



第二章

分布式进程与通信

第二章、分布式进程与通信

01

2.1 基本概念

02

2.2 分布式进程和线程设计

03

2.3 远程过程调用

04

2.4 面向消息的通信

05

2.5 *多播通信



2.1 基本概念

- **2.1.0 硬件知识**

- 冯诺依曼体系结构
- 程序执行过程

- **2.1.1 进程与线程**

- 进程模型
- 线程模型

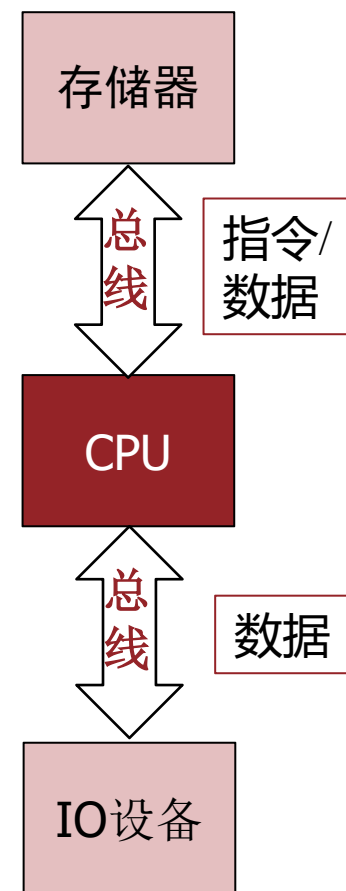
- **2.1.2 进程间通信**

- 通信协议
- 通信类型

2.1.0 硬件知识

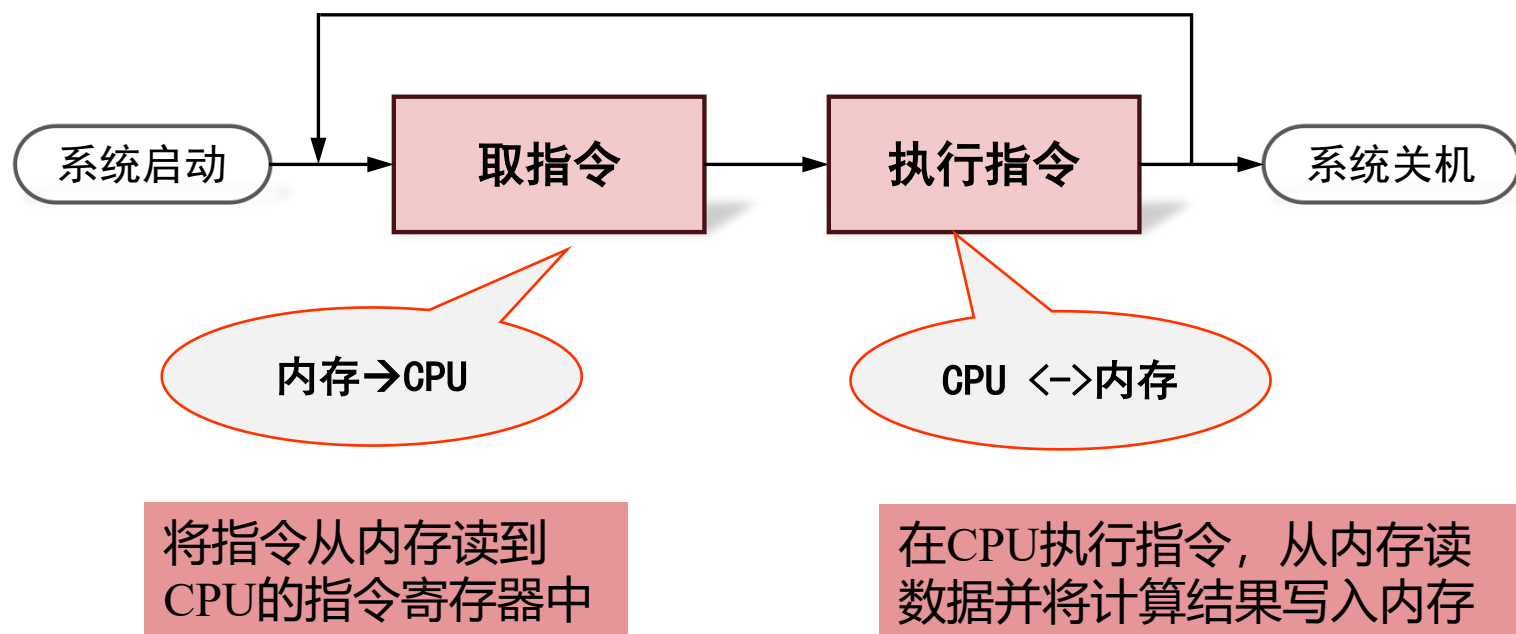
• 冯·诺依曼体系结构

- **程序**由一系列的“**指令**”组成; “**指令**”在**CPU**执行。
- **指令**由操作码和地址组成 $\langle \text{Op}, \text{Addr1}, \dots \rangle$
- **存储程序方式**, 指令和数据不加区别混合存储在同一个存储器中, 指令和数据通过一条共同的**总线**, 从存储器载入**CPU**
- **存储器**是按地址访问的**线性编址**的一维结构, 每个单元的位数是固定的
- **以运算器为中心**, I/O设备与存储器间的数据传送都要经过运算器



指令的执行过程

- 指令周期（执行一条指令花费的时间），分为取指令和执行指令两个阶段
- 程序指针（指令流的控制），顺序执行下一条指令，或判断分支跳转



程序vs指令

- C语言程序
- MIPS指令程序(如龙芯)

1	
2	void main() {
3	int b;
4	int i = 0, j = 0;
5	while (i < 100) {
6	if (b)
7	j++;
8	else
9	j--;
10	i++;
11	}
12	}

Entering
Loop

退出循环

```
main():
simple.c:2
004001f0 <main>      addiu   $29,$29,-24
004001f8 <main+0x8>    sw      $30,16($29)
00400200 <main+0x10> addu     $30,$0,$29
simple.c:4
00400208 <main+0x18> sw      $0,4($30)
00400210 <main+0x20> sw      $0,8($30)
simple.c:5
00400218 <main+0x28> lw      $2,4($30)
00400220 <main+0x30> slti    $3,$2,100
00400228 <main+0x38> bne     $3,$0,00400238
00400230 <main+0x40> j       004002b8
simple.c:6
00400238 <main+0x48> lw      $2,0($30)
00400240 <main+0x50> beq     $2,$0,00400270
simple.c:7
00400248 <main+0x58> lw      $3,8($30)
00400250 <main+0x60> addiu   $2,$3,1
00400258 <main+0x68> addu    $3,$0,$2
00400260 <main+0x70> sw      $3,8($30)
00400268 <main+0x78> j       00400290
simple.c:9
00400270 <main+0x80> lw      $3,8($30)
00400278 <main+0x88> addiu   $2,$3,-1
00400280 <main+0x90> addu    $3,$0,$2
00400288 <main+0x98> sw      $3,8($30)
simple.c:10
00400290 <main+0xa0> lw      $3,4($30)
00400298 <main+0xa8> addiu   $2,$3,1
004002a0 <main+0xb0> addu    $3,$0,$2
004002a8 <main+0xb8> sw      $3,4($30)
simple.c:11
004002b0 <main+0xc0> j       00400218
simple.c:12
004002b8 <main+0xc8> addu    $29,$0,$30
004002c0 <main+0xd0> lw      $30,16($29)
004002c8 <main+0xd8> addiu   $29,$29,24
004002d0 <main+0xe0> jr      $31
004002d8 <main+0xe8> nop
end_addr
```

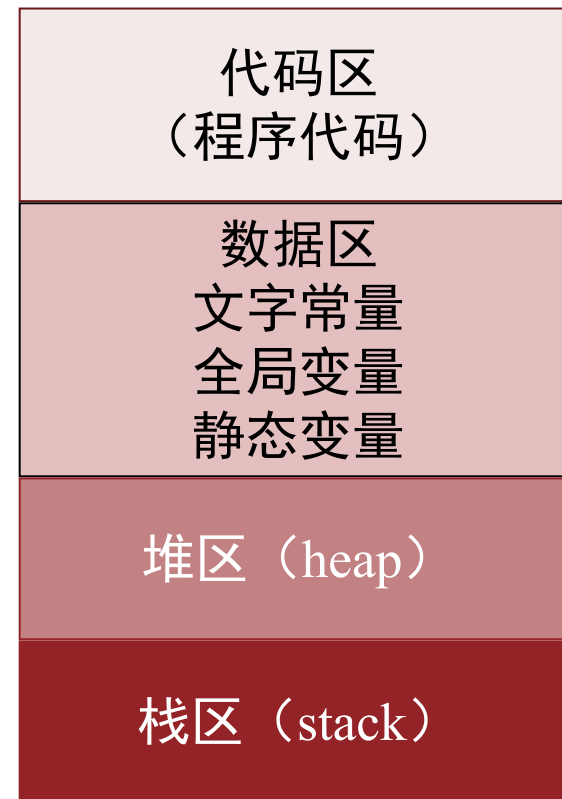
if块

else块

下一轮迭代

举例：C程序内存管理

- 代码区
 - 存放C函数编译后的代码
- 数据区
 - 字符串常量
 - 全局变量
 - 静态变量
- 堆区
 - 动态开辟的存储空间
- 栈区
 - 函数的参数、返回值、局部变量



举例：C程序内存管理

```
int a = 0;
char *p1;

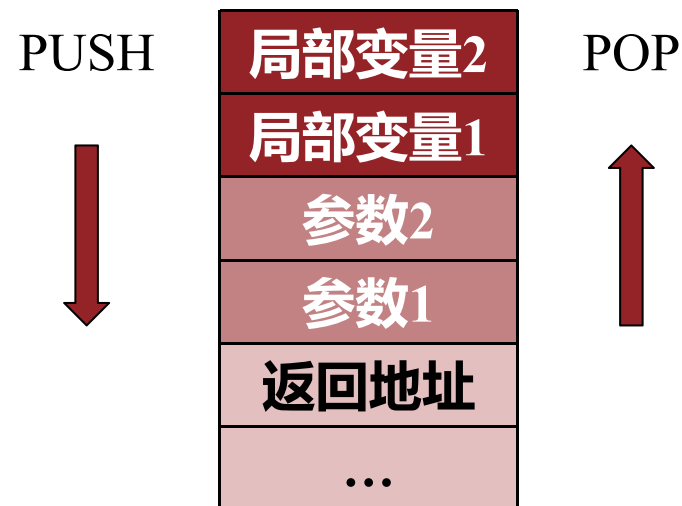
main()
{
    int a,b; //全局变量
    char s[] = "abc";
    char *p2;
    char *p3 = "123456";
    static int c =0; //静态全局变量
    p1 = (char *)malloc(10); //动态全局变量
    strcpy(p1, "123456");
    Foo(a,b);
}

Foo(int p1, int p2){
    int x, y; //局部变量
    static int z; //静态局部变量
}
```



举例：C函数调用过程

- 函数调用
 - 返回地址最先入栈
 - 各个参数入栈
 - 函数的局部变量入栈
 - 函数内部定义的静态变量不入栈
- 函数返回
 - 以上数据逆序从栈中弹出
 - 跳转至返回地址

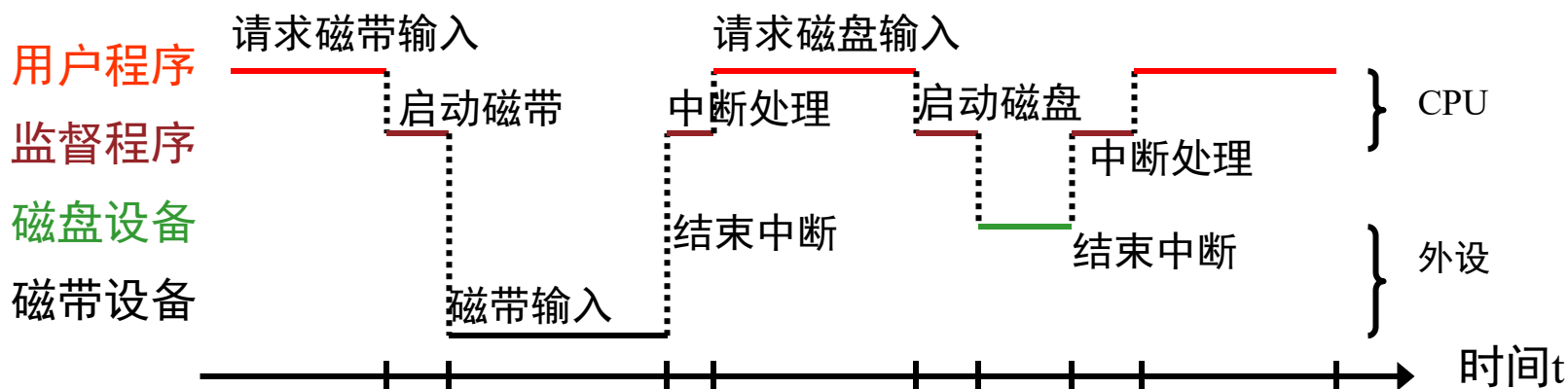


举例：C程序内存管理

- 栈 (Stack)
 - 大小固定，C程序中压栈和弹栈操作由编译器负责
 - 栈空间一旦用完，系统将报错
 - 操作按LIFO方式，访问栈指针指向的数据
- 堆 (Heap)
 - 有一个初始大小，超出后，通过系统调用申请更多
 - 地址空间可能不连续
 - C程序需要程序员显式管理堆空间，极易出现“内存泄露”
 - 操作无次序，可访问任何地址的数据

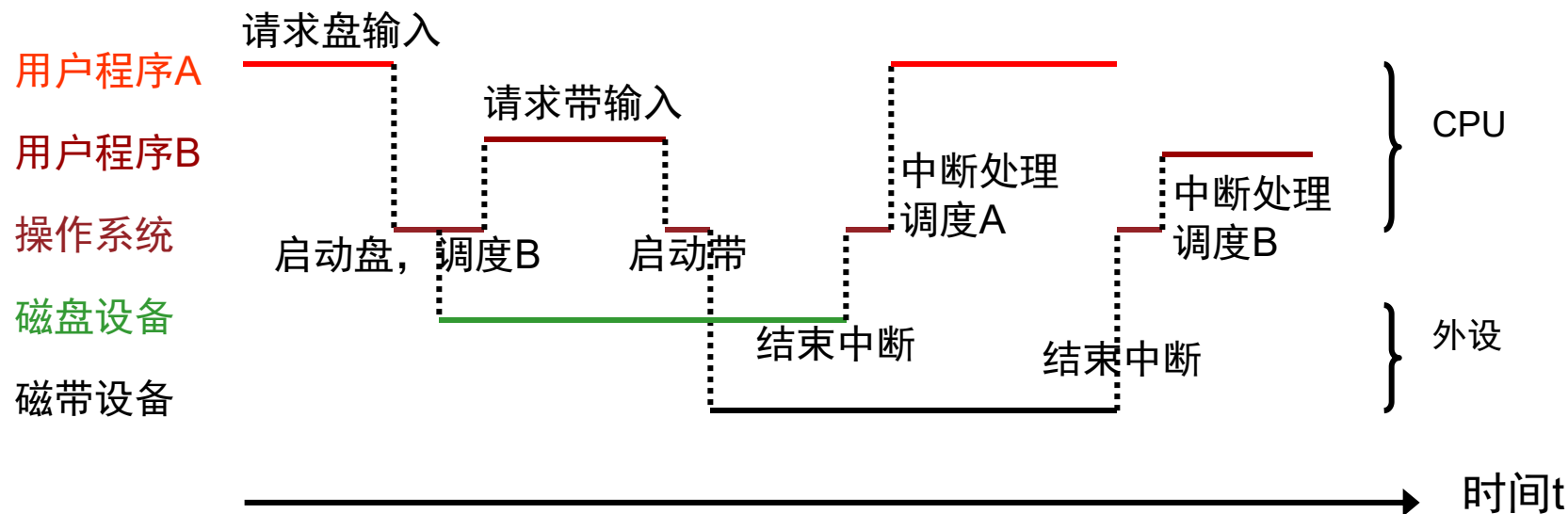
程序的顺序执行

- 顺序执行的问题：系统角度（单处理器）
 - 程序的执行，不仅仅需要CPU，还需要各种外设
 - 资源使用率低下，尤其CPU算力资源



程序的并发执行

- 不同于串行执行方式，在程序A的执行结束之前，程序B就开始执行
- 一组在**逻辑上相互独立**的程序在**执行时间上互相重叠**



2.1.1 进程模型

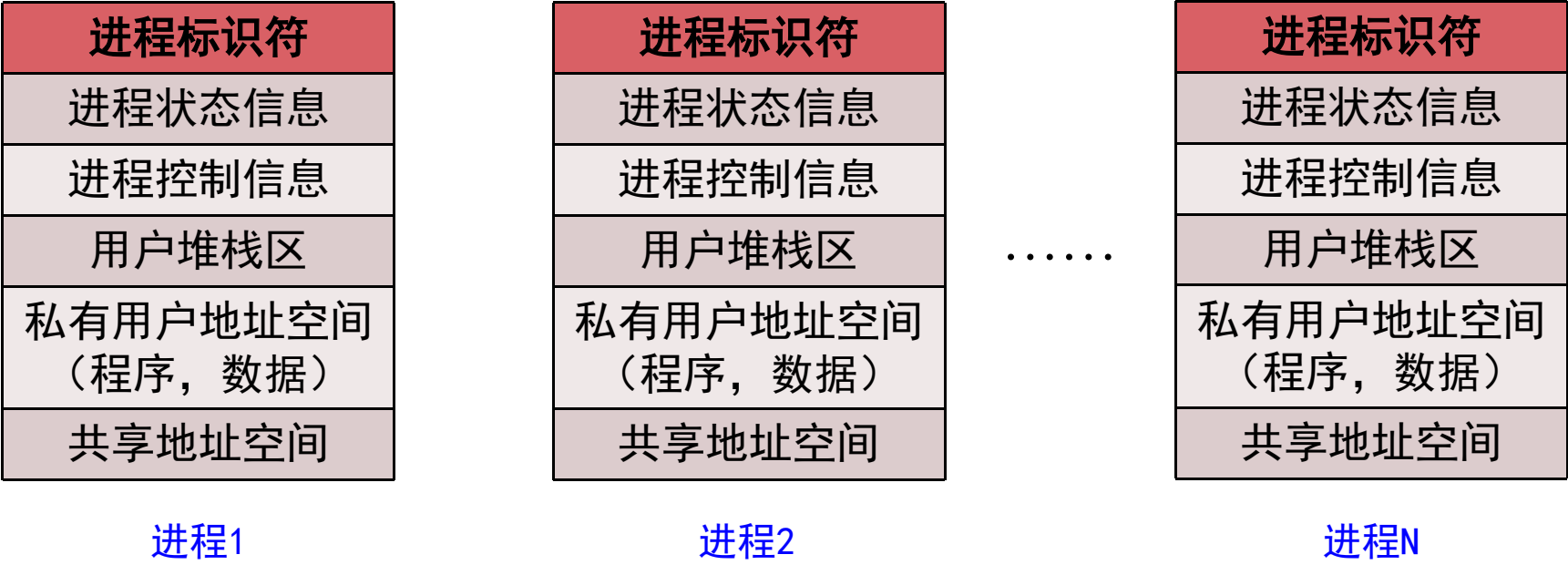
进程的定义

- 通常是指一个具有一定独立功能的程序在一个数据集上的一次动态执行过程
- 简言之，一个进程是程序的一次执行活动，或一个正在运行程序的实体、一个执行程序的载体

- 进程描述了程序的动态执行过程；
- 它对应处理机、存储器和外设等资源的分配和回收；
- 特别的，是分配处理机资源的基本单位（如不考虑线程）；
- 反映系统中程序执行的并发性、随机性和资源共享；
- 系统引入多进程，提高了对硬件资源的利用率，但又带来额外的空间和时间开销，增加了操作系统的复杂性

进程在操作系统中的描述

- 进程上下文：用于执行指令序列的数据集，主要有：存储在寄存器和内存中的数据值
- 进程控制块(PCB)：用于保存进程上下文

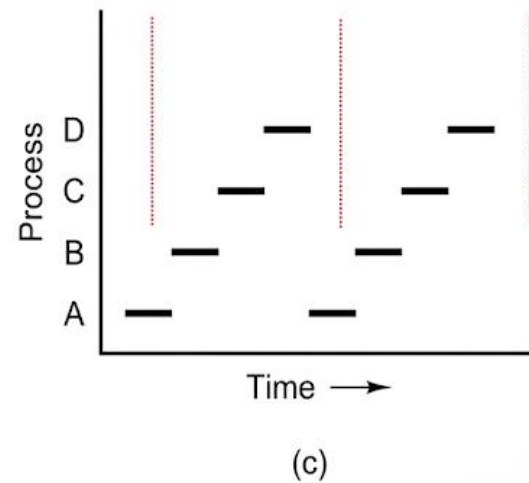
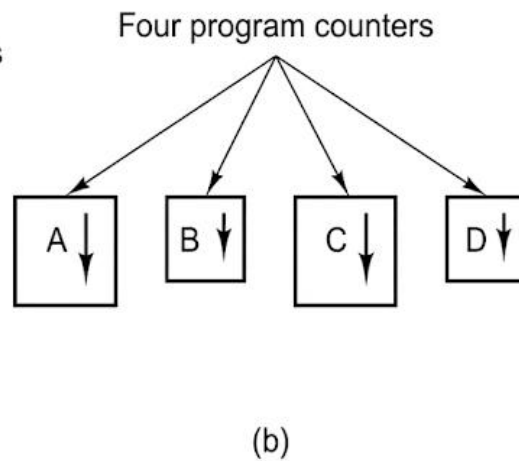
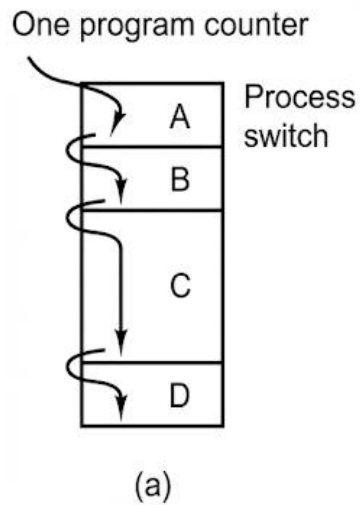


进程的本质

- 本质上实现了 “**并发透明性**” 与 “**虚拟处理器**”
 - 硬件处理器：物理存在，提供一套指令集，能够自动执行指令序列。
 - 软件处理器(Software Processor)：虚拟存在，能够执行指令序列。分为**进程级**和**线程级**
- 从进程的角度，感知不到其他进程的存在
 - 进程之间是隔离的
- 从系统的角度，进程之间相互独立，互不干扰
 - 每个进程认为是独占CPU和内存

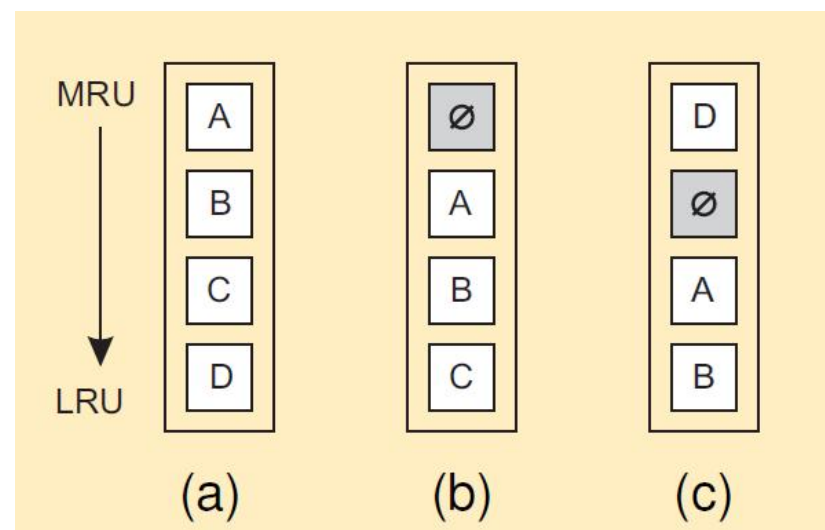
进程和线程管理

- 进程：系统中的运行程序的实体
- 例：4个程序组成的多道程序执行
 - 逻辑上，4个独立的、顺序执行的进程，概念模型(a,b)
 - 物理上，任意时刻只有一个是活动的(c)



