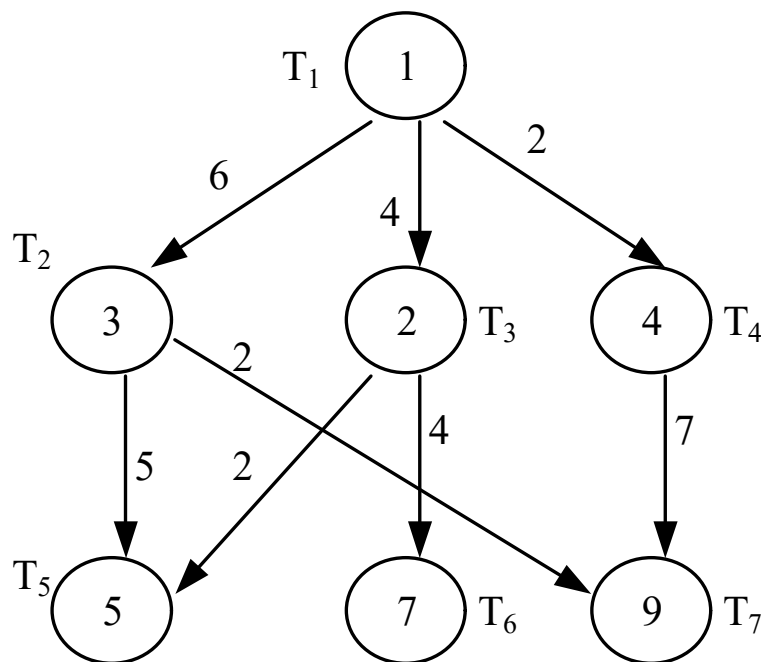
第十章 分布式调度

10.2 静态调度

❖任务划分与分配

任务复制:

时间	处理机 1	处理机 2	处理机 3
1	T ₁	T ₁	T ₁
2	T ₂	T ₃	T ₄
4	T ₂	T ₂	T ₄
5	T ₃	T ₂	T ₄
6	T ₃	T ₂	T ₂
7	T ₅	T ₆	T ₂
9	T ₅	T ₆	T ₇



第十章 分布式调度

10.2 静态调度

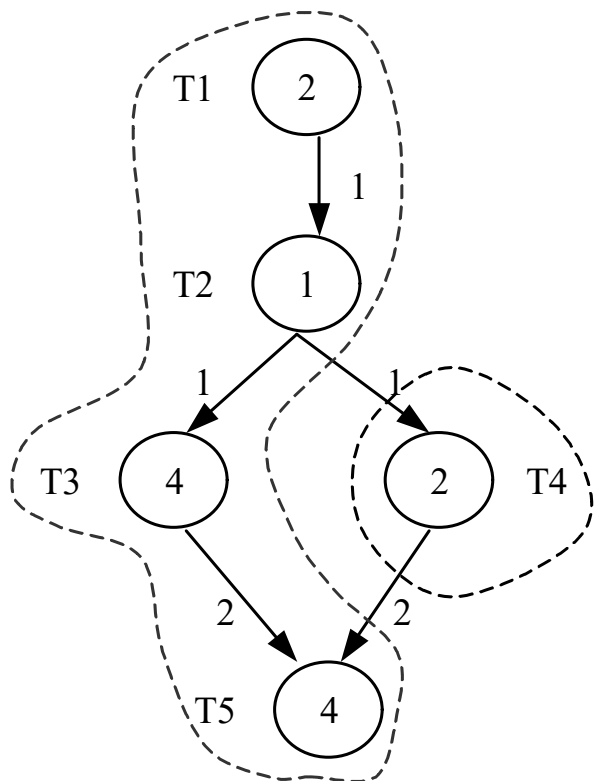
❖ 基于任务优先图的任务调度

甘特图 (gantt chart) 能够最有效描述进程对处理器的分配情况。甘特图以处理器为纵坐标，以时间为横坐标。图中的每个方块表示进程在某个系统中的开始时间、持续时间和结束时间。处理器内的时间延迟和处理器间的时间延迟都能够在图中体现。

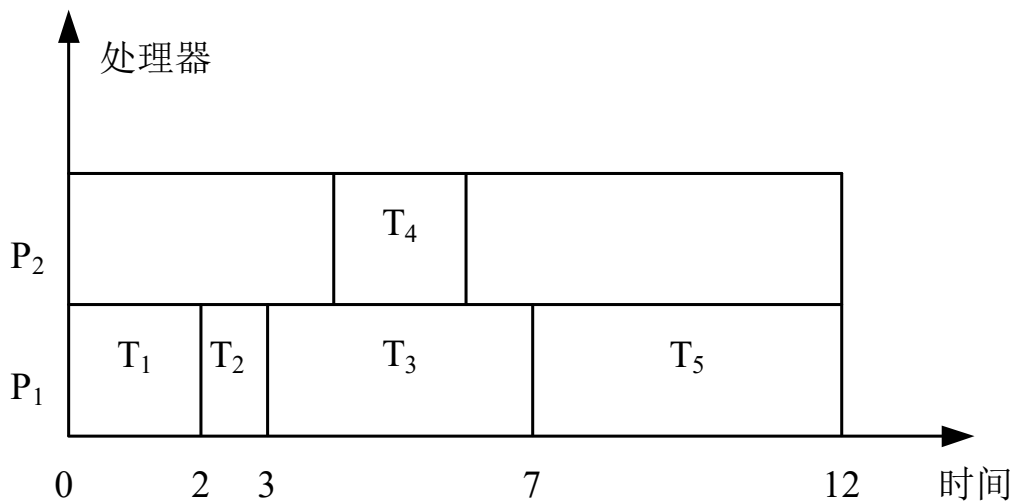
第十章 分布式调度

10.2 静态调度

❖ 基于任务优先图的任务调度



(a) 任务优先图



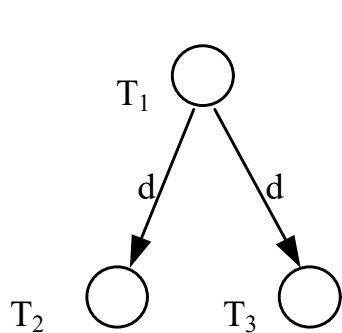
(b) 甘特图

第十章 分布式调度

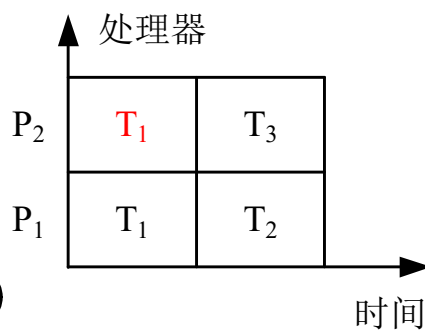
10.2 静态调度

❖ 基于任务优先图的任务调度

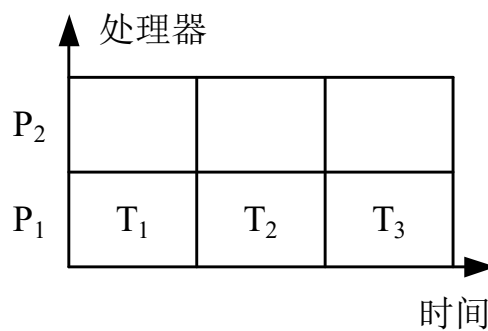
通信延迟和任务复制对调度的影响：



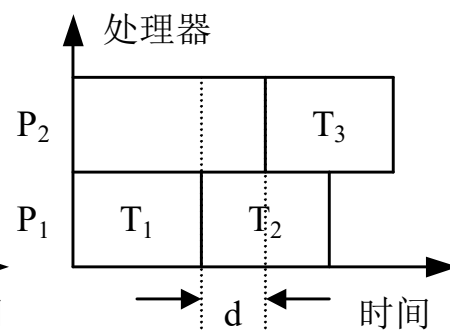
(a) 任务优先图



(b) 使用任务复制的调度



(c) 任务分配在一个处理器上



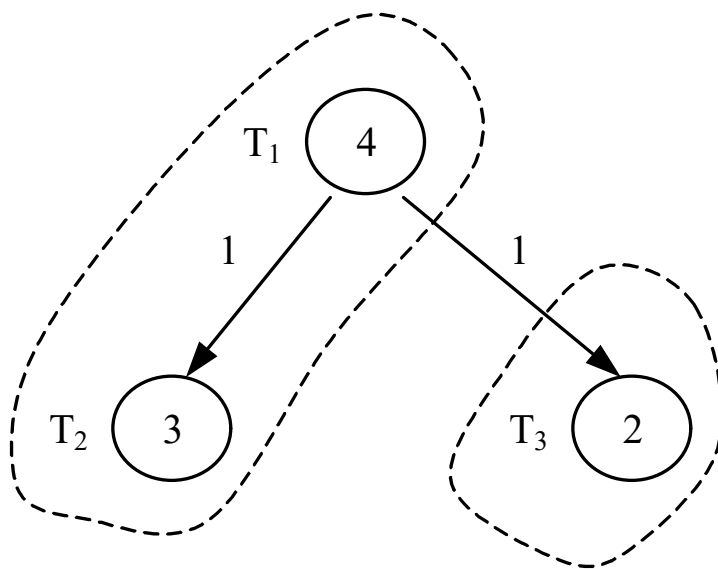
(d) 通信延迟对调度的影响

第十章 分布式调度

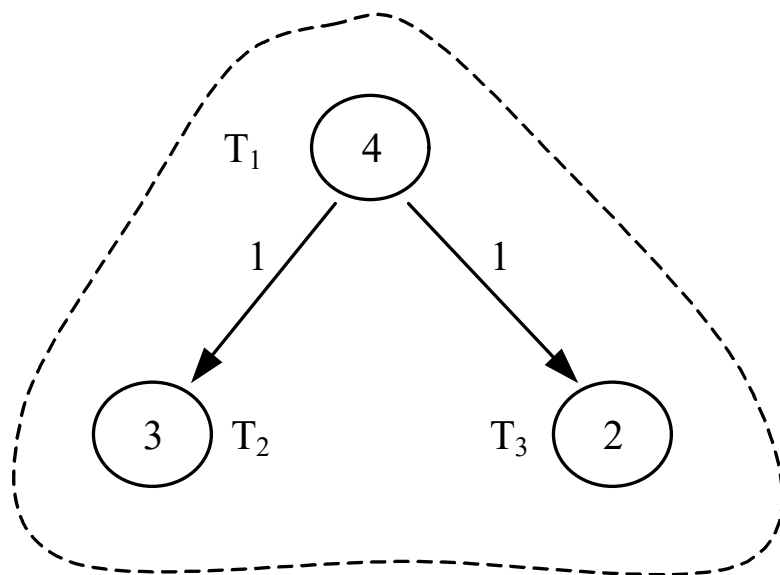
10.2 静态调度

❖ 基于任务优先图的任务调度

线性聚类与非线性聚类： 如果至少有一个聚类中包含两个独立的任务，则聚类是非线性的；否则，聚类就是线性的。



(a) 线性聚类



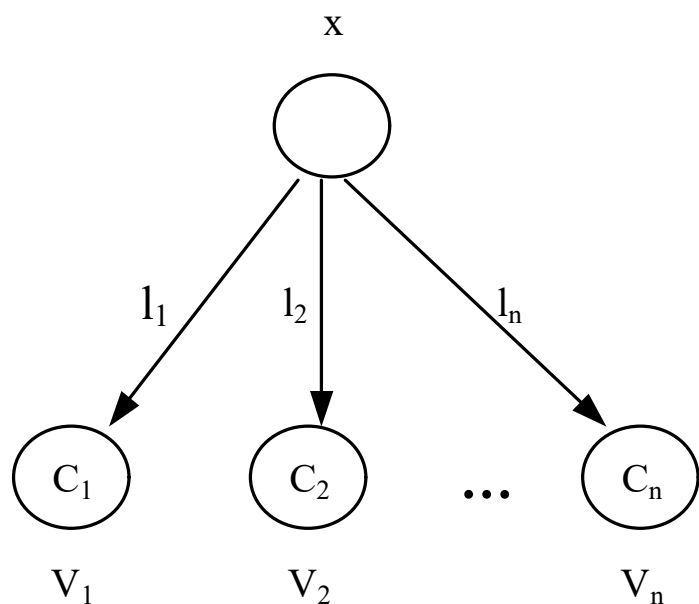
(b) 非线性聚类

第十章 分布式调度

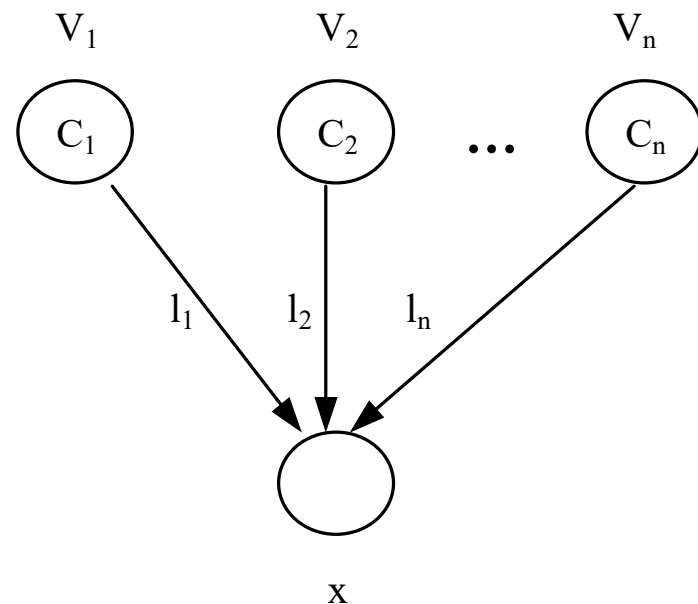
10.2 静态调度

❖ 基于任务优先图的任务调度

一个任务优先图可以认为是许多分叉和合并操作的集合，分叉 x (合并 x) 的粒度是：
$$g(x) = \min_{1 \leq k \leq n} \{C_k\} / \max_{1 \leq k \leq n} \{l_k\}$$



(a) 分叉操作



(b) 合并操作

第十章 分布式调度

10.2 静态调度

❖ 基于任务优先图的任务调度

给定任务优先图 G 的粒度是：
$$g(G) = \min_{\forall x \in G} \{g(x)\}$$

如果 $g(x) > 1$ ，合并 x 或分叉 x 就是粗粒度；否则就是细粒度。同样如果 $g(G) > 1$ ，图 G 就是粗粒度，否则就是细粒度。当表示一个应用程序的给定的有向无环图DAG(任务优先图)是粗粒度时，也就是它的一个链接上的通信代价小于分叉或者合并操作连接的相邻节点的计算代价，任何非线性聚类可以被转换成具有更少或相等执行时间的线性聚类。注意，上面的结论暗示了一个粗粒度程序的线性聚类性能优于任何非线性聚类。然而，对细粒度程序而言，可能存在也可能不存在一个非线性聚类优于线性聚类。

第十章 分布式调度

10.2 静态调度

❖ 两种最优调度算法

两种方法都假设通信代价可以忽略，优先图中每个节点的执行时间是一样的，即一个时间单元。具体限制如下：

- (1) 在第一个有约束的调度问题中，优先图是一棵树。
- (2) 在第二个有约束的调度问题中，只有两个处理器可用。

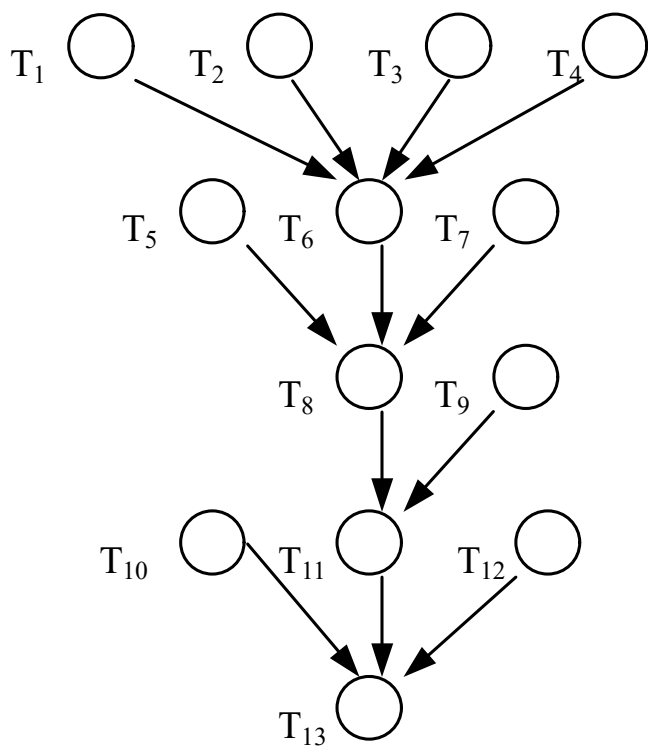
两种调度算法都是最高层优先(highest-level-first)方法，也就是说，通过节点的优先级来选择节点。

第十章 分布式调度

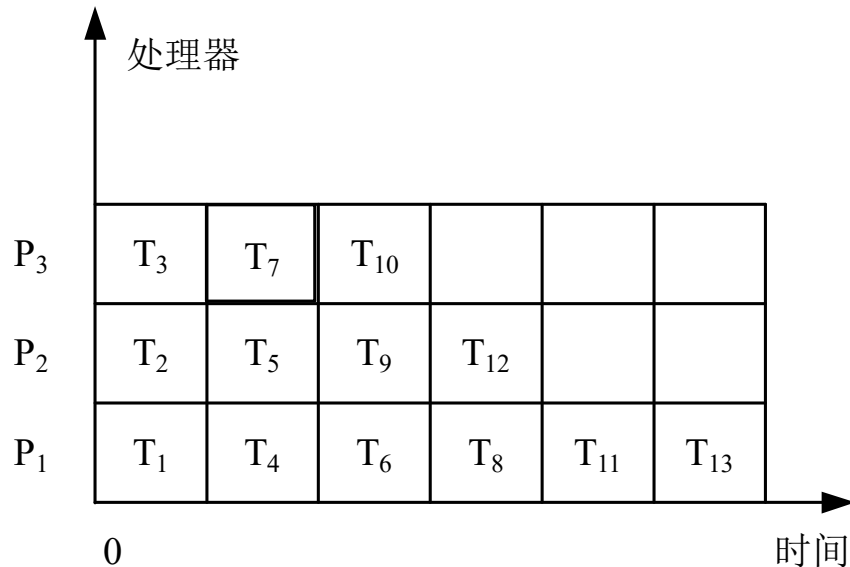
10.2 静态调度

❖ 两种最优调度算法

树结构的优先图 and 这个图在三个处理器上的最优调度：



(a) 树结构的任务优先图



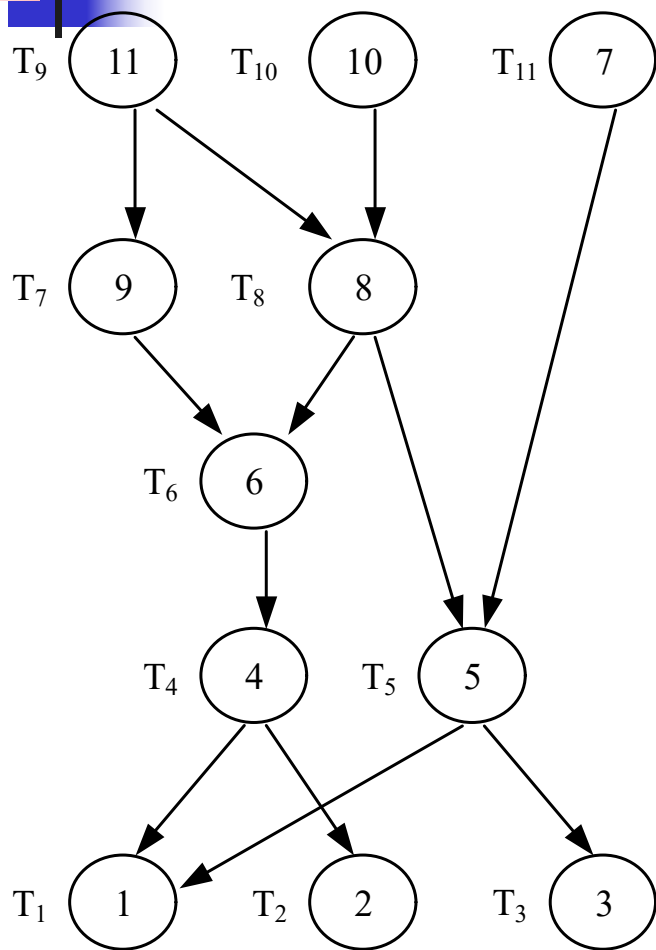
(b) 对三个处理器的调度

第十章 分布式调度

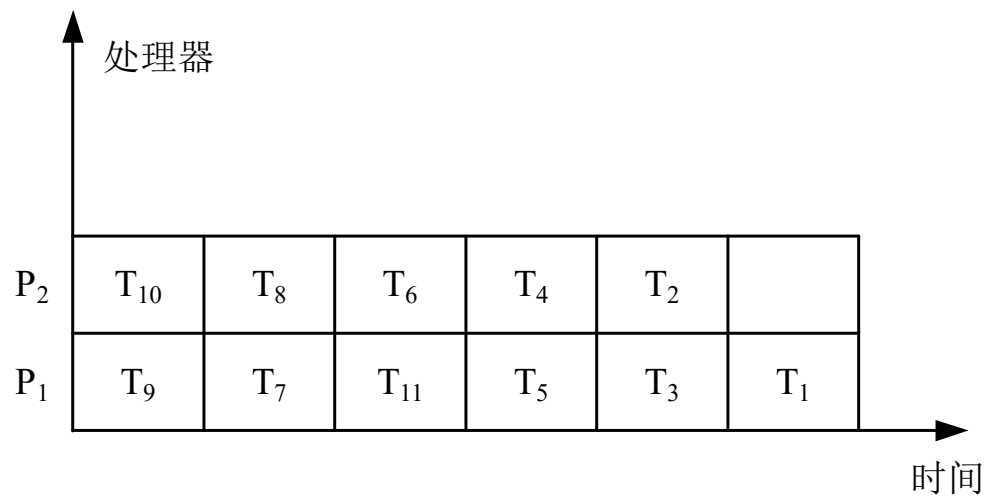
10.2 静态调度

❖两种最优调度算法

只有两个处理器可供使用的调度：



(a) 优先级的标记



(b) 对双处理器的调度

第十章 分布式调度

10.2 静态调度

❖ 基于任务相互关系图的任务调度

- 1) 任务相互关系图由无向图 $G_t(V_t, E_t)$ 表示, V_t 是进程集合, E_t 是边集合, 每条边用相关两个进程的通信代价标记;
- 2) 处理器图 $G_p(V_p, E_p)$ 用顶点集 V_p 和边集 E_p 表示, V_p 中的每个元素是一个处理器, E_p 中的每个元素是一个通信信道;
- 3) 然后进行分配M: 进行 $V_t \rightarrow V_p$ 的变换和执行时间的估计。假设 $w(u)$ 和 $w(u, v)$ 分别表示节点 u 和链接 (u, v) 的代价。

