



# 分布式系统

## Distributed Systems

陈鹏飞

数据科学与计算机学院

[chchenpf7@mail.sysu.edu.cn](mailto:chchenpf7@mail.sysu.edu.cn)

办公室：超算5楼529d

主页：<http://sdcs.sysu.edu.cn/node/3747>



中山大學

SUN YAT-SEN UNIVERSITY

数据科学与计算机学院

School of Data and Computer Science



## 第三讲 — 分布式中的进程和虚拟化



# 大纲

1

分布式系统中的线程

2

虚拟化

3

客户—服务典型结构

4

计算迁移

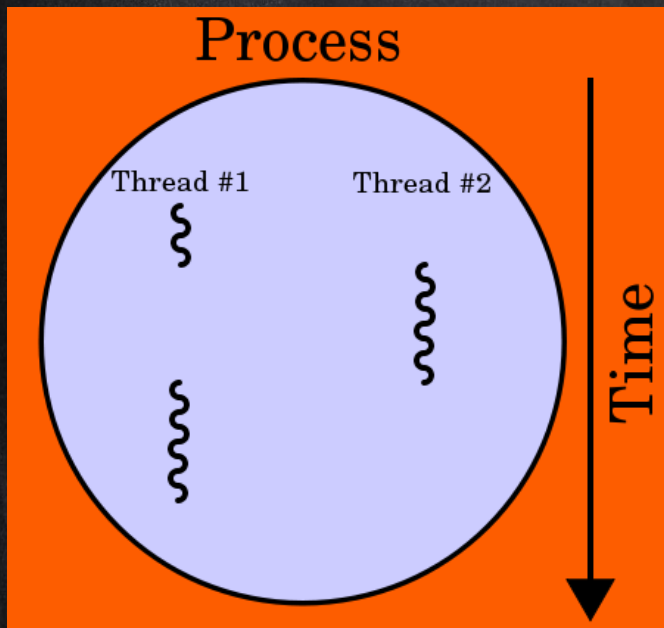




## 分布式系统中的线程



# 进程



- 何为进程？
- 何为线程？
- 操作系统如何管理进程？

初始化、分配资源、调度、销毁



# 分布式系统中的线程

## ➤ 基本思想：

在物理处理器上用软件创建虚拟处理器：

- ❑ 处理器： 提供和运行一系列指令集合的硬件平台；
- ❑ 线程： 一个最小的可执行一系列指令的软件处理器。 保存线程的上下文意味着终止线程当前的执行，在其他时刻装载保存的线程上下文后，线程可以继续执行；
- ❑ 进程： 包含多个线程的软件处理器，线程需要在进程的上下文中执行。



## 上下文切换

### ➤ 上下文

系统运行过程中的一系列状态，状态的含义因系统的不同而不同

#### □ 处理器上下文

处理器用于运行一系列指令的保存在寄存器中的**最小**数据集合（如：栈指针、地址寄存器、程序计数器）；

#### □ 线程上下文

用于执行一系列指令的保存在寄存器和内存中的**最小**的数据集合（如：处理器上下文、状态等）；

#### □ 进程上下文

用于执行线程的保存在寄存器和内存中的**最小**的数据集合（线程上下文、MMU寄存器值、TLB）





## 上下文切换

### ➤ 观察发现

- ❑ 线程共享相同的地址空间。线程上下文的切换可以独立于操作系统；
- ❑ 一般来讲进程之间的切换要更复杂、代价更高，因为需要陷入到OS内核才能完成；
- ❑ 创建和销毁线程的代价要远远小于对进程的创建和销毁；