进程切换开销

- 直接开销
 - 主要是进程上下文(Context)切换
- 间接开销，可能占80%
 - 主要是缓存扰乱造成的代价
 - 举例：内存Cache
 - (a) 上下文切换之前
 - (b) 切换之后，页面D被替换出内存
 - (c) 访问页面D之后。
需要从下一级存储（磁盘）读D
高于(a)的访问代价

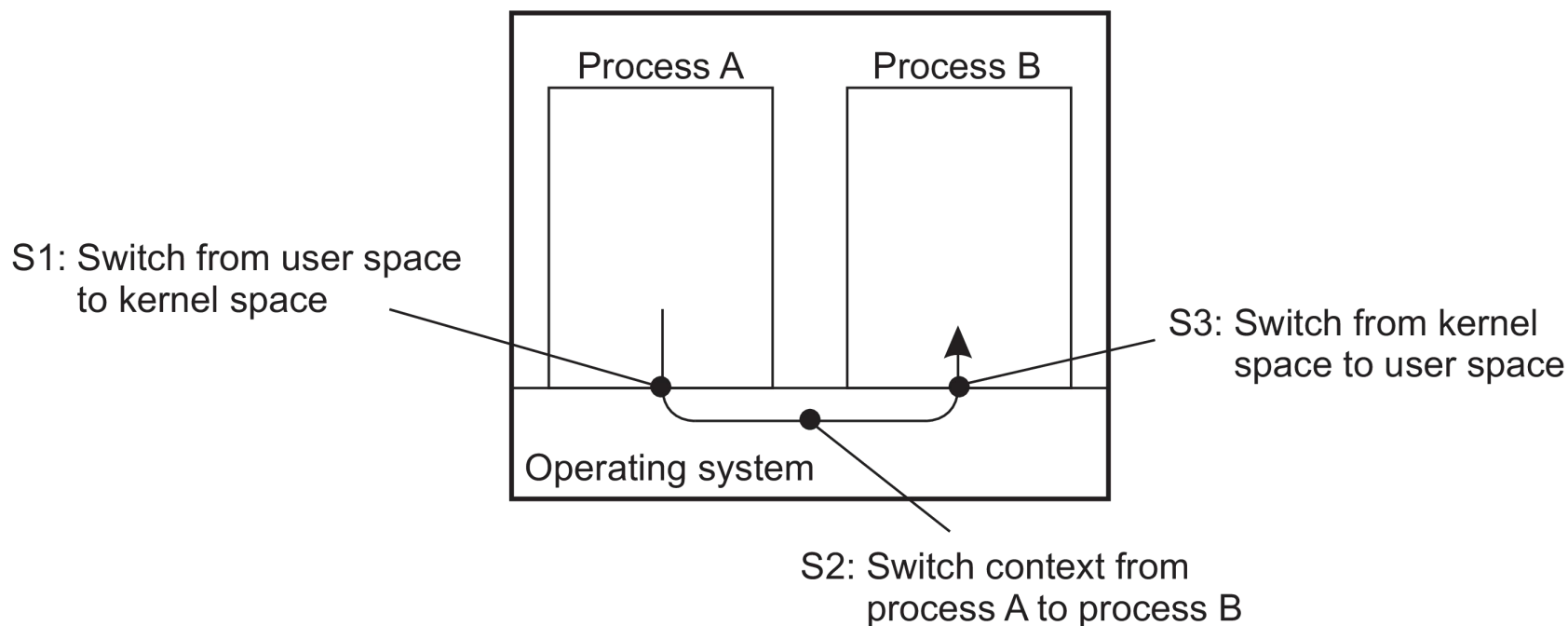


常用缓存替换策略：LRU：（最近最少使用）； MRU（最近最多使用）

多进程模型的问题

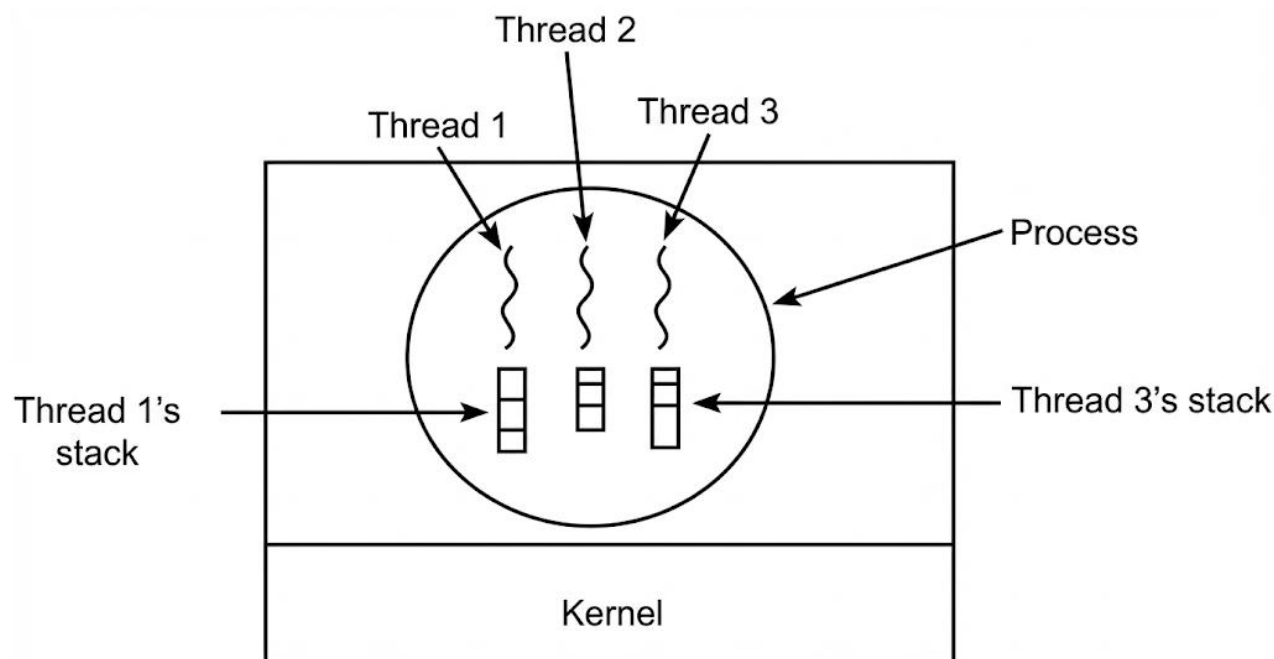
- 进程切换开销大
- 进程间通信开销大

这是为实现透明性
所付出的代价!



2.1.1 线程模型

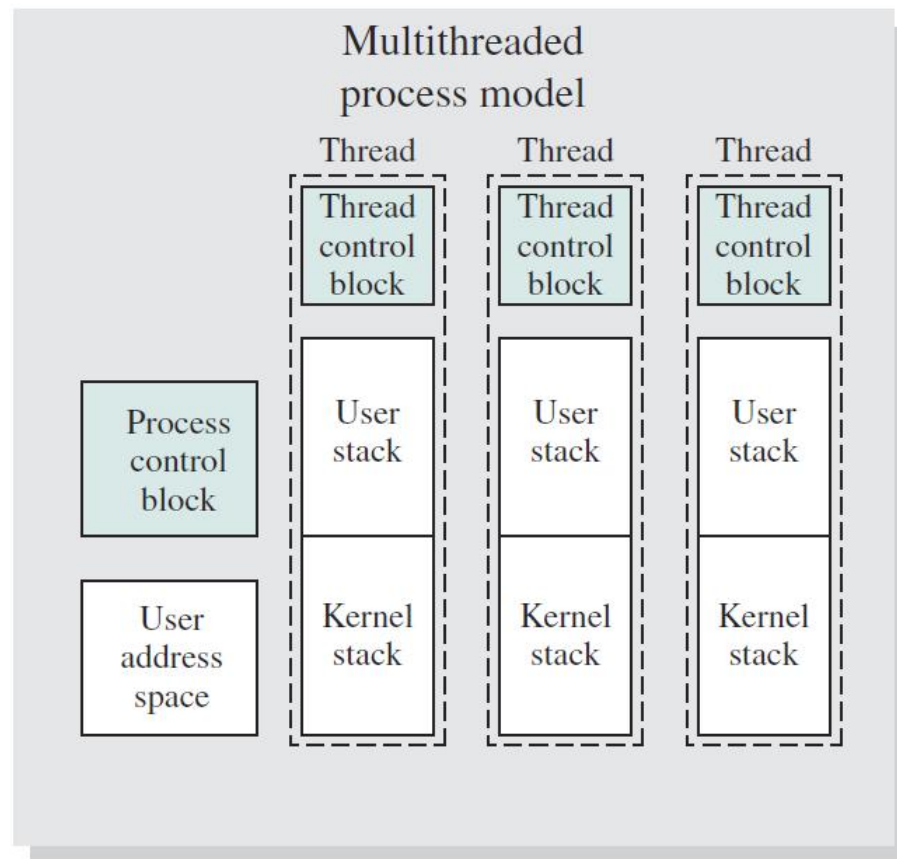
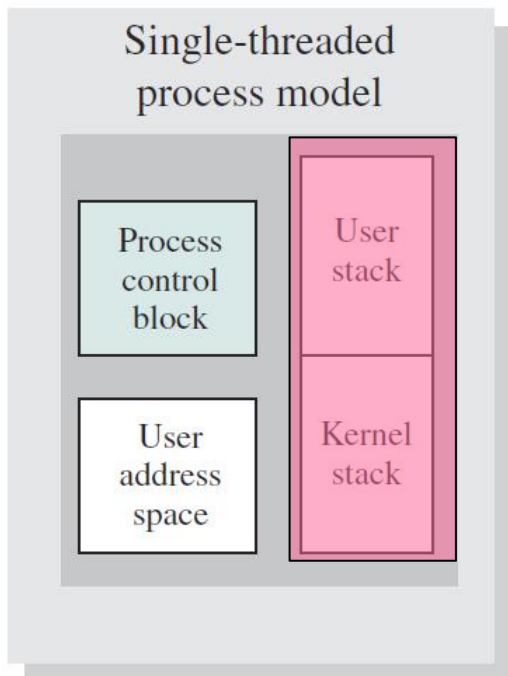
- 多线程是指操作系统的这样一种设计
 - 它允许在一个进程内部，有多个执行路线“同时”进行
 - 每个线程拥有自己的堆栈、局部变量、返回地址



线程与进程的区别

- 进程：相关的运行资源的管理单位
 - 内存空间
 - 打开的文件
 - 子进程、信号处理、记账信息等
 - 资源分组,独立实体
- 线程(线索)：在CPU上执行的脉络
 - 程序计数器、寄存器、栈
 - 执行,依赖于进程
 - 每个线程可共享所属进程的资源

线程与进程的区别



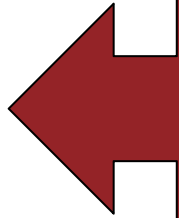
线程的优点

- 线程上下文

- 用于执行指令序列的最小数据集，主要是保存在寄存器和内存中的数据和状态值。

- 主要是性能考虑

- 创建和撤销一个线程的开销，比进程小很多
- 线程上下文线程切换，独立于操作系统，因而切换开销小
- 同一进程内部的线程共享同一地址空间，线程间通信开销小

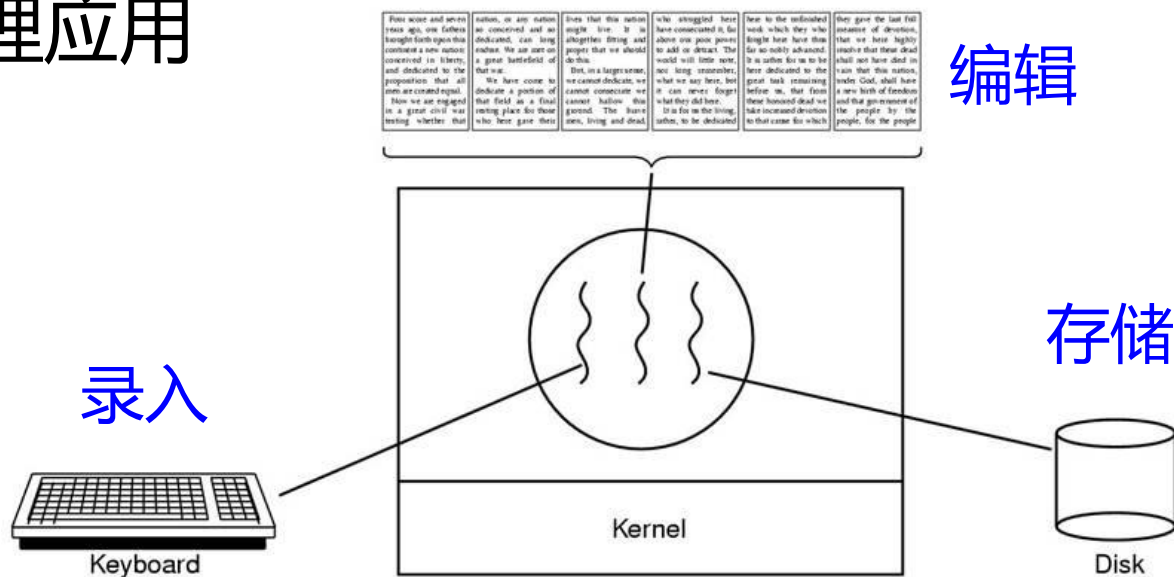


线程是在
性能与透
明性之间
的一种折
中

线程的优点

- 提高响应性、协作、异步操作
 - 避免无用**阻塞**。多线程进程当一个进程IO等待时，可执行其他线程
 - 利用**并行性**。在多机和多核系统上，多线程并行
 - 避免进程**切换**。大的应用不需要用多个进程实现

• 例1：文档处理应用



进程与线程

■ 进程模型比较

模型	特点
单线程进程	无并行性、 阻塞式系统调用，实现简单
多线程进程	并行性、 阻塞式系统调用，实现较简单
有限状态自动机	并行性、 非阻塞式调用，实现复杂

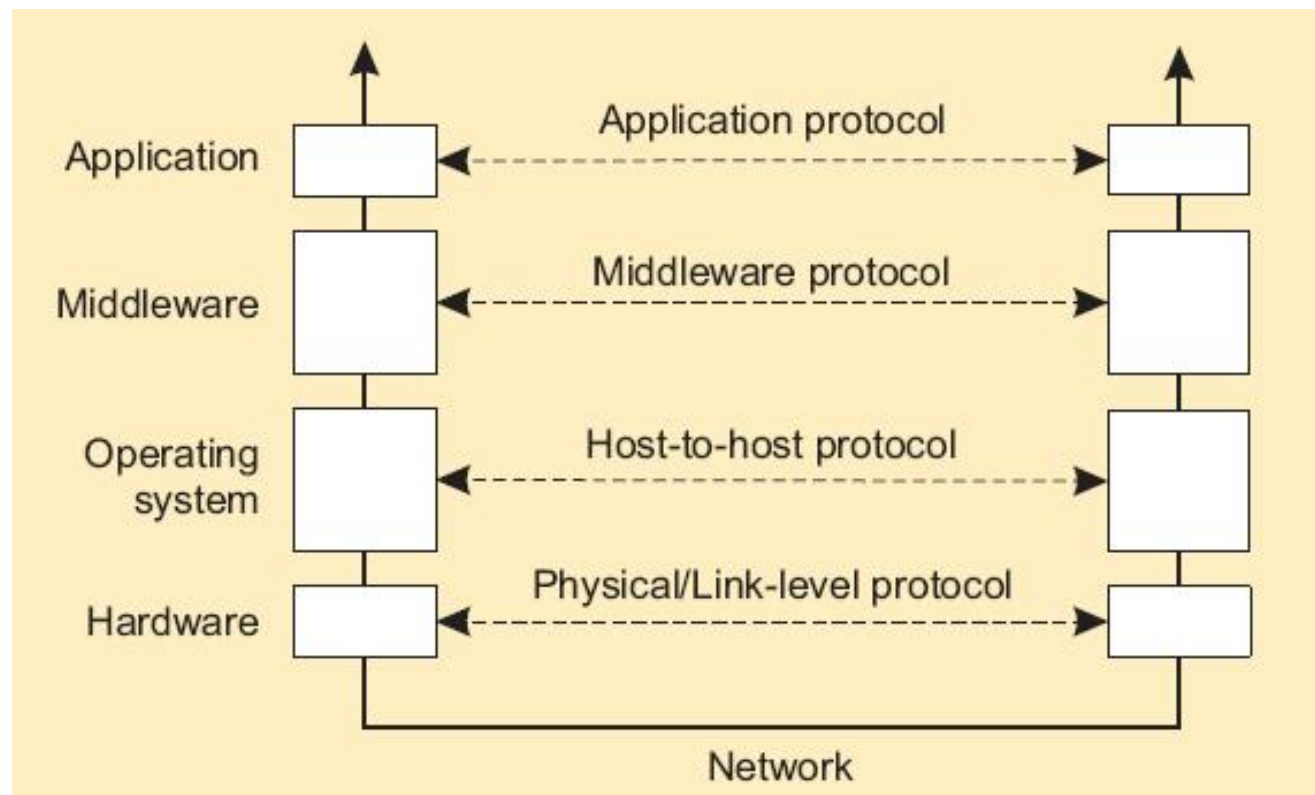
2.1.2 进程之间的通信

- 网络通信基本概念
- 分布式环境下，**消息**是信息传递的重要手段



- **协议**：消息的格式、内容和意义
 - 协议好比语言，大家只有说同样的语言才能相互交流
- **开放式系统**：可按照协议进行通信

适合的简化分层协议



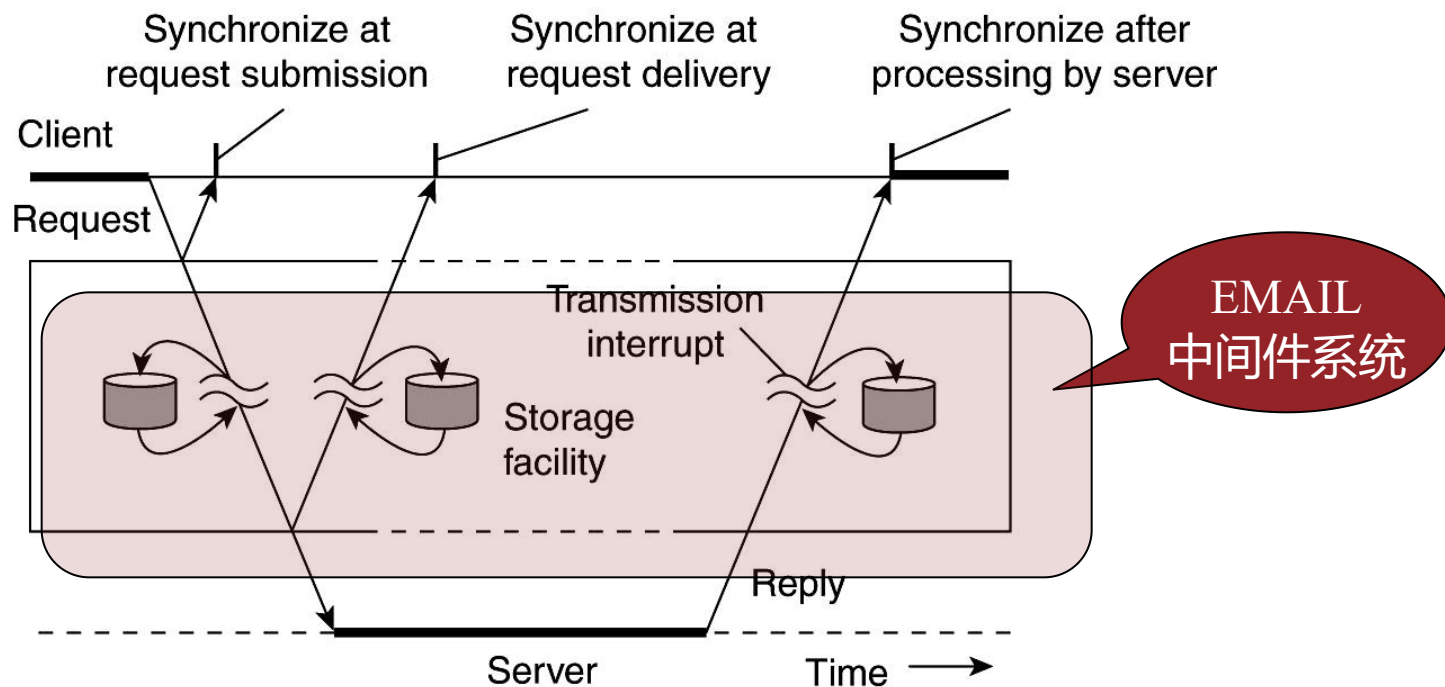
- **中间件**：提供中间件协议：会话层、展示层协议
- **操作系统**：实现主机-主机协议，网络层、传输层协议
- **硬件层**：实现物理层、链路层协议

中间件协议

- 在应用之间的通信，需要设计特定的通信协议
- 共性服务需求
 - 数据的编排/解编(marshaling/unmarshaling): 用于集成系统之间的数据传递
 - 命名协议: 用户对资源的共享
 - 安全协议, 身份认证协议: 保障安全的通信
 - 可伸缩性机制: 如进行复制、缓存管理
 - *面向事务的分布式提交协议: 原子性保证
 - *共享资源的封锁协议: 一致性保证
- 将这些共性需求设计为协议, 被各种应用复用
→ 中间件协议

举例：通信服务中间件协议

- 电子邮件系统
 - 三个可能的同步点



- 需要考虑通信是否可持久、是否要同步

通信类型

- 存储方式

- 持久性通信 (Persistent)

- 发送方提交的消息，由通信中间件持久存储直到对方接收

- 瞬时性通信 (Transient)

- 传输中断或接收方不活动，则丢弃消息。通信中间件没有存储机制

- 交互方式

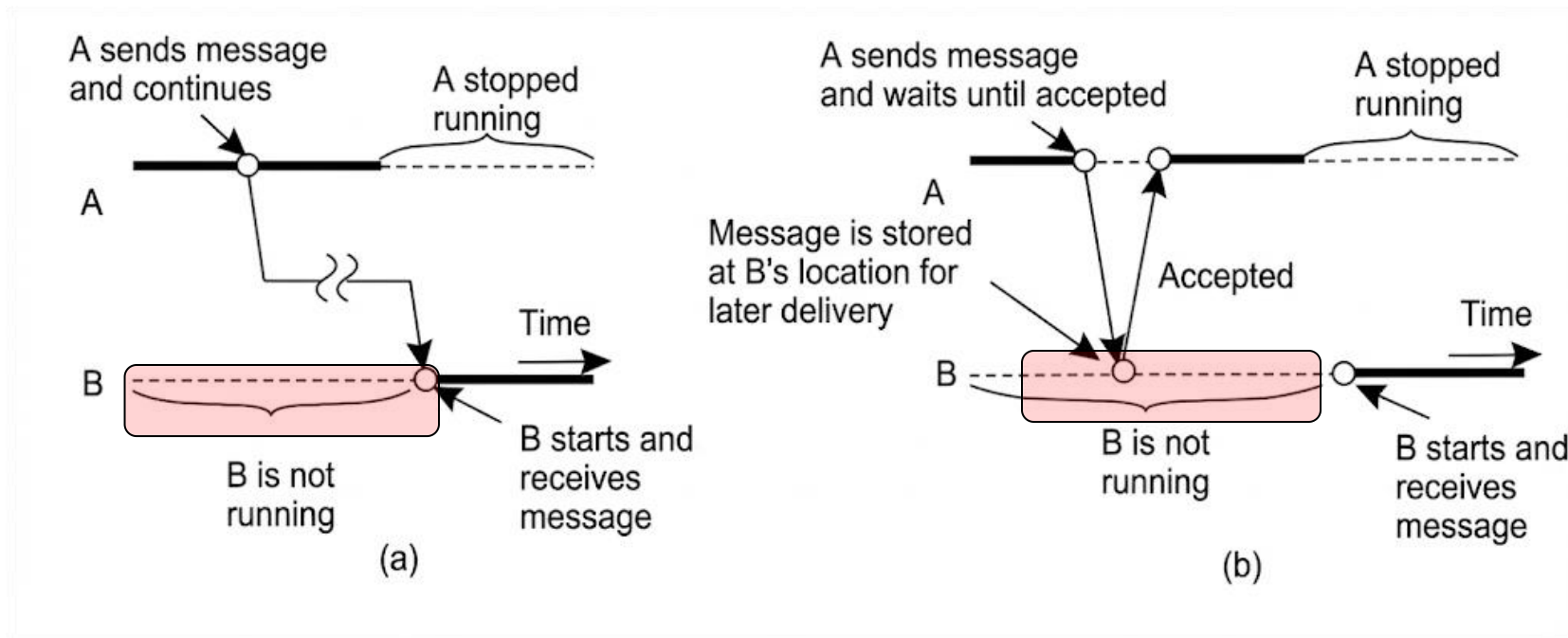
- 异步通信 (Asynchronous)