第十章 分布式调度

10.2 静态调度

❖ 基于任务相互关系图的任务调度

➤ 对分配M的评估:

- 1) 处理器p的计算负载为: $Comp(p) = \sum_{u \in V_t} w(u) \mid M(u) = p$
- 2) 处理器p的通信负载为: $Commp(p) = \sum_{(u,v) \in E_t} w(u,v) \mid M(u) = p \neq M(v)$
- 3) 整个应用程序中总的计算量是:

$$Comp = \sum_{p \in V_p} Comp(p) = \sum_{p \in V_p} \sum_{u \in V_t} w(u) \mid M(u) = p$$

- 4) 整个应用程序中总的通信量是:

$$Comm = \frac{1}{2} \sum_{p \in V_p} Commp(p) = \frac{1}{2} \sum_{p \in V_p} \sum_{(u,v) \in E_t} w(u,v) \mid M(u) = p \neq M(v)$$

第十章 分布式调度

10.2 静态调度

❖ 基于任务相互关系图的任务调度

➤ 对分配M的评估:

5) 程序总的执行时间大概是:

$$T = \max \{ \alpha \text{Comp}(p) + \beta \text{Comm}(p) \}, p \in V_p$$

α 是依据处理器的执行速度确定的值, β 是依据每个通信信道的通信速度和通信进程间的距离确定的值。注意如果两个进程 u 和 v 在 G_t 中邻接, 它们在 G_p 的映像(M的映像结果)可能邻接也可能不邻接。理想的情况下, 所有通信进程被分配在邻接的处理机上, 以此减少处理器间通信。

第十章 分布式调度

10.2 静态调度

❖ 基于任务相互关系图的任务调度

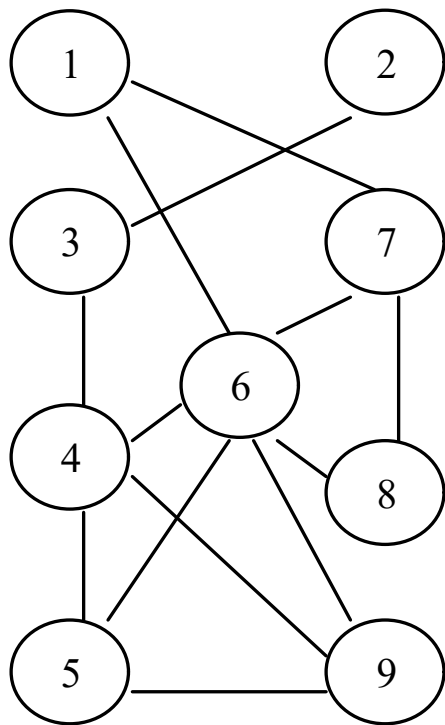
➤ **映射的势：** 评估映射质量的一个指标是任务图 G_t 中的边映射到处理器图 G_p 中的边的数目。这个数目被称作映射的势 (cardinality)，就是 G_t 中映射到 G_p 中邻接处理器的通信进程对的数目。映射的势不会超过 G_t 中的链接数目。如果一个映射的势最大，它就是一个理想的映射。

第十章 分布式调度

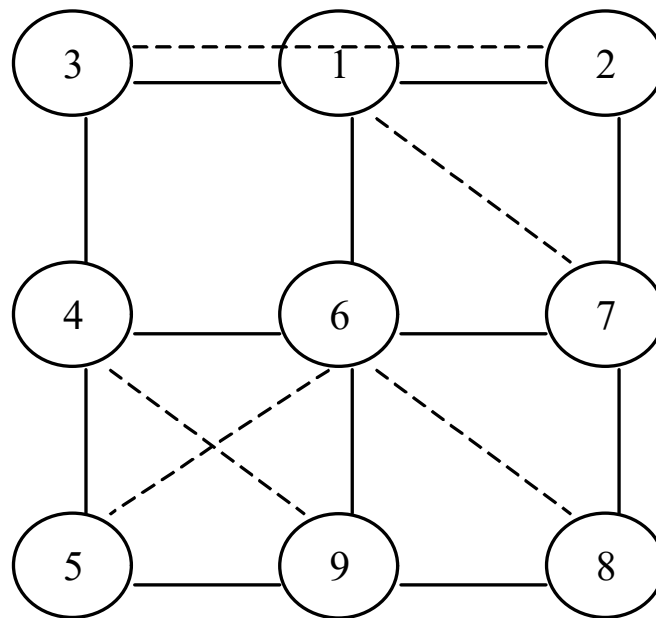
10.2 静态调度

❖ 基于任务相互关系图的任务调度

图中，映射的势是8，任务关系图中边的为13条。



(a) 任务相互关系图



(b) 任务到处理器的映射

第十章 分布式调度

10.2 静态调度

❖ 基于任务相互关系图的任务调度

嵌入：设想任务相互关系图和处理图被各自看作 G_t 和 G_p 。为了通过 G_t 得到对 G_p 的有效模拟(emulation)，也就是在 G_p 中嵌入 G_t 。

嵌入的不同代价指标：

(1) G_t 的边的膨胀。 G_t 的边的膨胀定义为被映射成 G_p 里的一条边的 G_t 中对应的路径的长度。嵌入的膨胀为 G_t 中的最大边膨胀。

(2) 嵌入的扩大。嵌入的扩大定义为 G_t 里的节点数对 G_p 里的节点数的比率。

(3) 嵌入的拥塞。嵌入的拥塞定义为包含 G_p 中的一条边的最大路径数， G_p 中的每条路径表示 G_t 中的一条边。

(4) 嵌入的负载。嵌入的负载是 G_t 分配给 G_p 中任意处理器的进程的最大数目。

第十章 分布式调度

10.3 动态调度

❖ 动态调度的组成要素

动态调度算法有六个策略：启动策略、转移策略、选择策略、收益性策略、定位策略和信息策略。

- 1) **启动策略**的责任是决定谁应该激活负载平衡活动。
- 2) **转移策略**决定一个节点是否在合适的状态参与负载转移。
- 3) **选择策略**选择最适合转移最能起平衡作用的任务，并发送给合适的目标处理器。
- 4) **收益性策略**量化系统中负载不平衡程度，并且作为系统负载平衡潜在受益的估计，评估系统负载平衡是否是有收益的。
- 5) **定位策略**是寻找合适的节点共享负载。
- 6) **信息策略**决定收集系统中其他节点状态信息的时机、收集的方法和收集的信息。

第十章 分布式调度

10.3 动态调度

❖ 动态负载均衡算法的分类、设计决策和使用的参数

动态负载均衡算法可以分成以下几类：

(1) 全局的和局部的。局部负载均衡算法在相邻的节点间转移工作负载。全局负载均衡算法不仅在相邻节点间转移负载，还在全系统内计算负载，根据全局情况调整处理器负载。

(2) 集中控制的和分散控制的。在集中控制算法中，中心控制器收集状态信息，做出负载均衡决策。分散控制算法把控制机制分散到全系统的各个节点。混合式负载均衡算法是集中控制和分散控制算法的折衷。

第十章 分布式调度

10.3 动态调度

❖ 动态负载均衡算法的分类、设计决策和使用的参数

动态负载均衡算法可以分成以下几类：

(3) 不协作的和协作的。在不协作方法中，各个节点不知道系统中其他节点的状态，独立决定自己的定位和负载转移规则。在协作算法中，节点间相互配合来决定负载均衡决策。

(4) 适应性的和非适应性的。在适应性算法中，负载均衡策略根据系统状态变化而改变；而非适应性方法中，这些策略是不变的。

第十章 分布式调度

10.3 动态调度

❖ 动态负载均衡算法的分类、设计决策和使用的参数

动态负载均衡算法的设计决策包括如下一些内容：

(1) **非抢先式的和抢先式的**：抢先式的主要目的是负载共享，节点只分配新到达的任务，又称为任务放置(placement)。抢先式的算法的主要目的是充分利用系统资源，能够重新分配正在运行的任务，又称为进程迁移(migration)。

(2) **采用何种信息策略**。与信息策略有关的问题有：(a)周期性收集信息还是非周期性收集信息；(b)收集局部信息还是全局信息；(c)处理器负载指标。

(3) **集中控制算法和分散控制算法**：集中控制算法有一个中心处理器从系统中其他处理器收集负载信息。分散控制算法是通过每个处理器发送自己负载变化情况给所有处理器或者它的邻居来实现的。

第十章 分布式调度

10.3 动态调度

❖ 动态负载平衡算法的分类、设计决策和使用的参数

动态负载平衡算法的设计决策包括如下一些内容：

(4) **采用何种启动策略。**启动策略有三种：发送者启动的、接受者启动的和对称启动的。

(5) **资源复制。**任务转移的时候，涉及到的文件和数据也必须被复制到目标处理器。为了减少转移的代价，常用的任务和数据可以事先被复制和分配到不同的处理器。

(6) **进程分类。**依据特征来区分进程类型。如果系统中运行的进程有很大的区别，它们就必须分在不同的类。当系统中有多多个进程类型时，负载平衡算法必须考虑进程的类型，根据不同的类型做出改变。

第十章 分布式调度

10.3 动态调度

❖ 动态负载平衡算法的分类、设计决策和使用的参数

负载平衡算法使用的参数：

- (1) **系统的规模**。系统中处理器的数目是影响负载平衡决策的一个参数。
- (2) **系统负载情况**。 需要避免颠簸现象。
- (3) **处理器的输入流量**。进程可以以任何**随机**模式到达处理器，如果处理器能够测定自己的输入流量并且和其他处理器比较，它就能比较容易评估系统即时的负载水平，从而对任务转移做出更好的决策。
- (4) **转移的负载门限**。系统中触发任务转移的负载门限是一个关键参数，因为选择不当会导致系统不平衡和任务转移的连锁反应。

第十章 分布式调度

10.3 动态调度

❖ 动态负载平衡算法的分类、设计决策和使用的参数

负载平衡算法使用的参数：

(5) 任务大小。一般来说，转移一个运行时间太短的任务是不恰当的。类似的，太大的进程或者涉及到大量数据和文件的进程最好在本地处理器上执行。

(6) 管理成本。组成管理成本的主要因素是：处理器当前负载的测量、处理器决策使用的负载信息、决策发生的位置和处理器间任务的传送。

第十章 分布式调度

10.3 动态调度

❖ 动态负载均衡算法的分类、设计决策和使用的参数

负载均衡算法使用的参数：

(7) 负载均衡的视界。一个节点能够在其邻接节点范围内为一个任务寻找可能的目标节点，在其上运行该任务。这个邻接节点范围的直径称为视界(horizon)。这个参数设置了寻找目标节点过程中探查的邻接节点的数量。

(8) 资源要求。任务对系统资源的要求会影响它的转移。需要较多资源的进程可能会持续等待资源变得可用，这就可能影响系统的响应时间。

第十章 分布式调度

10.4 空闲工作站的调度结构

❖ 工作站共享问题

工作站共享问题包括以下一些内容：对工作站使用模式的分析，设计分配远程处理能力的算法和结构，研究远程执行设备。

全局调度机构的主要目标有：

- (1) **性能要求：**调度机构占用整个系统的开销最小，它们不应该占用不使用此机构的应用程序的时间，也不应该使被调度的应用程序的执行产生大的延迟。
- (2) **支持的系统规模：**应该能支持几百个甚至上千个工作站。
- (3) **容错：**一个或几个机器崩溃时，系统的远程执行设备应该在几秒钟之后能够继续工作。
- (4) **公平性：**不管分配作业到哪个机器上，为该作业提供的性能都是同样可接受的。
- (5) **自治性：**工作站属于个人所有，其他人使用不应影响主人的工作。

第十章 分布式调度

10.4 空闲工作站的调度结构

❖ 工作站共享问题

设计全局调度设施在结构上要解决三个主要问题：

- (1) 有关负载的信息是如何传送的，使用公布的还是回答查询的办法？即选择哪一种信息策略。
- (2) 谁主动发起远程执行的请求，是作业进入的顾客节点(源节点)还是处理此作业的节点(服务员节点)？一个物理节点处理机可以是一个源节点，也可以是一个服务员节点。在服务员主动的情况下，此服务员主动寻找工作。这里所要解决的是选择什么样的启动策略。
- (3) 谁来决策为一个作业(程序)选择一个合适的执行主机，请求的发起者还是一个集中的服务员节点？这里要解决的是定位策略的问题。

第十章 分布式调度

10.4 空闲工作站的调度结构

❖ 集中式调度

集中式调度是在系统中有一个中央调度服务员，负责搜集状态信息并做出全部调度决策。各机器周期性地向它发送状态更新报文，报告它们的负载信息；顾客向它发送远程执行请求。中央调度服务员根据负载情况，建立一个主机候选者的有序表，依次选择主机，对顾客的远程执行请求进行响应。使用中央调度服务员查询状态会减少报文传送数目。但是因为机器由于本地活动可以在任何时间改变其负载，所以将产生状态信息过时的问题。

解决集中式调度的**容错问题**的典型方法是提供多个备用服务员。集中调度的最后一个问题是在**何处运行调度程序**。调度程序没有任何特殊要求，可放到任何空闲机器上，并可根据需要迁移。

第十章 分布式调度

10.4 空闲工作站的调度结构

❖ 分散式调度

在全分散方案中，每个机器自己进行选择活动。它必须不断地记录整个系统状态或者当需要时查询系统状态信息。在前一种情况下，每个机器(即使是忙碌的机器)要定期地产生更新报文并向其他主机广播(公布)。而每个主机中维持一个主机状态表。在后一种情况下只有对主机选择有兴趣的那些主机才关心状态信息(查询)。采用查询方法，即每个需要获得空闲主机的顾客机发送查询报文请求得到当前状态信息，请求中包括所需资源的说明。该顾客从所有愿意成为候选主机的机器那里得到回答，并从中选取一个最合适的机器。

第十章 分布式调度

10.4 空闲工作站的调度结构

❖ 分散式调度

两个要解决的问题。第一是查询者可能要求接收大量的、几乎是同时到来的回答报文，以及 N^2 报文的传送要消耗网络的带宽。第二是可能产生冲突。

第一个问题的解决方法：一个相当简单的办法是放宽选择主机的标准，它可以不是最佳的，即不是负载最轻的，但可以是较轻的、较好的。查询者只考虑全部回答报文中的一部分，扔掉其余部分。

第二个问题解决办法是在迁移程序前先发送一个执行请求，被选择机只对第一个请求回答并等待申请者传送被执行的程序。

第十章 分布式调度

10.4 空闲工作站的调度结构

❖ 混合式调度

集中式方法支持的规模较大，但集中式方法可靠性较差，不易扩充。分散式方法具有较高可靠性，实现简单，容易扩充，但效率较低。

混合式调度结构中，每个工作站有一个局部调度程序，还有一个后台作业队列，用户提交的作业和远程作业都放到此队列中。有一个工作站除了局部调度程序和作业队列外，还有一个协调程序（协调者）。协调者定期（例如每两分钟）向各个工作站查询，看有哪些工作站可用作远程执行的源，哪些工作站后台作业队列中有作业等待处理。中央协调者为有后台作业等候的工作站上的调度程序分配空闲工作站资源。各工作站如果其队列中有多个后台作业，则由本地调度执行程序决定下次应执行哪个作业。

第十章 分布式调度

10.5 进程转移和远程执行

❖ 进程转移和远程执行的目的和方法

- 进程转移的主要目的是使由个人工作站组成的系统的计算资源容易共享：用户在执行若干相对独立的任务时，可把它们从某些重负载工作站移到另外一些轻负载工作站上加快完成。
- 进程转移的形式有两种：抢先方式和非抢先方式，非抢先方式又称为进程放置，抢先方式又称为进程迁移。进程放置是为进程选择一个执行节点，在此节点上启动此进程。进程迁移是把进程转移到一个较好的执行节点继续执行。

第十章 分布式调度

10.5 进程转移和远程执行

❖ 进程转移和远程执行的目的和方法

➤ 进程转移和远程执行的一般要求有以下两点：

(1) **透明性**。进程运行的结果与该进程在系统中什么地方执行无关。为了转移此进程不必用特定方式重新编写程序。也就是说，这些进程转移到远程执行环境后必须与在原地一样（名字、操作和数据，但不包括硬件）。

(2) **有效性**。迁移一个进程需要时间，支持该进程远程执行也需要时间，这些时间应尽量短。

第十章 分布式调度

10.5 进程转移和远程执行

❖ 进程转移和远程执行的目的和方法

➤ 判断是否值得进行进程迁移和远程执行：

- (1) 有多个计算量很大的进程在一个工作站上运行；
- (2) 运行时间远远超过在远程启动执行一个进程的时间；
- (3) 从所选择的远程节点上被驱逐的可能性很小；
- (4) 进程刚建立不久，还未来得及使用很多地址空间。

第十章 分布式调度

10.5 进程转移和远程执行

❖ Sprite的进程迁移和远程执行设备

Sprite系统实现透明性的方法：当一个被迁移的进程调用一个与位置有关的系统调用（这种调用在不同位置执行可能产生不同的结果）时，该系统调用由RPC设备转发到此进程的基地节点执行。

Sprite系统把系统调用分成以下几类：

- (1) 和地址无关的，共有38个。远程节点处理与位置无关的系统调用。
- (2) 和地址有关的，有24个。基地节点为大多数和进程位置有关的系统调用服务，对进程的这种环境的任何操作都被转发给基地。

第十章 分布式调度

10.5 进程转移和远程执行

❖Sprite的进程迁移和远程执行设备

Sprite系统实现透明性的方法：当一个被迁移的进程调用一个与位置有关的系统调用（这种调用在不同位置执行可能产生不同的结果）时，该系统调用由RPC设备转发到此进程的基地节点执行。

Sprite系统把系统调用分成以下几类：

(3) 合作的，有5个。在一些情况下，远程节点和基地节点必须合作处理一个系统调用。

(4) 不可迁移的，有一个。这个调用把内核的地址空间的一部分变换到用户存储器中。一个远程进程不可能把存储器从内核变换到它的基地节点上，所以任何使用此调用的进程必须在调用完成前迁移到基地。