# 为什么要利用线程

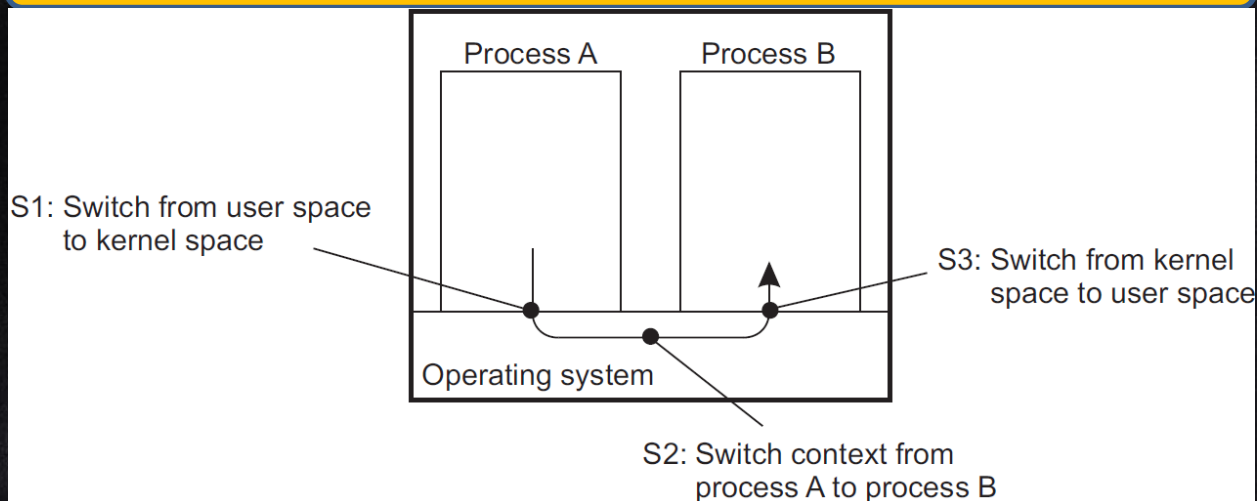
## 主要的原因：

- ❑ 避免不必要的阻塞：单线程的进程在进行I/O操作的时候会被阻塞；在多线程的进程中，操作系统可以将CPU切换到进程的另外一个线程；
- ❑ 更好地发挥并行性：一个具有多线程的进程可以在多核或者多处理器的CPU上并行执行；
- ❑ 避免进程上下文切换：架构大型应用的时候不是利用多个进程而是多个线程；



## 避免进程切换

### ➤ 避免昂贵的上下文切换



- ❑ 多个线程利用相同的地址空间：更容易出错；
- ❑ 在线程使用内存时，没有来自OS/HW的保护；
- ❑ 线程的上下文切换比进程的上下文切换要快得多；

平衡

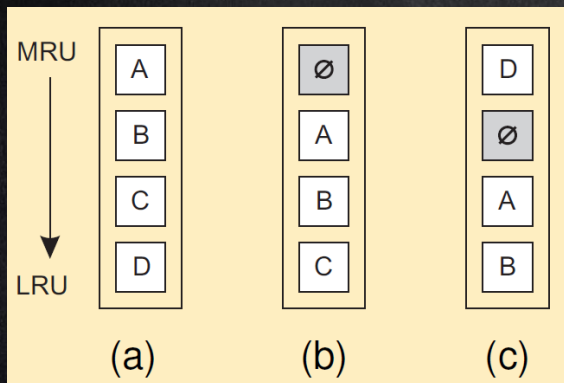


## 上下文切换的代价

### 考虑简单的时钟中断处理器：

- ❑ 直接代价：用于实际切换和执行中断处理代码的时间；
- ❑ 间接代价：其他代价，比较常见的是cache刷新的时间；

### 上下文切换导致的间接代价：



- (a) 在上下文切换之前；
- (b) 在上下文切换之后；
- (c) 在访问区块D之后；

代价高达 80%



## 线程和操作系统

### ➤ 内核解决方案

基本的想法是在内核中实现软件包，这也就意味着所有的操作变成系统调用；

- ❑ 用于阻塞线程的操作就不再是问题了，内核会在同一个进程中调度另外一个可用的线程；
- ❑ 处理外部事件也变得简单了：内核（捕捉所有的事件）直接调度与线程相关的事件；
- ❑ 存在的问题是效率，因为每个线程操作都需要陷入到内核；