- 发送方提交消息后，无需等待，继续执行

- 同步通信 (Synchronous)

- 发送方阻塞，直至得到接收方通知后方可继续执行

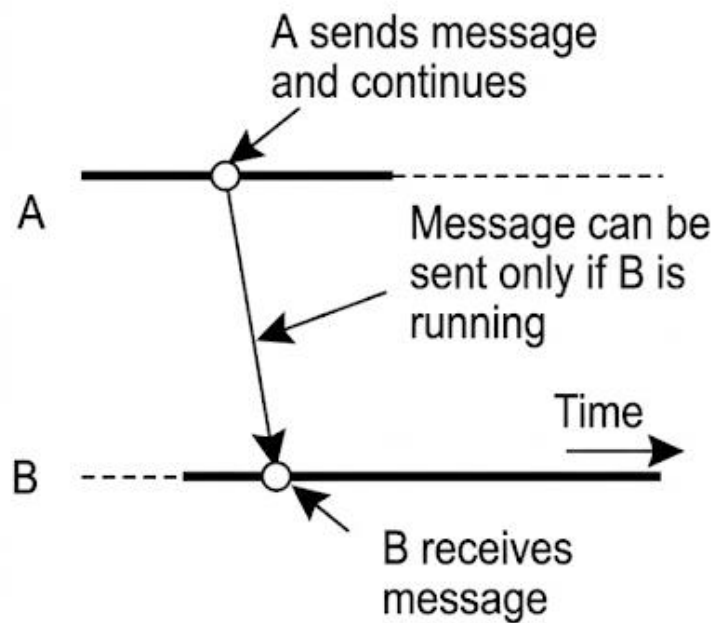
通信类型



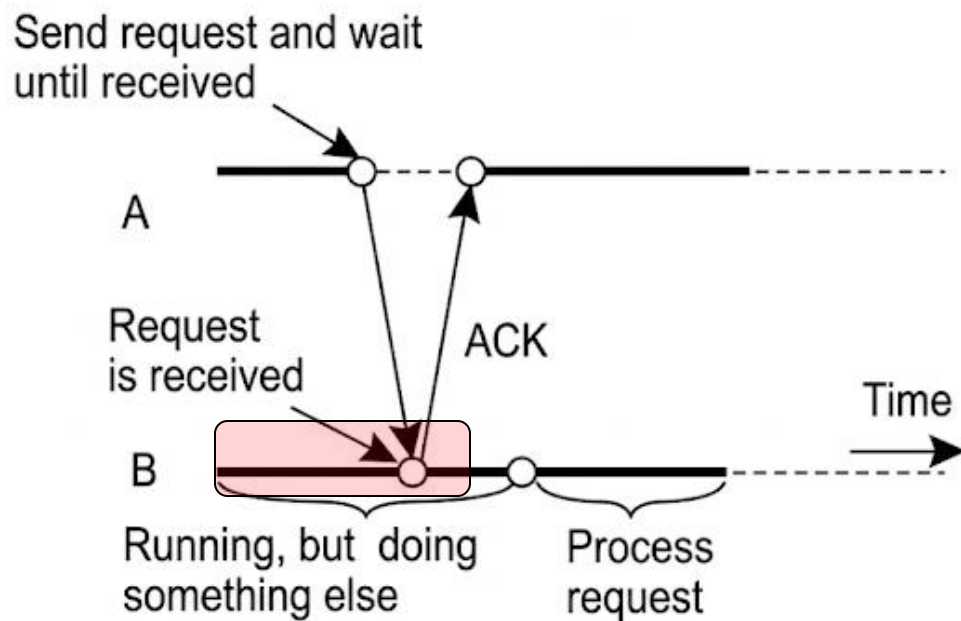
持久异步通信

持久同步通信

通信类型

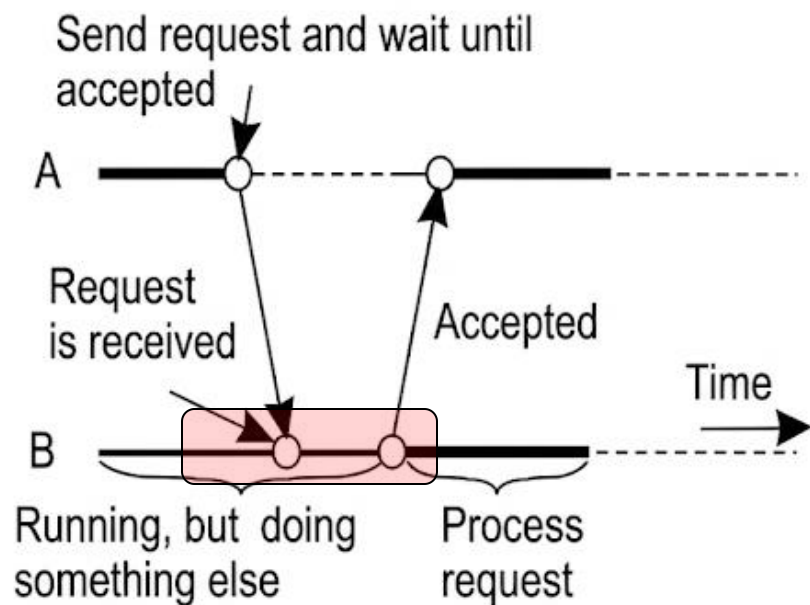


瞬时异步通信

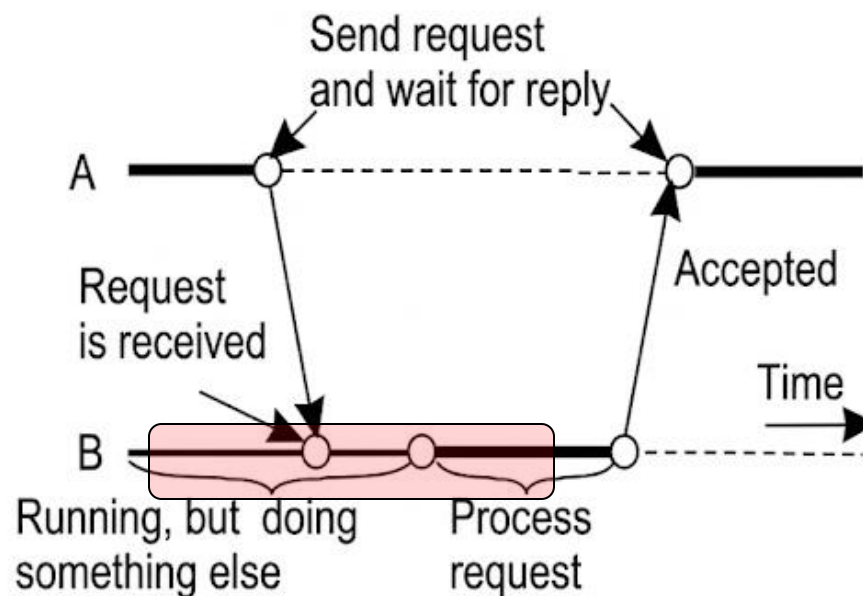


基于接收的瞬时同步通信

通信类型



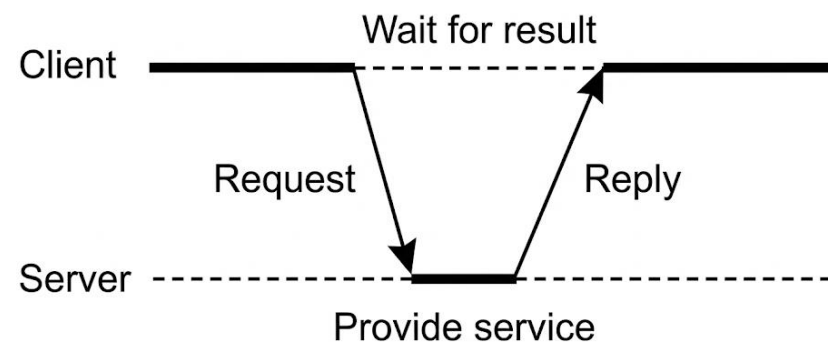
基于交付的瞬时
同步通信



基于响应的瞬时
同步通信

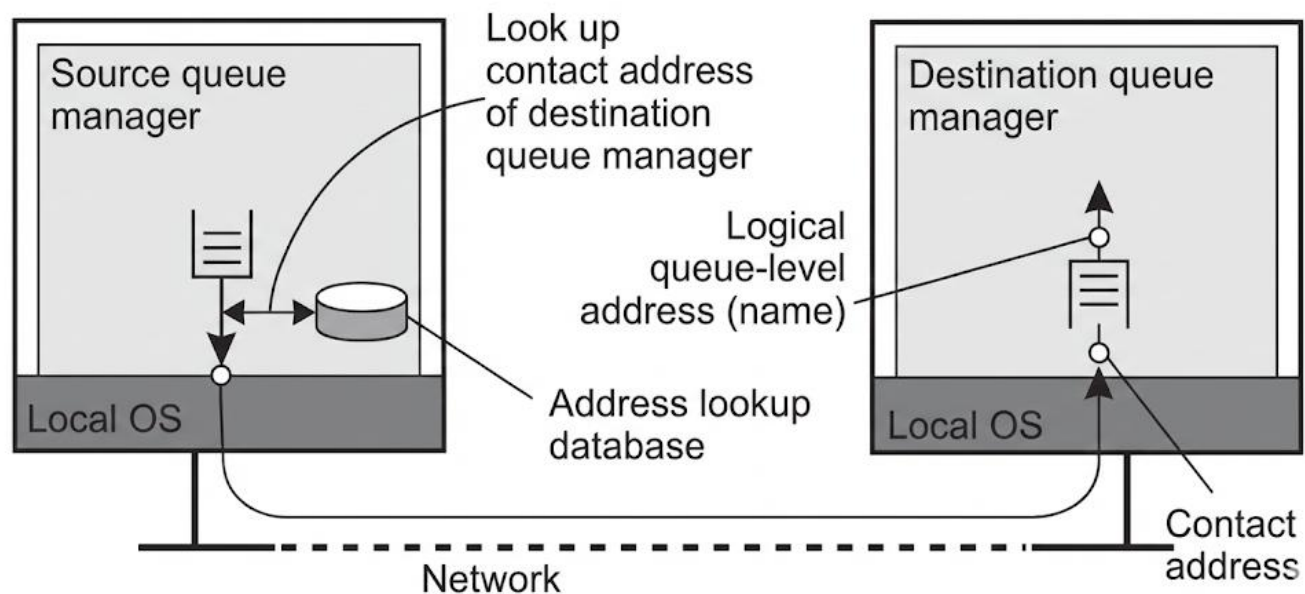
举例：客户机/服务器中的通信

- 客户/服务器计算基于**瞬时同步通信**
 - 客户和服务端必须同时在线
 - 客户发出请求，阻塞执行，直到收到应答
 - 服务器等待请求，然后依次逐个处理
- 同步通信的缺点
 - 客户在等待应答期间，处于空闲状态
 - 如果发生故障，必须立即处理，由于客户在等待
 - 模型对一些应用完全不适合（如邮件、在线新闻）



举例：消息队列系统

- 面向消息的中间件，提供高级的持久性异步通信
 - 进程之间相互发送的消息，保存在持久化队列中
 - 发送方无需等待响应，可以接续其他操作
 - 中间件通常提供容错功能



第二章、分布式进程与通信

01

2.1 基本概念

02

2.2 分布式进程和线程设计

03

2.3 远程过程调用

04

2.4 面向消息的通信

05

2.5 *多播通信



2.2 分布式进程和线程设计

- 分布式任务与进程
 - 客户机/服务器(C/S)模型
 - 客户机进程
 - 服务器进程
- 客户机进程的设计
 - 多线程客户机
 - 对透明性的支持
- 服务器进程的设计
 - 基本服务器设计
 - 多线程服务器
 - 可移动服务器

2.2.1 分布式任务与进程

- **分布式任务的执行过程**
- **分布式任务的分解**
 - 一个复杂的分布式任务分解为多个相互独立执行的子任务
- **分布式处理**
 - 将子任务分配给不同的计算机或节点进行处理
 - 资源管理系统：负责协调、监控和分派分布式任务的进程

2.2.1 分布式任务与进程

• 分布式进程的执行

- 每个子任务由一个或多个进程来执行，这些进程可以并行执行
- 利用适当的通信协议和机制，交换和共享数据。
- 利用同步机制，安全地访问和操作共享资源，使各个进程进行协同工作
- 利用容错机制，应对可能出现的故障和异常情况，以确保任务能够完成
- 当各个子任务完成时，它们的结果需要被合并，最终完成整个任务

多线程的好处

- 线程级并行性:TLP

- 设 i 个线程同时执行, c_i 为这 i 个线程执行的时间占比,
 N 为最多可同时执行的线程个数, 则有:

$$TLP = \frac{\sum_{i=1}^N i \cdot c_i}{1 - c_0}$$

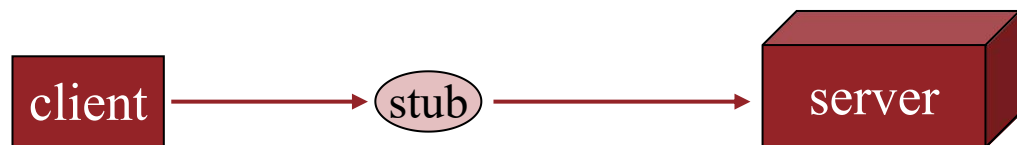
- WEB浏览器的TLP值区间为[1.5,2.5]
 - 为了有效利用线程平行性, 客户机CPU至少有2-3个核
 - 但更多的线程个数, 并不能提高性能。

2.2.2 客户机进程的设计

- 多线程客户机 (Client)
- 例1：HTML网页加载
 - 对一个网页，Web浏览器需要同时读取多个文件
 - 采用多线程，每个线程负责读取一个文件
 - 通过“边加载，边显示”的策略隐藏通信延迟
- 例2：多请求-响应调用 (RPC)
 - 客户机同时进行多个调用，每个调用使用一个线程
 - 如果一个服务器对应一个线程，则实现线性加速

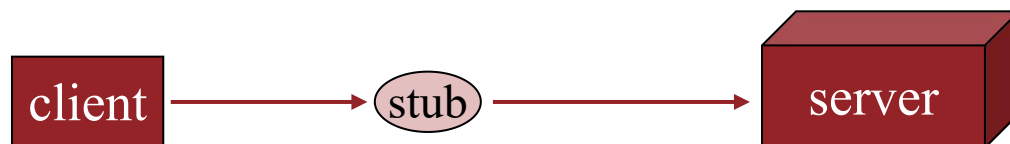
客户机对分布透明性的支持

- 客户机实现分布式透明性的主要问题
 - 如何隐藏广域网环境下的通信延迟
- 访问透明性
 - 客户机存根程序(stub): 实现远程过程调用 (RPC, 将在本章第3节介绍) 的代理程序
 - 隐藏客户和服务主机之间的硬件差异和通信



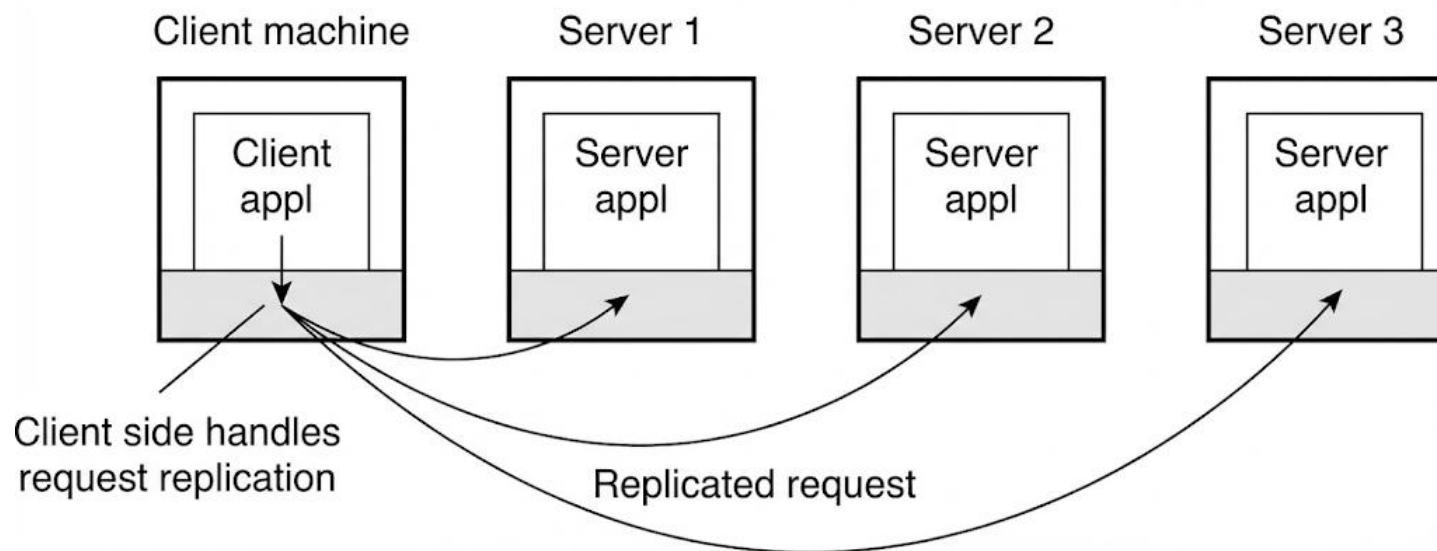
客户机对分布透明性的支持

- 位置透明性
 1. 由客户机程序跟踪服务器的实际位置
 2. 命名系统：全局逻辑名字
- 迁移透明性、重定位透明性
 - 重新绑定机制：当服务器端改变位置后，通知客户机重新自动绑定



客户机对分布透明性的支持

- 复制透明性
 - 由客户机的stub处理多重调用



- 失效透明性
 - 重新连接：通信失败时，重试
 - 本地数据缓存：例，脱机方式

2.2.3 服务器进程的设计

- 服务器：
 - 为一组客户提供专门服务的进程
- 服务器的两种组织方式
 - 迭代（Iterative）型服务器：自己处理请求
 - 并发（concurrent）型服务器：交给进（线）程处理
- 服务器端点（endpoint）的设置
 - 专用端口。例：
 - 运行时分配端点

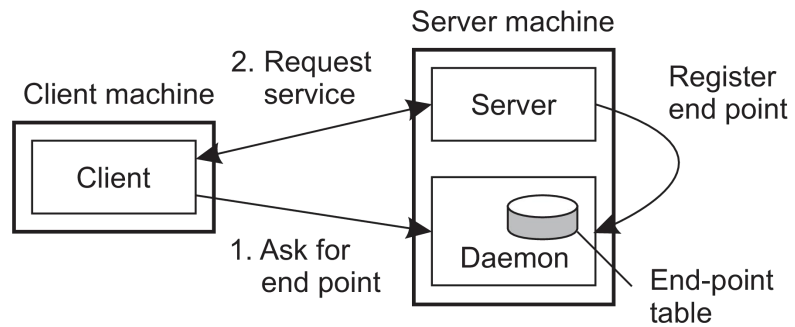
ftp	21	传输文件
telnet	23	远程终端
smtp	25	邮件传输
www	80	Web服务器

基本服务器设计

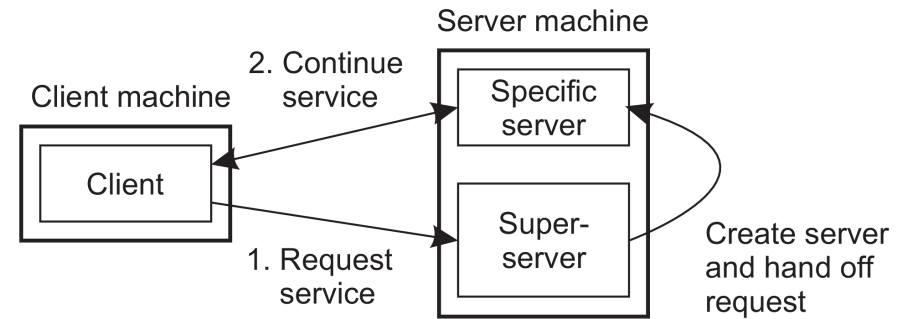
□ Client-to-Server两种绑定策略：

(a)静态: 由daemon 进程负责 (如, DCE)

(b)动态: 由superserver进程负责 (如, Unix *inetd*)



(a)静态策略



(b)动态策略

基本服务器设计

- client中断服务器端执行的策略：
 - 客户重启：中断连接
 - 客户发送out-of-band命令
 - ❖ 发送方法：
 - ✓ 利用可发送紧急消息的**另外端口**。服务器端设有另外的线程/进程处理紧急消息。OS支持优先级调度。
 - ✓ 利用**传输层的设施**，如，TCP允许在同一链路中发送紧急消息。OS signaling处理可捕获这些消息

基本服务器设计

- Client-to-Server两种连接方式
 - ✓ **无状态服务器**：服务器端不保存客户机程序的状态
 - 例如，Web服务器：不记录文件是否已被打开，不承诺清除客户缓存，不跟踪客户
 - 客户机和服务器端完全独立
 - ✓ 由于客户或服务端崩溃所造成的状态不一致性大大减少
 - × 可能损失性能（如服务器无法预料客户行为）

基本服务器设计

- Client-to-Server两种连接方式
 - ✓ **有状态服务器**：服务器端保存客户机的状态
 - 例如，NFS文件服务器
 - ❖ 文件使用表（客户，文件，可更新否）
 - ❖ 服务器重启时：恢复故障前状态
 - ✓ 例1：由于知道文件已被打开，可以进行预取，以提高性能
 - ✓ 例2：知道客户缓存了哪些数据，可以保证数据一致性

基本服务器设计

- 用无状态服务器实现有状态服务
 - 仿真方式：
 - 例：Read文件→Open, Read, Close
 - Cookie: 客户的标识数据
 - 在Web服务器中记录客户的行为历史信息，利用Cookie机制进行管理
 - 对用户是透明的。如保存用户的登录信息
 - × 缺点：违反隐私权，有安全性隐患（仿冒ID）

服务器进程的设计

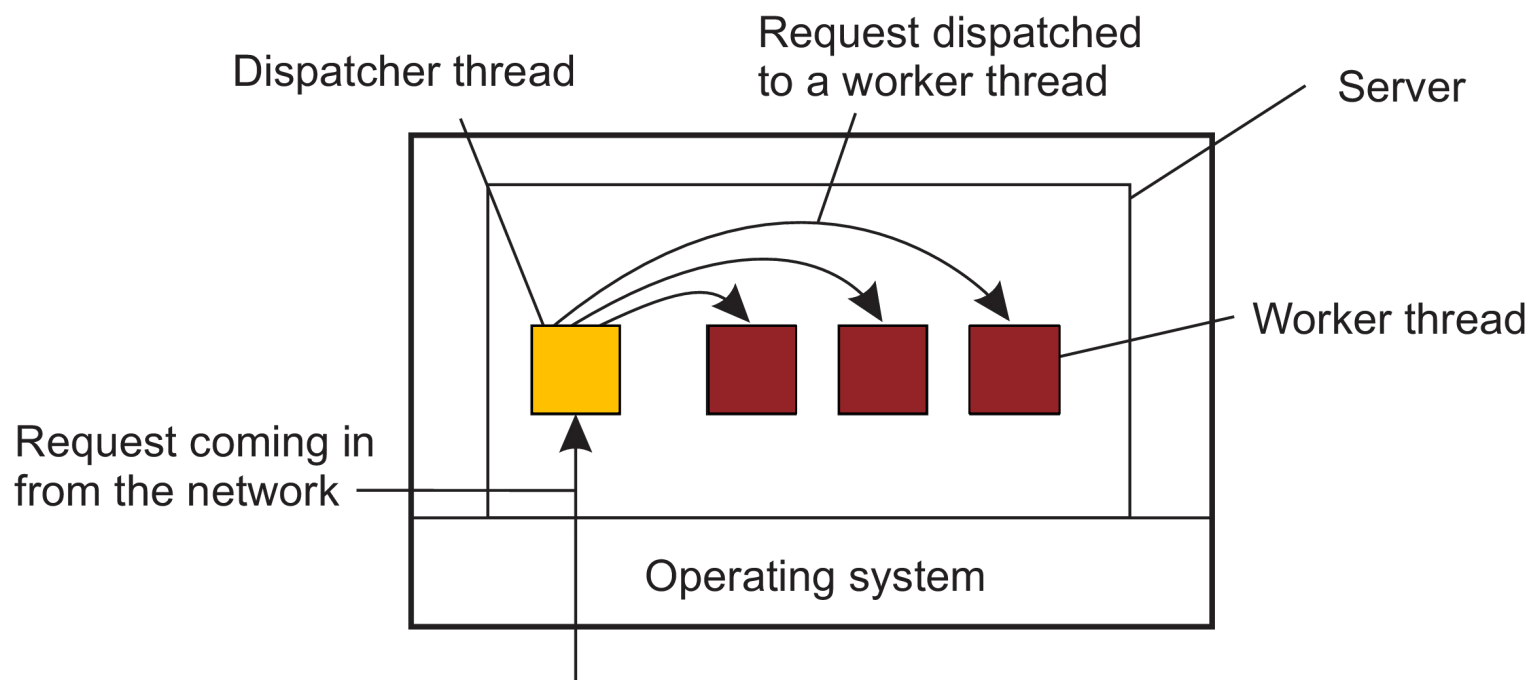
• 多线程服务器 (Servers)