第十章 分布式调度

10.5 进程转移和远程执行

❖ Sprite的进程迁移和远程执行设备

Sprite系统的进程迁移包括以下几个步骤：

- (1) 向目的节点发送一个RPC，确认是否允许迁移该进程。
- (2) 当要迁移该进程时，使用标准信号中断该进程的执行。
- (3) 传送该进程的“进程状态”，包括各寄存器的内容、用户标识符和小组标识符、信号处理信息、基地节点和该进程标识符。
- (4) 传送虚拟地址空间。把所有重写的页送到文件服务器，把对应的交换文件的页表和说明符送到目的节点。
- (5) 将该进程已打开的文件的说明符和当前工作目录打包并传送。
- (6) 发送一个RPC结束迁移，允许被迁移的进程在目的节点上恢复执行。

第十章 分布式调度

10.5 进程转移和远程执行

❖ V系统中的可抢先的远程执行设备

V系统的可抢先远程执行设备使用**预复制方法**提高性能：

为了减少冻结时间，可使用预复制方法，在冻结前就多次复制全部地址空间，每次复制上次复制以来被修改的页。多次复制后被修改的部分只剩下很少的部分，这时再冻结，把剩余部分复制过去。这样仅用最短时间冻结。

第十章 分布式调度

10.5 进程转移和远程执行

❖ NEST中的透明的远程执行设备

NEST使用逻辑机的概念。它被定义为属于一个物理机的所有进程集合(包括本地的和远程的)。在逻辑机边界内透明地保持全部UNIX系统的能力, 具体有以下几点:

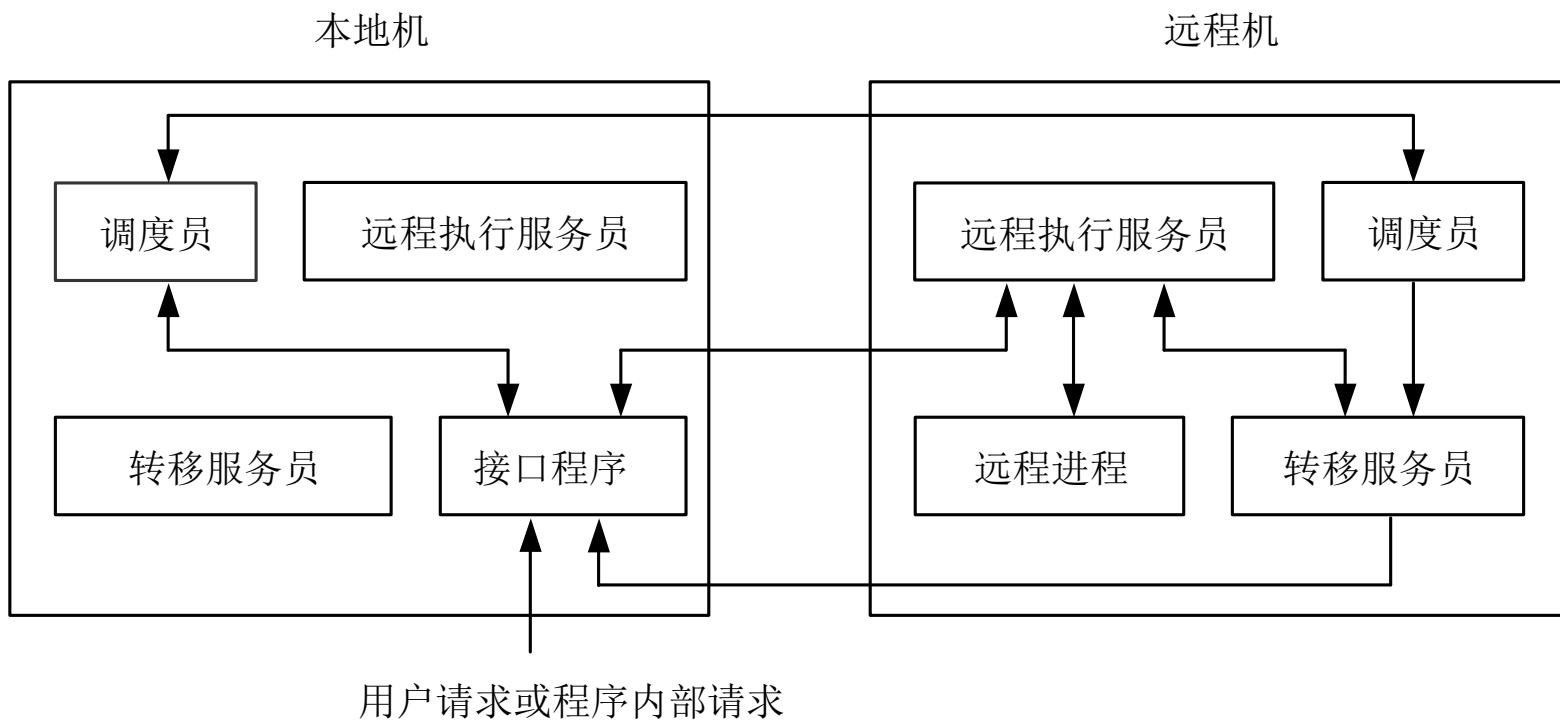
- (1) 远程机上运行的远程进程所访问的文件都是原主机上的文件。
- (2) 远程进程能够向逻辑机边界内任何进程发送所有标准UNIX系统信号, 也能接收来自逻辑机边界内任何进程的标准UNIX系统的信号。
- (3) 原有的UNIX进程组关系仍在逻辑机内保持。
- (4) 在逻辑机边界内保持原有的进程之间的父子关系。
- (5) 远程进程能以透明方式访问其远程终端, 即原来登录的终端。

第十章 分布式调度

10.6 空闲工作站共享系统Sidle

❖ Sidle的组成

Sidle由以下四个部分组成：空闲机调度员，远程执行服务员，转移服务员和用户接口程序。



第十章 分布式调度

10.6 空闲工作站共享系统Sidle

❖ Sidle的调度

Sidle采用完全分散的调度方案，但使用以下措施减少网络上的报文数：

- (1) 不维持整个系统的状态信息，既不维持空闲工作站表，所以不必定期地相互发送广播查询报文，只有在有远程执行请求时，才发送广播查询报文。在没有远程执行请求的情况下，Sidle系统不占用网络带宽，只需要定期地更新本地的负载状态信息。
- (2) 减少回答报文数。某主机仅当可以作为远程执行的目标节点时，才对查询报文予以回答。
- (3) 减少处理报文的数目。不考虑负载最轻或者最佳调度，只需处理最先到达的回答报文即可。

第十章 分布式调度

10.6 空闲工作站共享系统Sidle

❖ Sidle的透明远程执行设备

(1) 支持程序内部的并行。Sidle系统的远程执行设备由远程执行服务员RES和支持程序内部并行执行的接口程序组成。这个接口程序称为通信服务员CS。通信服务员CS将并行程程序的远程执行申请转发给远程机的远程执行服务员，后者派生一个子进程执行并行程程序所派生的子任务，远程子任务的执行结果送回基地主机，由基地主机的通信服务员CS保存。并行程程序需要子任务的执行结果时，从通信服务员CS处获得。

第十章 分布式调度

10.6 空闲工作站共享系统Sidle

❖ Sidle的透明远程执行设备

(2) 远程执行的透明性。在Sidle系统中，远程执行的透明性主要表现在三个方面：第一，文件的远程透明访问；第二，进程间关系的保持和物理主机间UNIX标准信号的传送；第三，远程控制终端。

第十一章 分布式共享存储器

11.1 基本概念

❖什么是分布式共享存储器系统

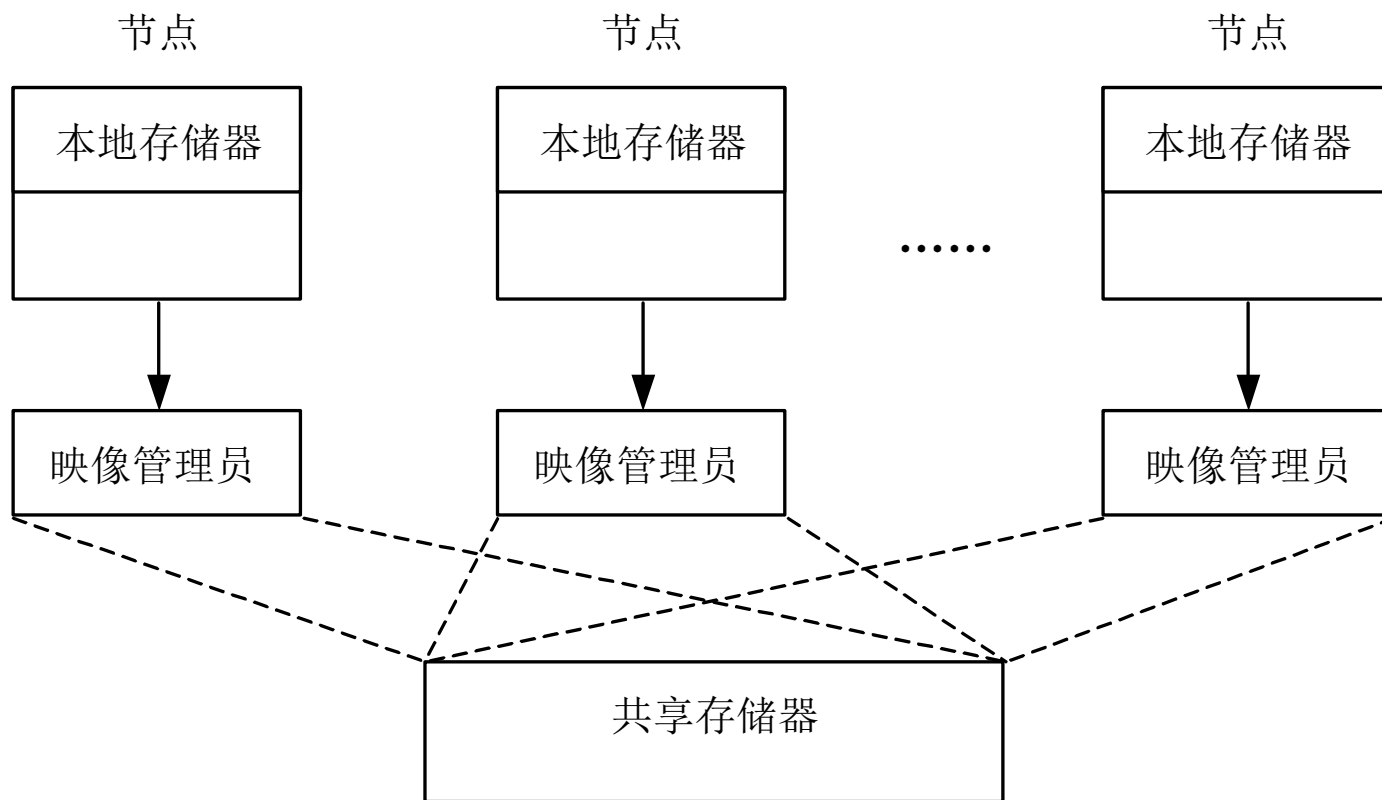
➤分布式共享存储器系统是分布式操作系统中的一个资源管理部件，它在没有物理上共享的存储器的分布式操作系统中实现了共享存储器模式。这种共享存储器模式在分布式系统中提供了一个可供系统内所有节点所共享的虚拟地址空间。程序设计者可以像使用传统的存储器一样使用该虚拟地址空间。这种物理上分布逻辑上共享的存储器就叫做分布式共享存储器(Distributed Shared Memory—DSM)。

每一个节点都可以拥有存储在共享空间的数据，数据的所有者也可以跟随数据从一个节点移到另一个节点。当一个进程访问共享地址空间中的数据时，映像管理员就把共享存储器地址变换到本地地址或远程的物理存储器地址。

第十一章 分布式共享存储器

11.1 基本概念

❖ 什么是分布式共享存储器系统



第十一章 分布式共享存储器

11.1 基本概念

❖为什么需要分布式共享存储器

DSM的计算模型和RPC的计算模型相比各有优缺点：

- 1) DSM的计算模型支持数据在系统内移动，使数据更容易访问。
- 2) RPC计算模型是把操作移到数据所在位置。RPC不支持程序利用其访问的局部性优点，对一块远程数据的每个操作都产生通信，对数据的操作必须先定义好。但是RPC支持异构型。
- 3) DSM可把数据移到本地节点，允许程序利用其访问的局部性优点，使用缓存器可以改善响应时间。移动性要求对数据位置进行跟踪；缓存要求解决各副本的一致性。当数据正向某个主机移动时，不能对它进行处理。如果数据经常修改，RPC模型可能更好些。

第十一章 分布式共享存储器

11.1 基本概念

❖为什么需要分布式共享存储器

从通信机制来看，DSM与报文传递方式有以下不同：

- (1) 访问的透明性。使用报文传递方式访问共享的数据变量时，程序必须明确地使用节点地址信息和通信原语(如SEND和RECEIVE)。而在DSM中系统提供了一种抽象的共享地址空间从而隐匿了物理地址和通信细节，使得程序直接面向共享的数据。
- (2) 共享数据结构的复杂性和异构性。使用报文传递方式，由于数据是在不同的地址空间之间传递，从而使得具有复杂数据结构的数据难于在不同类型的计算机及进程之间传递。而在DSM中，可以借助引用机制(reference)去实现上述数据访问，复杂性与异构性的问题由引用机制去处理，从而进一步简化了并行程序设计。

第十一章 分布式共享存储器

11.1 基本概念

❖ 为什么需要分布式共享存储器

从通信机制来看，DSM与报文传递方式有以下不同：

- (3) 数据的局部性。在DSM中，新访问的数据项与其周围的数据一起按块或按页移动，而不是只移动新访问的数据本身。根据程序的局部性原理，这样可以大大地减小网络的通信开销。

第十一章 分布式共享存储器

11.1 基本概念

❖为什么需要分布式共享存储器

与紧密耦合的多机系统相比，DSM系统具有以下特点：

- (1) 规模可扩充。在紧密耦合的多机系统中，由于各处理机共享的是一个单一的物理存储器，主存访问都要经过一个集中环节(例如总线)进行，这就限制了多机系统的规模(一般为几十台处理机)。DSM不存在这样的限制，可以扩充至很大的规模(多至上千个节点)。
- (2) 廉价。由于DSM系统可以用现有的硬件来构造，并且无需连接共享存储器与处理机的复杂接口，因而DSM的构造成本要低于紧密耦合的多机系统。
- (3) 兼容性。在共享存储器多机系统上编写的程序原则上都可以无需修改或稍加修改后转换到DSM系统上运行。

第十一章 分布式共享存储器

11.1 基本概念

❖ 共享存储器中缓存一致性方法

有两类基本方法实现缓存一致性：即探听缓存方法和使用目录的方法。

➤ 探听(snooping)缓存方法用于具有广播能力的通信介质中，例如共享总线。每个缓存器为了保持自己数据的一致性要监听共享总线上进行的由其他处理机发出的存储器操作。

Berkeley是一个典型例子，它是一种写无效协议，它假设通过单总线访问共享的物理存储器。此协议采用一个所有权方案。一个数据块的所有者是一个缓存器，是上次对该数据块的修改者，如果该块被其所有者清除，则主存作为其所有者。

第十一章 分布式共享存储器

11.1 基本概念

❖ 共享存储器中缓存一致性方法

Berkeley 探听协议数据块有四种状态：重写(dirty)、共享重写、有效和无效：

- (1) 无效。该缓存块不包含有效数据。
- (2) 有效。该缓存块中数据是有效的。
- (3) 重写。共享存储器中的数据是不正确的，该缓存块是唯一有效的数据副本。该缓存块是数据的所有者。
- (4) 共享重写。共享存储器中的数据是不正确的，该缓存块是数据的所有者，其他缓存中有同样的副本。

第十一章 分布式共享存储器

11.1 基本概念

❖共享存储器中缓存一致性方法

探听协议的写操作：数据只能由所有者提供。有效块和无效块在替换时可以简单地扔掉。重写块和共享重写块在替换时要写回共享存储器，并把共享存储器设置为所有者。如果对缓存块进行写，而缓存块的状态是重写的，则写操作可以直接进行；但是如果缓存块是共享重写、或有效，则必须向其他缓存器发送“无效”信号，然后可以进行写操作；如果缓存块是无效的，要从当前所有者那里取得数据块，再向其他缓存器发送“无效”信号，然后可以进行写操作。

第十一章 分布式共享存储器

11.1 基本概念

❖共享存储器中缓存一致性方法

目录协议：在共享存储器中设置存储器块的目录。当发生缓存不命中时，先把请求转到此目录。通常目录项中包含所有权、副本集(copyset)和该块的重写位。副本集指出该块数据在哪些缓存器中有副本，可用位向量来实现。发生读未命中时，先检查重写位，如果该块不处于重写状态，则共享存储器中的版本是有效的，于是简单地返回该块，并对副本集信息进行更新；如果该块的重写位置位，则该块的所有者必须修改该块，并且要更新共享存储器中的版本，向读者提供读副本。写未命中或者从读权变成写权时，要求目录的副本集使其他副本无效。与探听缓存方案不同，读副本的位置都已经知道，因此，可以用顺序方式而不是以广播方式发送“无效”报文。目录方案不要求广播介质，但在每次缓存未命中时要增加一次查表。

第十一章 分布式共享存储器

11.1 基本概念

❖ DSM的设计与实现问题

- (1) 共享地址空间结构和粒度。共享地址空间的结构指的是存储器中共享数据的布局方法，它依赖于应用程序类型，地址空间可以是平面的，分段的或物理的。粒度是指共享单元的大小，可以是字节、字、页或复杂的数据结构，它也是可用的并行性的度量，依赖于通信开销和应用程序表现的局部性类型。结构和粒度是密切相关的。
- (2) 缓存一致性协议。不同的协议有不同的假设，选择协议依赖于存储器访问模式和支持环境。在写无效协议中，一块共享数据可能有很多个只读副本，但仅有一个可写副本，每进行一次写时，除了一个以外，其他副本均变成无效。在写更新协议中。每次写都要对所有副本进行更新。

第十一章 分布式共享存储器

11.1 基本概念

❖DSM的设计与实现问题

- (3) 同步原语。在并发访问下，光有缓存一致性协议还不能维持共享数据一致性。尚需要同步原语对访问共享数据的活动进行同步，例如信号灯、事件计数和锁等。
- (4) 替换策略。在允许数据迁移的系统中，当共享数据占满了缓存器的有效空间时，必须决定将那些数据转移出去并且放到哪里去。
- (5) 可扩充性。DSM系统比起紧密耦合系统来，一个重大的优点是具有可扩充性。限制可扩充性有两个因素：集中的瓶颈（像紧密耦合系统中的总线）和全局公用信息的操作及存储（如广播报文或目录等）。

第十一章 分布式共享存储器

11.1 基本概念

❖DSM的设计与实现问题

- (6) 异构性。如何实现对两个具有不同体系结构的机器的存储器共享是个很困难的问题。两个机器甚至对基本数据类型(如整数、浮点数等)都使用不同的表达方式。
- (7) 数据定位和访问。为了在一个DSM系统中共享数据, 应用程序必须能找到并且检索所需要的数据。对于一个支持数据迁移的系统, 实现这一点就更为复杂。
- (8) 颠簸。DSM系统特别容易出现颠簸, 例如若两个节点对一个数据项同时进行写, 就可能产生以高速率来回传送数据的现象(乒乓效应), 使得任何实际工作都不能进行。

第十一章 分布式共享存储器

11.1 基本概念

❖一致性语义

共享存储器中常使用的一些一致性语义：

- (1) 严格一致性。对一个数据项所进行的任何读操作所返回的值总是对该数据项最近一次进行写操作的结果。
- (2) 顺序一致性。所有进程对数据项的所有操作可以认为是按照某个顺序进行的，任何进程对这个顺序的观点是一样的。
- (3) 处理机一致性。不仅要求一个进程中的所有写操作能够以它在该进程中出现的顺序被所有其他进程看见，还要求不同进程对同一个数据项的写操作，应该被所有进程以相同的顺序看见。

第十一章 分布式共享存储器

11.1 基本概念

❖一致性语义

共享存储器中常使用的一些一致性语义：

- (4) 弱一致性。程序员使用同步算符，使得对数据的多个操作组来说是顺序一致性的。即不同进程的多个操作组可以认为是按照某个顺序进行的，任何进程对这个顺序的观点是一样的。但是操作组内的多个操作其他进程是不可见的。对同步算符是顺序一致性的。
- (5) 释放一致性。使用了“获得”和“释放”这两类同步算符，对同步算符是处理机一致的。