### ➤ 结论

尽可能融合用户级别和内核级别的线程，发挥各自的优势。但是，事实上这种做法带来的性能提升也难以弥补其复杂性带来的困难。





# 线程的实现

## ➤ 线程的实现方式

- ❑ 往往以线程包的形式存在；
- ❑ 完全在用户空间下创建线程库；
- ❑ 由内核来掌管线程并进行调度；

## ➤ 用户级线程

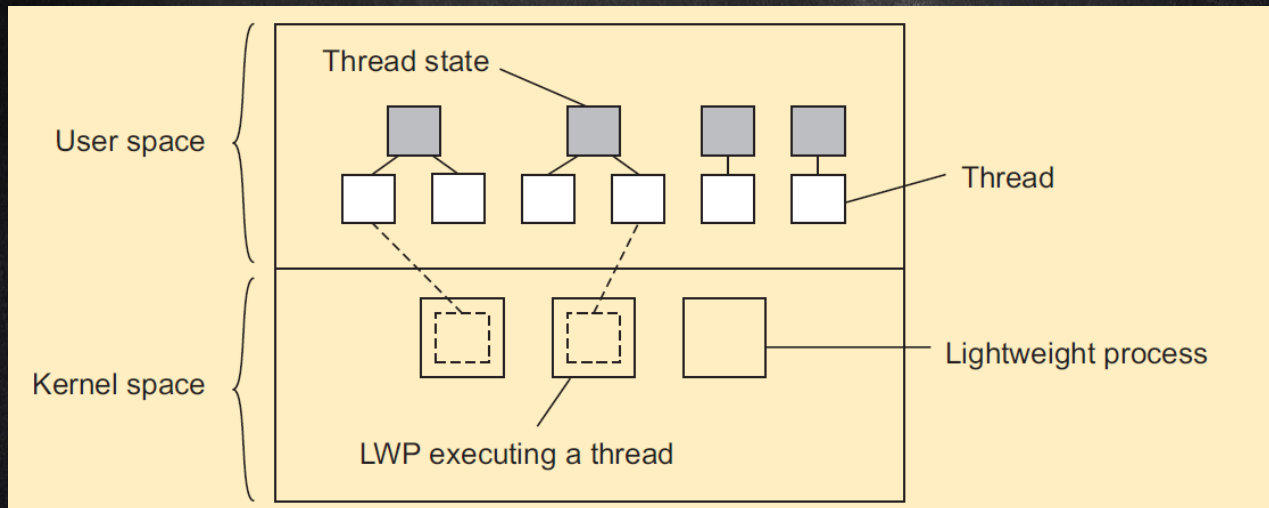
- ❑ 好处：上下文切换代价小；
- ❑ 缺陷：线程阻塞 -> 进程阻塞；



## 轻量级进程 (LWP)

### ➤ 基本思想

引入一种两层线程方法：轻量级的进程 (LWP) 能够执行用户空间的线程：





## 轻量级进程 (LWP)

### ➤ 主要的操作

- ❑ 用户级的线程执行系统调用 => LWP执行相应的操作，同时该线程被挂起。线程维持绑定在该LWP上；
- ❑ 内核调度负责调度另外一个LWP，该LWP绑定了一个活跃线程。注意，这个线程可以切换到用户空间中的另外的活跃线程；
- ❑ 一个线程调用阻塞用户层的操作 => 上下文切换到一个运行的线程，并绑定到相同的LWP；
- ❑ 当没有线程调用时， LWP 保持空闲甚至有可能被内核清除、回收；

概念被废弃!!!





## 在客户端使用多线程

- 多线程在分布式系统中的重要意义
- 多线程的Web客户端

隐藏了网络的延迟；

- ❑ Web浏览器扫描到达的HTML页面，发现需要获取更多页面；
- ❑ 每一个页面由一个特定的线程获取，每个线程执行HTTP请求；
- ❑ 随着文件的到达，浏览器将这些文件展示出来；

### ➤ 服务器之间多个请求-响应调用

- ❑ 客户端同时产生多个调用，每一个线程负责一个调用；
- ❑ 之后，客户端等待结果返回
- ❑ 注意： 如果调用的不同的服务器，将会得到线性加速；





## 多线程客户端是否有用？

### Thread-level parallelism: TLP

Let  $c_i$  denote the fraction of time that exactly  $i$  threads are being executed simultaneously.

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

with  $N$  the maximum number of threads that (can) execute at the same time.

### Practical measurements

A typical Web browser has a TLP value between 1.5 and 2.5  $\Rightarrow$  threads are primarily used for **logically organizing** browsers.