- 提高性能：
 - 启动一个线程的代价比启动一个进程低
 - 易于Scale-Up。单线程服务器端不能利用多核系统
 - 隐藏网络延迟。在回答前一个客户端请求的同时，处理下一个客户端请求。
- 改善程序结构：
 - 简化总体结构。对于频繁的IO需求，实现简单、易于理解的阻塞式调用
 - 小而容易理解。多线程程序使用简化的控制流。

服务器进程的设计

- 举例：多线程文件服务器端

- 分派器线程 
- 工作者线程 



服务器进程的设计

- 多线程代码示例

- (a): 分派者线程 (dispatcher) 源代码
- (b): 工作者线程 (worker) 源代码

```
while(TRUE){  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

```
while (TRUE) {  
    wait_for_request(&buf)  
    look_for_page_in_cache(&buf,&page);  
    if (page_not_in_cache(&page)  
        read_page_from_disk(&buf,&page);  
    return_page(&page)  
}
```

(b)

*可移动服务器（广域网）

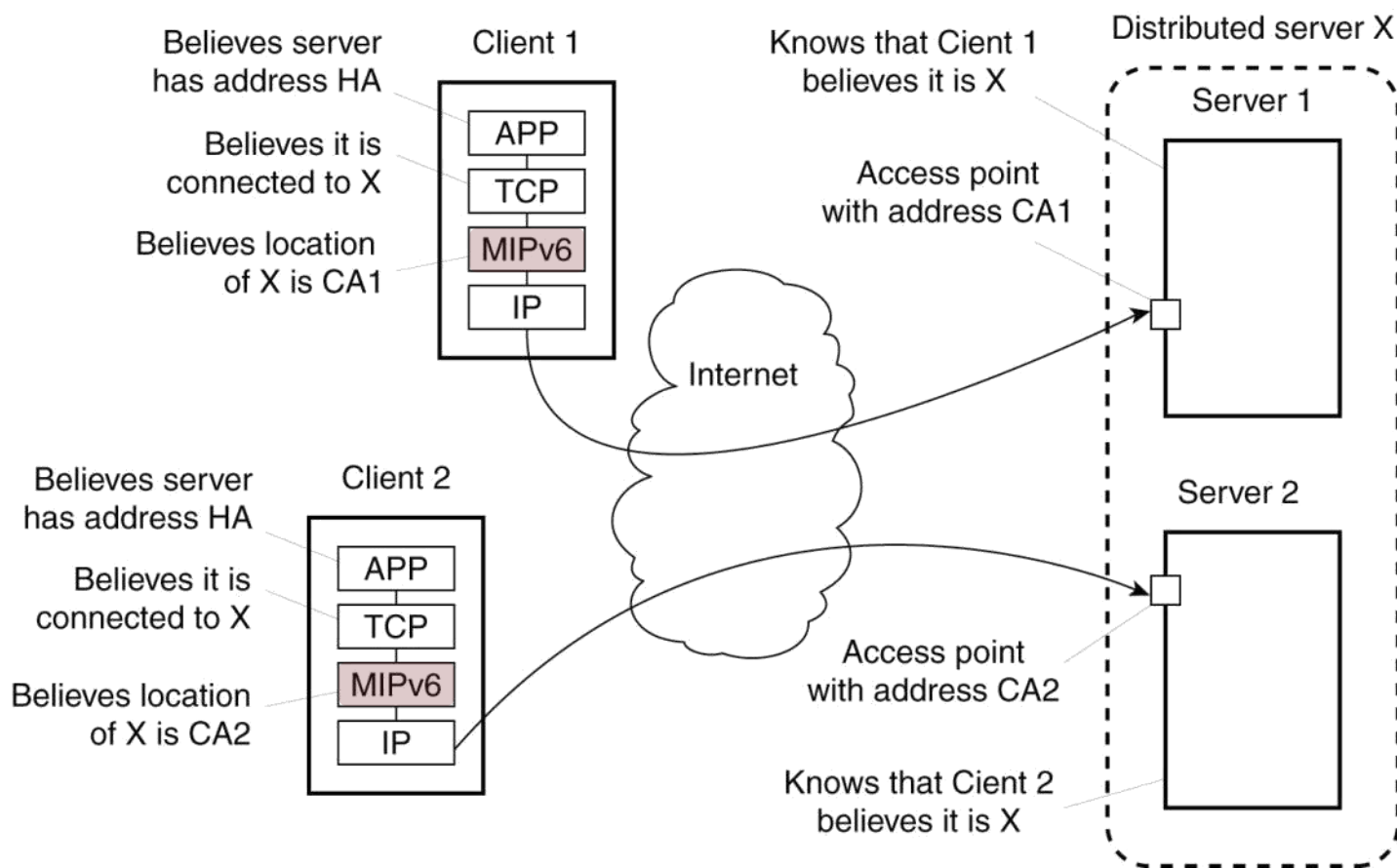
- 作用：
 - 配置可动态变化的机器群
 - 允许访问点变化
- 实现技术
 - 利用MobileIPv6支持移动节点的机制
 - 移动节点
 - 原籍网络 (home network)
 - 原籍地址HA (home address) : 稳定、不变化
 - 原籍代理 (home agent) : 特别的路由器
 - 暂住地址CA (care of address) : 临时的网络地址
 - 客户机进程维护(HA,CA), 通过系统自动转换, 实现移动IP透明性

*可移动服务器（广域网）

- MobileIPv6寻址方法
 - 客户机C建立与HA的连接
 - HA由原籍代 HA_a 维护， HA_a 返回对应的CA
 - C申请路由优化，将消息包转发给CA
- 利用DNS的寻址方法
 - 客户机C查找DNS（域名服务系统）
 - DNS服务器保存有服务器的地址，返回最近的服务器的最新地址CA给C
 - C发送消息给CA

*可移动服务器（广域网）

- 利用MIPv6的访问透明性
 - 例：C1, C2分别用CA1, CA2请求S1, S2



小结

- 多线程 - 提高处理的效率
 - 概念、实现技术、应用
- 客户进程的设计
 - 通信、透明性
- 服务器进程的设计
 - 通信、状态管理

第二章、分布式进程与通信

01

2.1 基本概念

02

2.2 分布式进程和线程设计

03

2.3 远程过程调用

04

2.4 面向消息的通信

05

2.5 *多播通信



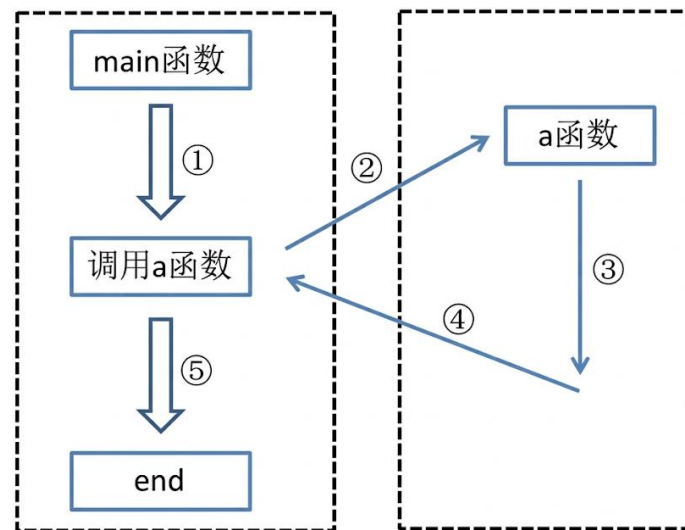
2.3 远程过程调用 (RPC)

• 举例：函数(子程序)调用流程

- ① 保存返回地址
- ② 传递参数
- ③ 执行
- ④ 传递执行结果
- ⑤ 弹出返回地址

• 特点：

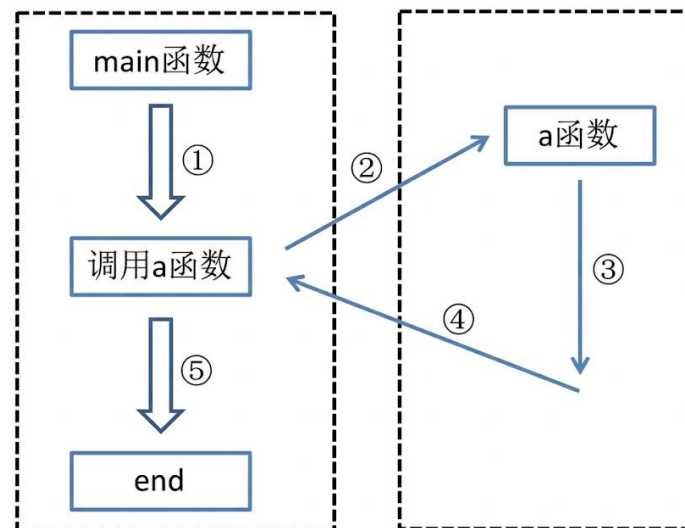
- 对于主函数（调用者）和子函数（被调用者）的**参数和地址**
- **位于同一地址空间**
- **具有同样的格式**



2.3 远程过程调用 (RPC)

• 举例：函数(子程序)调用流程

- ① 保存返回地址
- ② 传递参数
- ③ 执行
- ④ 传递执行结果
- ⑤ 弹出返回地址



• RPC基本概念

- 分布式系统中一种常见的通信模式，支持在不同计算机节点之间像调用本地函数调用远程函数（过程）
- 一个位于计算机A上的进程调用一个位于计算机B上的函数之后，A上的调用者等待，被调函数在B上执行。
- 参数通过网络从A传到B，结果从B传回A。所有信息传递过程对编程者透明（访问透明性）

单机函数调用过程

■ 举例：C程序存储模型

```
int a = 0;
char *p1;
main() {
    int a,b;
    char s[] = "abc";
    char *p2;
    char *p3 = "123456";
    static int c =0;
    p1 = (char *)malloc(10);
    strcpy(p1, "123456");
    Foo(a,b);
}
```

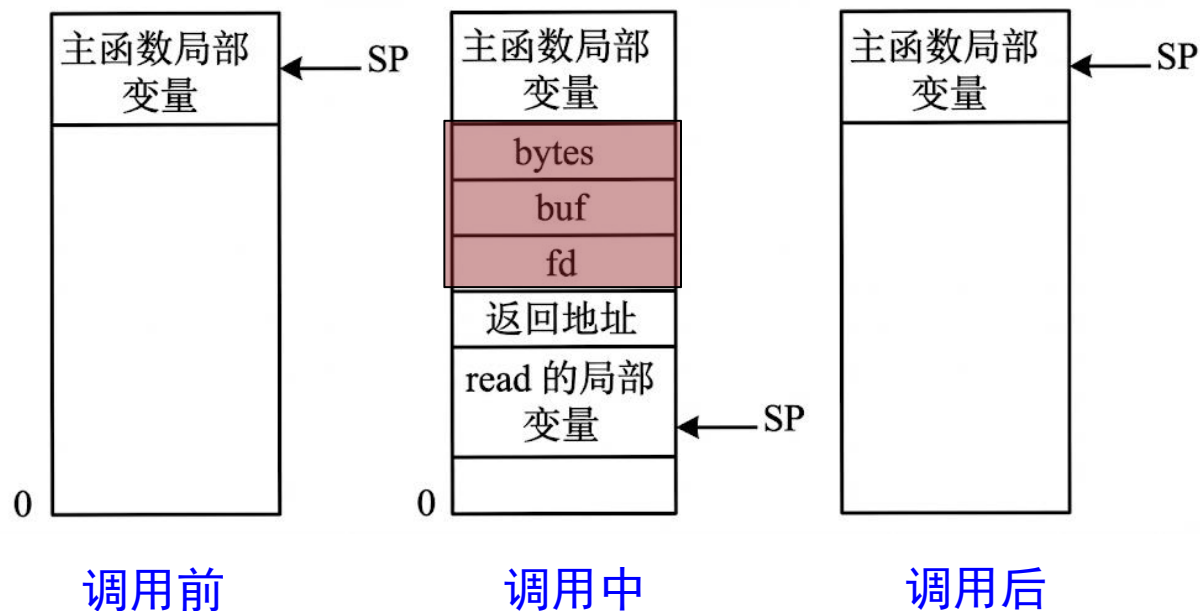
```
Foo(int p1, int p2){
    int x, y;
    static int z;
    z = x+y;
}
```



单机函数调用过程

- 举例

- `int read(int fd, char * buf, int bytes);`
- `count = read(fd, buf, bytes);`

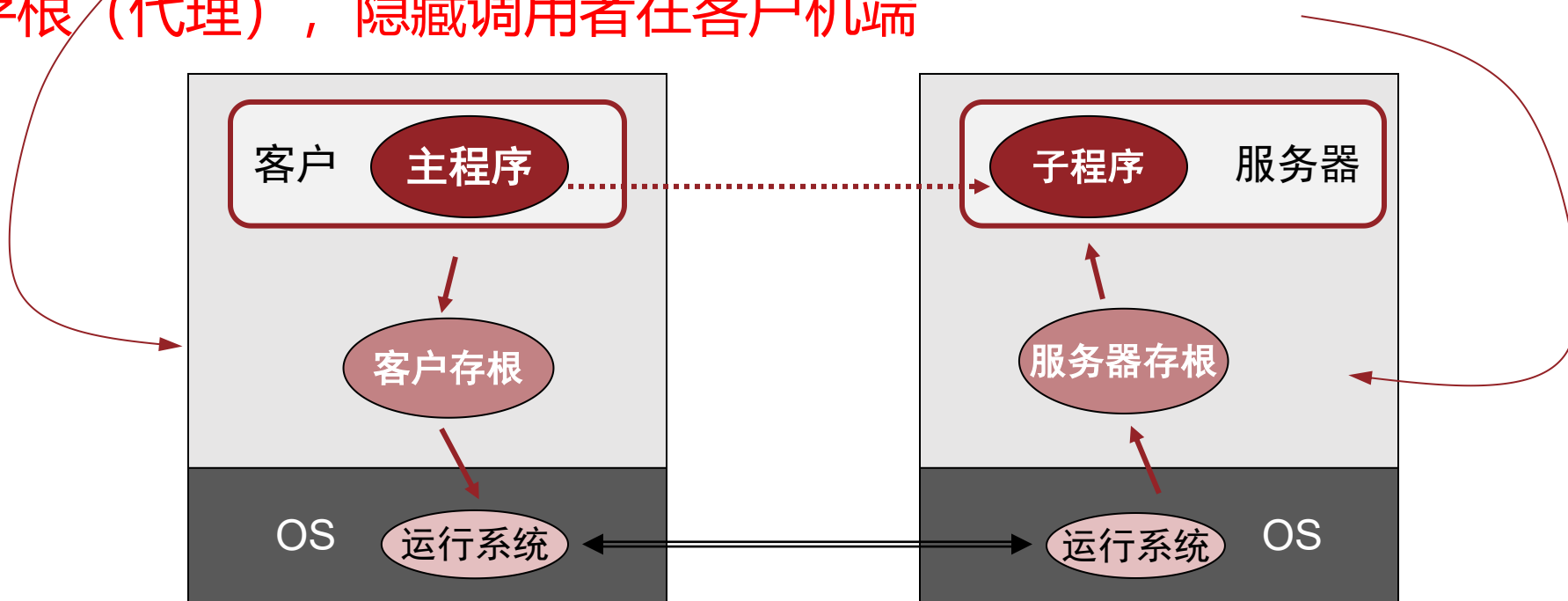


两种典型的参数调用：
call-by-value, call-by-reference

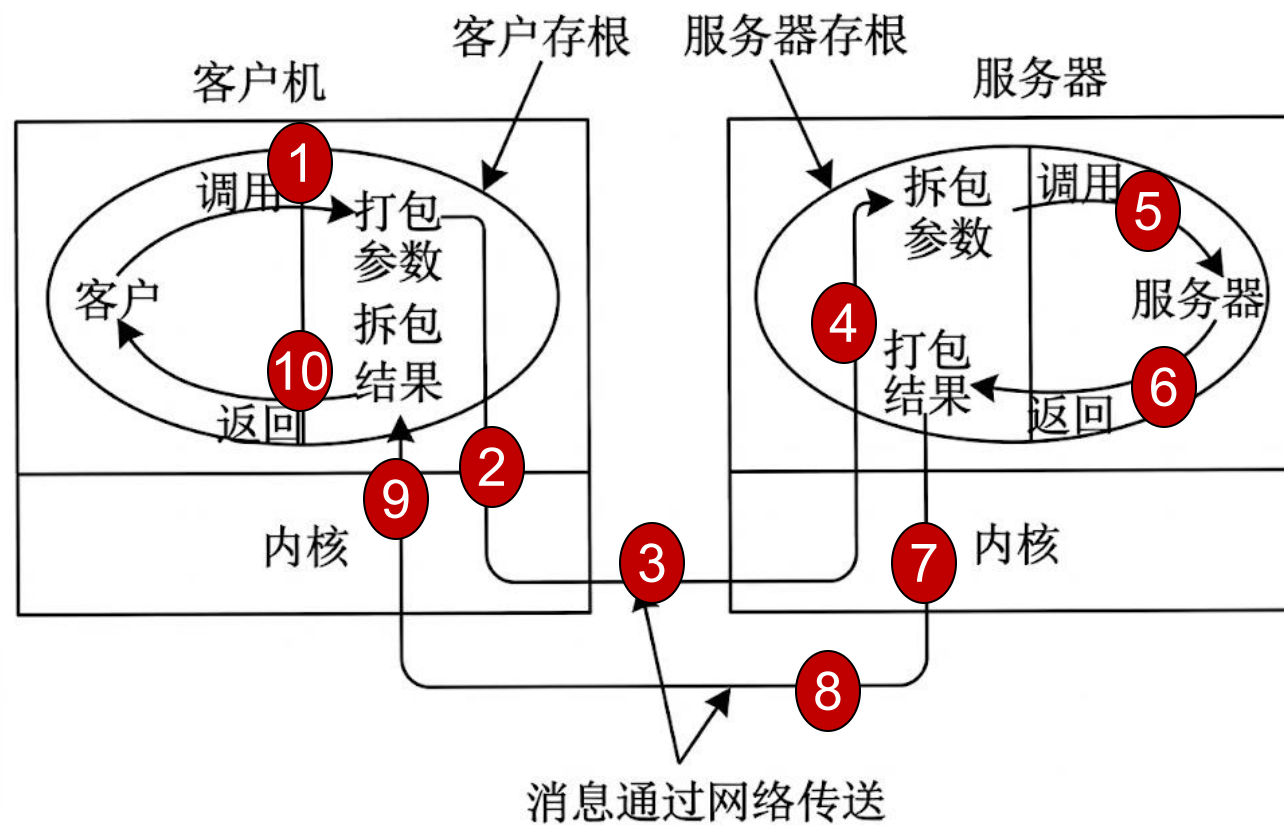
客户存根与服务器存根

- 为实现子程序调用的透明性

- 客户存根（代理），隐藏被调用者在服务器端
- 服务器存根（代理），隐藏调用者在客户机端



RPC的主要执行步骤

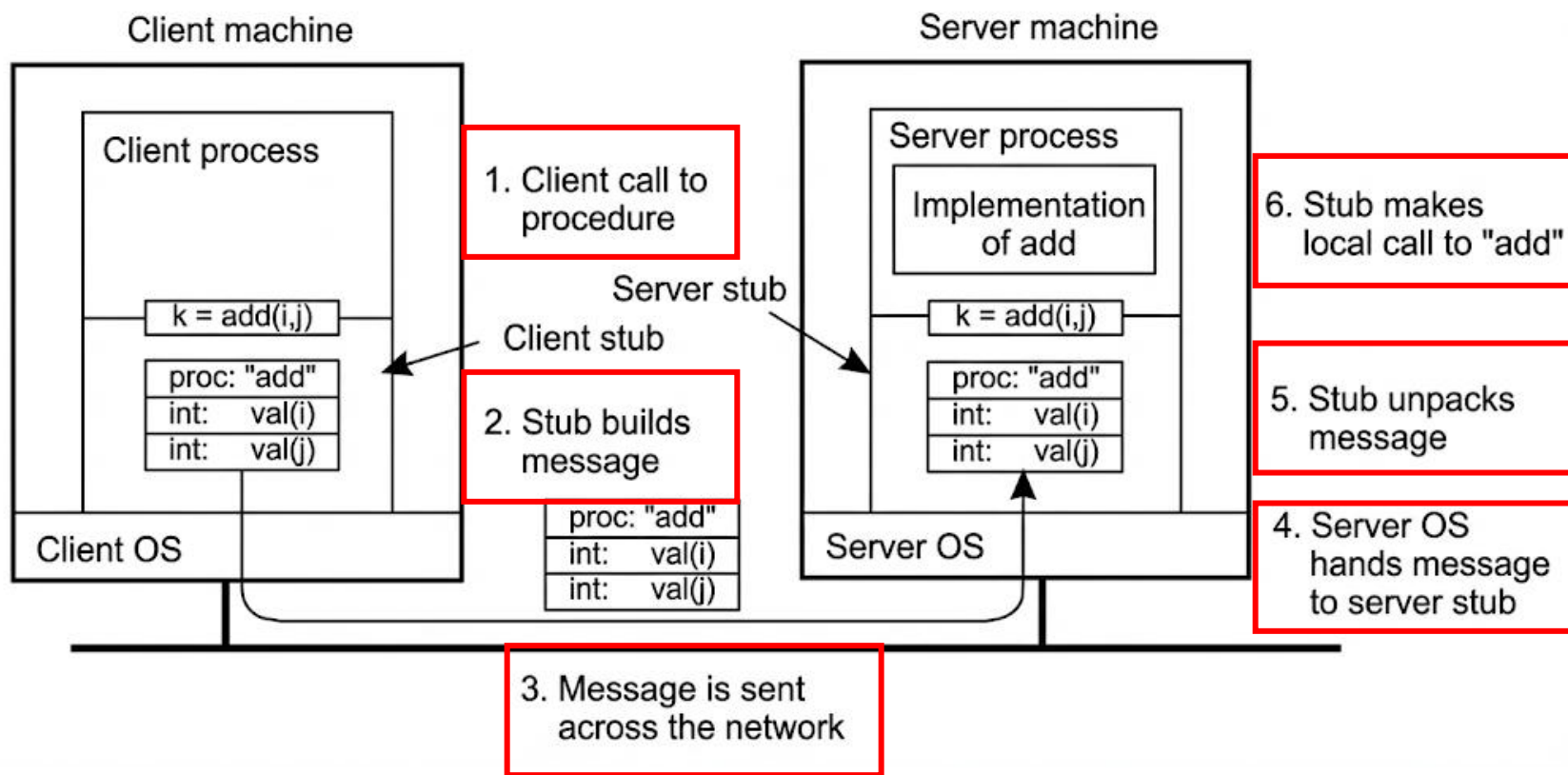


RPC的主要执行步骤

- 1) 客户过程以普通方式调用相应的**客户存根**
- 2) **客户存根**建立消息并激活内核陷阱
- 3) 内核将消息发送到远程内核
- 4) 远程内核将消息送到**服务器存根**
- 5) **服务器存根**取出消息中的参数后调用服务器的过程
- 6) 服务器完成工作后将结果返回至**服务器存根**
- 7) **服务器存根**将它打包并激活内核陷阱
- 8) 远程内核将消息发送至客户内核
- 9) 客户内核将消息交给**客户存根**
- 10) **客户存根**从消息中取出结果返回给客户

参数传递—基本过程

- 举例：远程调用`add(i, j)`



参数传递—数据表示问题

- 在异构体系结构下，大小端模式的问题
- 例：数据 5,JILL

3	2	1	0
0	0	0	5
7	6	5	4
L	L	I	J

(a) X86格式（小端）
发送方的格式

0	1	2	3
5	0	0	0
4	5	6	7
J	I	L	L

(b) SPARC格式（大端）
接收方出现整型值错误

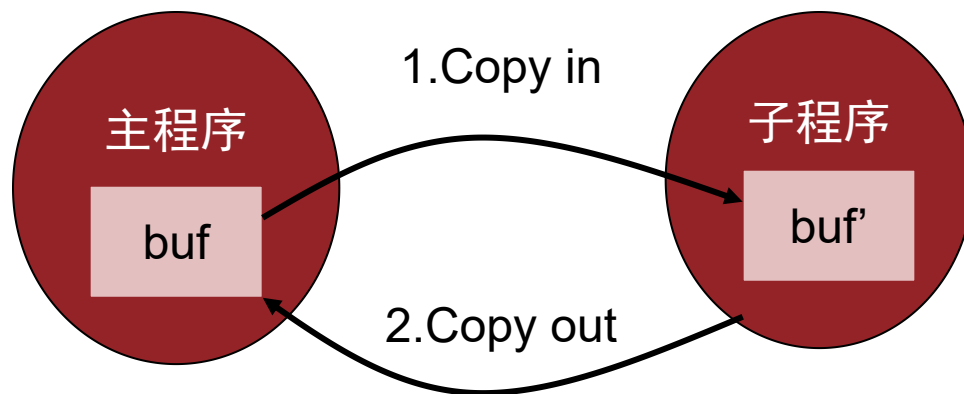
0	1	2	3
0	0	0	5
4	5	6	7
L	L	I	J

(c) 全部逆转的格式
接收方出现字符串错误

- 解决方案：
 - 将消息转换成独立于机器的表示形式
 - 编排/解编(marshaling/unmarshaling)

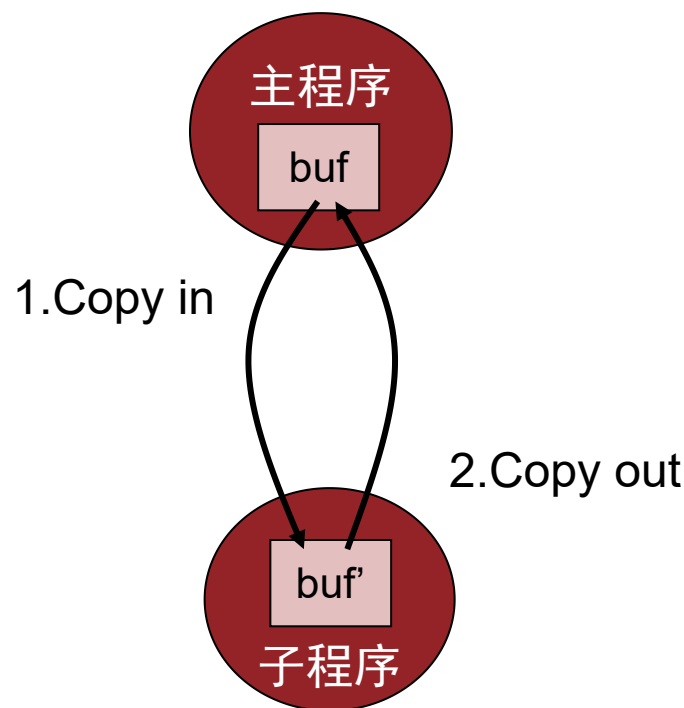
参数传递—引用(指针)的问题

- 问题：
 - 客户机传送的指针参数指向本地某个内存区域内的数据，例如数组，同样的指针值在服务器端所指向的内存区域无定义
- 解决方案
 - Copy in/copy out方法，也叫copy-by-value/restore
 - 在客户机和服务器之间拷贝数据



参数传递—引用(指针)的问题

- Copy in/copy out的不足
 - 当远程过程被执行时，不能保证对参数值的及时更新
 - 所有被操作的数据是通过参数传递，不能传递全局数据引用（如全局文件句柄）
 - 不能全部实现访问透明性
- 解决方案-远程引用（remote reference）
 - 可提供对远程数据的统一访问
 - 可作为RPC中的参数传递
 - 存根有时可用作全局引用



RPC协议

- 定义消息格式
 - 数据表示、编码规则
 - 例： `foobar(char x; float y;int z[5]) { }`
- 定义接口代码
 - 采用IDL（接口定义语言）实现
 - 通过IDL编译器，编译IDL接口源代码，生成客户存根程序和服务器存根程序

发送的消息

局部变量	
	x
y	
5	
z[0]	
z[1]	
z[2]	
z[3]	
z[4]	

RPC编程举例

- RPC语言源文件（UNIX系统）

FILE_RPC.x /* 举例程序，实现远程文件读写 */

[root]# rpcgen File_RPC.x /* 执行rpcgen命令 */

- 相关的c语言源代码文件（由rpcgen自动生成）

1、参数定义 源代码文件

FILE_RPC.h /* 声明变量和函数头文件*/

FILE_RPC_xdr.c /* 遵从XDR标准的参数编排和解编程序*/

2、服务器端程序源代码文件

FILE_RPC_svc.c /* server存根程序，不需修改 */

FILE_RPC_server.c /* 服务器模版程序，需要用户修改 */

3、客户机程序 源代码文件

FILE_RPC_clnt.c /* client存根程序，不需修改 */

FILE_RPC_client.c /* 客户模版程序，需要用户修改 */

RPC声明-- File_RPC.x