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖ 算法使用的模型和环境

➤ 共享存储器模型为访问共享数据提供了两个基本操作：

`data:=read(address)`

`write(data, address)`

`read`返回由`address`指出的数据项。`Write`把由地址`address`指出的内容设置为`data`。

➤ 根据是否允许迁移或复制，可以将DSM的实现算法分成四类：中央服务员算法、迁移算法、读复制算法和全复制算法。

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖ 算法使用的模型和环境

DSM 的四种算法

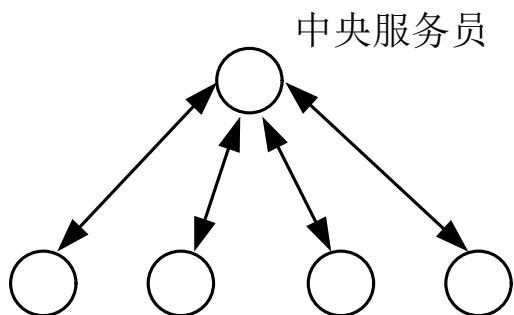
	非复制	复制
非迁移	中央服务员	全复制
迁移	迁移	读复制

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖ 中央服务员算法

使用一个中央服务员，负责为所有对共享数据的访问提供服务并保持共享数据唯一的副本。读和写操作都包括由执行该操作的进程向中央服务员发送请求报文，中央服务员执行请求并回答，读操作时回答数据项，写操作时回答一个承认。



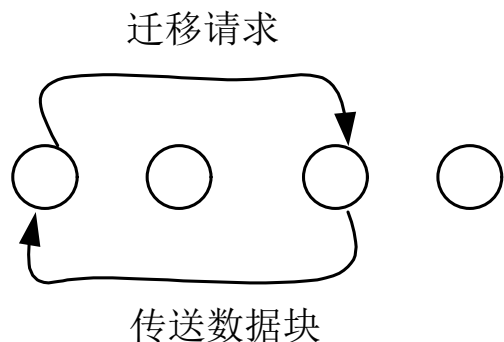
顾客	中央服务员
发送数据请求	接收请求，执行数据访问，发送回答
接收回答	

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖ 迁移算法

数据总是被迁移到访问它的节点。这是一个“单读者/单写者” (SRSW) 协议，因为在整个系统中，一次只有一个进程读或写一个给定的数据项。



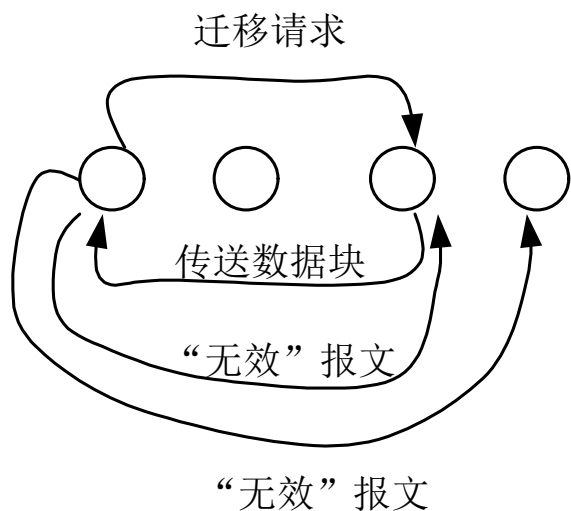
顾客	远程主机
如果数据块不在本地，则确定位置，发送请求	
	接收请求，发送块
接收回答，访问数据	

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖ 读复制算法

对于一个当前不在本地的块中的一个数据项进行读操作时，先与远程节点通信以获得那个块的一个只读副本，然后再进行读操作。若被执行写操作的数据所在的块不在本地或在本地但主机无写权时，必须先使此块在其他节点的所有副本无效，之后再进行写操作。



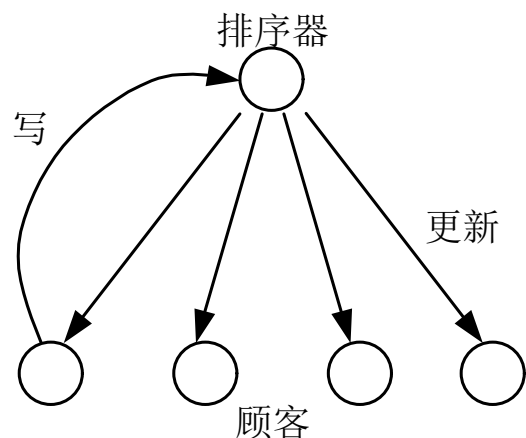
顾客	远程主机
如果数据块不在本地，则确定位置，发送请求	接收请求，发送块
接收块，广播“无效”报文	接收“无效”报文，使块无效

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖全复制算法

全复制算法允许数据块在进行写时也可以复制，因而它遵从了“多读者/多写者”（MRMW）协议。保持复制数据一致性的一种可能的方法是对所有的写操作进行全局排序，而只对与发生在执行读操作节点上的写操作相关的那些读操作进行排序。



顾客	排序器	主机
若写，则发送数据到排序器	接收数据，添加序号，广播	接收数据，更新本地存储器
接收承认，更新本地存储器		

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖ 算法性能

基本代价：

- (1) p : 一个包事件的代价，即发送或接收一个短包的处理代价，包括可能的任务切换、数据复制及中断处理开销。实际系统的典型值的变化范围是1到几个毫秒。
- (2) P : 发送或接收一个数据块的代价。这与 p 十分相似，但 P 值要高得多。对于一个通常需要多个包的8K字节的块来说，典型值的范围是20至40个毫秒。
- (3) S : 参与分布式共享内存的节点数。
- (4) r : 读/写比，即平均有 r 个读操作时才有一个写操作。这个参数也用于整个块的访问模式。显然这两个比可能不同，但为了简化分析假定相等。

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖ 算法性能

基本代价：

(5) f ：非复制数据块(用于迁移算法)访问故障的概率。它等于单一节点连续访问一个块(以后由另一个节点访问此块导致故障)的平均次数的倒数。它说明迁移算法数据访问的局部性。

(6) f' ：读复制算法用于对复制数据块访问故障的概率。它是连续访问本地数据块中数据项(以后访问一个非本地数据块中某数据项)的平均次数的倒数。它说明读复制算法数据访问的本地性。

第十一章 分布式共享存储器

11.2 实现DSM的算法

❖ 算法性能

四种算法的平均访问代价：

中央服务员算法： $C_c = (1 - 1/S) \cdot 4p$

迁移算法： $C_m = f \cdot (2P + 4p)$

读复制算法： $C_{rr} = f' \cdot [2P + 4p + Sp / (r + 1)]$

全复制算法： $C_{fr} = [1 / (r + 1)] \cdot (S + 2)p$

每一个表达式都有两部分，第一部分在“ \cdot ”的左边，表示访问远程数据项的概率。第二部分在“ \cdot ”的右边，等于访问远程数据项的平均代价。

第十一章 分布式共享存储器

11.3 使用目录的DSM

❖ 目录方案的分类

- 目录：不用广播的缓存器一致性协议必须保存每块共享数据的所有缓存器副本的位置。此缓存位置表，不管是集中的还是分散的，都叫做目录。
- 每个数据的目录项包括许多指针，用来指出此块各副本所在位置。每一个目录项还有一个“重写”位用来指明是否允许某一个(只有一个)缓存器进行写。
- 目录协议有三种主要类型：全映像目录、有限目录和链式目录。全映像目录的每个目录项保持 N 个指针，这里 N 是系统中处理器的个数。有限目录和全映像目录的不同之处在于，有限目录的每个目录项具有固定数量的指针，与系统中处理机数量无关。链式目录与全映像目录相似，只是它将目录分布于各缓存器。

第十一章 分布式共享存储器

11.3 使用目录的DSM

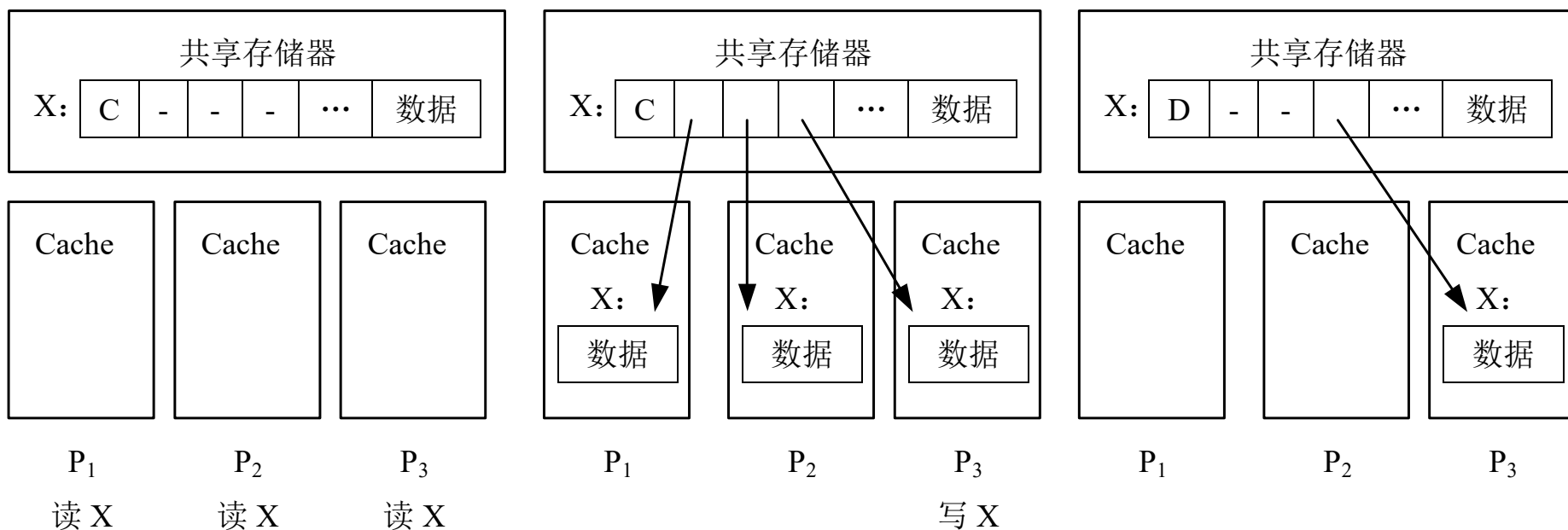
❖全映像目录

全映像目录协议使用的目录每项包含每个处理机，有一个指针并且有一个“重写”位。指针所对应的每一位代表该块在相应处理机缓存器中的状态(存在或不存在)。如果“重写”位置位，那么有且只有一个处理机的指针位被置位，允许这个处理机对该数据块进行写操作。缓存器保存每块数据的两个状态位：一位表明此数据块是否有效，另一位表明一个有效的数据块是否可写。缓存器一致性协议必须在存储器目录中保存这些状态位，并维持缓存一致性。

第十一章 分布式共享存储器

11.3 使用目录的DSM

❖ 全映像目录



第十一章 分布式共享存储器

11.3 使用目录的DSM

❖全映像目录

写过程：

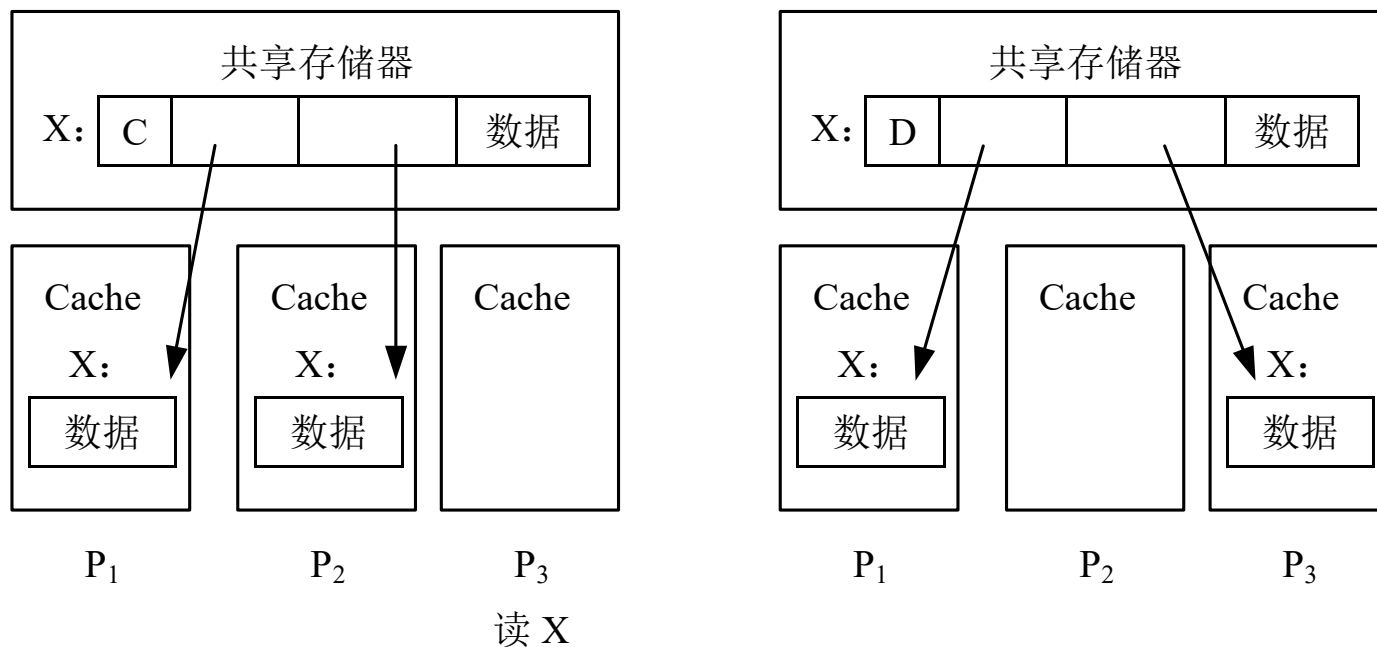
- (1) C_3 检测到包含单元X的数据块是有效的，但是该处理机对数据块无写的权限，这由块的允许写位表示。
- (2) C_3 发出一个对包含单元X的存储模块的写请求，并且停止处理机 P_3 。
- (3) 存储器模块向 C_1 和 C_2 发出无效请求。
- (4) C_1 和 C_2 收到无效请求后，设置对应的位指出包含单元X的数据块是无效的，并向存储器模块发回一个承认。
- (5) 存储器模块收到这个承认，将“重写”位置位，清除指向 C_1 和 C_2 的指针，并向 C_3 发出写允许报文。
- (6) C_3 收到写允许报文，更新该缓存器中的状态，并且激活处理机 P_3 。

第十一章 分布式共享存储器

11.3 使用目录的DSM

❖有限目录

有限目录协议是为了解决目录大小问题而设计的。限制对同一数据块同时进行缓存的任务数目，即限制一个数据块的缓存数目，就可以将每个目录项的大小限定为一个常数。



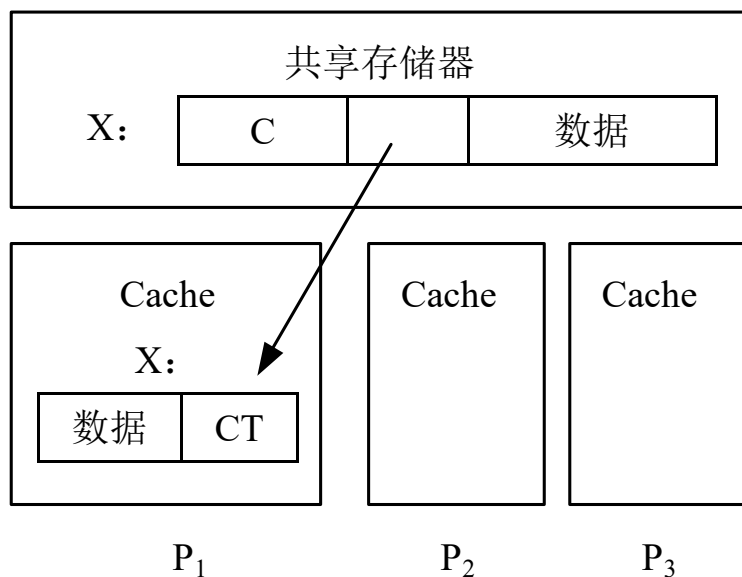
第十一章 分布式共享存储器

11.3 使用目录的DSM

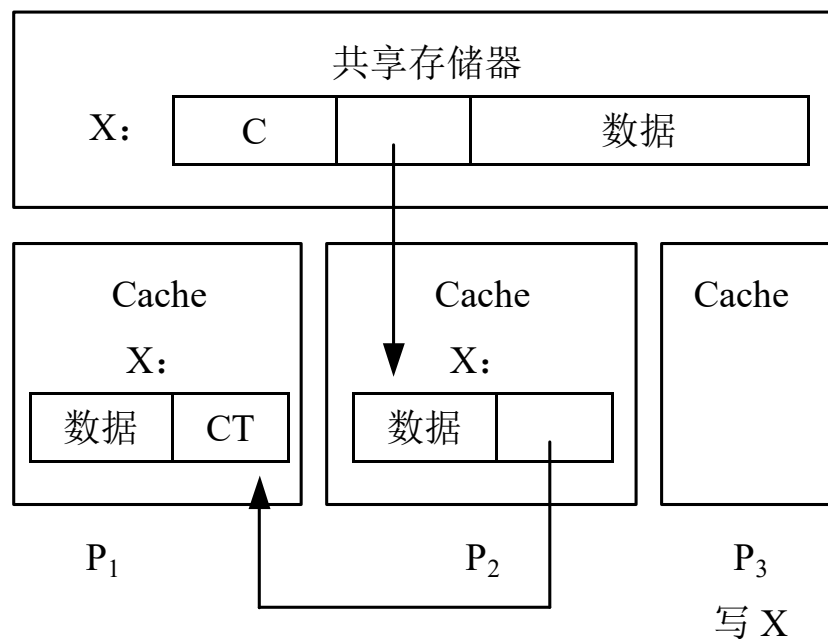
❖链式目录

它通过保持一个目录指针链对共享副本进行跟踪。

单向链式结构：



读 X



写 X

第十一章 分布式共享存储器

11.3 使用目录的DSM

❖链式目录

缓存器块的替换：假设从 C_1 到 C_N 都有单元 X 的副本，并且单元 X 和单元 Y 都直接映射到缓存器同一行上。如果处理机 P_i 读单元 Y ，必须从它的缓存器中先驱逐单元 X 。在这种情况下，有两种可能性：

- (1) 沿着链路向 C_{i-1} 发送一个报文，将 C_{i-1} 的指针指向 C_{i+1} ，将 C_i 从链路中脱离开。
- (2) 使从 C_i 到 C_N 中的单元 X 无效。

第十一章 分布式共享存储器

11.3 使用目录的DSM

❖链式目录

双向链式结构：另外一种解决替换问题的方法是使用双向链。这种方案为每个缓存器副本保持一个向前和一个向后的指针，这样当缓存器替换时，协议不必遍历整个链。双向链目录优化替换条件是以更大的平均报文长度（由于传送更多的目录指针）、缓存器中的指针的存储空间加倍和更为复杂的一致性协议为代价的。

第十一章 分布式共享存储器

11.4 DSM系统的实现

❖实现DSM的基本方法

DSM有三种实现方法，有的系统使用了不止一种方法。

- (1) 硬件实现。把传统的高速缓存技术扩展到可扩充的体系结构中。
- (2) 操作系统和程序库的实现。通过虚拟存储器的管理机构达到共享和一致性。
- (3) 编译程序的实现。把共享访问自动转换成同步和一致性原语。

一部分 DSM 系统的主要实现技术

系统名称	当时实现	结构和粒度	一致性语义	一致性协议	改进性能方法	同步支持
Dash	硬件, 4D/340 工作站, 网状网	16 字节	释放	写无效	松弛的一致性, 预取	排队锁, 原子增减
Ivy	软件, Apollo 工作站, Apollo 环	1KB 页	严格	写无效	指针链断开, 可选广播	同步的页, 信号灯事件计数
Linda	软件, 各种不同的环境	元组	无可变数据	可变	散列法	
Memnet	硬件, 令牌环	32 字节	严格	写无效	控制流的向量中断	
Mermaid	软件, Sun 工作站, DEC Firefly 多处理机, Mermaid 固有操作系统	8KB(Sun), 1KB (Firefly)	严格	写无效		信号灯, Signal/wait 报文
Mirage	软件, Vax11/750, Locus 操作系统, UNIX 系统 V, 以太网	512 字节页	严格	写无效	内核级实现, 时间窗口一致性协议	UNIX 系统 V 信号灯
Munin	软件, SUN 工作站 UNIX 内核及 Presto 并行程序设计环境, 以太网	对象	弱	指定类型用于读为主的协议, 延迟写更新	延迟修改队列	对象同步
Plus	软件和硬件, Motorola88000, Caltech 网状网, Plus 内核	页用于共享, 字用于一致性	处理器	非请求写更新	延迟操作	复杂的同步指令
Shiva	软件, Intel iPSC/2, 超立方体, Shiva 固有操作系统	4KB 页	严格	写无效	数据结构紧凑, 存储器用作后备存储	信号灯, Signal/wait 报文