## 多线程服务器

### ➤ 提高性能

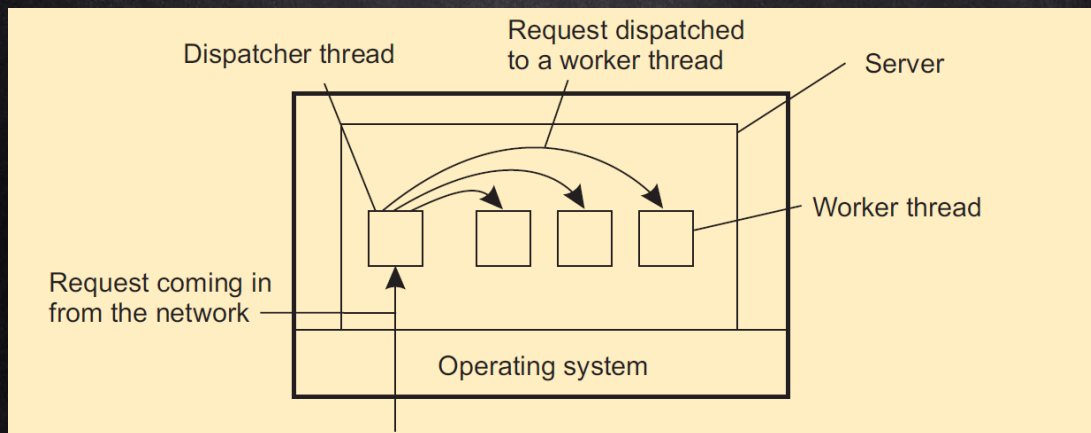
- ❑ 启动一个线程远比启动一个进程代价要小很多；
- ❑ 单线程服务很难再多处理器系统中实现 scale-up（纵向扩展）；
- ❑ 与多线程客户端相呼应：通过并行响应请求隐藏网络延迟；

### ➤ 更好的结构

- ❑ 大部分服务器都具有较高的I/O需求，使用简单容易理解的阻塞调用可以简化整体结构；
- ❑ 多线程的程序的代码数量较少，比较容易理解，因为控制流被简化了；



## 多线程服务器举例：文件服务器



模型	特征
多线程	并行，使用会导致阻塞的系统调用
单线程进程	非并行，使用会导致阻塞的系统调用
有限状态机	并行，使用非阻塞系统调用

图 3.4 构建服务器的 3 种方式



# 虚拟化 (Virtualization)

## ➤ 观察发现

虚拟化（软硬件的多路复用）非常重要：

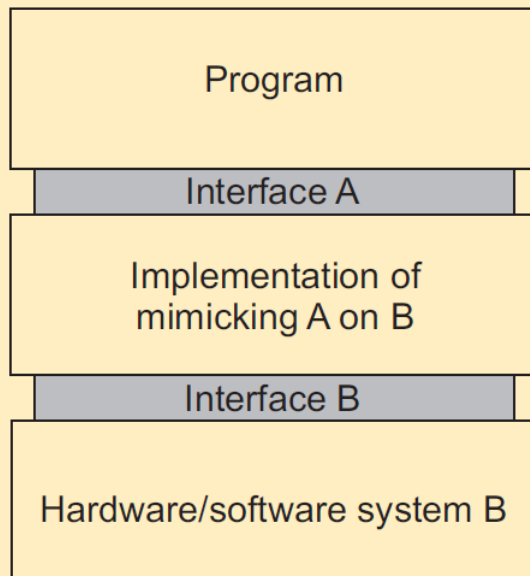
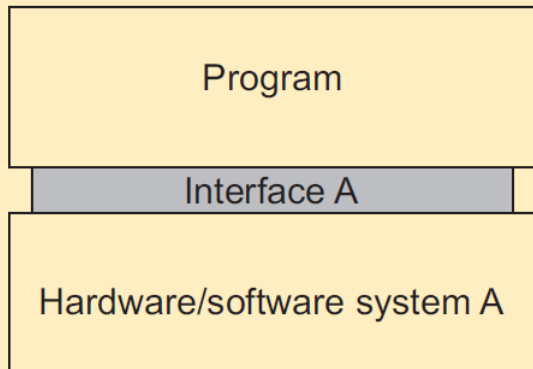
- ❑ 硬件比软件变化的快；
- ❑ 需要灵活的可移植性和代码迁移；
- ❑ 失效和攻击隔离