```
#define RPC_ARGSIZ  256
#define RPC_RETSIZE 256
/* the interface version number */
#define uSERVERVERS 1

/* program number range established by
   ONC */
#define uSERVERPROG 0x20000051

struct CallArg_t {
    char body[RPC_ARGSIZ];
}; /* RPC's arg data form */

struct Return_t {
    char body[RPC_RETSIZE];
}; /* RPC's arg data form */
```

返回参数

```
/* program definition*/
program SERVERPROG {
    version SERVERVERS {

        Return_t FILE_CREATE(CallArg_t) = 1;
        Return_t FILE_READ(CallArg_t) = 2;
        Return_t FILE_WRITE(CallArg_t) = 3;
        Return_t FILE_DEL(CallArg_t) = 4;

    } = uSERVERVERS;
} = uSERVERPROG;
```

RPC程序编号

版本号

输入参数

[root]# rpcgen File_RPC.x

RPC服务器存根程序 -- File_RPC_svc.c

```
#include <rpc/rpc.h>
#include FILE_RPC.h
.....
void main2() {
    SVCXPRT *transp;

    transp = svcudp_create(RPC_ANYSOCK);
    svc_register(transp, SERVERPROG, SERVERVERS, serverprog_1,
        IPPROTO_UDP); /* 注册 */

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    svc_register(transp, SERVERPROG, SERVERVERS, serverprog_1,
        IPPROTO_TCP); /* 注册 */

    svc_run(); // 启动RPC

    exit(1);}
}
```

RPC服务器程序 -- File_RPC_server.c

```
static void serverprog_1(struct svc_req *
    rqstp, SVCXPRT * transp)
{
    union { CallArg_t file_create_1_arg;
            CallArg_t file_read_1_arg;
            :
        } argument;
    char *result;

    xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);

    switch (rqstp->rq_proc) {
        :
        case FILE_READ:
            xdr_argument = xdr_CallArg_t;
            xdr_result = xdr_Return_t;
```

```
        local = (char *(*)(char *, struct svc_req
            *))
            file_read_1_svc; /* 被调用的子程序*/
            break;
        :
        default:
            vcerr_noproc(transp);
    return;
    }

    (void) memset((char *)&argument, 0,
        sizeof (argument));
    svc_getargs(transp, _xdr_argument,
        (caddr_t) &argument)
    result = (*local)((char *)&argument,
        rqstp);
}
```

RPC客户存根程序 -- File_RPC_clnt.c

```
.....  
/* Default timeout can be changed using clnt_control() */  
static struct timeval TIMEOUT = { 25, 0 };  
  
Return_t * file_read_1(CallArg_t *argp, CLIENT *clnt)  
{ static Return_t res;  
  
    bzero((char *)&res, sizeof(res));  
  
    if (clnt_call(clnt, FILE_READ, xdr_CallArg_t, argp, xdr_Return_t,  
        &res, TIMEOUT) != RPC_SUCCESS)  
        return (NULL);  
}  
return (&res);  
.....
```

RPC客户程序 -- File_RPC_client.c

```
#include <rpc/rpc.h>
#include <rpc/auth.h>
#include <rpc/clnt.h>
#include FILE_RPC.h
int read(char *file, int position , int nbytes,
char *buf){
    struct param p1; /* 参数缓冲区 */
    p1.offset = position; /* 读位置 */
    p1.count = nbytes; /* 读长度 */
    strcpy(&p1.filename,file) /* 文件名 */

    CommonCallArg_t * argument =
    (CommonCallArg_t *) p1;
}
```

```
hostname="localhost";
CLIENT* cl = clnt_create(hostname,
    SERVERPROG, SERVERVERS,
    "tcp");

clnt_control(cl, CLSET_TIMEOUT,
    (char *) &WS_TIMEOUT);

if ( file_read_1(argument, (CLIENT *)cl)
    <> 0 ) /*远程子程序调用*/
    strcpy(buf, &p1.data);

clnt_control(cl, CLSET_FD_CLOSE,
    NULL);
clnt_destroy(cl);
}
```

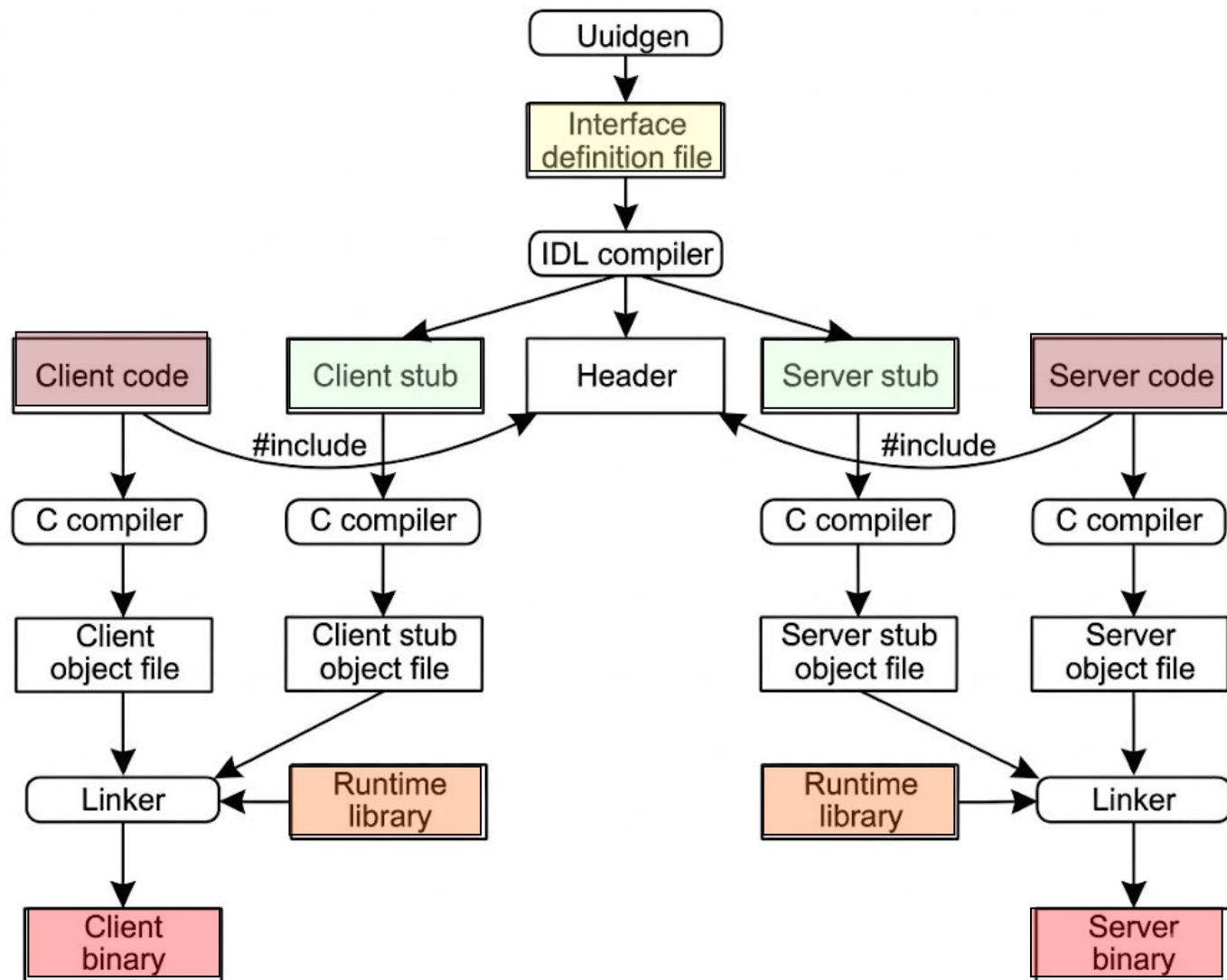
举例：DCE RPC

- DCE (Distributed Computing Environment)
- 设计目标
 - DCE RPC是一个纯中间件系统，目的是为客户机提供远程过程调用的功能，简化客户机程序开发
- 功能
 - 隐藏客户机与服务器端之间的通信
 - 自动定位服务器地址，自动建立网络连接
 - 处理双向信息传输
 - 消息分片与组合
 - 数据类型转换

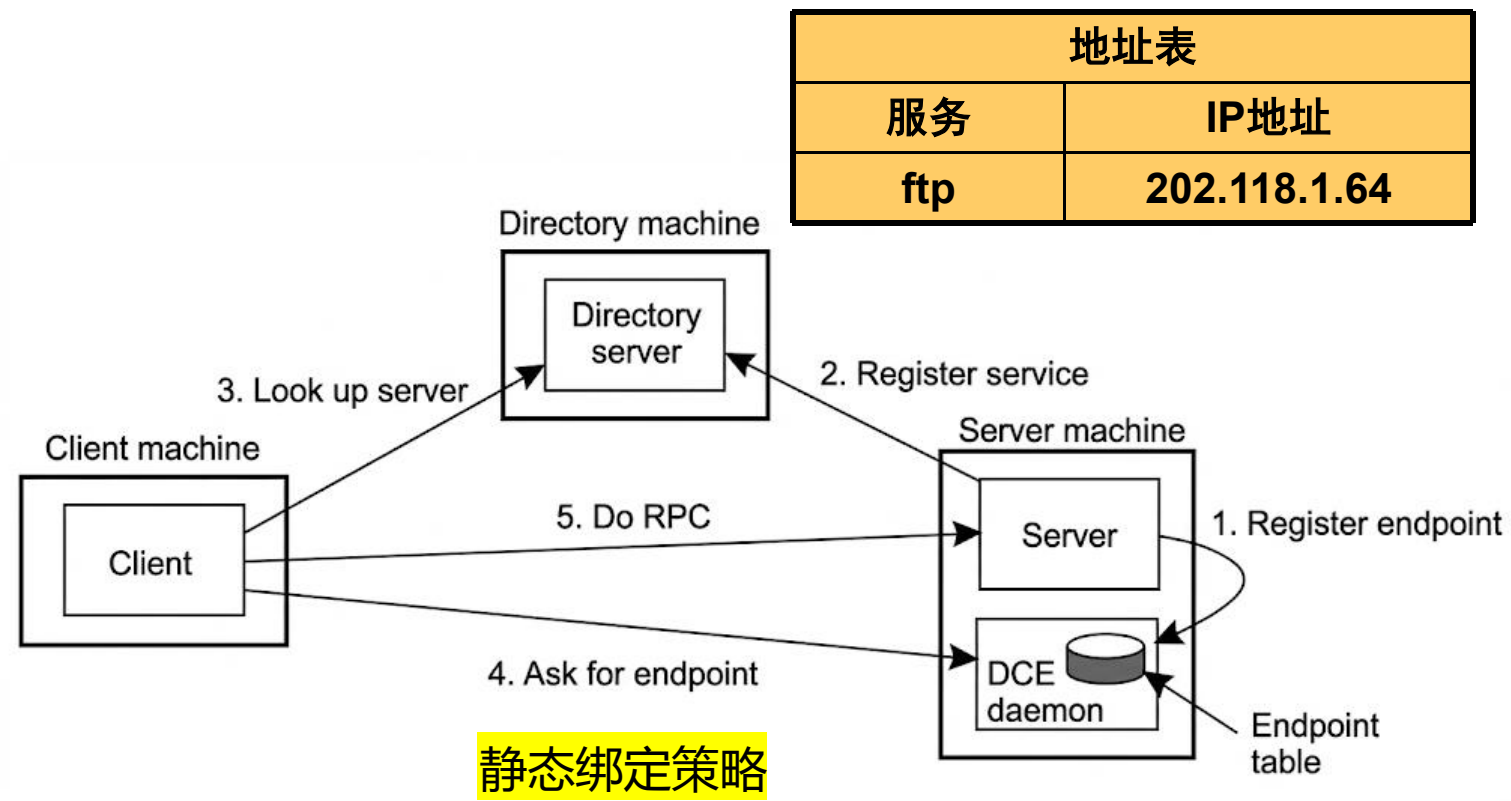
举例：DCE RPC

- 特点
 - 允许客户机程序和服务器端程序采用不同语言编写
 - 允许客户机与服务器端运行在不同的硬件体系结构上
 - 允许客户机与服务器端运行不同的操作系统
- 接口的定义
 - IDL：接口定义语言
 - 头文件，如C语言的.h文件
 - 客户存根程序
 - 服务器存根程序

客户机与服务器端程序的编写



客户进程与服务器进程的绑定



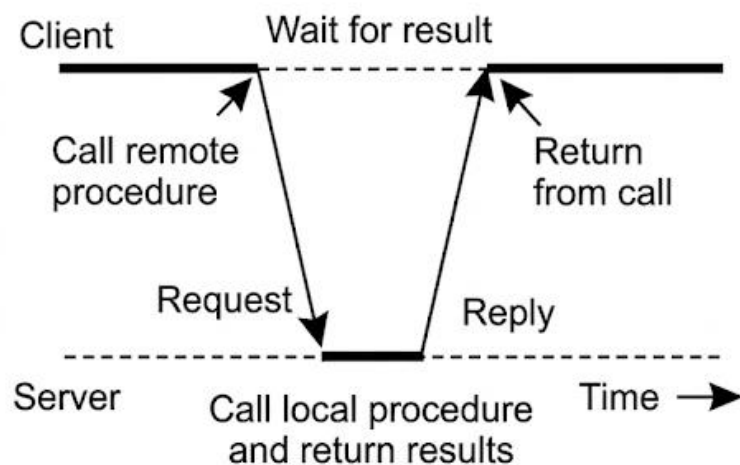
静态绑定策略

地址表	
服务	IP地址
ftp	202.118.1.64

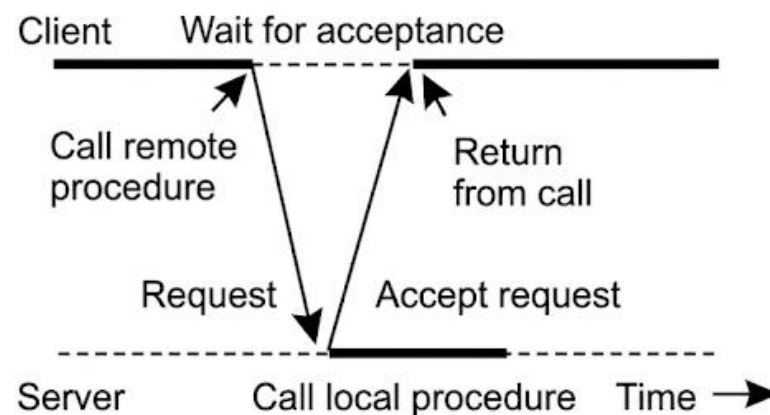
端点表	
服务器	端口号
ftp	8080

异步RPC

- 基本RPC过程中的问题
 - 客户机调用函数需要一直阻塞直至结果返回
- 在某些情况下，客户机调用者无需等待
 - 例如：从一个账户划款到另一账户，向数据库中添加一条记录



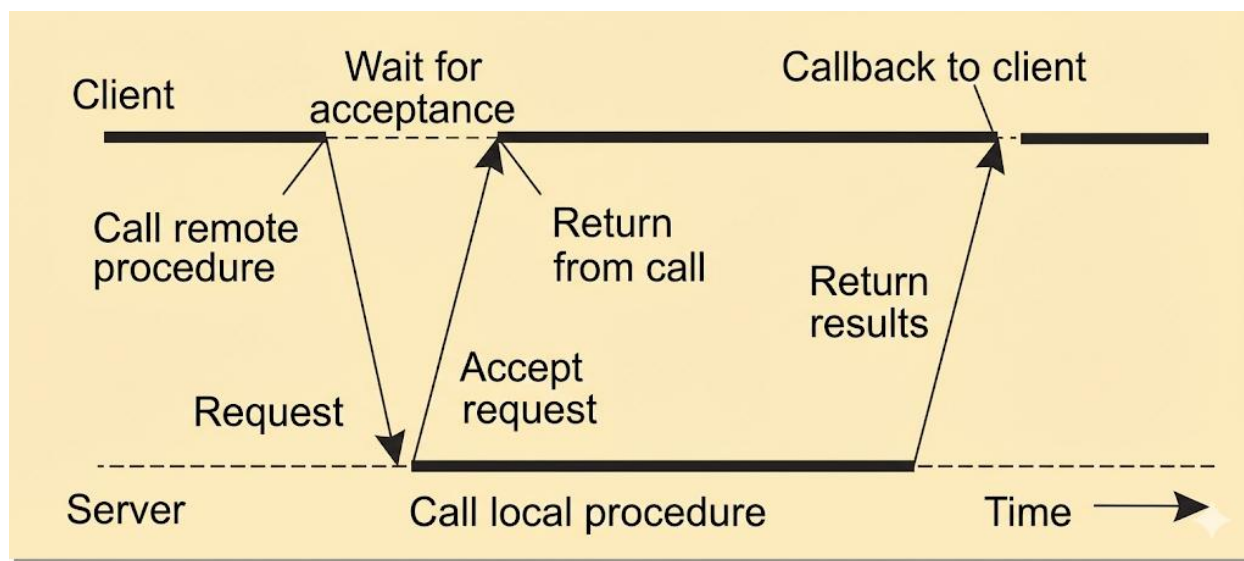
传统PRC



异步PRC

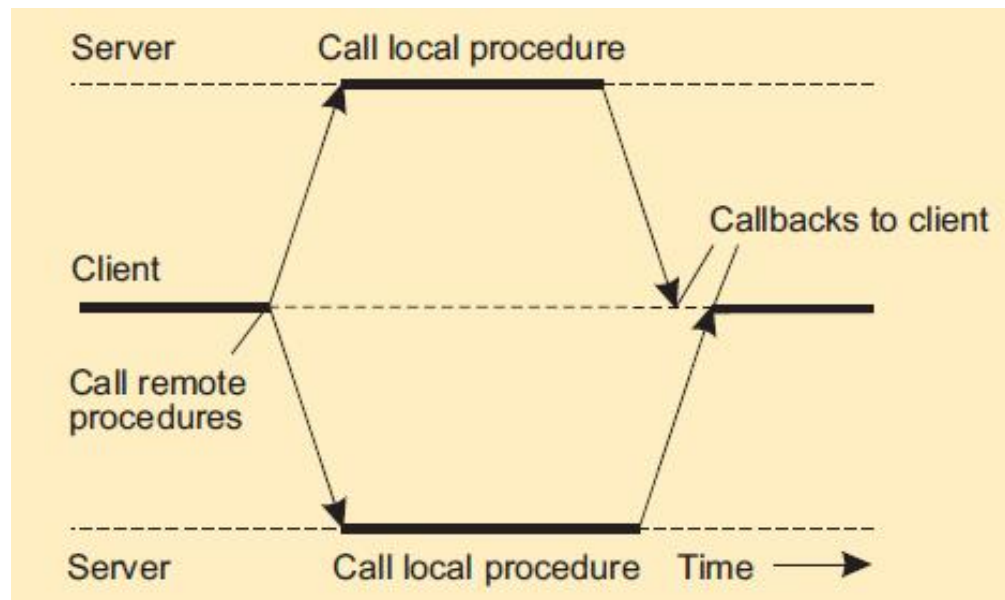
异步RPC

- 需要传回运算结果的异步PRC
 - 也称为**延迟的同步RPC**
 - 回调操作（callback），用户定义的函数，在特定事件（如传入消息）发生时启动执行



多播RPC

- 发送一个RPC请求到一组服务器
 - 按异步或延迟同步执行RPC
 - 各服务器并行执行
 - 通过回调（Callback），返回结果
- 用途
 - 复制透明性
 - 支持容错



第二章、分布式进程与通信

01

2.1 基本概念

02

2.2 分布式进程和线程设计

03

2.3 远程过程调用

04

2.4 面向消息的通信

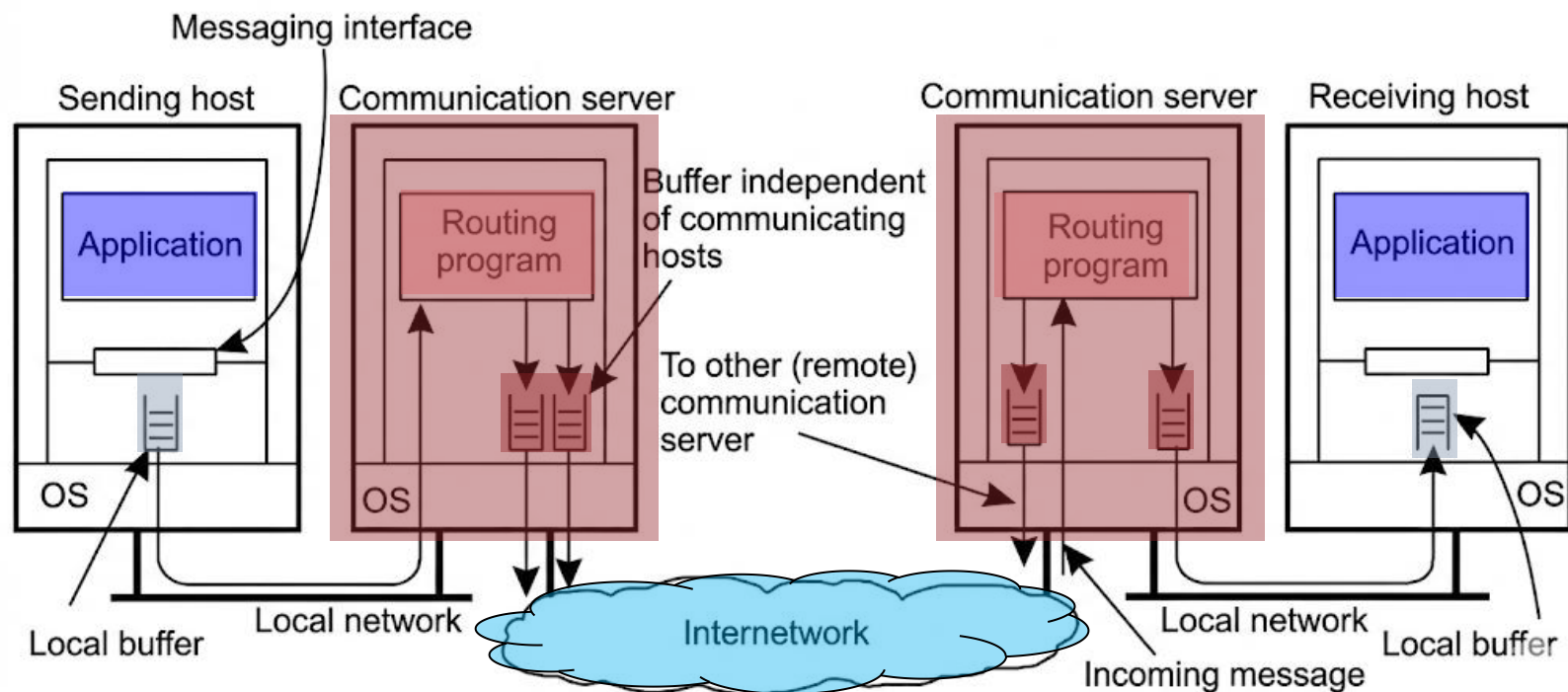
05

2.5 *多播通信



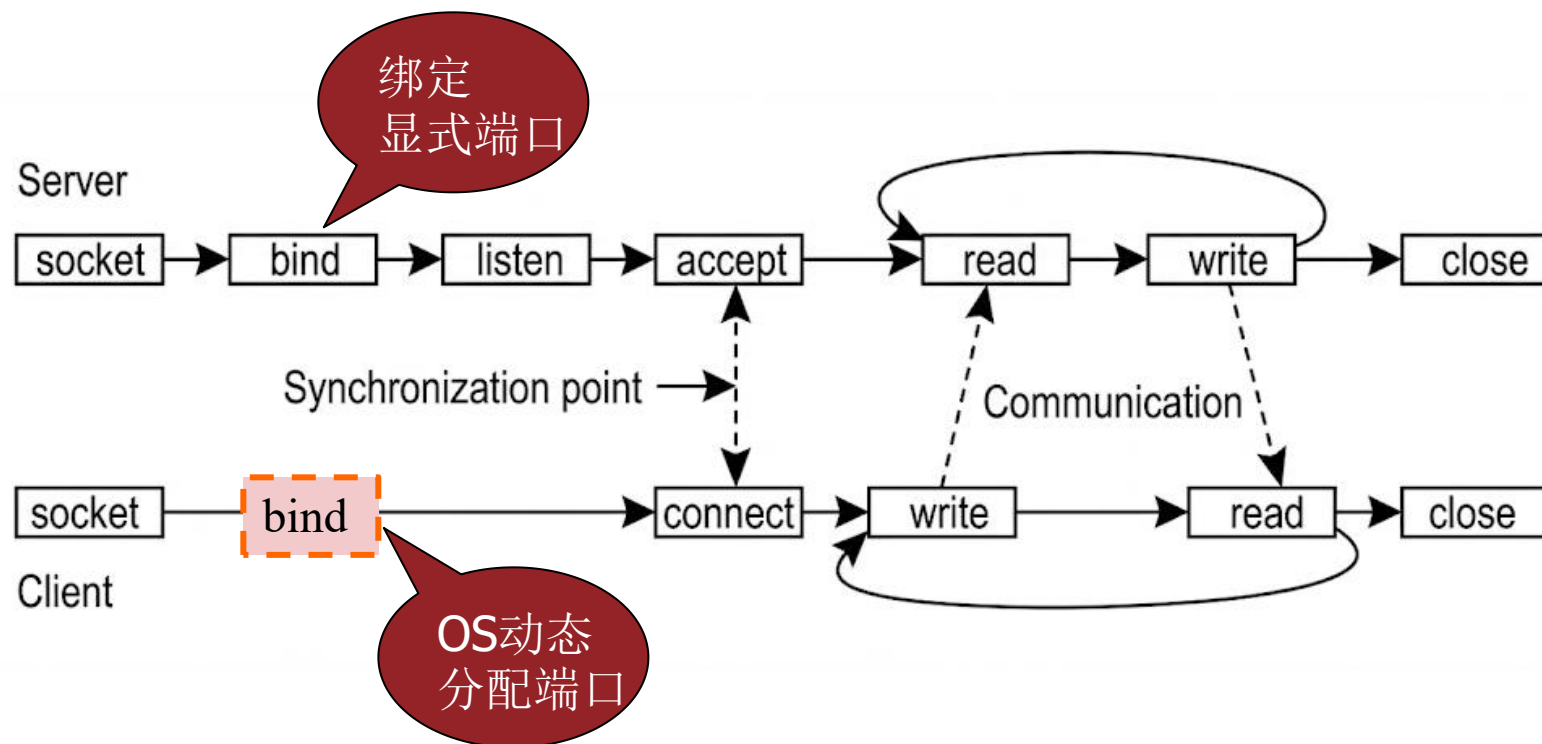
2.4 面向消息的通信

- 通信系统的一般结构
- 通信服务器：中间件系统
 - 缓冲区：收发的消息
 - 路由程序：建立远程连接通道



面向消息的瞬时通信

- 简单方法：Berkeley Socket接口
 - 在传输层实现
 - Socket：通信端点，用于读写信息



面向消息的瞬时通信

- Berkeley Socket中的操作原语

原语	意义
socket	创建新的通信端点
bind	将本地地址绑定(attach)到套接字上
listen	宣布已准备好接受连接，用于有连接通信
accept	在收到连接请求之前，让服务器等待
connect	在连接到指定地址， 建立好连接通道之前，让客户等待
send	通过连接通道，发送数据
receive	通过连接通道，接受数据
close	释放连接

Berkeley Sockets 举例

• server程序

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
.....
int sock_server(char* message ) { ....
    struct sockaddr_in addr;;
    int sock, slave;
    int len = sizeof(struct sockaddr_in) ;
    addr.sin_family = AF_INET;
    addr.sin_port = 80;
    addr.sin_addr.s_addr =
    INADDR_ANY ;
    sock = socket( addr->sa_family,
    SOCK_STREAM , 0 ) ; /* 创建 */
```

```
bind(sock, addr, len ); /* 绑定地址 */
```

```
listen( sock, 16 ); /* 侦听连接 */
```

```
struct sockaddr_in client ;
```

```
slave=accept(sock, (struct
sockaddr*) & (client), &len );
/* create new slave socket */
```

```
recv( slave, buffer, sizeof(ulong), 0 );
put_message(message, buffer);
close(slave); /* 关闭 */
close(sock); /* 关闭 */
```

```
}
```