第十一章 分布式共享存储器

11.4 DSM系统的实现

❖ 结构和粒度

- 1) DSM的硬件实现方法典型地支持了较小的粒度。
- 2) 页的大小：较大的页能够减少分页的开销，但是可能引起争用可能性越大。另一个影响页大小选择的因素是必须保留该系统有关页的目录信息：页越小，则目录越大。
- 3) 结构化共享存储器的一个实现方法是根据数据类型进行共享。这种方法是把共享存储器作为面向对象的分布式系统中的对象而进行构造。
- 4) 另一个方法是把共享存储器构造成像一个数据库。Linda就是一个这种模式的系统。它把它的共享存储器安排成为一个相联存储器，叫做元组(tuple)空间。

第十一章 分布式共享存储器

11.4 DSM系统的实现

❖ 数据定位与访问

- 1) 集中的服务员：集中的服务员来跟踪所有共享数据。这种集中的方法有两个缺陷：服务员串行执行定位查询，从而削弱了并行性；服务员负载过重，降低了整个系统的速度。
- 2) 广播数据请求：不幸的是，广播的可扩充性不好，所有的节点(不仅是数据所在的节点)都必须处理广播请求。广播在网络上的等待有可能使访问花费很长时间才能完成。
- 3) 基于所有者的分布式的模型：每一块数据都有一个与之相联系的所有者，这个所有者就是拥有数据主副本的节点。当数据在整个系统中迁移时，它的所有者也会随之而改变。当另一个节点需要数据的一个副本时，就向所有者发送请求。所有者如果仍保留着这个数据，就返回该数据；若所有者已将数据发送给其他节点，则把这一请求转发给那个新所有者。缺点是一个请求可能被转发多次后才能到达当前所有者。

第十一章 分布式共享存储器

11.4 DSM系统的实现

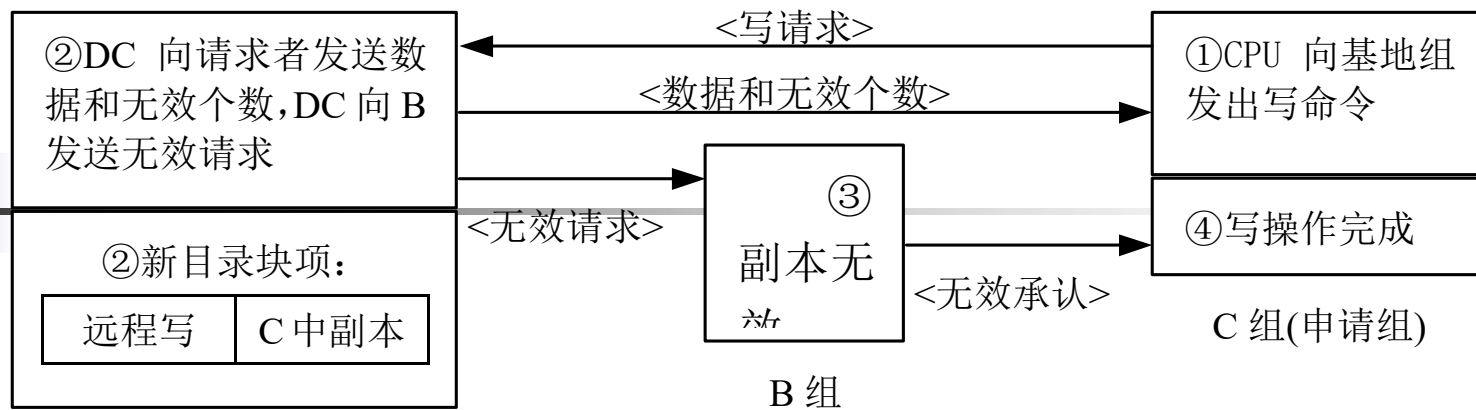
❖一致性协议

两类一致性协议：写无效协议和写更新协议

在写无效协议中，一块数据可能有很多个只读副本，但是，只有一个可写副本。这种协议之所以被称作写无效协议，是因为在开始一次写操作之前，除了将被写的那个副本之外，其他副本均变成无效。

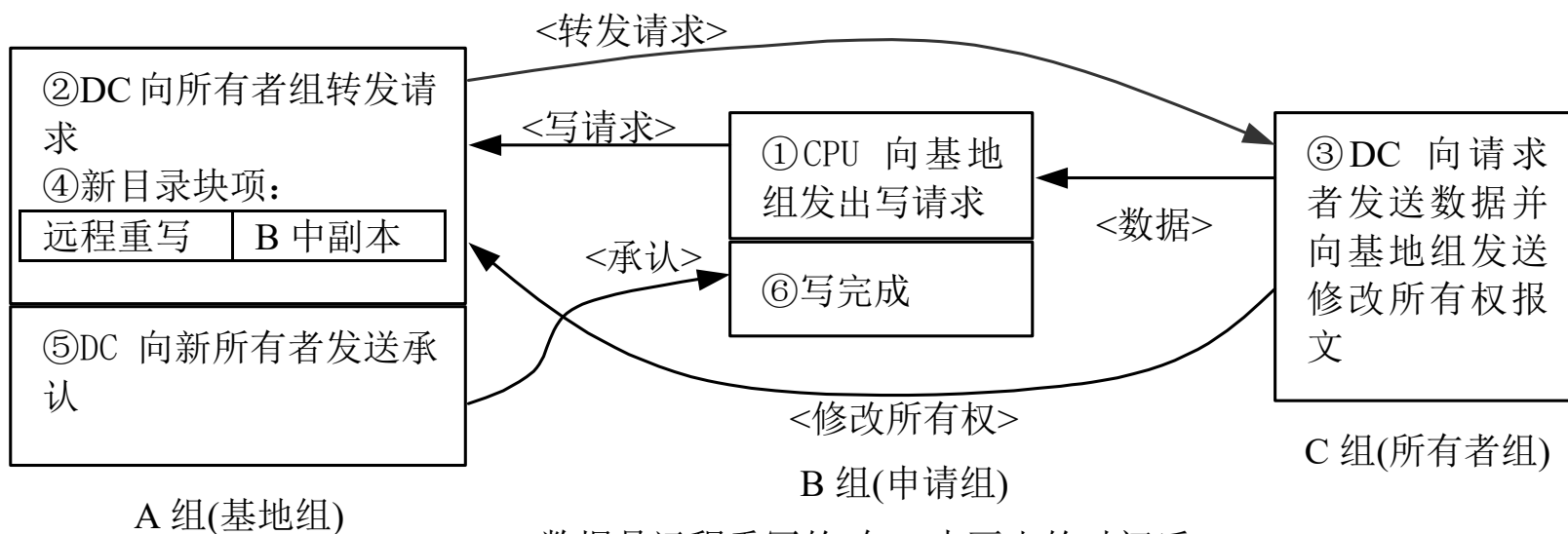
在写更新方式中，一次写操作将更新所有副本。

Dash系统简化的写无效协议 (DC代表目录控制器)



A 组(基地组)

(a)数据是远程共享的



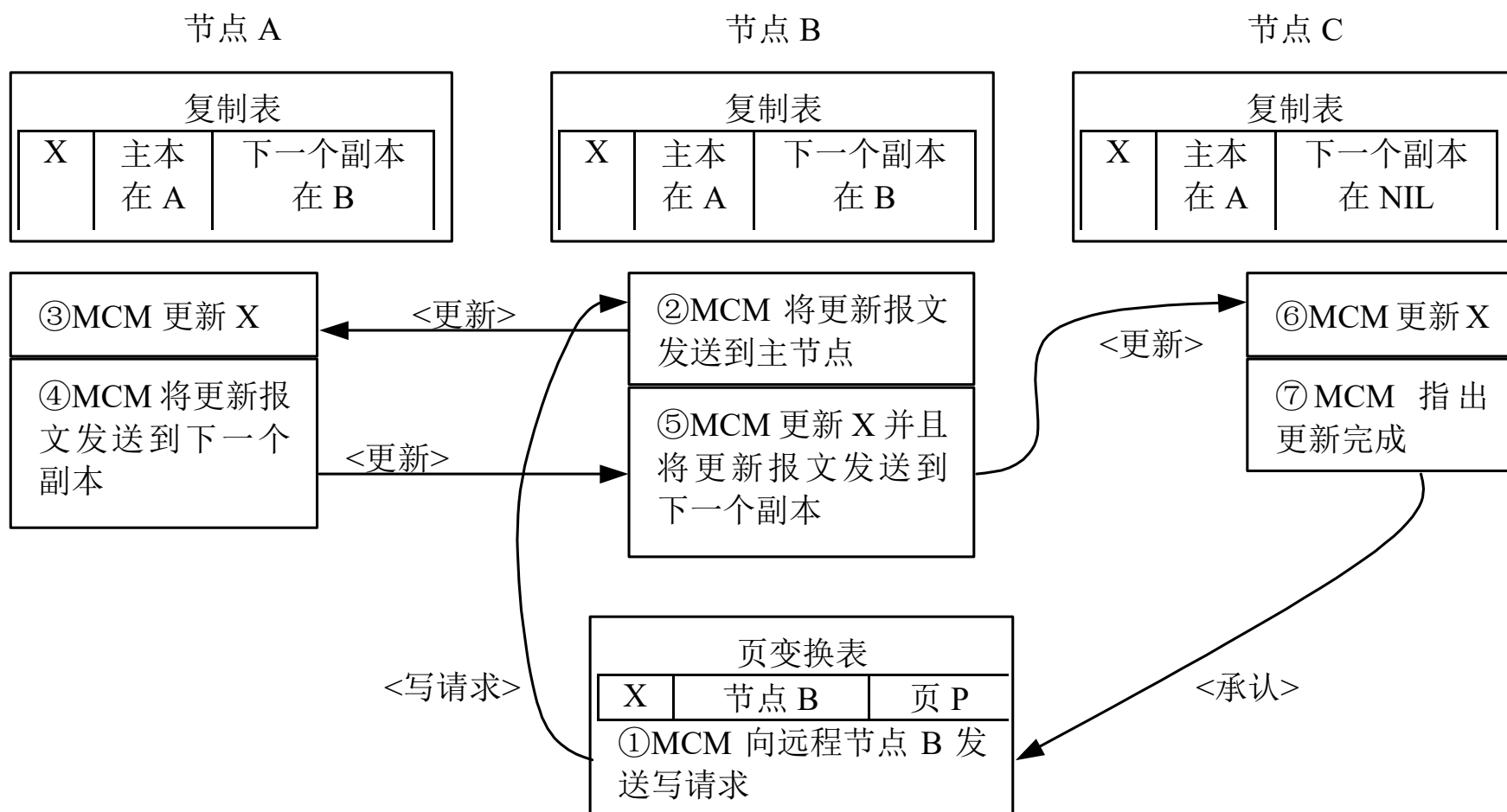
A 组(基地组)

B 组(申请组)

C 组(所有者组)

(b)数据是远程重写的(在(a)中画出的时间后)

Plus写更新协议，MCM代表存储一致性控制器



第十一章 分布式共享存储器

11.4 DSM系统的实现

❖ 颠簸

DSM系统特别容易出现颠簸。如果两个节点对一个数据项同时进行写，该数据项就有可能以高速率来来回回地被传送(乒乓效应)，任何实际工作都做不成。

- 1) Munin系统允许程序员把共享数据和类型联系起来：写一次、写多次、生产者—消费者、专用、迁移、结果、常读、同步及一般的读/写。为避免两个竞争写者的颠簸，一个程序员可以把类型指定为写多次，系统将使用延迟写策略。
- 2) Mirage系统在一致性协议中，增加了一个动态可调整参数，它决定一页在一个节点上保持可用的最小时间量(Δ)。例如若一个节点对一个共享页执行一次写操作，则此页在该节点上时间 Δ 内是可写的。

第十一章 分布式共享存储器

11.4 DSM系统的实现

❖ 可扩充性

DSM系统一个理论上的优点是它们比紧密耦合系统具有更好的可扩充性。前面说过，对可扩充性限制有两种因素：集中瓶颈（例如紧密耦合系统中的总线）和全局公用信息的操作及存储（例如广播报文或目录，它的大小与节点数成比例）。

第十一章 分布式共享存储器

11.4 DSM系统的实现

❖ 异构性

- 1) 在Agora系统中，把存储器构造为在异构性机器之间共享对象。
- 2) Mermaid探索了另一种不同寻常的方法：存储器以页方式共享，并且一页只包含一种数据类型。当在不同体系结构的两个系统之间移动一页时，变换子程序都会把该页上的数据转换成适当的格式。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy—软件实现的DSM

➤ Ivy中进程地址空间分成专用和共享两部分。专用部分不能由其他进程寻址。共享部分由虚拟共享存储器实现，是个平面地址空间，为运行在不同节点上的所有进程所共享，也就是被各线程共享的单地址空间。

➤ 地址空间是分页的。一页是同步的最小单位，当需要时它可以从一个节点迁移到另一个节点。每个节点上有一个存储器管理程序，满足本地和远程请求并实现缓存器一致性协议。当访问共享空间的一个地址时，阻塞故障进程，Ivy存储器管理程序检查待访问页是否在本地。如果不在本地，向远程存储器发送请求。得到该页后，发生页故障的进程恢复执行。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy一致性协议

➤ Ivy所使用的一致性概念是多个读/单个写的语义。对某地址的读操作总是得到最近对该地址写的值，一致性协议保证执行这一语义。Ivy的每个处理机都有自己的页表。表中的每一项纪录着该主机的访问权，可以对一页拥有读、写权或无任何权利。一页的访问权是与缓存器中一个块的状态相当的。

缓存器	Ivy
重写	写
共享重写	所有者读
有效	读
无效	无任何权力

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy一致性协议

当访问一个共享地址时，该主机检查它是否有权在指定方式下访问含有此地址的那一页。如果没有，则根据访问方式产生一个读或者写故障，其步骤如下：

读故障：

- (1) 找出谁是所有者；
- (2) 所有者把该故障主机填入副本集；
- (3) 所有者把自己的访问权变成只读；
- (4) 所有者向故障主机发送该页。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy一致性协议

当访问一个共享地址时，该主机检查它是否有权在指定方式下访问含有此地址的那一页。如果没有，则根据访问方式产生一个读或者写故障，其步骤如下：

写故障：

- (1) 找到所有者；
- (2) 所有者向故障主机发送该页和该页的副本集，并将它的那项标为无效；
- (3) 故障主机根据副本集送出“无效”报文；
- (4) 返回对“无效”报文的承认，进程继续执行。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy一致性协议

三种不同的一致性协议：

- 1) 集中管理者方法类似于管程，管程由一个数据结构和一些过程组成，提供对数据结构的互斥访问。集中管理者固定在一个处理机上，维持一张页表。每个处理机也有一张页表，每一项有两个域：访问和锁。访问域记录本地处理机上的页面的可访问信息。每个处理机知道中心管理者，并且在本地没有数据对象时能够向管理者发出请求。当一个处理机上有多个进程等待同一个页面时，加锁机制防止处理机发出多个请求。对一个页面成功执行写操作就会成为页面的所有者。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy一致性协议

三种不同的一致性协议：

- 2) 有两种类型的固定分布式管理者算法：固定的和广播的。固定算法中，每个处理机预定管理一部分页面。通常一个适当的散列函数用于把页面映射到处理机。广播算法中，访问页面出故障的处理机发出广播确定页面的真正所有者。广播算法性能比较差，因为所有处理机必须处理每个请求，减慢处理机的计算。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy一致性协议

三种不同的一致性协议：

- 3) 动态分布式管理者方法的核心是每个处理机的页表中记录所有页的所有者。页表中，集中管理者算法中的所有者域被替换成另一个域，叫做可能所有者 (proowner) 域，它可能是页面的真正所有者，也可能是页面的可能所有者。可能所有者域构成一个链，从一个节点到另一个节点，最终会指向真正的所有者。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy存储器管理

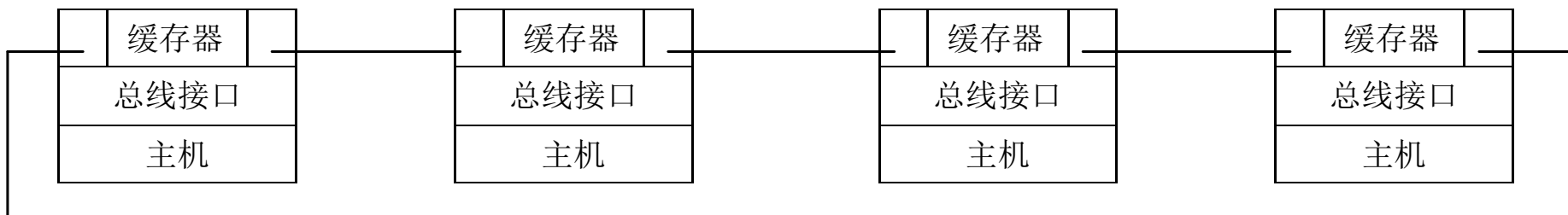
- (1) 页替换。 Ivy的虚拟共享存储器的页有五种：可写的、所有者可读的、只读的、空的和不使用的。空页和不用页都具有最高的被替换优先权，即如果需要一页则先替换它们。由于只读页可被其所有者制造备份，所以可以简单地丢弃。对于所有者读的页和可写页的丢弃当然需要所有权的转让。
- (2) 存储器分配。 为了支持动态数据结构，使用动态存储器分配方案是必要的。 Ivy有两种方法。第一种是集中式方法，即所有进程向集中式的“存储器分配程序”请求存储器和归还存储器。这是个简单的方法。第二个方法，除了集中式分配程序外，每个节点还有它自己的分配子程序。每个节点请求一大块存储器并执行来自本地进程的存储器请求。仅在本节点用完存储器时才和集中式管理程序联系。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ MemNet—硬件实现的DSM

在MemNet中，全部主机共享一个公共的地址空间。此地址空间是分页的，这些页可根据要求在系统内迁移。对此地址空间的访问被直接送到主机内的接口部件(作为一个智能存储器模块)。这个部件能缓存一部分共享存储器并和其他主机的这种部件相互作用调进另外的页。



第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ MemNet缓存一致性协议

MemNet使用的缓存一致性协议和Ivy使用的相同，即读操作必须总是返回该数据最近的值。每个MemNet部件有一块表，表中每项对应整个共享地址空间中的每一块，还包括下述状态标志：

- (1) 有效。指出此缓存器对该块是否有一个有效副本。
- (2) 独占。指出本主机对该块是否有独占的访问权。
- (3) 保留。指出该块的保留空间是否在本主机中。
- (4) 位置。如果块是在本主机内，指出该块副本的位置。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ MemNet缓存一致性协议

- 1) 当该部件收到从处理机来的读请求时，先检查是否有一个覆盖该地址的块的有效副本。如果有，则请求很容易满足，否则发送一个带有空白的数据请求报文。每个主机依次检查MemNet请求。第一个具有有效副本的主机满足读请求。
- 2) 当MemNet部件收到对共享空间某地址的写请求时，先检查它是否有一个有效副本和对该块的独占访问权。如果是，则请求被满足。否则，如果该部件有该块的一个有效副本但对其无独占访问权，则发送“无效”请求。最后，如果该部件没有该块的有效副本，则发送一个独占数据请求。当一个部件收到一个“无效”请求时，如果该块被缓存则使其无效。独占的数据请求除有类似的作用外，具有该块有效副本的第一个部件还必须在无效前供给该块。当最初的请求返回时，被阻塞的进程就恢

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ MemNet缓存一致性协议

- 3) MemNet的缓存替换方法是为每块设置一个保留区，使用随机替换策略。当一个部件要从其缓存器中替换一个块时，发送更新块请求并带走此块。为此块保留空间的部件为此请求服务，将此保留空间更新。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy与MemNet的比较

- 1) Ivy的目标是支持并行处理并证明用软件实现共享虚拟存储器是可行的； MemNet则侧重于DSM的硬件实现，使用共享总线把部件连到处理机上，这样很自然地要共享物理地址空间。
- 2) Ivy对页定位问题研究得很全面，认为固定分布是最简单和有效的方案，转发方案具有全分布式控制的优点。MemNet则在一个令牌环上实现，使用广播简单地确定最近的副本的位置。
- 3) Ivy对每页维持一个副本集以避免使用广播方法的“无效”操作，而MemNet则利用其令牌环的优点并使用广播方式。