# 虚拟化 (Virtualization)

## ➤ 主要原理：模拟接口





# 模拟接口

## ➤ 三个层次上的四种类型的接口

□ **指令集架构**：一系列的机器指令，主要分为两类：

- a. 特权指令：允许操作系统执行的执行；
- b. 通用指令：可以被任何程序执行的指令；

□ **系统调用**

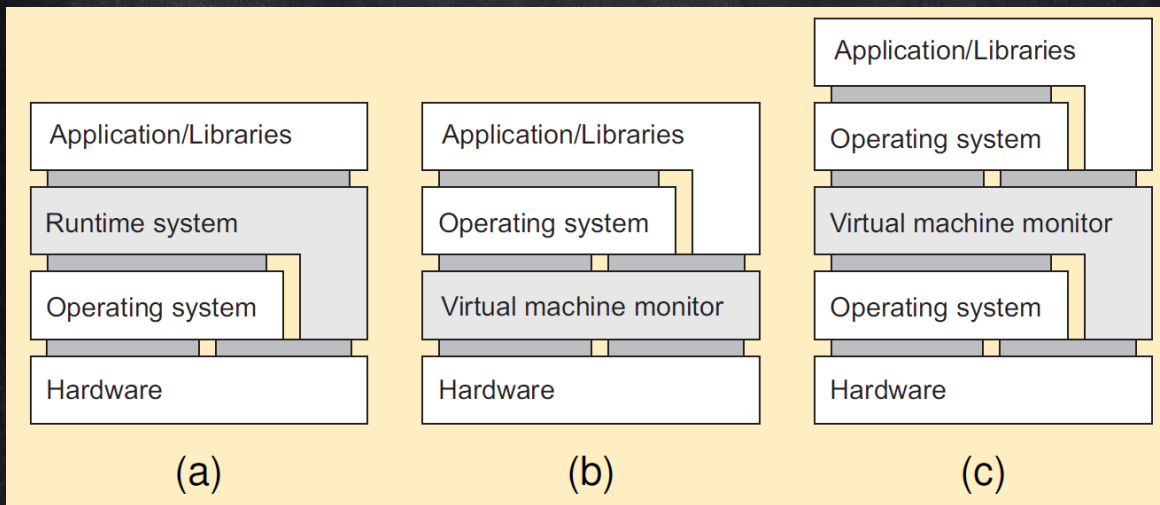
由操作系统提供的函数；

□ **库函数调用**，也成为**应用程序接口（API）**



## 虚拟化的不同方式

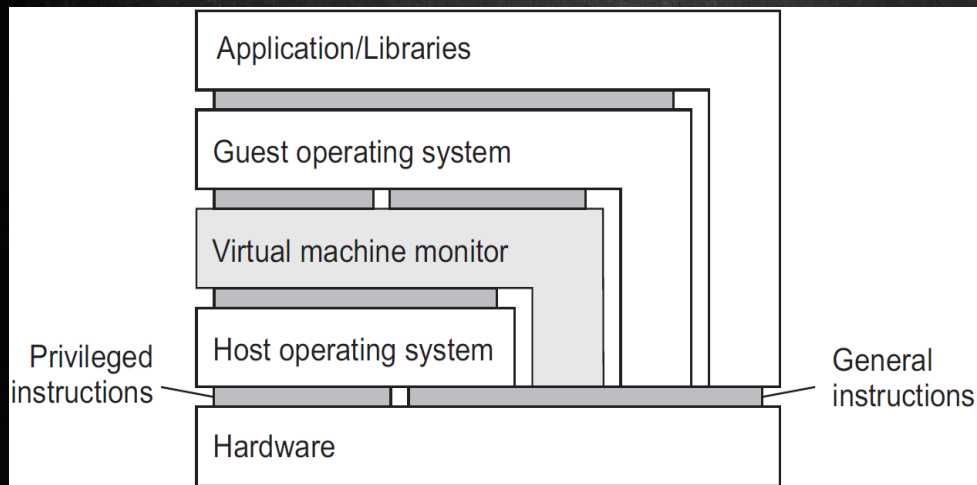
(a) 进程虚拟机 (Process VM)、(b) 原生虚拟机监控器 (Native VM M)、(b) 主机虚拟机监控器 (Hosted VMM)



区别: (a) 分离的指令集合, 实际是运行在操作系统之上的解释器 (J VM) 或者模拟器 (Qemu); (b) 底层的指令, 同时具有跑在硬件上的最小的操作系统; (c) 底层指令, 但是需要一个完整的OS;