Berkeley Sockets 举例

• client程序

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

.....

int sock_client (char * message, struct
sockaddr_in server ) { ....
    struct sockaddr_in addr;
    int sock;
    int len;

    addr.sin_family = AF_INET;
    addr.sin_port = 80;
    addr.sin_addr.s_addr =
    INADDR_ANY ;
```

```
sock = socket( addr->sa_family,
SOCK_STREAM , 0 ) ; /* 创建 */
```

```
len = sizeof(struct sockaddr_in) ;
```

```
connect( sock, (struct sockaddr
*)server, &len) ;
```

```
set_message( buffer, message ) ;
sendto(sock, buffer, sizeof(ulong) +
message_size(), 0, (struct
sockaddr*)server, &len )
```

```
close(sock); /* 关闭 */
}
```

消息传递接口 (MPI)

- 传输层消息传递方式的不足
 - 仅提供非常简单的发送/接收消息的原语
 - TCP/IP不适用于高速网络。高速网络有专用协议。
- MPI (Message Passing Interface) :
 - 一种硬件和平台独立的消息传递标准
- MPI的特点
 - 支持并行计算
 - 消息传递发生在设定好的一组进程内
 - 提供多种通信方式，便于灵活的性能优化

消息传递接口 (MPI)

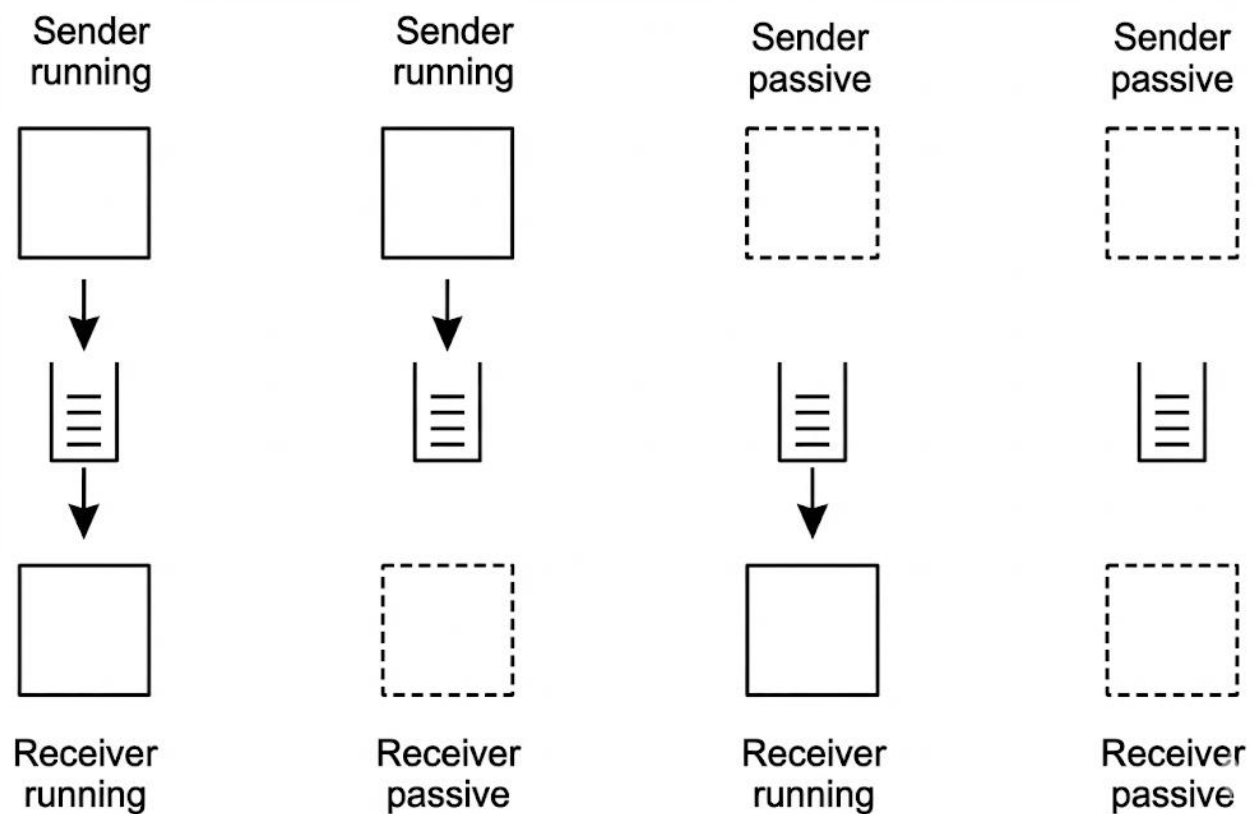
原语	意义
MPI_bsend	将要送出的消息追加到本地发送缓冲区中
MPI_send	发送消息，并等待直到消息复制到本地或远程缓冲区中为止
MPI_ssend	发送消息，并等待直到对方开始接受为止
MPI_sendrecv	发送消息，并等待直到收到应答消息为止
MPI_isend	传送要送出消息的引用，随后继续执行
MPI_issend	传送要送出消息的引用，并等待直到对方开始接受为止
MPI_recv	接受消息，如果不存在等待的消息则阻塞
MPI_irecv	检查是否有输入的消息，但是无论有没有消息都不会阻塞

面向消息的持久通信

- 消息队列系统（Message-Queuing System）
 - 遵守**公共消息协议**：所有应用在消息格式上（数据结构和数据表示）一致
 - 通过提供中间存储能力，支持**持久性异步通信**
 - 支持传输时间较长的通信，如延迟为几分钟的通信
 - 实现了发送方和接收方的松散耦合：发送方仅确保所发送的消息能够到达接收方的队列，但并不保证消息何时被接收方阅读（取决于接收方自身行为）

消息队列模型

- 四种方式
 - 双主动、主动/被动、被动/主动、双被动



消息队列通信原语

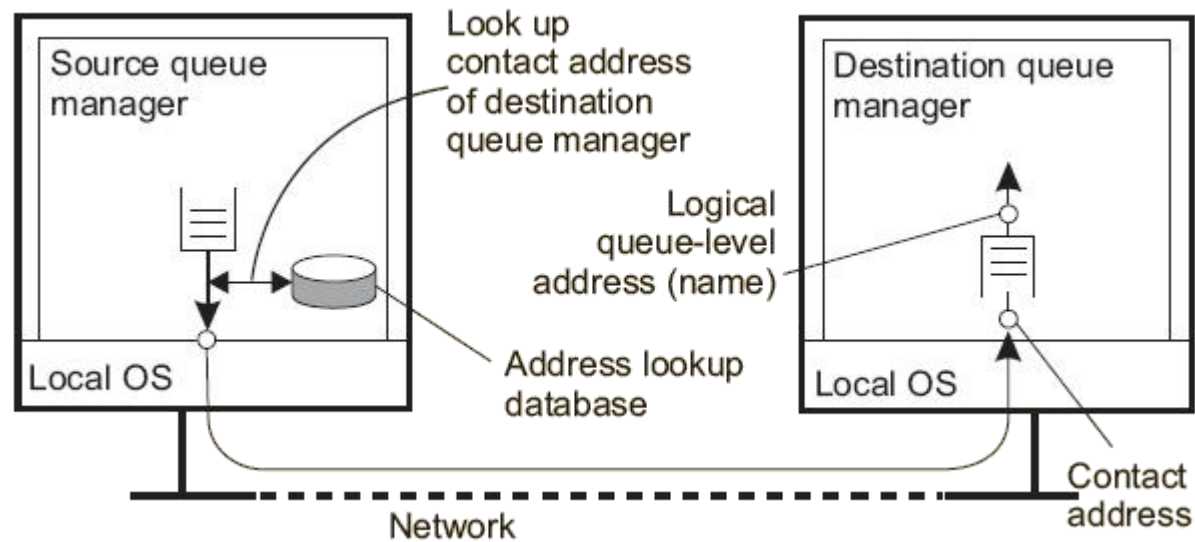
原语	意义
Put	将消息追加进指定队列
Get	调用进程阻塞，直到指定队列变为非空为止， 然后取出队列中的第一个消息
Poll	查看指定队列中的消息， 并且取出队列中的第一个消息。不阻塞调用进程
Notify	注册一个处理程序， 在有消息进入指定队列时调用该处理程序

消息队列一般结构

- 队列分类
 - 源队列 (source queue) : 发送
 - 目的队列 (destination queue) : 接收
- 队列寻址
 - 逻辑地址: 队列名, 如 steen@cs.vu.nl
 - 网络地址: IP地址
- 队列管理器
 - 发送/接收
 - 中转 (relay)

消息队列通用模型

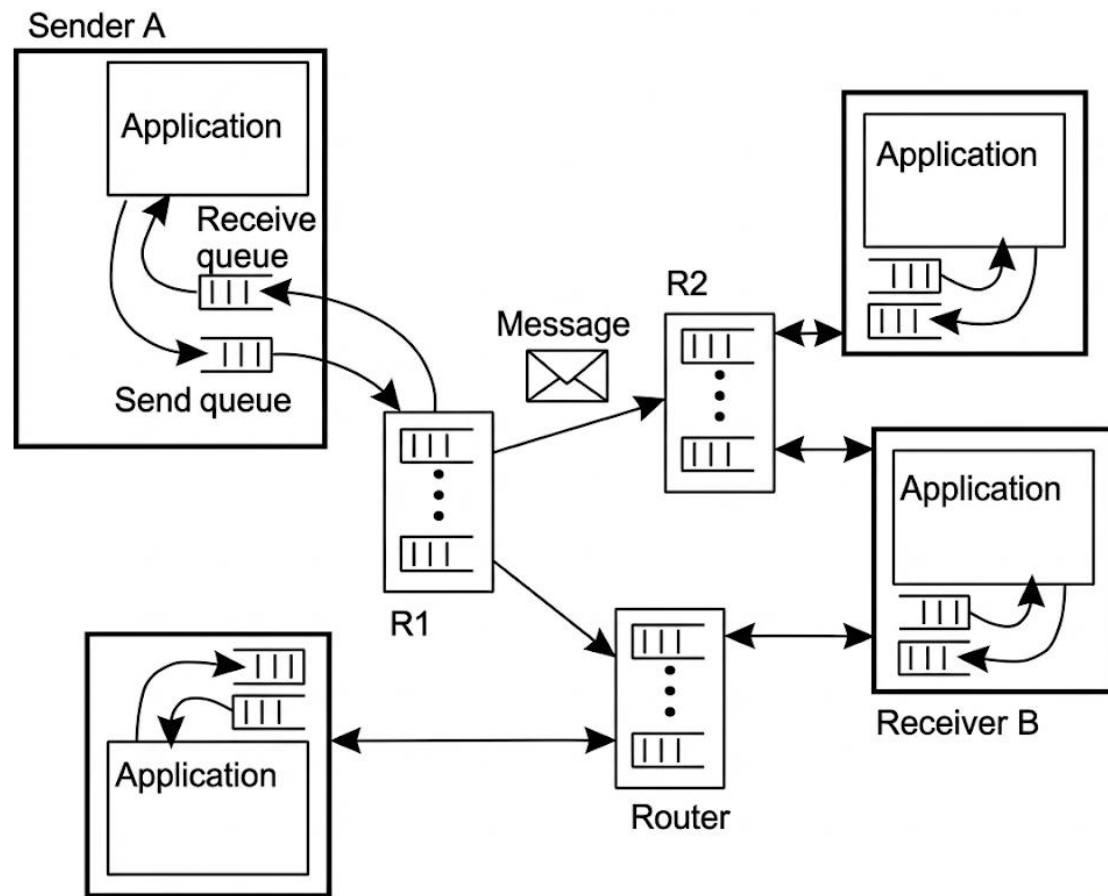
- 队列管理器：
 - 应用进程只能将消息放进本地队列（写操作）
 - 或者，只能从本地队列中获得消息（读操作）
 - 对消息进行路由



队列名与网络地址之间的关系

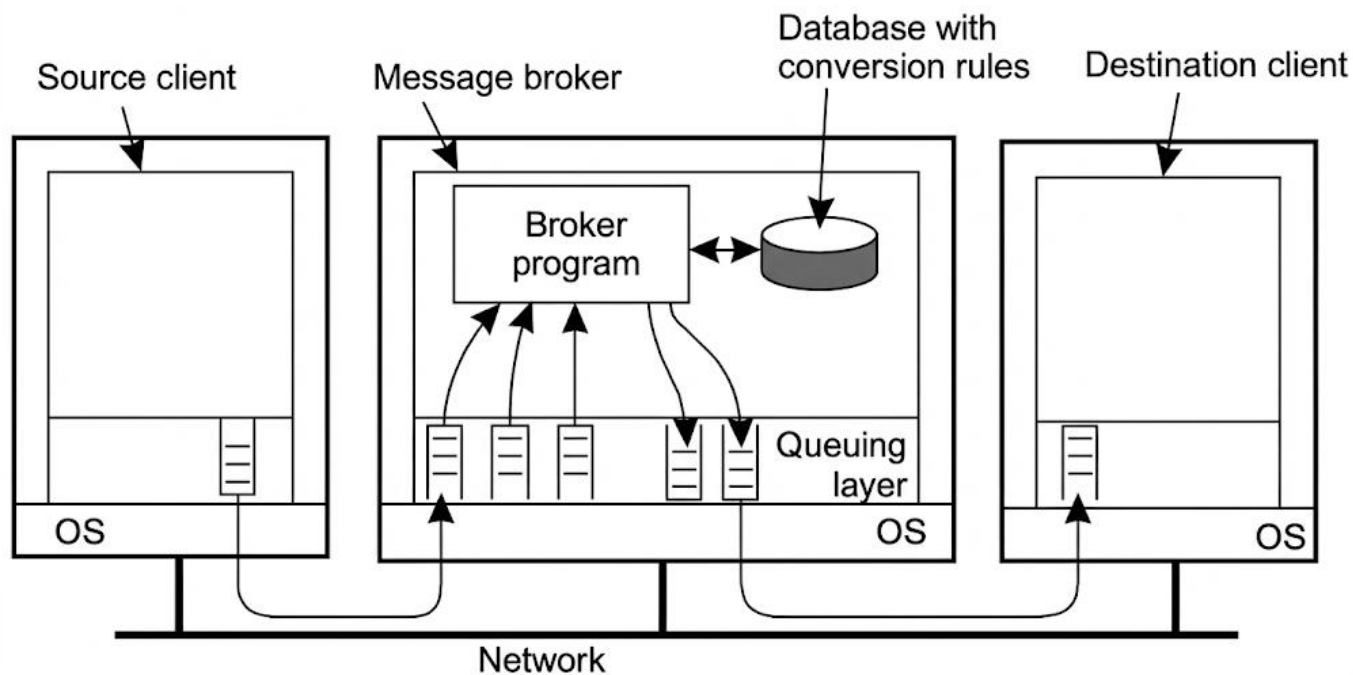
消息队列一般结构

- 带有中继的消息队列系统结构



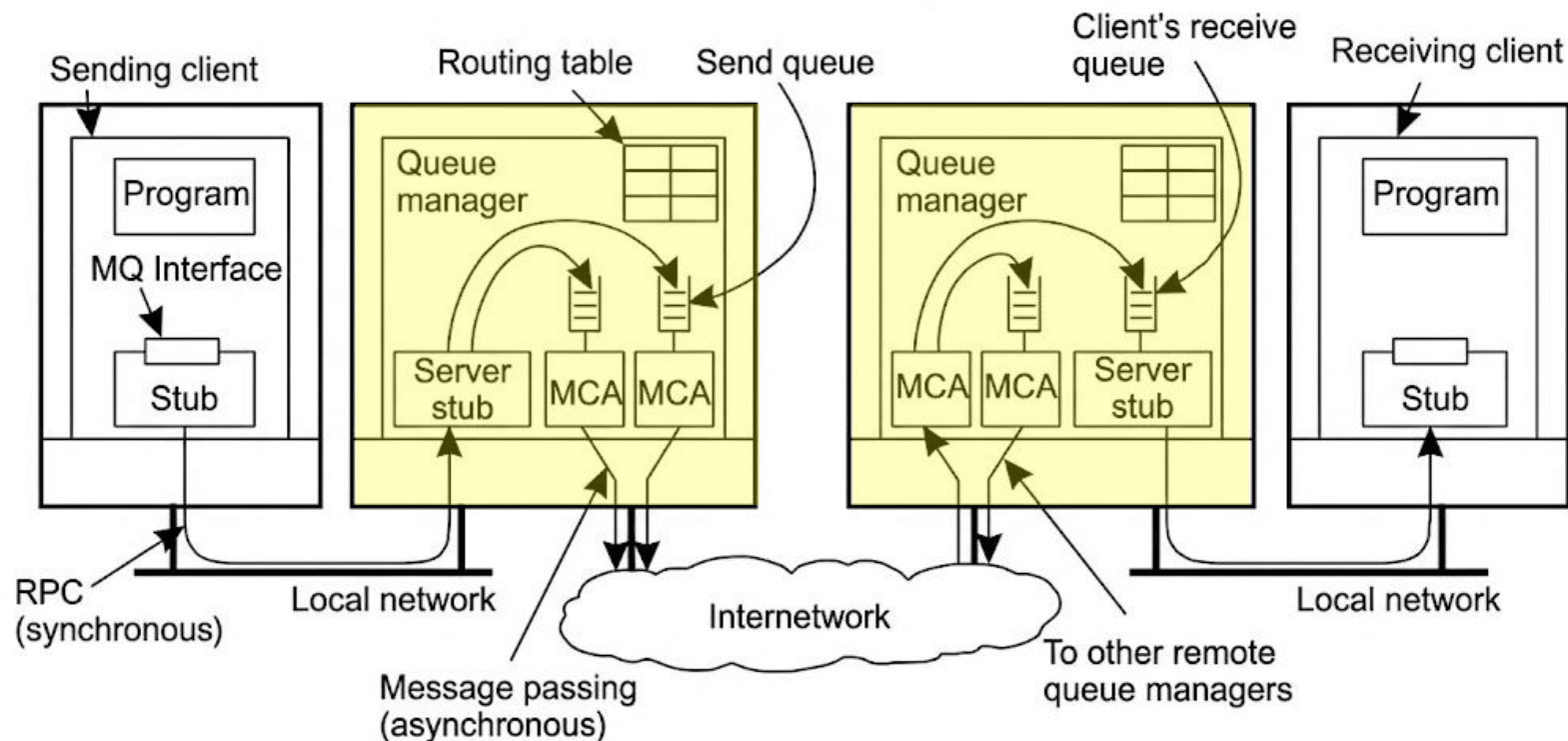
消息代理器 (Broker)

- 主要负责消息格式的转换
- 经常作为应用网关(gateway)
- 可提供基于主题的路由功能 (如发布-订阅功能)



举例：IBM WebSphere MQ

- 应用进程指定的消息可放入队列中或从队列中移除
- **队列管理器**：负责从发送队列取出消息，转发到其他QM
- 消息的缺省大小4MB；发送队列大小4GB



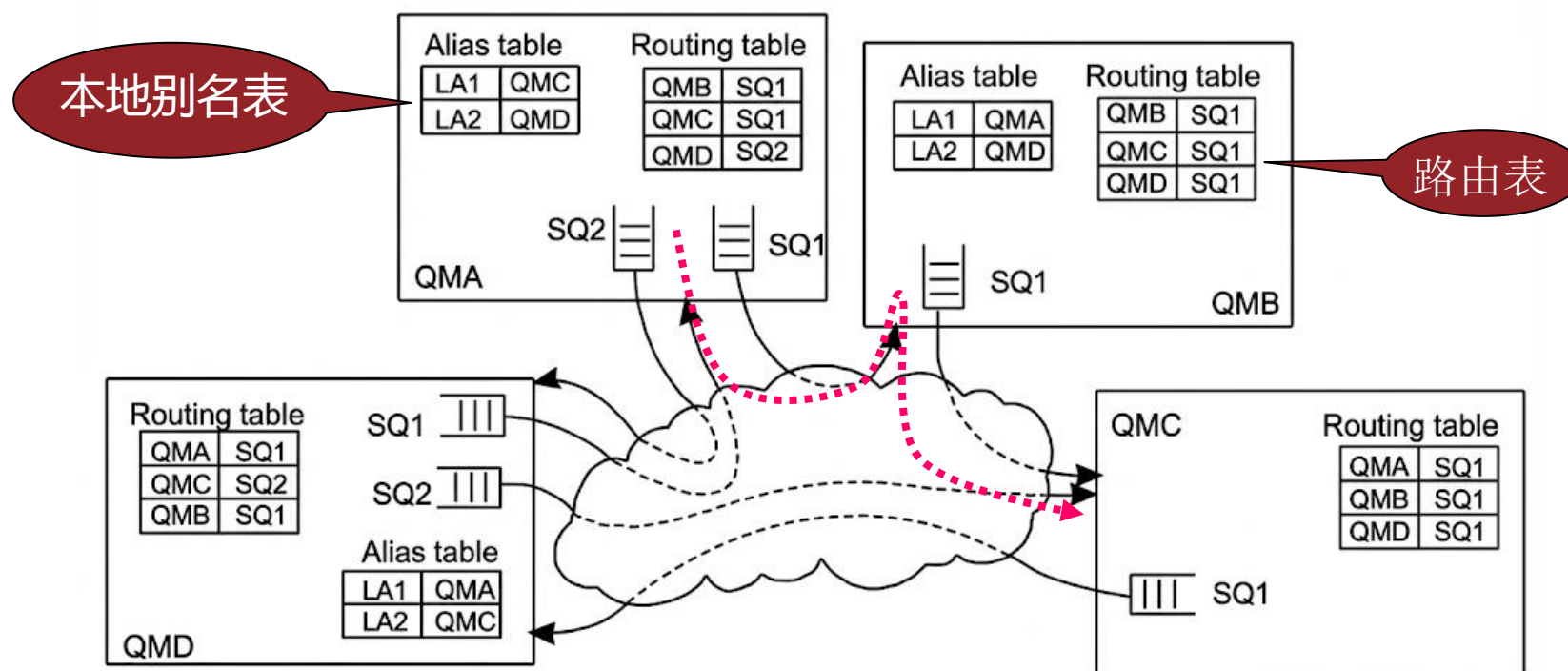
通道和通道代理

- 通道：实现发送方和接收方QM之间的单向的、可靠的连接
- 通道代理(MCA)：使用底层网络设施（如TCP/IP）建立通道；从传输层数据包中打包消息，或解包消息；发送/接受数据包
- 一个通道与一个发送队列对应
- 通过MCA设置配置通道的属性

属性	描述
Transport type	决定要采用的传输协议
FIFO delivery	表明消息将要按照发送的次序到达
Message length	单个消息的最大长度
Setup retry count	指定启动远程MCA的最大重试次数
Delivery retries	MCA将收到的消息放入队列的最大重试次数

消息传递的过程

- 路由：利用路由表（destQM, sendQ），对目的名进行名字解析，得到本地发送队列的名字，将待发送消息放到远程队列中
- 例：从QMA出发，LA1-QMC-SQ1-QMB-SQ1-QMC



第二章、分布式进程与通信

01

2.1 基本概念

02

2.2 分布式进程和线程设计

03

2.3 远程过程调用

04

2.4 面向消息的通信

05

2.5 *多播通信

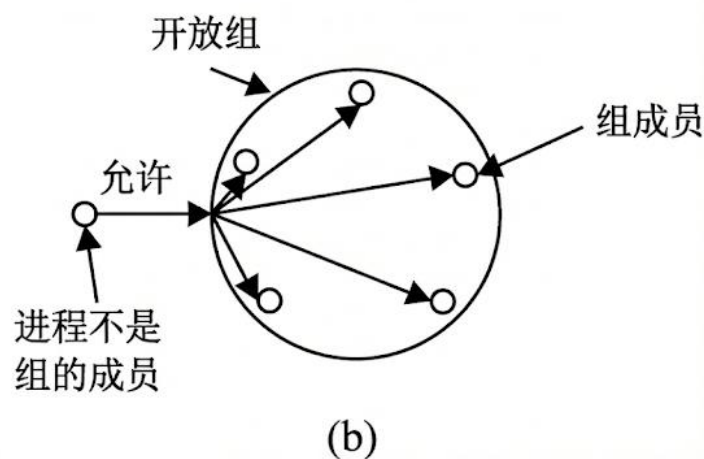
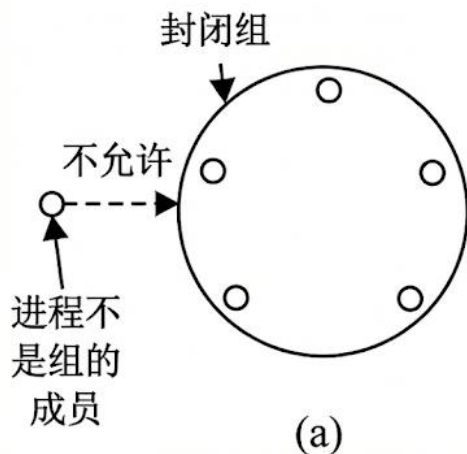


2.5 *多播通信

- 基本通信方式
 - 点到点通信 (point-to-point) : 单播 (unicast)
 - 一到多通信(one-to-many) : 多播 (multicast) 、广播 (broadcast)
- 多播需要保证原子性
 - 一个消息要么被所有进程接收, 要么一个进程都没有接收到

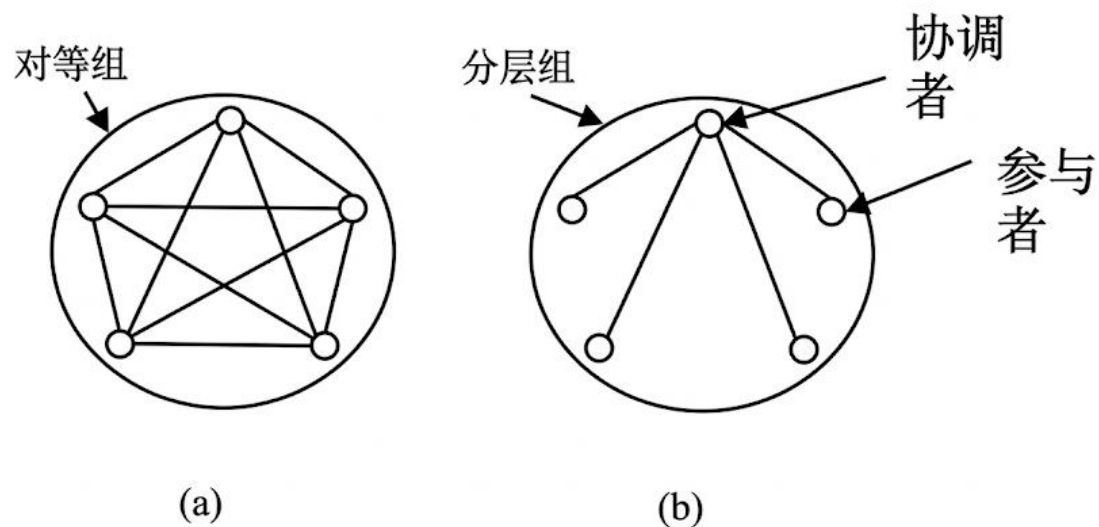
多播通信的设计

- 多播组(group):
 - 由系统或用户确定的若干个进程的集合
 - 组的成员籍 (membership) 管理
- 封闭组与开放组
 - 封闭组: 只允许组内成员之间多播
 - 开放组: 组外进程可以对组内进程进行多播



多播通信的设计

- 对等组与分层组
 - 对等组：成员之间的地位是平等的