第十一章 分布式共享存储器

11.5 DSM实例：Ivy和MemNet

❖ Ivy与MemNet的比较

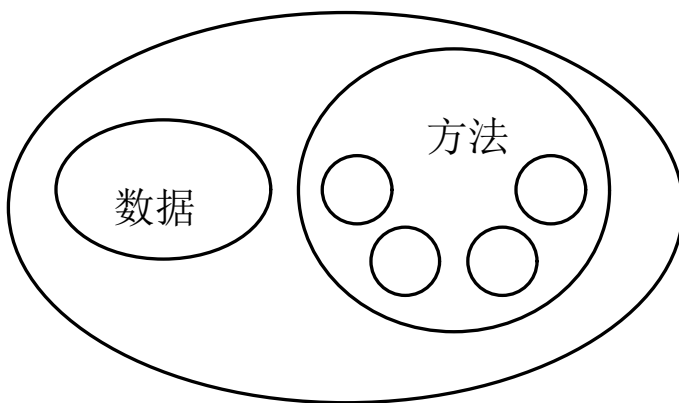
- 4) Ivy和MemNet都使用单个写多个读的协议。读操作总是返回一页的最新内容。
- 5) Ivy的同步单位是页。如果使用Ivy开发细粒度并行性是不现实的，所以页长为1KB是合理的。MemNet的高速令牌环和硬件实现允许开发较细粒度的并行性，因此其同步单位较小，仅为32字节。
- 6) Ivy研究了使用磁盘和其他节点主存储器进行替换的方法。后一方法由于未来工作站可能具有更大主存而更加有意义。
MemNet的不寻常之处是保留主存对块进行后备。这对MemNet是很重要的，因为它依赖于可预言的短的故障修复时间以避免任务切换。

第十二章 基于对象的分布式系统

12.1 分布式对象

❖对象的概念

对象是一个抽象体，它将相关的服务和数据封装在一起。对象服务以函数的形式提供，是对数据进行的操作，被称为对象的方法(method)。对象里的数据也被称为状态(state)。



第十二章 基于对象的分布式系统

12.1 分布式对象

❖对象的概念

对象的接口和对象对接口的实现：

通过对象的接口调用对象的方法，从而可对对象的状态进行访问或操作，除此之外，无任何对对象的状态进行访问和操作的合法手段。一个对象可以实现多个接口；同样，给定一个接口的定义，多个对象可能对这一定义提供实现。接口和对象对接口的实现是独立的。

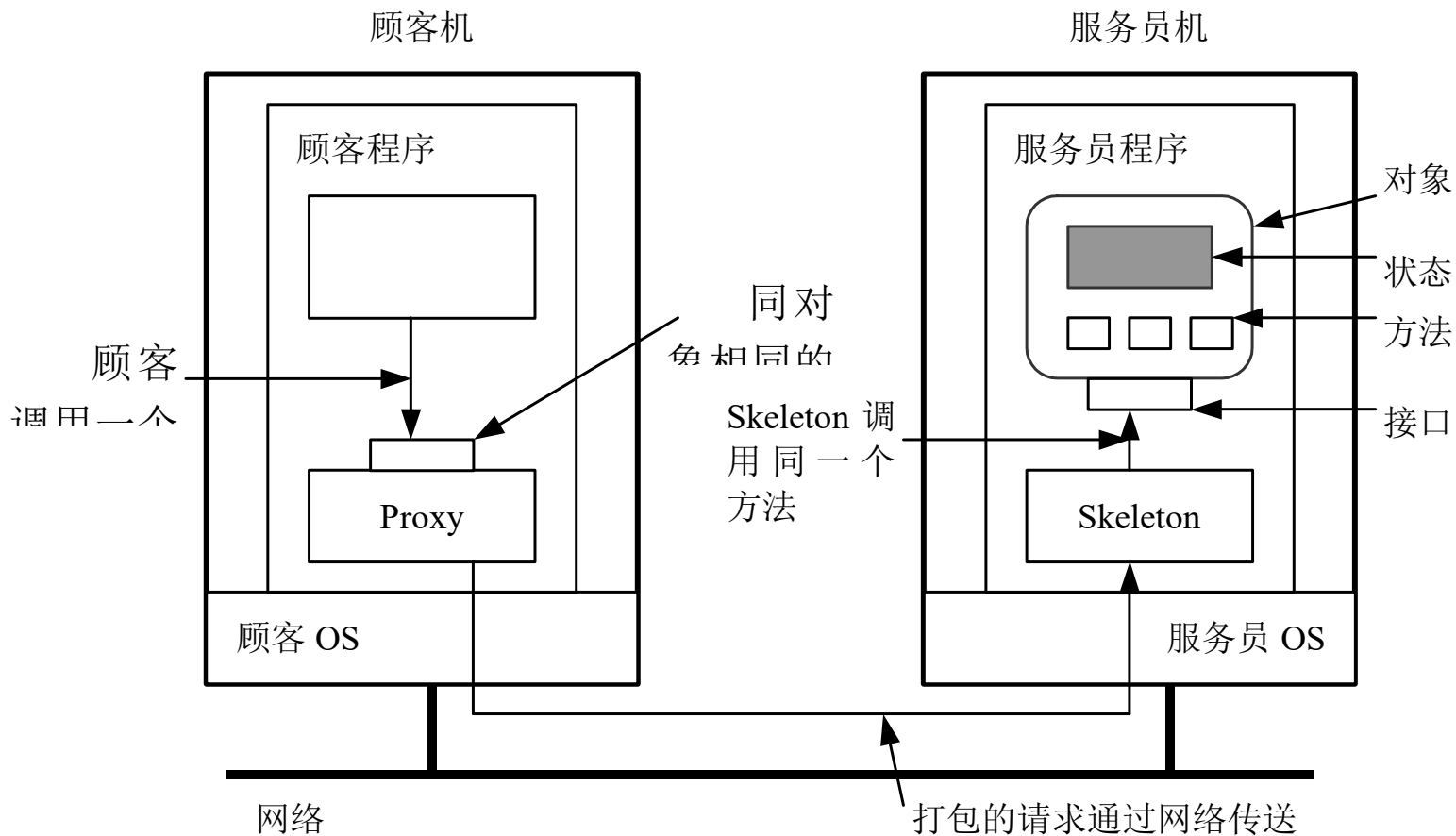
分布式对象：

在一个机器上设置一个接口，而对象本身驻留在另一个机器上，这通常被称为分布式对象。

第十二章 基于对象的分布式系统

12.1 分布式对象

❖ 对象的概念



第十二章 基于对象的分布式系统

12.1 分布式对象

❖ 对象的类型

➤ 编译时对象和运行时对象：

- 1) 编译时对象(compile-time object)。在这种情况下，一个对象是程序所定义的类(class)的一个实例，一个类是一个模块的抽象描述，这个模块由一组数据元素和对这些数据的一组操作组成；
- 2) 另外一种对象是在运行时通过明确的方式构建的，称为运行时对象(runtime object)。由于运行时对象不依赖于分布式应用程序是使用什么语言编写的，所以这种方法在许多基于对象的分布式系统中得到采用。特别是，一个应用程序可以使用由多种语言编写的对象来实现。

第十二章 基于对象的分布式系统

12.1 分布式对象

❖ 对象的类型

➤ 运行时对象的实现

- 通常的方法是使用一个对象适配器(object adapter)，它的主要作用是将用户编写的程序包装成具有对象的外观。对象适配器允许将一个接口转变成顾客所希望的形式和功能。例如一个适配器可以将上面所描述C库函数动态地和一个对象结合(bind)并打开一个对应的文件作为该对象的当前状态。
- 对象适配器在基于对象的分布式系统中具有重要的作用。为了使包装尽量容易，一个对象由它所实现的接口进行定义。实现的接口在适配器中进行注册，然后这个接口对远程请求来说就是可用的。适配器监视着所发出的对对象的调用请求，给顾客提供一个远程对象的映像。

第十二章 基于对象的分布式系统

12.1 分布式对象

❖ 对象的类型

➤ 持久性对象和暂时性对象：

- 一个持久性对象是一个持续存在的对象，即使是在当前它并不包含在一个服务员进程的地址空间中，它仍然是存在的。也就是说，一个持久性对象并不依赖于它当前的服务员。在实际中，这就意味着当前管理这个持久对象的服务员可以将这个对象的状态保存在磁盘中，然后退出。随后，一个新启动的服务员能够从磁盘中将这个对象的状态读取到自己的地址空间中去，然后处理调用请求。
- 相反，一个暂时性对象只能随管理它的那个服务员的存在而存在。一旦服务员退出，对象也就随之消失了。

第十二章 基于对象的分布式系统

12.2 CORBA

CORBA是Common Object Request Broker Architecture的简称，即通用对象请求代理结构。

❖CORBA的总体结构

CORBA系统由四组构造元素组成，这四组构造元素由一个叫做ORB(Object Request Broker)的机构连接。

- 1) ORB是任何一个CORBA分布式系统的核心，它负责在对象和其顾客之间建立通信，并将分布和异构性的问题隐藏起来。在许多系统中，ORB是以库函数的形式实现的，由顾客程序和服务员程序连接，提供基本的通信服务。

第十二章 基于对象的分布式系统

12.2 CORBA

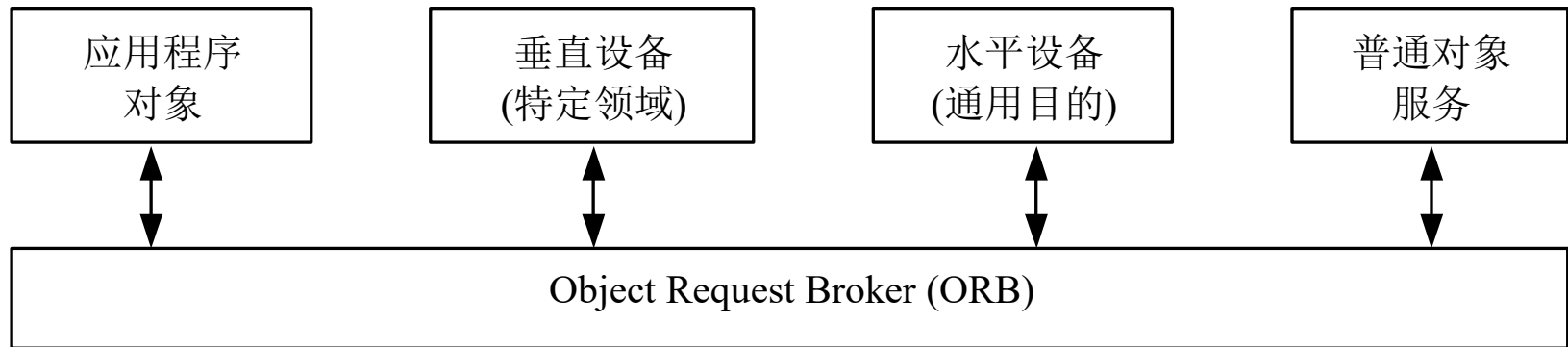
❖CORBA的总体结构

- 2) 应用对象是作为应用程序的一部分而建立的。
- 3) 除此之外，CORBA还提供了一些设备，这些设备用于构建CORBA服务。参考模型对两类CORBA设备进行了区分，它们被分成两组，一组为水平设备，另一组为垂直设备。水平设备由通用的高层服务组成，这些高层服务和特定的应用领域无关。这些服务当前包括为用户接口提供的服务、为信息管理提供的服务、为任务管理(用于定义工作流系统)提供的服务等。
- 4) 垂直设备由那些为特定的应用领域而提供的高层服务组成，例如电子商务、银行、制造业等。
- 5) 第四组构造元素是普通的对象服务。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的总体结构



第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的对象模型

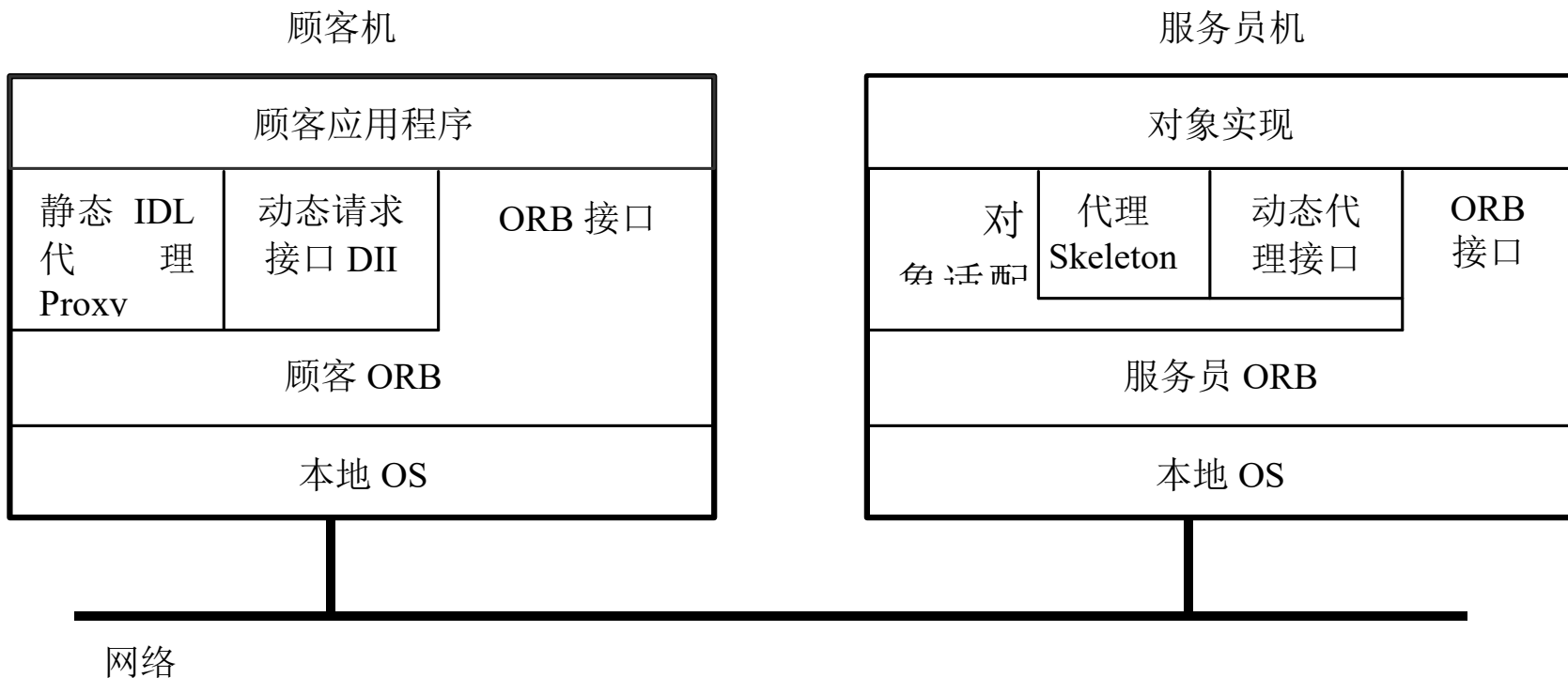
- CORBA使用远程对象模型，在这种模型中，CORBA对象只驻留在一个服务员的地址空间中。
- 在CORBA中，使用CORBA的接口定义语言IDL (Interface Definition Language) 对对象和服务进行说明。CORBA IDL和其他的接口定义语言类似，在这种语言中，它提供了精确的语法用于表达方法和参数。
- 接口说明可以只用IDL语言给出，但是在CORBA中，必须提供一个严格的规则用于将IDL描述的接口说明转换成所使用的程序设计语言。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的对象模型

CORBA系统的一般组织方式：



第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的对象模型

CORBA系统的一般组织方式：

- **ORB**：ORB是一个运行时系统，负责处理顾客和对象之间的基本通信问题，这些通信保证请求被传送给对象的服务员和应答返回给顾客。ORB本身只提供少量的服务：
 - 1) 处理对象引用(object reference)，对象引用一般来说依赖于特定的ORB。
 - 2) ORB提供对对象引用的打包和拆包操作，使得进程之间能够交换信息，同时还提供引用之间的比较操作。
 - 3) ORB所提供的其他操作还包括为一个服务找到对应的进程，这需要用到名字服务。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的对象模型

CORBA系统的一般组织方式：

- **顾客静态代理：**一个顾客应用程序有一个Proxy代理，Proxy代理将调用请求打包传送给服务员，并且将来自服务员的响应进行拆包传送给顾客程序。需要注意的是，Proxy和ORB之间的接口不必是标准化的，因为CORBA假定所有的接口都是由IDL描述的，CORBA为用户提供了一个IDL编译器，这个编译器能够生成必要的代码用于处理顾客和服务员的ORB之间的通信。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的对象模型

CORBA系统的一般组织方式：

- **动态调用接口DII**：在某些情况下，静态定义的接口对顾客来说是不适用的，相反顾客需要在运行时期间确定一个特定对象的接口是什么样的，然后构建一个对该对象的调用请求。为此目的，CORBA为顾客提供了一个动态调用接口DII(Dynamic Invocation Interface)。DII提供了一个通用的调用操作，该操作从顾客那里得到一个对象引用、一个方法标识符、一些作为参数的输入值和一些由调用者给出的变量用于存放返回的结果。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的对象模型

CORBA系统的一般组织方式：

- **服务员方的构件：**在服务员方，CORBA提供了一个对象适配器(object adapter)，它负责将收到的请求转发给正确的对象。实际的拆包操作由CORBA中的服务员方的代理Skeleton负责处理。同顾客方的代理一样，服务员方的代理可以是静态的，也可以是动态的。静态的代理由IDL所定义的说明编译得到。当使用动态的通用代理，对象必须实现一个调用函数。

第十二章 基于对象的分布式系统

12.2 CORBA

❖ 接口库和实现库

- 1) CORBA提供了一个接口库，里面保存所有接口定义，允许对调用请求的动态解释。在许多CORBA系统中，有一个独立的进程提供了一个标准的接口用于向接口库保存和检索接口定义。当一个接口定义被编译的时候，IDL编译器会赋予这个接口一个库标识符，这个标识符是从库中检索一个接口定义的基本手段。
- 2) 一个实现库中包含了所有需要实现和激活的对象。对象适配器负责激活一个对象，保证对象能够在一个服务员的地址空间中运行，从而使得对象的方法能够被调用。对一个给定的对象引用，适配器能够与实现库联系从而准确地找到一个对象的实现。例如，实现库应该维持一张表，该表记录了哪一个服务员需要启动，服务员需要为特定的对象监听哪一个端口号。实现库还要提供服务员需要装载和执行哪一个可执行文件等信息。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的服务

CORBA 的服务

服务	功能描述
收集	对象被分组聚集到一个链表中、一个队列中、一个堆栈中或一个集合中
查询	提供了一种手段构建对象组，使对象可以使用一种声明的方式进行查询
并发	允许对共享对象进行并发访问
事务处理	允许顾客在一个事务处理中访问多个对象，支持平面的和嵌套的事务处理
事件	通过事件支持异步通信
通告	提供一个高级设备支持基于事件异步通信
外观化	处理对对象打包或拆包，使得对象可以被存储在磁盘上或者通过网络发送。
生存周期	支持创建、删除、拷贝、和移动对象
许可	允许对象的开发者将一个许可附加到一个对象
命名	提供对象名字到对象标识符之间的映射
性质	使用一些(参数，值)对来对对象的性质进行描述
交易	提供一些关于该对象的一些广告信息
持久性	支持对象的持久性存储
关系	表达对象之间的关系
安全性	提供诸如认证、权限、安全通信和管理
时间	在一定的误差范围内提供当前的时间

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的通信

➤对象调用模型

CORBA 支持的三种不同请求模型

请求模型	失效语义	功能描述
同步	最多一次	调用者阻塞直到收到响应或者出现异常
单程	尽力传输	调用者立即继续执行不等待服务员的响应
延期同步	最多一次	调用者立即继续执行，在以后某个时刻阻塞等待响应