## 进一步了解虚拟机：性能



**特权指令：**当且仅当在用户模式下执行时，引起操作系统陷入；

**非特权指令：**其他所有指令

### 特殊指令：

控制敏感性指令：可能影响到机器配置的指令（如：寄存器重定位或者中断表）

行为敏感性指令：指令效果由上下文确定；





## 虚拟化的条件

### ➤ 采用虚拟化的必要条件

对于任何通用计算机，如果敏感指令集是特权指令的子集，则可以构建虚拟机监控器；

### ➤ 问题：条件并不总是被满足

存在这样的敏感指令集，这些指令集在用户空间执行，但是不会引起操作系统的陷入；

### ➤ 解决方案：

模拟所有指令；

包装非特权敏感指令，将其交给VMM控制执行；

半参数化（Paravirtualization）：修改客户OS，要么阻止非特权的敏感指令，或者将其变为非敏感指令（也就是改变上下文）；



## 客户-服务器之间的交互

客户机器主要是让个人用户和远程服务器交互。主要包含两种模式：1、对于每种远程服务，客户机都有一个独立的网络模块联系这些服务；2、通过一个方便统一的用户接口来对远程服务直接访问；

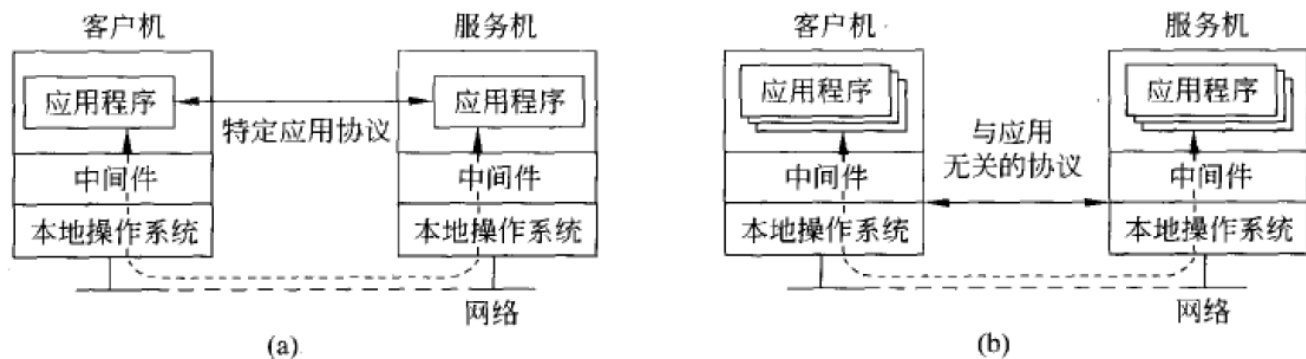


图 3.8

(a) 带自用协议的网络连接的应用程序；(b) 允许访问远程应用程序的通用解决方案



### 样例：X windows系统

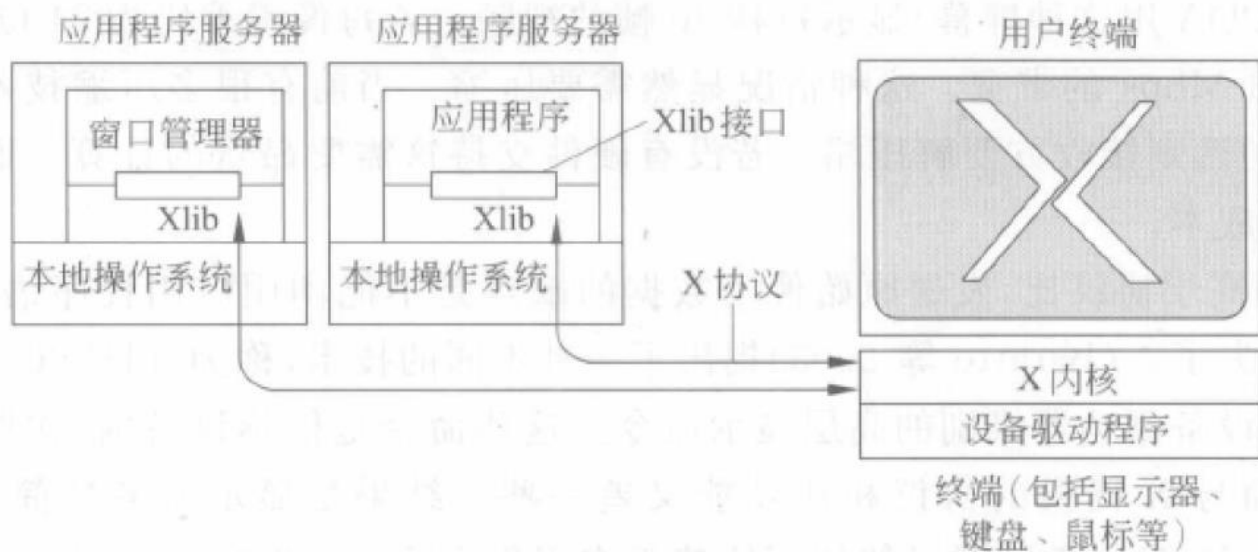