 - 分层组：存在控制其他节点的超级节点，成为**协调者**



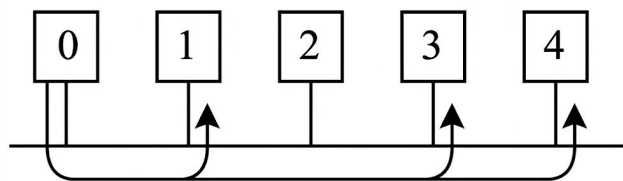
多播通信的设计

- 组的成员籍管理 (membership)
 - 集中式方法: 组服务器登记所有组成员
 - 分布式方法: 所有节点自己登记同组成员
- 组的加入/退出
 - 集中式方法: 通知组服务器
 - 分布式方法: 通知所有节点
- 组的寻址
 - 组地址: 唯一ID标识号
 - 目的地址列表: 包含所有成员的地址, 如IP+端口号
 - 谓词寻址: 关于成员地址的布尔表达式

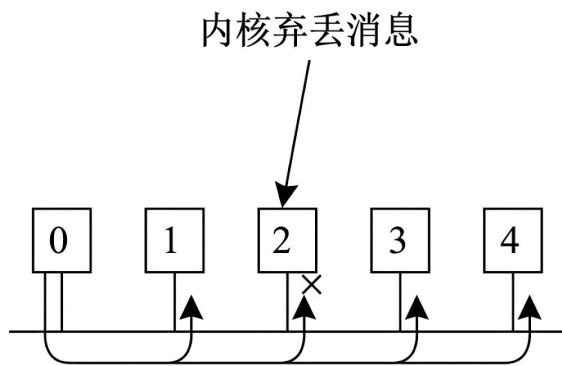
多播通信的设计

- 多播消息发送的实现方式

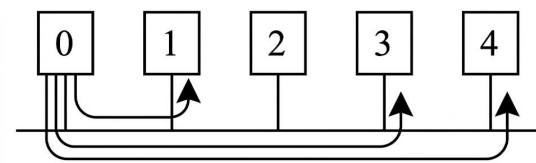
- 多播：一对多发送
- 广播：一对所有发送
- 单点：一对一发送



1 多播传送



2 广播传送



3 单点传送

多播通信的设计

- 消息的收发顺序

- 全局时间顺序

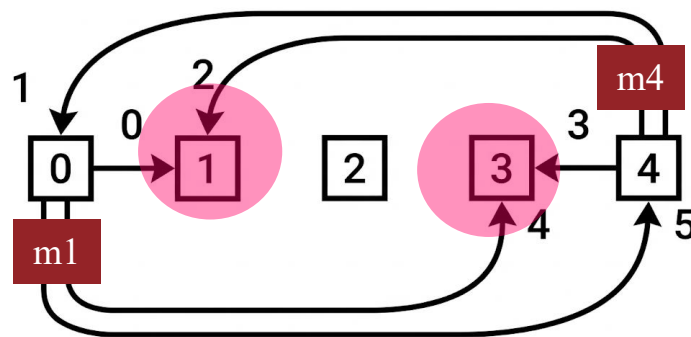
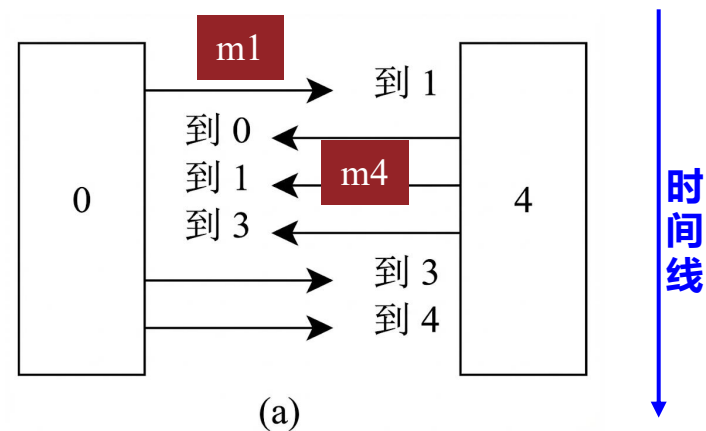
- If $ts(m1) < ts(m2)$, then $G(m1) < G(m2)$
 - $G(m)$ 表示组内所有成员接收到 m 的时间

- 一致时间顺序

- $G(m1) < G(m2)$ 或者 $G(m2) < G(m1)$

- 事件类型

- 因果关系：事件A的发生影响到事件B的发生
 - 并发关系：事件A的发生与事件B的发生无关



消息接收次序不一致

多播通信的设计

- 消息的收发顺序

- 全局时间顺序

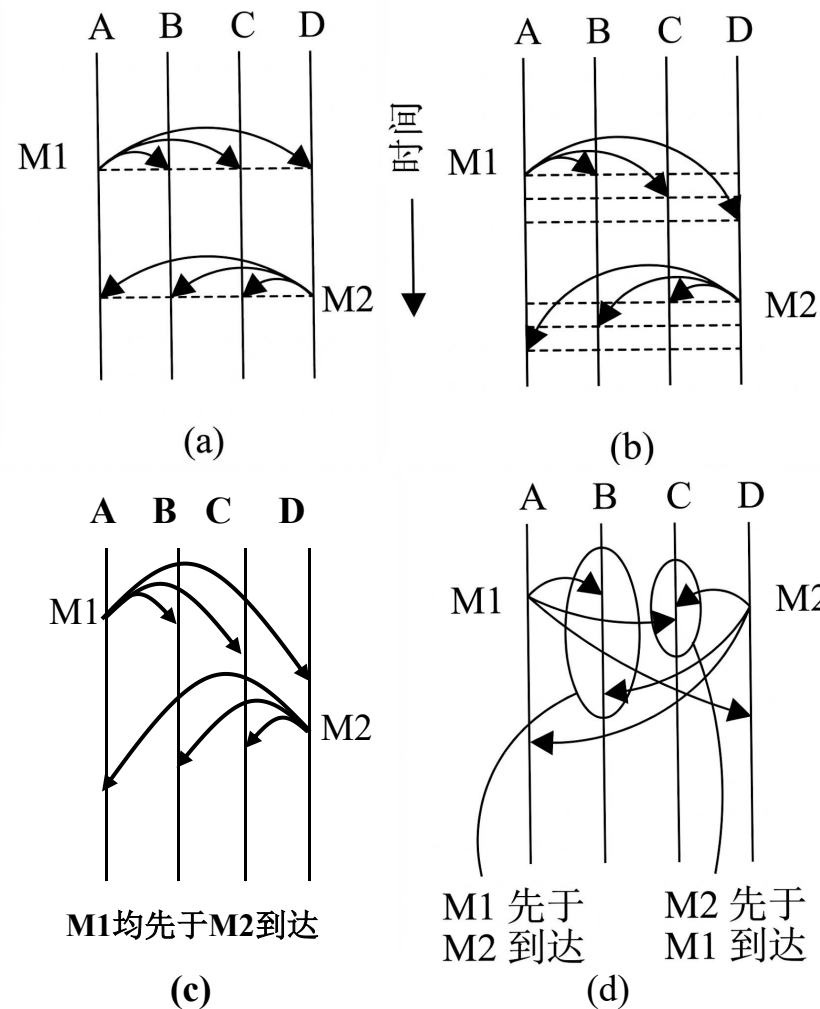
- If $ts(m1) < ts(m2)$, then $G(m1) < G(m2)$
 - $G(m)$ 表示组内所有成员接收到 m 的时间

- 一致时间顺序

- $G(m1) < G(m2)$ 或者 $G(m2) < G(m1)$

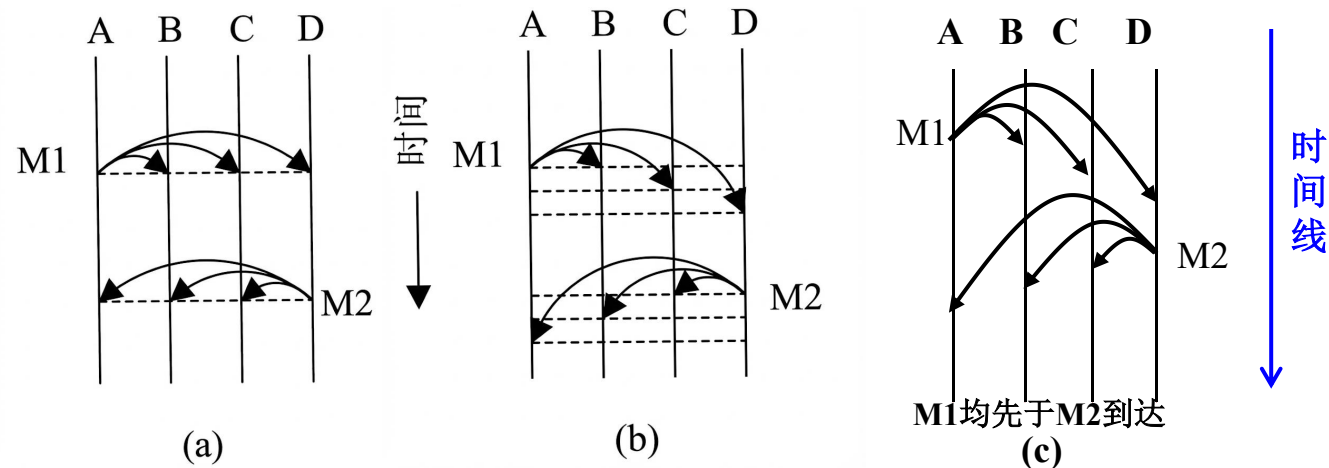
- 事件类型

- 因果关系：事件A的发生影响到事件B的发生
 - 并发关系：事件A的发生与事件B的发生无关



ISIS多播通信

- ISIS：康奈尔大学开发的组通信工具集
- 同步类型
 - A) 同步：同一事件在各处的发生严格遵照同一时间，即服从**全局时间顺序**)
 - B) 松散同步：事件在各处的发生可滞后一定的时间，但顺序保持一致，即服从**一致时间顺序**)
 - C) 虚拟同步：具有因果关系的事件必须服从同一次序，即服从**因果顺序**



ISIS多播通信

- ISIS中的广播原语

- ABCAST:

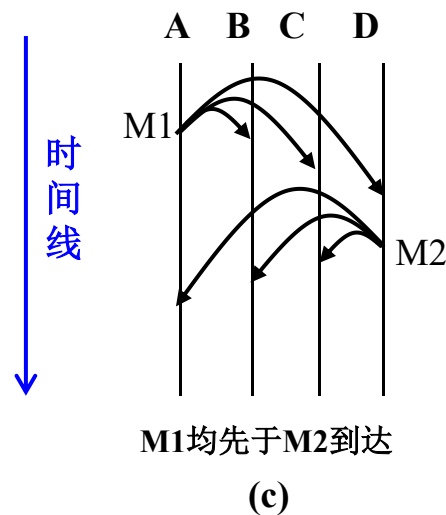
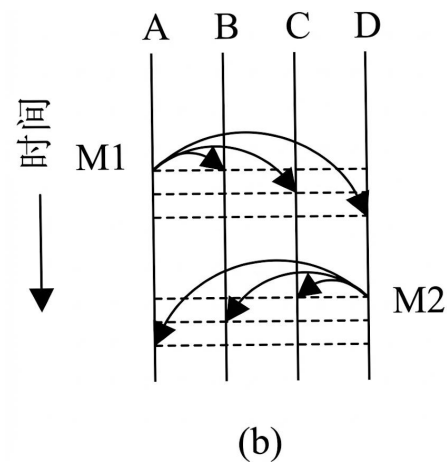
- 用于支持松散同步的分组通信
 - 采用两段提交协议 (2PC) 实现

- CBCAST:

- 用于支持虚拟同步的分组通信
 - 用时间戳向量实现

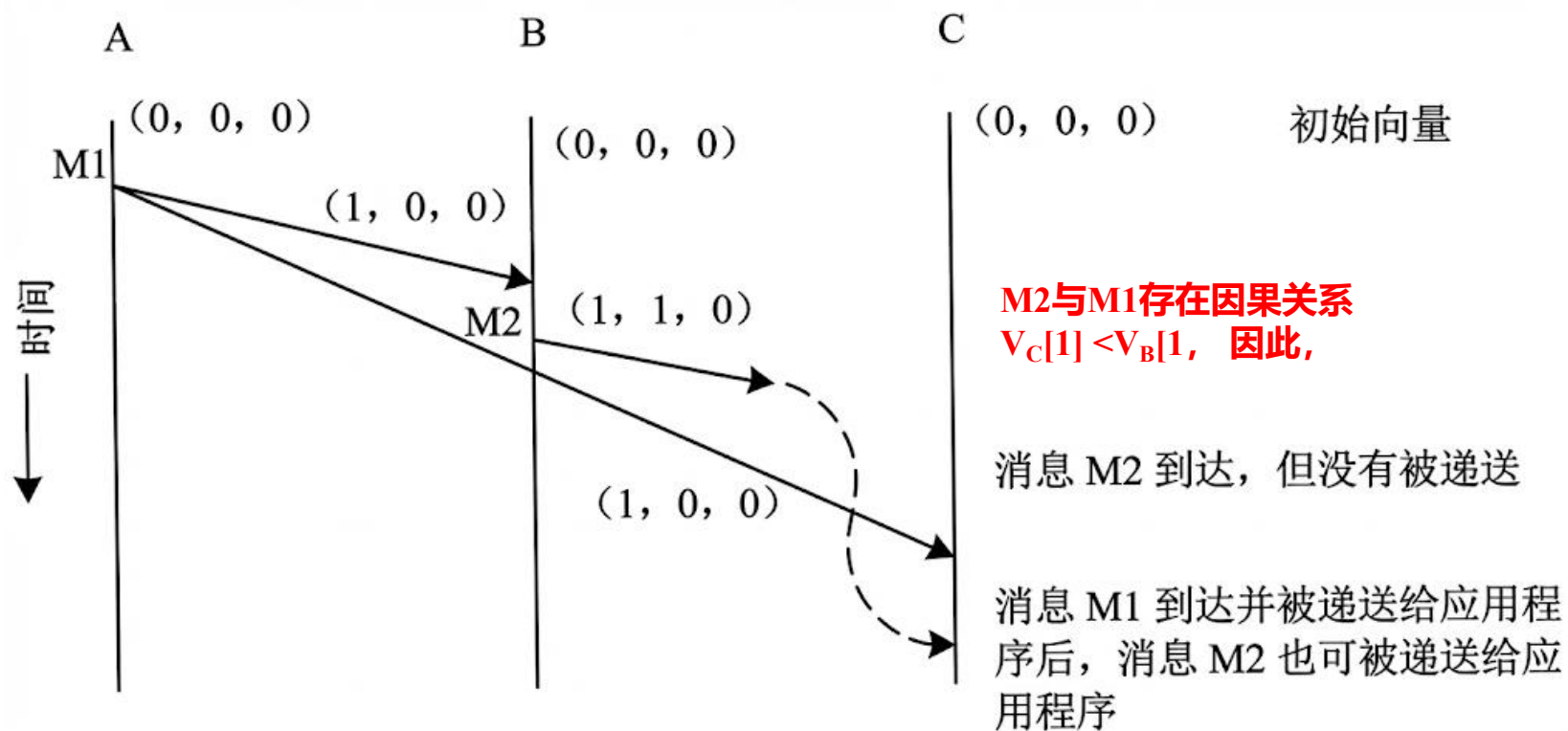
- GBCAST:

- 用于组成员管理的ABCAST通信



ISIS中的虚拟同步 (CBCAST)

- 例1: 3个进程



ISIS中的虚拟同步 (CBCAST)

- 时间戳向量
 - 设组内有 n 个进程，每个进程设置一个 n 元向量 $V[n] = \{V[1], V[2], \dots, V[n]\}$ 。 $V[i]$ 表示从进程 i 收到的最近消息的时间戳 ts
 - 进程当前发送的消息与 V 中记录的消息都存在因果关系
- 举例：有6个进程， $P1, P2, P3, P4, P5, P6$
 - 对于进程 $P1$ ，其下一个待发送的消息，与 $P2$ 的消息($ts=6$)， $P3$ 的消息($ts=8$)， $\dots$ ， $P6$ 的消息($ts=5$)都存在有因果关系

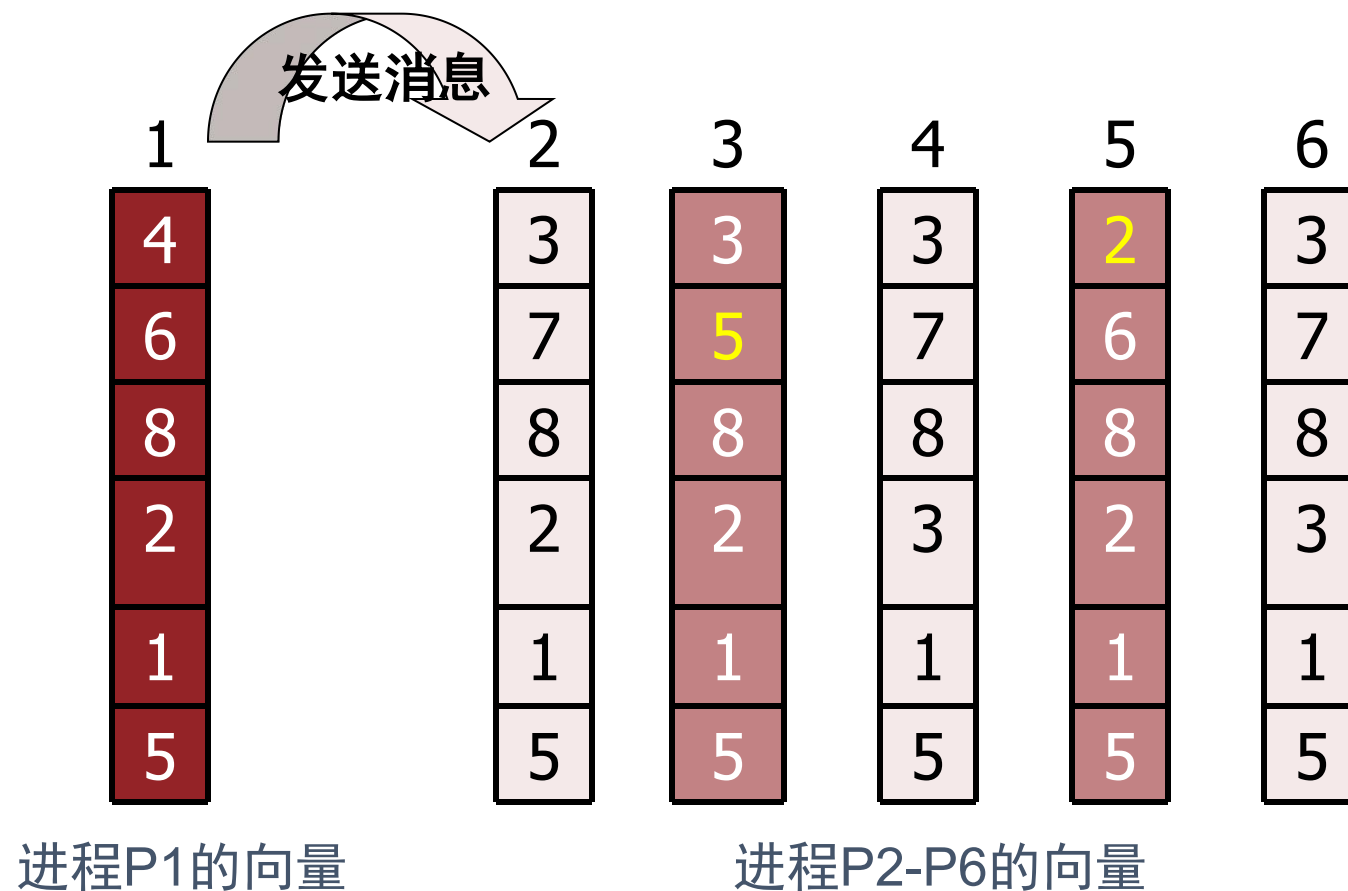
P1	P2	P3	P4	P5	P6
4	6	8	2	1	5

ISIS中的虚拟同步 (CBCAST)

- CBCAST算法
- 发送者 P_j (发送消息 m_j , 时间戳向量 V_j) :
 - 设置新的时间戳: $V_j[j] = V_j[j] + 1$; (对应进程 P_j)
 - 发送消息 m_j (携带时间戳向量 V_j)
- 接收者 P_k (收到消息 m_j , 时间戳向量 V_k) :
 - 判别接收条件 (是否满足因果关系) :
 - (1) $V_k[j] == V_j[j] - 1$; (已收到 P_j 的上一个消息)
 - (2) $V_k[i] \geq V_j[i]$ ($i \neq j$); (**i已收到 P_j 已收到的消息**)
 - 如果可以接收 P_j 的消息 m_j
修改向量: $V_k[j] = V_j[j]$;
 - 否则, 延迟接收消息 m_j

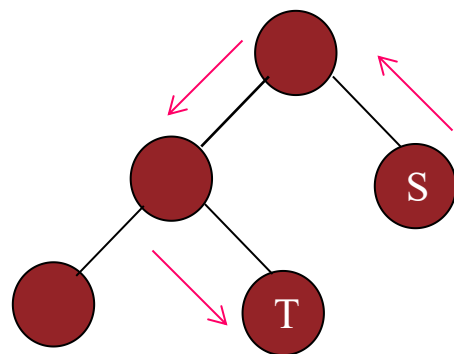
ISIS中的虚拟同步 (CBCAST)

- 例2：6个进程P1-P6

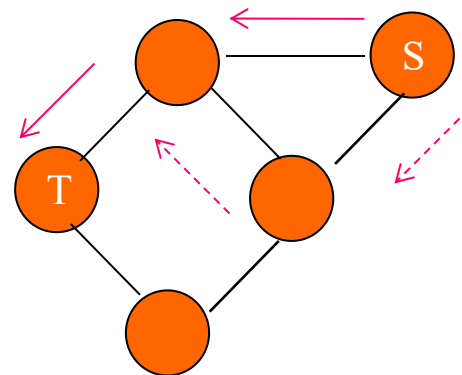


应用层多播(ALM)

- 将节点组成一个**覆盖网络**，用来实现数据分发
- 覆盖网络的拓扑结构
 - 树形覆盖网络：节点之间只有唯一通信路径
 - 网状覆盖网络：节点之间有多条通信路径



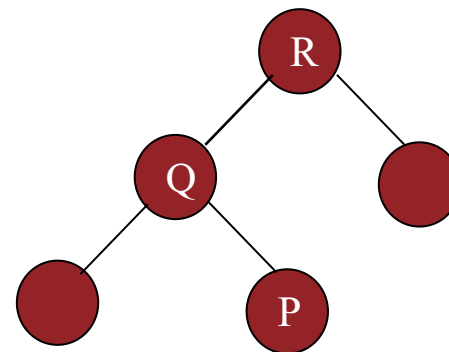
树形覆盖网络



网状覆盖网络

例：Chord中的应用层多播

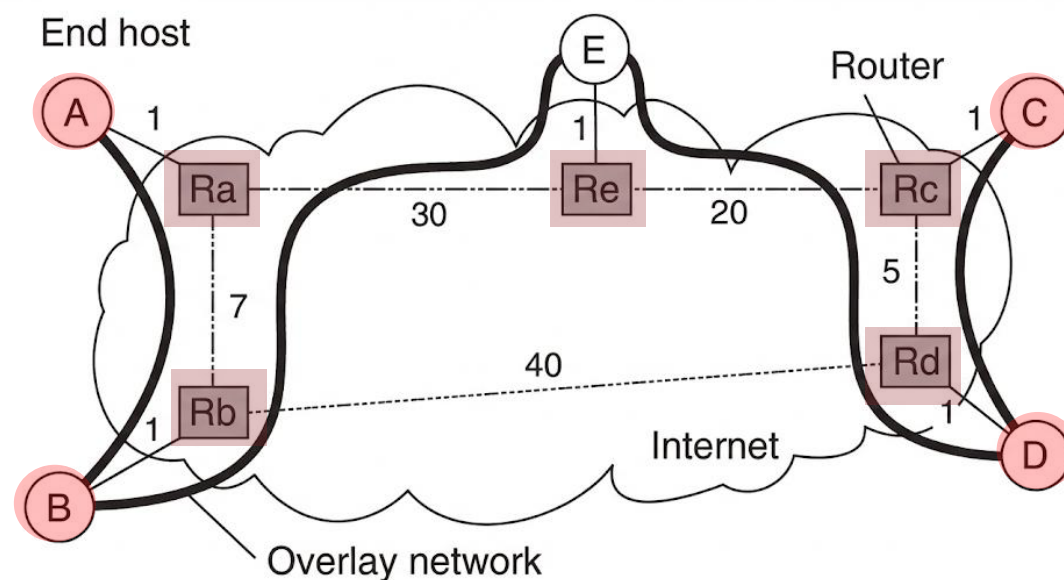
- Pastry构造多播树的基本方法
- 树的创建
 1. 发起者 (Initiator) 生成多播标识符 mid.
 2. 执行succ(mid), 找到负责mid的节点R.
 3. 将构造树请求经过路由到节点R, R变成根节点.
- 节点的加入
 1. 节点P 想要加入多播树, 发送加入请求到节点R
 2. 当请求到达节点Q:
 - 如果Q 没有收到加入请求, Q称为转发者; P成为Q的孩子节点。Q继续转发加入请求
 - Q 已知树的存在; P成为Q的孩子节点。不需转发



*覆盖网络的性能

• 覆盖网络的链接与实际网络拓扑的关系

- 例: $B \rightarrow C$
- 方案1 (覆盖网) : $B \rightarrow Rb \rightarrow Ra \rightarrow Re \rightarrow E \rightarrow Re \rightarrow Rc \rightarrow Rd \rightarrow D \rightarrow Rd \rightarrow Rc \rightarrow C$
(Re,Rc,Rd都经过2次)
- 方案2 (底层互联网) : $B \rightarrow Rb \rightarrow Rd \rightarrow Rc \rightarrow C$ (Ra,Rc只经过1次)

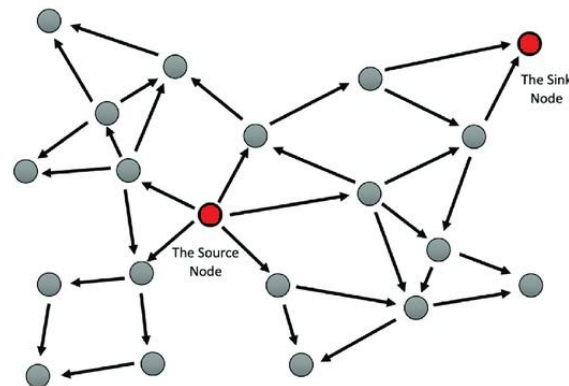