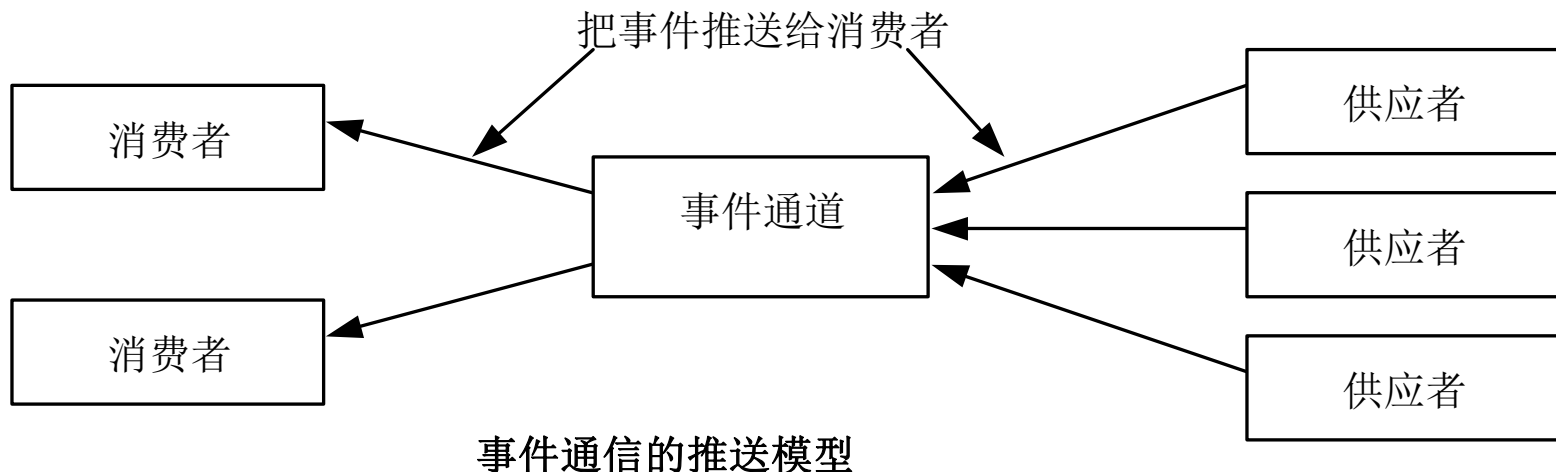
第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的通信

➤事件和通告服务

CORBA定义了一个事件服务，用于提供基于事件的通信。事件由供应者产生，由消费者接收，事件通过事件通道传输。



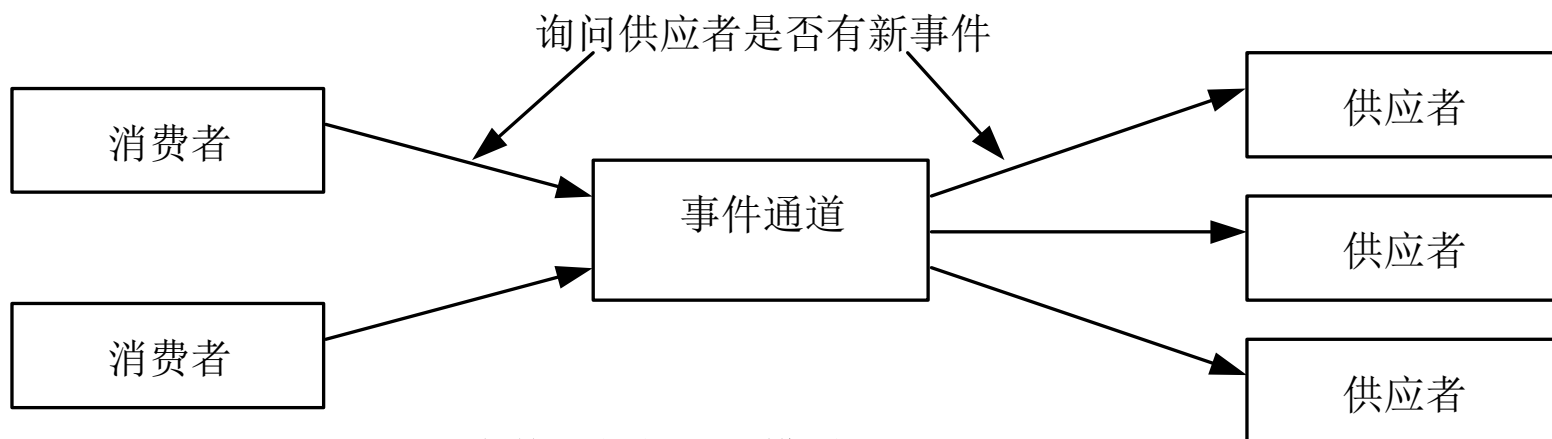
第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的通信

➤事件和通告服务

CORBA定义了一个事件服务，用于提供基于事件的通信。事件由供应者产生，由消费者接收，事件通过事件通道传输。



事件通信的拉取模型

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的通信

➤事件和通告服务

CORBA的事件服务存在如下缺点：

第一，为了传播事件，供应者和消费者必须都连接到事件通道上。如果消费者是在一个事件已经产生之后才连接到事件通道上的，那么这个事件就会丢失。也就是说，CORBA的事件服务不支持持久性事件。

第二，消费者无法对事件进行过滤，每个事件被传递给全部的消费者。如果要区分不同的事件类型，需要给每一类事件独立地设立一个事件通道。CORBA在通告(notification)服务中增加了过滤能力，在这种服务中，如果没有消费者对某个特定的事件感兴趣，则这个事件不会传播。

第三，事件的传播是不可靠的。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的通信 ➤报文队列

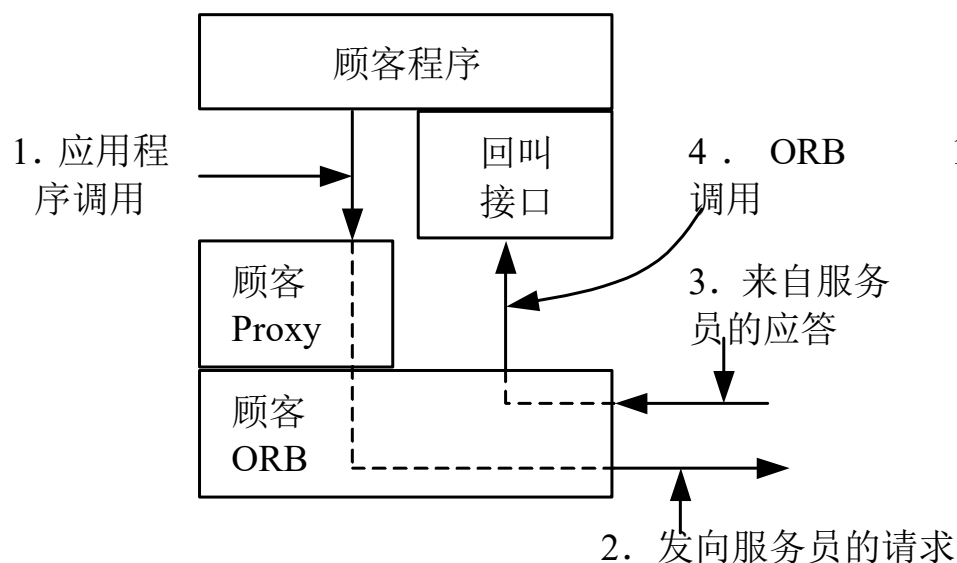
CORBA的支持持久性通信的模型称为报文排队(message-queuing)模型，CORBA使用了一个附加的报文服务(messaging service)来支持这种通信模型。在报文服务的情况下，有两种附加的异步的方法请求方式，一种是回叫(callback)模式，另一种是收集(polling)模式。

为了支持持久性通信，ORB的底层通信系统应该能够保存报文，直至它们被接收为止。

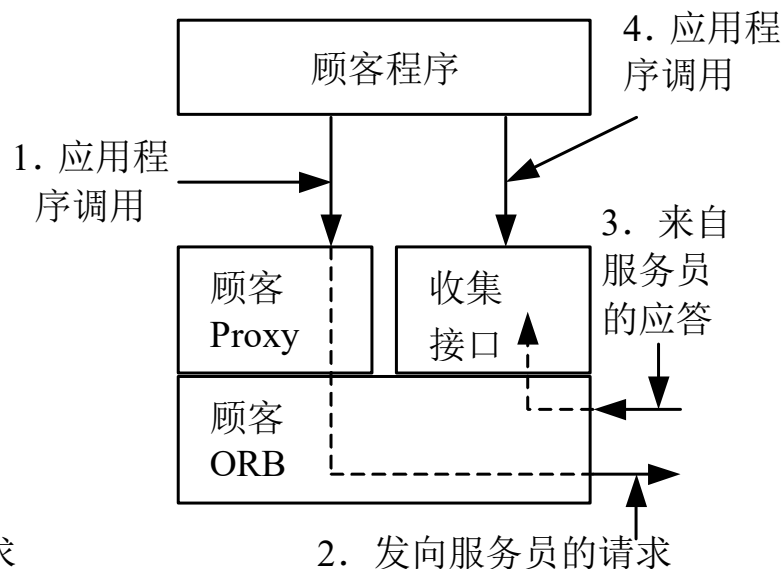
第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的通信 ➤报文队列



(a) 回叫模式



(b) 收集模式

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的通信

➤互操作性

不同厂商提供的CORBA系统中，顾客和对象服务员之间的通信有自己的方式。为了解决互操作性问题，CORBA引入了一个标准的协议，称为通用的ORB间互操作协议GIOP (General Inter-ORB Protocol)，顾客和服务员都遵守这个协议。

其中最重要的两个报文类型是Request和Reply，Request报文包含了一个被打包的调用请求，请求中包含对象的引用，被调用方法的名字，所需要的输入参数。每个Request报文中还有自己的请求标识符，用于随后对对应的响应进行匹配。

Reply报文包含了被打包的返回值和输出参数，不必要明确指出对象和方法，只需用一个和对应的Request报文中的请求标识符相同的标识符即可。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的通信 ➤互操作性

GIOP 报文类型

报文类型	发送者	功能描述
Request	顾客	包含调用请求
Reply	服务员	包含对一个请求的响应
LocateRequest	顾客	请求一个对象的准确位置
LocateReply	服务员	包含一个对象的位置信息
CancelRequest	顾客	取消一个请求，不再希望得到对应的应答
CloseConnection	顾客或服务员	指出要关闭连接
MessageError	顾客或服务员	包含一个错误信息
Fragment	顾客或服务员	对大的报文进行分段

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的POA

POA: 在CORBA中，对象适配器称为POA (Portable Object Adapter)，即可移动的对象适配器。它负责将服务员方的程序代码改编成顾客可调用的CORBA对象。依赖于POA的支持，使用何种语言书写服务员方程序代码不依赖于特定的ORB。

雇佣: CORBA采取了对象部分地由一个被称为雇佣(servant)的部件提供实现。雇佣是一个对象的一部分，它实现顾客可以调用的方法。雇佣不可避免地要依赖于程序设计语言。例如用C++或者JAVA实现雇佣时，典型的做法是提供一个类的实例。另一方面，用C或其他过程式语言编写的雇佣一般包含一组函数，这组函数对一个数据结构进行操作，这个数据结构代表对象的状态。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的POA

如何使用一个雇佣来构建一个CORBA对象的映像的呢？

- 每个POA提供如下操作：

`Objectid activate_object(in Servant p_servant);`

这个操作已经在CORBA IDL中说明。这个操作的输入参数是一个指向某个雇佣的指针，返回值是一个CORBA对象的标识符。

- C++中，雇佣被映射成为一个指针，该指针指向一个事先定义好的ServantBase类。这个类中包含了一组方法定义，这些方法定义要求每个C++雇佣必须实现。

第十二章 基于对象的分布式系统

12.2 CORBA

❖CORBA的POA

如何使用一个雇佣来构建一个CORBA对象的映像的呢？

- 假定我们编写了一个ServantBase的子类My_Servant，并且产生了一个C++对象作为My_Servant的一个实例，这个实例可以通过如下方式转换成一个CORBA对象：

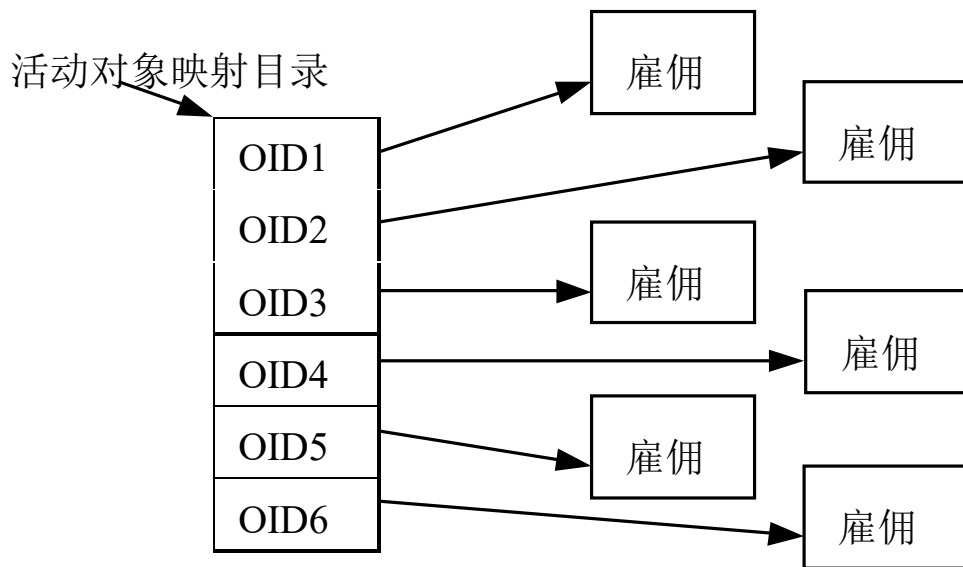
```
My_Servant *my_object;           //定义一个C++对象的指针
CORBA::Objectid_var oid;         //定义一个CORBA对象
标识符
my_object=new My_Servant;        //创建一个新的C++对象
oid=poa->activate_object(my_object); //将C++对象注册为一个CORBA对象
```


第十二章 基于对象的分布式系统

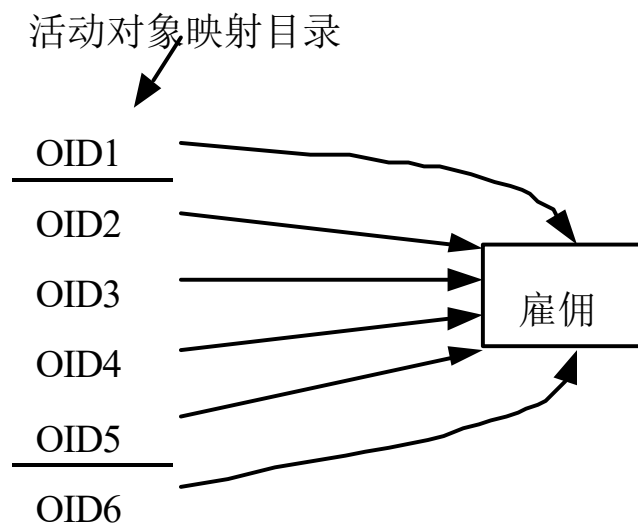
12.2 CORBA

❖CORBA的POA

一个雇佣可能只与一个对象标识符对应，也可能与多个对象标识符对应。



(a) 一个雇佣对应一个对象



(b) 一个雇佣对应一个类的所有对象

第十二章 基于对象的分布式系统

12.3 DCOM

❖COM和DCOM

- 基本的COM模型能够将一些逻辑元素看作是独立的，同时使得灵活的二进制组件适应不同的配置和不同的机器，COM被认为是ActiveX的核心技术。任何支持COM组件的软件工具自动地支持DCOM，DCOM是COM的分布式扩展，它允许软件组件能够通过网络直接进行通信。
- COM的目标是支持这样一类组件的开发，这类组件能够被动态激活，组件之间能够相互作用。COM中的每个组件是一个可执行代码，可以包含在一个动态连接库中，也可以以可执行程序的形式存在。

第十二章 基于对象的分布式系统

12.3 DCOM

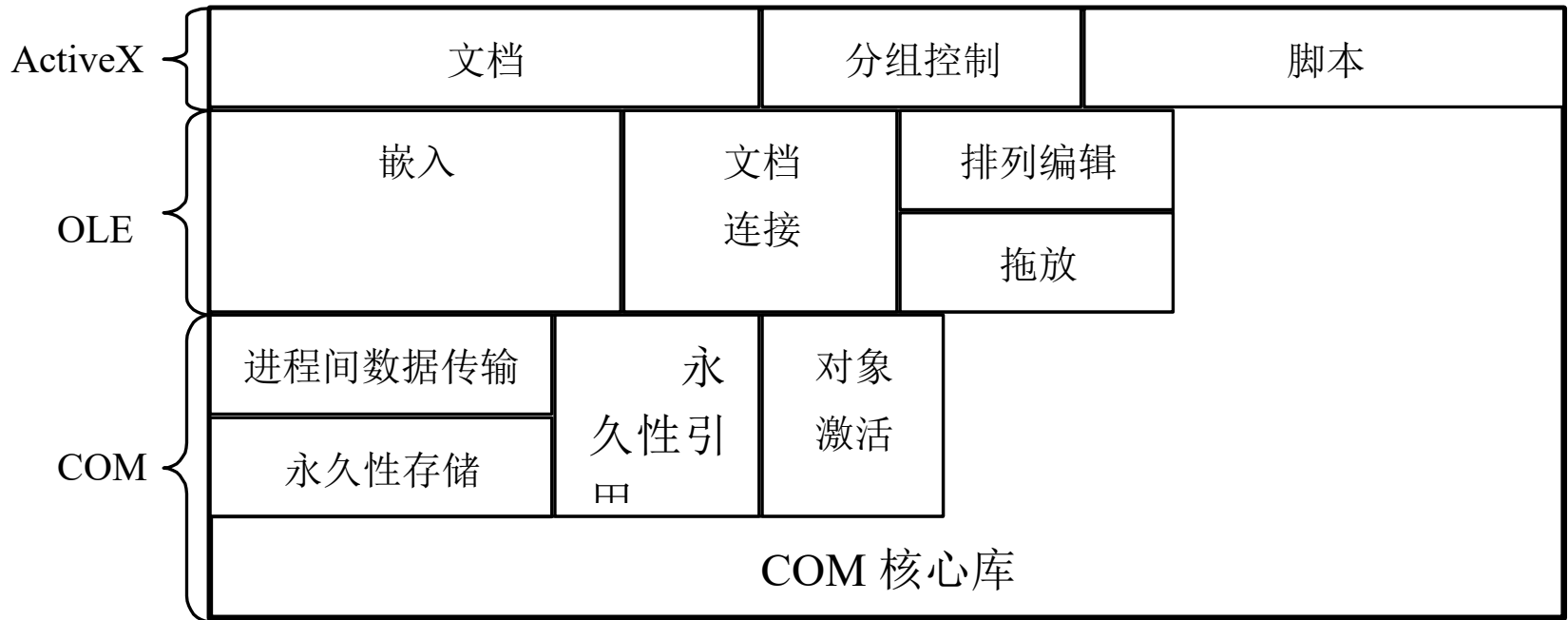
❖COM和DCOM

- 为了支持千变万化的复合文档，Microsoft需要一种通用的方法来区分文档的不同部分并且能够将它们粘合在一起。这就导致了OLE的出现，OLE代表对象连接和嵌入(Object Linking and Embedding)。
- ActiveX覆盖了OLE的所有内容并增加了一些新的特点。这些新特征包括能够在不同的进程中灵活地启动组件，支持脚本，将对象分组(分组被称为ActiveX控制)。