图 3.9 X Window 系统基本组织结构



## 提高X Windows系统性能

### ➤ 实际观察

- ❑ 应用程序的逻辑和用户接口之间没有清晰的界限；
- ❑ 应用程序与X内核之间的交互倾向于采用同步模式；

### ➤ 改进方法

- ❑ 应用程序完全控制显示，可以到像素级别（VNC）；
- ❑ 基金提供几种高层次的显示操作；

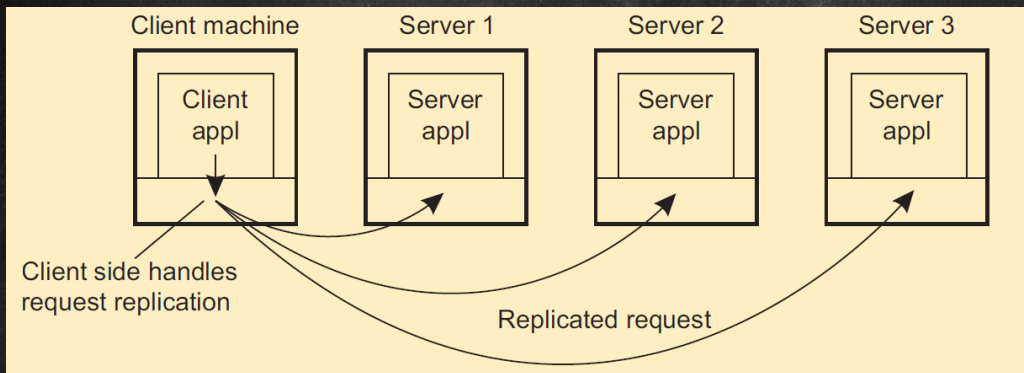




## 客户端软件

客户端软件包含用于获得分布式透明性的组件

- ❑ 访问透明性：客户端拥有用于 RPC 访问的存根；
- ❑ 位置/迁移的透明性：让客户端记录服务器的实际位置；
- ❑ 副本透明性：客户端存根多个副本的调用



- ❑ 故障透明性：经常放在客户端（用于屏蔽服务器和通信问题）



## 服务器软件：常见设计问题

### ➤ 基本模型：

服务器是实现特定服务的进程，这些服务是为一组客户提供的。本质上，每个服务器的组织方式都一样，等待来自客户的请求，随后负责处理该请求。



# 服务器的组织方式

## ➤ 两种基本类型

- ❑ 迭代服务器 (Iterative server)：服务器顺序处理到达的请求；
- ❑ 并发服务器 (Concurrent server)：利用分派器 (dispatcher)，选择到达的请求并将它传递给分离的线程或者进程处理；