*覆盖网络的性能

- 链路应力(Link stress)
 - 计算每个物理链接上, 数据包通过的频率
- 伸张度(Stretch), 也称为相对延迟损失(RDP, Relative Delay Penalty)
 - 定义为两个节点之间的覆盖网络延时与底层网络延时的比例, 例如:
 - 覆盖网Cost (B-Rb-Ra-Rc-Rd-Rc-C) =73
 - 互联网Cost (B-Rb-Rd-Rc-C) =47。则有, $\text{stretch} = 73/47 = 1.55$
- 树代价(Tree Cost)
 - 所有链路上的开销总和
 - 最小化Tree Cost等价于找到一棵最小生成树, 使得信息传播到所有节点的总体时间最小

基于洪泛的多播通信

- 洪泛协议 (flood protocol)
 - 节点P将消息m发给它的每个邻居
 - 邻居节点Q如果从来没有收到过m, 则转发m到Q的每个邻居 (P节点除外)
- 性能: 覆盖网络中的边越多, 开销M越大
 - 如果G是一棵树, $M=N-1$
 - 如果全连通图, 则 $M = \frac{N*(N-1)}{2}$
- 优化策略:
 - 减少开销M, 即传播边的个数
 - 概率式洪泛协议



无结构P2P网络

$$\binom{N}{2}$$

基于洪泛的多播通信

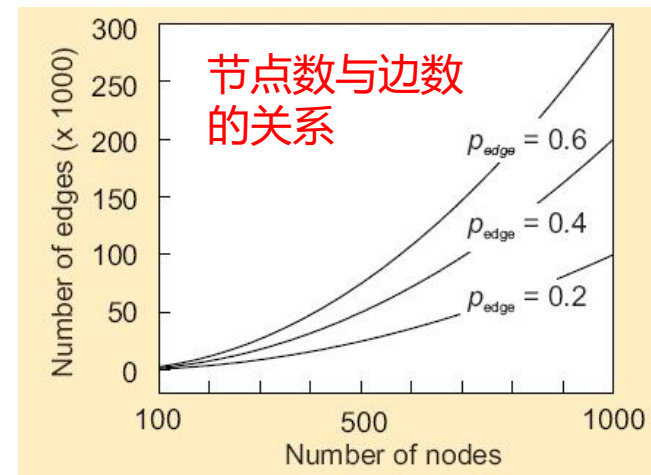
- 随机覆盖网络（随机图）

- 设 P_{edge} =两个节点存在边的概率
- 边的个数 S 是其节点个数 N 的函数

$$S = P_{edge} * N * (N-1) / 2$$

- 概率式洪泛协议

- 节点Q收到消息m后，以概率 P_{flood} 向其邻居转发消息m
- 性能：与全洪泛算法相比传播消息数量约减少到 P_{flood}
- 不足：如果 P_{flood} 太小，对邻居少的节点可能永远收不到消息。
- 改进方法：根据邻居个数，动态设置 P_{flood} 的大小



基于Gossip的数据传播分发

- 传染协议 (epidemic protocol)
 - 借用传染病 (epidemic) 行为描述数据传播 (传言, gossip), 使用局部信息在大规模节点中快速传播信息
 - 基于医学上成熟的传染病理论
- 节点状态:
 - **感染者**: 已得到数据, 并愿意参加传播
 - **易感者**: 还未得到数据
 - **隔离者**: 不参加传播
- 特点:
 - 没有**写-写冲突**: 规定更新操作只在一个节点上执行
 - 局部性: 副本只将其更新状态传递给几个邻居节点
 - 滞后性: 更新传播是滞后的, 不是立即的
 - 最终一致性: 每个更新都能到达它的所有副本



反熵传播模型

- 反熵
 - 熵用来描述混乱程度。反熵指将混乱的状态达成一致，即所有的节点得到相同的数据副本
- 反熵Gossip协议
 - 推(push)方法：节点P将自己的新数据推送给Q
 - 拉 (pull) 方法：节点P从Q中拉取新数据
 - 推-拉(push/pull)方法：P与Q相互发送新数据
- 推-拉方法的性能
 - 传播轮(round)：每一个节点都进行一次消息交换
 - 将更新消息分发到所有N个节点，需要执行 $O(\log(N))$ 轮

反熵传播模型

- 性能分析

- 设 p_i 为一个节点在第 i 轮仍未收到新数据的概率

- Pull策略:

$$p_{i+1} = (p_i)^2$$

- Push策略:

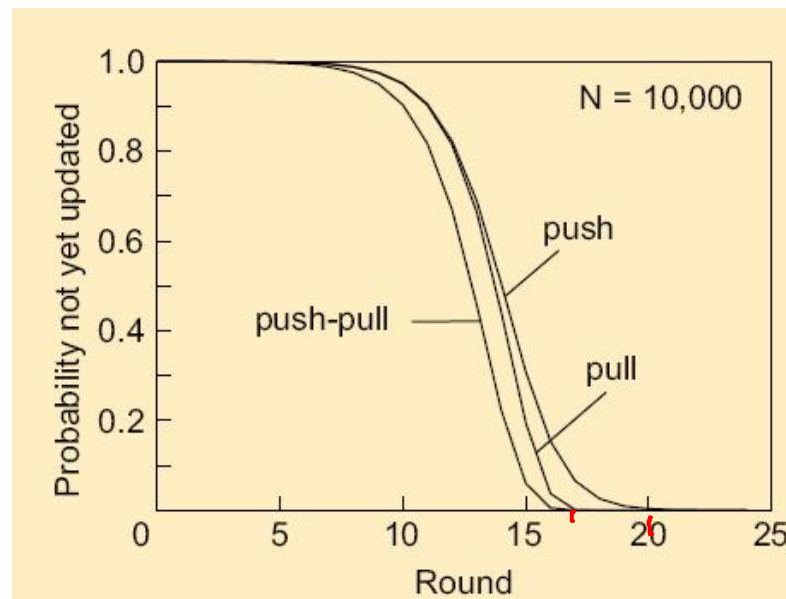
$$p_{i+1} = p_i \left(1 - \frac{1}{N-1}\right)^{N(1-p_i)}$$

$\approx p_i e^{-1}$ (当 p_i 很小 N 很大时)

- push-pull策略:

$$p_{i+1} = (p_i)^2 * p_i e^{-1}$$

- 如果节点都一直运行，反熵协议是极其有效的传播协议



*Rumor传播协议

- 传染协议的变种
 - 假设节点具有停止传播消息的概率 p_{stop}
- 基本算法
 - 如果节点P刚刚收到了数据x的更新，则随机选择一个节点Q将更新发送给Q
 - 如果P得知Q已经得到了这个新数据，则P退出传播（变为隔离节点），不再继续传播该消息
- 不足：有些节点可能永远收不到消息
 - 假设P不再传播的概率 p_{stop} ，覆盖网络中仍为易感的节点比值为 $s = e^{-\left(\frac{1}{p_{stop}} + 1\right)(1-s)}$
 - 例： $p_{stop} = 0.25$, 则有 $s < 0.007$, 即0.7%未被感染

*Rumor传播协议

- Rumor性能分析

- 设 s 为未被更新的比值（易感节点）， i 为已更新且活跃的比值（感染节点）， r 为已更新但放弃传播的比值（隔离节点）。 $s + i + r = 1$

- 传染病理论公式

$$(1) \quad ds/dt = -s \cdot i$$

$$(2) \quad di/dt = s \cdot i - p_{stop} \cdot (1 - s) \cdot i$$

$$\Rightarrow \quad di/ds = -(1 + p_{stop}) + \frac{p_{stop}}{s}$$

$$\Rightarrow \quad i(s) = -(1 + p_{stop}) \cdot s + p_{stop} \cdot \ln(s) + C$$

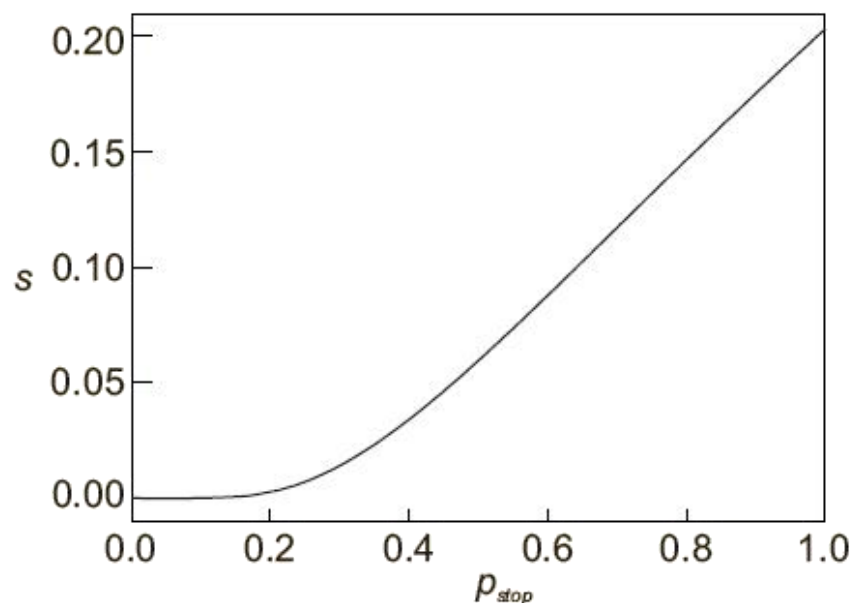
$$i(1) = 0 \Rightarrow C = 1 + p_{stop} \Rightarrow i(s) = (1 + p_{stop}) \cdot (1 - s) + p_{stop} \cdot \ln(s).$$

当达到不再有传播时, $i(s)=0$

$$s = e^{-\left(\frac{1}{p_{stop}} + 1\right)(1-s)}$$

*Rumor传播协议

- P_{stop} 停止概率的影响
 - 如果真正保证所有的节点被更新，基于谣言的传播是做不到的
 - 设 s 为未被更新的节点比值， N_s 为对应的节点个数



Consider 10,000 nodes		
$1/p_{stop}$	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

“删除数据”问题

- 通过传播“删除数据”信息，效果不好
 - 原因：删除一个数据时删除了所有有关信息，当数据在一个节点上被删除后，由于通信滞后性，可能依然会收到旧版本数据，又当做新数据被接收
- 解决方案
 - 将“数据删除”作为一个版本更新来传播——死亡证书。但死亡证书又带来占用空间问题
- 死亡证书的两种清理方法
 1. 运行一个全局算法，检测该删除操作是否已全部完成，如果是，则回收死亡证书
 2. 给死亡证书加一个时间范围。过期后，自动清除

通信管理小结

- 分层通信协议
 - ISO协议、中间件协议、通信类型
- 远程过程调用
 - RPC步骤、参数传递、扩展RPC
- 面向消息的通信
 - Socket、MPI、消息队列系统
- 多播通信
 - 组管理、虚拟同步、洪泛式协议、gossip传播协议

第二章 完