第十二章 基于对象的分布式系统

12.3 DCOM

❖ COM和DCOM



第十二章 基于对象的分布式系统

12.3 DCOM

❖ DCOM的对象模型

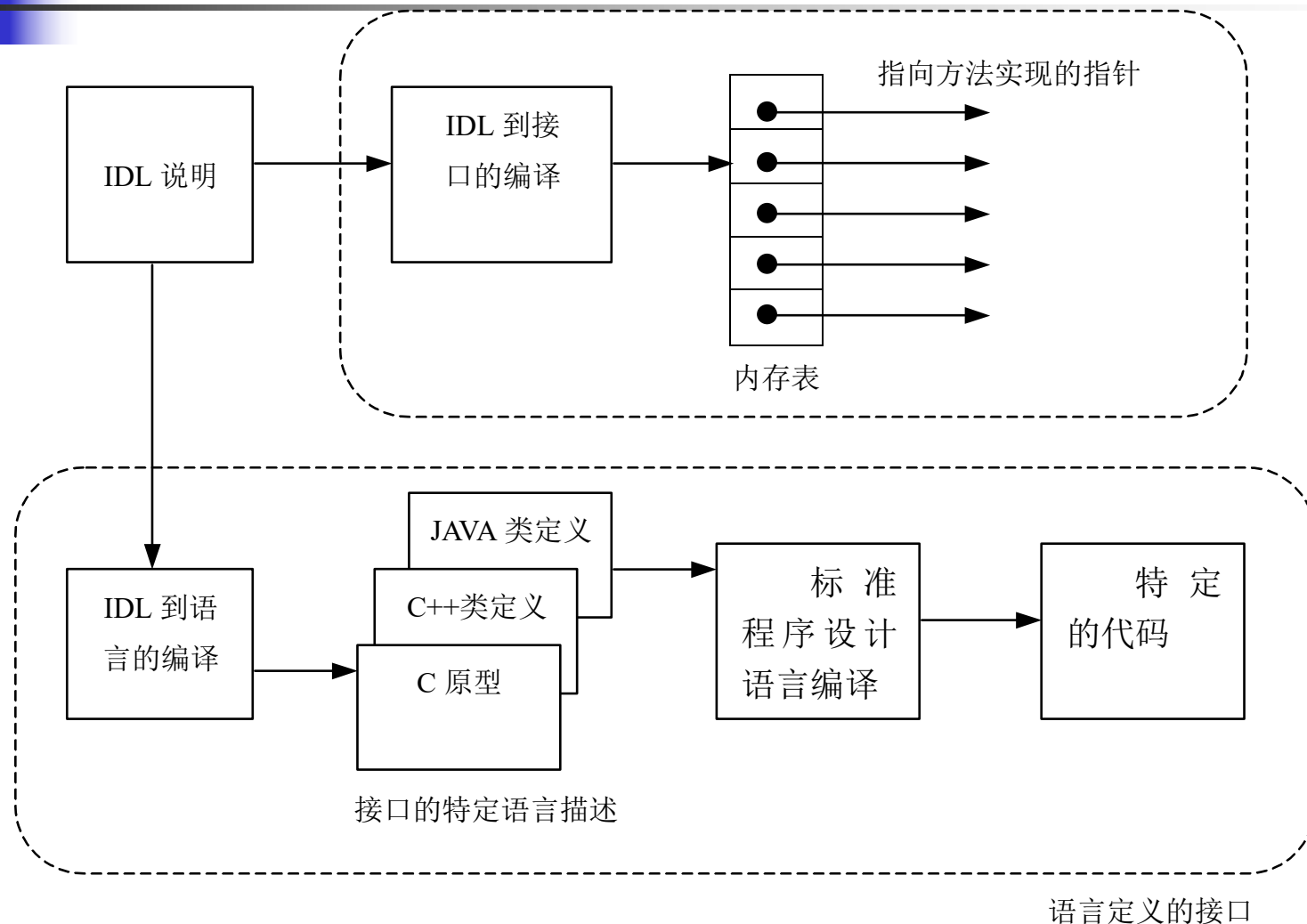
同所有的其他基于对象的分布式系统一样，DCOM采用的是远程对象模型。

- DCOM对象模型的中心问题是其接口的实现，一个对象可以同时实现多个接口。同CORBA所不同的是，DCOM中对象的接口都是二进制接口。
- 在CORBA的情况下，每当需要支持另外一种程序设计语言，那么从IDL说明到该语言的映射就需要进行标准化。在二进制接口的情况下，这种标准化是不需要的。
- 同CORBA对象模型的一个重要区别是，在DCOM中，所有对象都是暂时性的。
- DCOM也支持动态地对象调用。

第十二章 基于对象的分布式系统

12.3 DCOM

二进制接口



第十二章 基于对象的分布式系统

12.3 DCOM

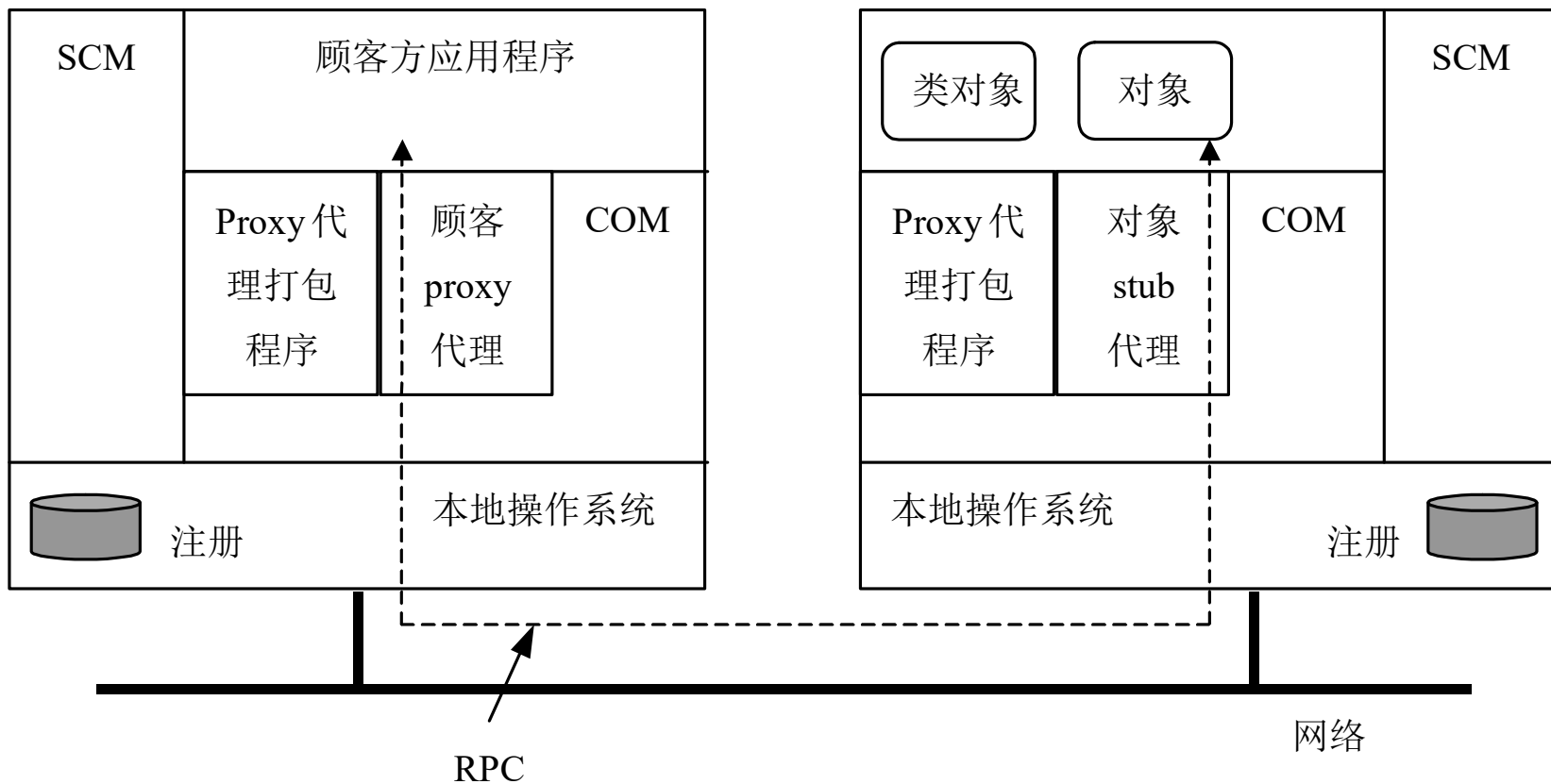
❖DCOM的类型库和注册

- DCOM的类型库和CORBA中的接口库相同。一个类型库一般对应一个应用程序或者由不同类对象组成的一个较大的组件。
- DCOM使用了Windows的注册机制，该机制和一个被称为服务控制管理员SCM(Service Control Manager)的特殊进程结合在一起。注册用来记录一个对象标识符CLSID到一个本地文件名的映射，该文件包含了对应的类的实现。
- 当一个对象要在一个远程机上执行时，顾客需要和那个主机上的SCM联系，SCM负责激活对象，这和CORBA中的实现库类似。远程机上的SCM浏览它自己的本地注册查找和指定的CLSID对应的库文件，然后启动一个进程，要执行的对象驻留在这个进程中。

第十二章 基于对象的分布式系统

12.3 DCOM

❖ DCOM的类型库和注册



第十二章 基于对象的分布式系统

12.3 DCOM

❖DCOM的服务

DCOM 服务和 CORBA 服务的对应关系

CORBA 服务	DCOM/COM+服务
收集	ActiveX 数据对象
查询	无
并发	线程并发
事务处理	COM+的自主(Automatic)事务处理
事件	COM+事件
通告	COM+事件
外观化	打包设施
生存周期	类工厂, JIT 激活
许可	特殊的类工厂
命名	名字(monikers)
性质	无
交易	无
持久性	结构化存储
关系	无
安全性	授权(Authorization)
时间	无

第十二章 基于对象的分布式系统

12.3 DCOM

❖DCOM的通信

➤对象调用模型

- 对象的同步调用模型是DCOM的默认的对象调用模型，当出现错误的时候，顾客得到一个错误代码。在这种情况下，DCOM本身试图重新调用该对象，即DCOM提供最多一次的调用语义。
- DCOM也支持异步调用，这种异步调用对应于CORBA的异步方法请求中的收集(polling)模式，其实它们构建支持异步调用的接口的方案也是相同的。对于一个接口的MIDL说明来说，MIDL编译器为这个接口中的每个方法m生成两个独立的方法：Begin_m和Finish_m。Begin_m只有输入参数，输入参数来自于方法m，而Finish_m只有输出参数。

第十二章 基于对象的分布式系统

12.3 DCOM

❖ DCOM的通信

➤ 事件

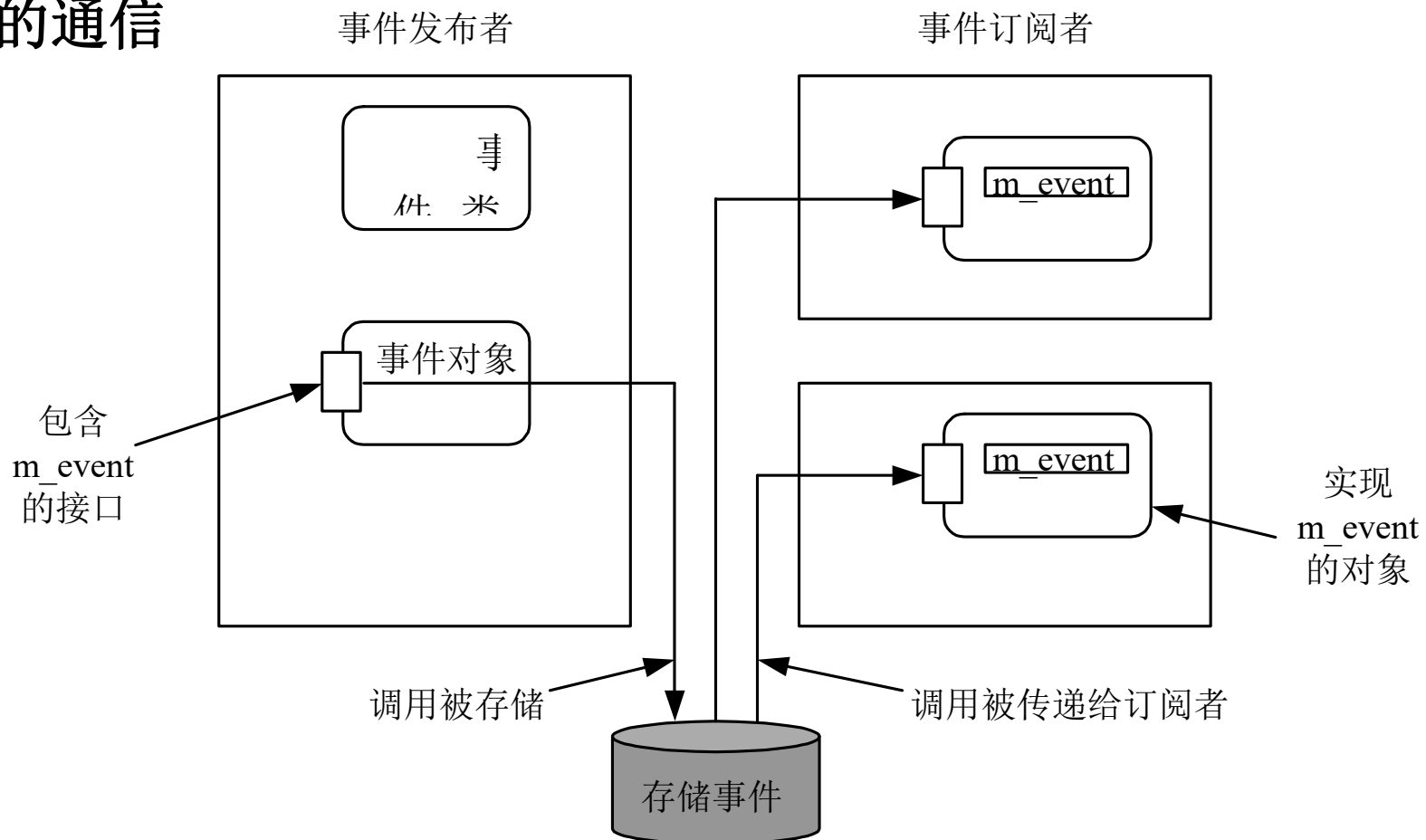
- DCOM通过COM+提供了一个更为成熟的事件模型，这种模型更适合于实现我们所熟知的发布/订阅 (publish/subscribe) 系统。其基本思想同CORBA的推送风格的事件模型非常类似。
- DCOM中的事件可以被存储起来。假设订阅者处于非活动状态，如果这时事件发生了，在这种情况下事件被存储起来，以后再被订阅者获取。

第十二章 基于对象的分布式系统

12.3 DCOM

❖ DCOM的通信

➤ 事件



第十二章 基于对象的分布式系统

12.3 DCOM

❖ DCOM的通信

➤ 报文

- 除了支持暂时性异步通信之外，DCOM还通过队列组件QC (Queued Components) 支持持久性的异步通信。QC实际上是一个与Microsoft报文队列MSMQ (Microsoft Message-Queuing) 的接口，QC的作用就是将MSMQ对顾客和对象隐匿起来，使得通过持久性异步通信进行的方法调用的处理方式和暂时性的异步通信相同。

- 每次顾客请求这样一个方法时，该请求自动被打包并存储在顾客方的QC子系统中，当顾客完成了方法请求之后，可以调用Release来释放这个接口。被打包和存储起来的方法请求会通过MSMQ传送。

第十二章 基于对象的分布式系统

12.3 DCOM

❖ DCOM的通信

➤ 报文

- 当打包的调用到达目的地之后，QC子系统对每个请求进行拆包然后调用对象。不保证请求的处理顺序和它们的请求顺序是一致的，除非这些调用请求是一个事务处理的一部分。
- MSMQ同大多数报文队列系统一样，提供事务处理队列。事务处理队列中的报文要么全部处理，要么一个也不会处理。另外当出现失效的时候，队列中的报文能够恢复。

第十二章 基于对象的分布式系统

12.3 DCOM

❖ DCOM的Moniker

- 基本的DCOM对象模型只支持暂时性对象，也就是说当一个对象不再被引用时会被销毁。为了对一个对象的生命周期进行扩展，使得其生命周期超过调用它的顾客，DCOM提供了一个moniker作为这个对象的持久性引用。这个持久性引用moniker被保存在磁盘上，这个moniker包含了重新构造这个对象所需的必要信息，例如这个对象的最后一个顾客退出时，这个对象的状态。
- DCOM提供了各种不同类型的moniker，其中一个重要的类型是文件moniker。对于文件moniker，它包含了一个本地的文件名字，这个文件用于构造对应的对象。另外它还包含一个类对象的标识符CLSID，这个类对象通过对应的类实际创建一个对象。
- 一个文件moniker提供了一个BindToObject操作，顾客通过调用这个操作与这个moniker对应的对象结合。

第十二章 基于对象的分布式系统

12.3 DCOM

❖ DCOM的Moniker

顾客通过文件 **moniker** 和持久性对象结合

步骤	参与者	描述
1	顾客	在一个 moniker 上调用 BindMoniker
2	Moniker	查找对应的 CLSID 并指示 SCM 创建一个对象
3	SCM	装入类对象
4	类对象	创建一个对象并返回指向一个 moniker 的接口指针
5	Moniker	指示对象装载以前保存的状态
6	对象	从一个文件装载它的状态
7	Moniker	返回一个指向对象的接口指针给顾客

第十二章 基于对象的分布式系统

12.4 Clouds系统

❖Clouds的对象

- Clouds的对象全部是持久性的，被放在一个虚拟地址空间中，不允许多重继承。
- 对象之间通过报文进行通信。报文通信要依次请求一个方法的执行，要求通信的进程提交请求，这个方法修改对象中的数据值并使其他对象接收报文。对象具有给发送者发送的报文进行应答的能力，通过返回应答，被请求的方法的执行才能完成。
- 报文是由一个线程的运行产生的，一个对象请求一个进入点让一个报文传送到对象内，进入点有自己的输入参数。参数严格地被限定为数据，而不是数据的地址。一个用户可以通过向Clouds的shell指定一个对象、一个进入点和一组参数的形式调用一个Clouds的对象。

第十二章 基于对象的分布式系统

12.4 Clouds系统

❖ Clouds的对象

- 一个Clouds对象由以下部分组成：

- (1) 用户定义的代码；
- (2) 永久性数据；
- (3) 一块被临时分配的内存；
- (4) 一块被长久分配的内存。

第十二章 基于对象的分布式系统

12.4 Clouds系统

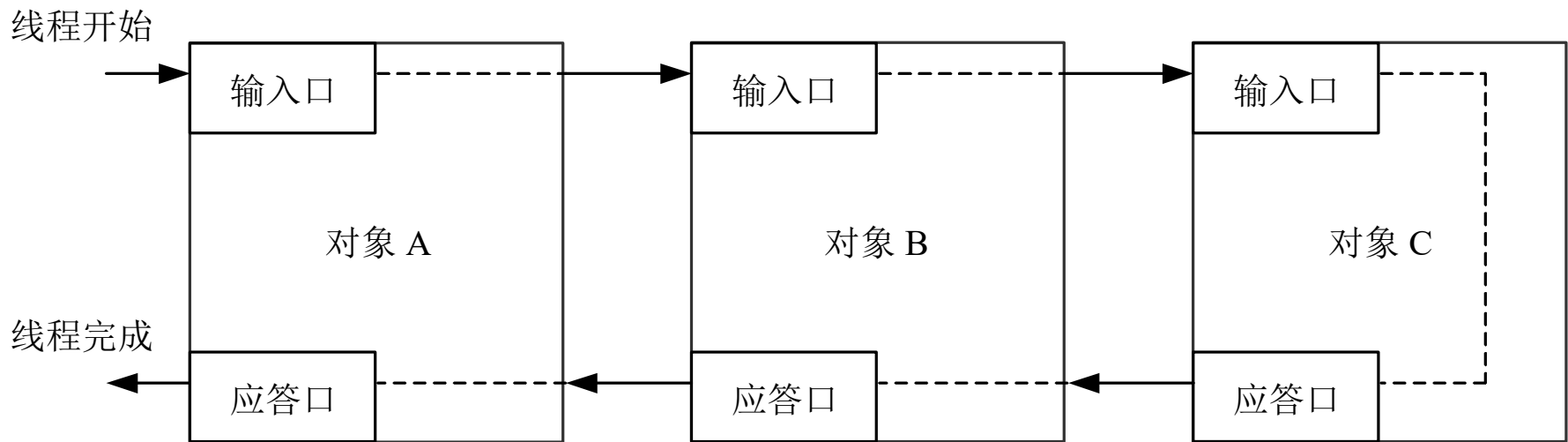
❖ Clouds的线程

线程的执行：线程在对象之间穿越并为一个对象执行一个进入点。当一个线程经过时，线程执行对象里的代码，所以，不同的线程代表了用户的所有活动，每个线程对应一个活动，因此代表了一个逻辑的路径。一个线程可以用如下两种方式之一产生：通过用户产生和通过程序产生。Clouds的线程不绑定在一个地址空间上，当一个对象的执行产生对另一个对象的请求时，线程可以临时地离开第一个对象进入到新被调用的对象，执行完成后，线程返回到第一个对象。对象请求可以是递归的或嵌套的。当整个操作执行完成时，线程终止。

第十二章 基于对象的分布式系统

12.4 Clouds系统

❖ Clouds的线程



第十二章 基于对象的分布式系统

12.4 Clouds系统

❖ Clouds的线程

线程在对象内并发执行：Clouds允许线程在对象内并发执行，也就是说，允许在一个对象内有多个线程同时存在。当多个线程同时在一个对象中同时存在的时候，它们共享这个对象的内容和地址空间。Clouds支持锁和信号灯，用于在一个对象内进行并发控制。所有的应用程序的对象必须被编写成支持并发执行。并发发生在执行的时候产生并发线程的情况下。

第十二章 基于对象的分布式系统

12.4 Clouds系统

❖ Clouds的存储器

在传统的系统中，永久性存储器用来保存文件而易失性存储器用来保存进程等易失性信息。Clouds使用对象将永久性存储器和内存统一成了一个完整的永久性的地址空间。Clouds在操作系统一级上不支持报文和文件，但如果需要，允许对它们进行模拟。Clouds共享数据的方法是把数据放在对象中，需要访问共享数据的计算调用有该数据的对象。