## ➤ 观察发现

并发服务器是一种常见的类型：他们能够处理多个请求，特别是针对某些阻塞性的操作；





## 服务器连接

```

EtherNet/IP-2 44818/tcp EtherNet-IP-2 # EtherNet/IP messaging
EtherNet/IP-2 44818/udp EtherNet-IP-2 # EtherNet/IP messaging
m3da 44900/tcp # M3DA (efficient machine-to-machine com
munication)
m3da-disc 44900/udp # M3DA Discovery (efficient machine-to-m
achine communication)
amp 45000/tcp # NSI AutoStore Status Monitoring Protoc
ol data transfer
amp-mon 45000/udp # NSI AutoStore Status Monitoring Protoc
ol device monitoring
amps 45001/tcp # NSI AutoStore Status Monitoring Protoc
ol secure data transfer
synctest 45045/tcp # Remote application control
invision-ag 45054/tcp # InVision AG
invision-ag 45054/udp # InVision AG
eba 45678/tcp # EBA FRISE
eba 45678/udp # EBA FRISE
dai-shell 45824/tcp # Server for the DAI family of client-se
rver products
qdb2service 45825/tcp # Qpnature Data Access Service
qdb2service 45825/udp # Qpnature Data Access Service
ssr-servermgr 45966/tcp # SSRServerMgr
ssr-servermgr 45966/udp # SSRServerMgr
sp-remotetablet 46998/tcp # connection between computer and a sign
ature tablet
mediabox 46999/tcp # MediaBox Server
mediabox 46999/udp # MediaBox Server
mbus 47000/tcp # Message Bus
mbus 47000/udp # Message Bus
winrm 47001/tcp # Windows Remote Management Service
jwl-mactalk 47100/udp # Configuration of motors connected to in
dustrial ethernet
dbbrowse 47557/tcp # Database Corporation
dbbrowse 47557/udp # Database Corporation
directplayervr 47624/tcp # Direct Play Server
directplayervr 47624/udp # Direct Play Server
ap 47806/tcp # ALC Protocol
ap 47806/udp # ALC Protocol
bacnet 47808/tcp # Building Automation and Control Networ
k
bacnet 47808/udp # Building Automation and Control Networ
k
nimbuscontroller 48000/tcp # Nimbus Controller
nimbuscontroller 48000/udp # Nimbus Controller
nimbuspooler 48001/tcp # Nimbus Spooler
nimbuspooler 48001/udp # Nimbus Spooler
nimhub 48002/tcp # Nimbus Hub
nimhub 48002/udp # Nimbus Hub
nimgtw 48003/tcp # Nimbus Gateway
nimgtw 48003/udp # Nimbus Gateway

```

```

tcp 0 0 127.0.0.1:43984 127.0.0.1:2379 ESTABLISHED
tcp 0 0 127.0.0.1:10252 127.0.0.1:57092 TIME WAIT
tcp 0 0 127.0.0.1:2379 127.0.0.1:43980 ESTABLISHED
tcp 0 0 222.200.180.178:40316 222.200.180.178:9999 ESTABLISHED
tcp 0 0 222.200.180.178:39780 222.200.180.178:6443 ESTABLISHED
tcp 0 0 222.200.180.178:60560 222.200.180.178:9090 ESTABLISHED
tcp 0 0 222.200.180.178:9090 222.200.180.178:60564 ESTABLISHED
tcp 0 0 127.0.0.1:2379 127.0.0.1:45296 TIME WAIT
tcp 0 0 127.0.0.1:43992 127.0.0.1:2379 ESTABLISHED
tcp 0 0 127.0.0.1:43918 127.0.0.1:2379 ESTABLISHED
tcp 0 0 222.200.180.178:39876 222.200.180.178:6443 ESTABLISHED
tcp 0 0 222.200.180.178:39920 222.200.180.178:6443 ESTABLISHED
tcp 0 0 127.0.0.1:2379 127.0.0.1:43938 ESTABLISHED
tcp 0 0 222.200.180.178:39802 222.200.180.178:6443 ESTABLISHED
tcp 0 0 222.200.180.178:36878 222.200.180.178:9306 ESTABLISHED
tcp 0 0 222.200.180.178:9090 222.200.180.178:60556 ESTABLISHED
tcp 0 0 127.0.0.1:43946 127.0.0.1:2379 ESTABLISHED
tcp 0 0 127.0.0.1:2379 127.0.0.1:43898 ESTABLISHED
tcp 0 0 10.96.0.1:59266 10.96.0.9:8081 TIME WAIT
tcp 0 0 127.0.0.1:2379 127.0.0.1:43880 ESTABLISHED
tcp 0 0 127.0.0.1:10251 127.0.0.1:60262 TIME WAIT
tcp 0 0 127.0.0.1:43948 127.0.0.1:2379 ESTABLISHED
tcp 0 0 222.200.180.178:60556 222.200.180.178:9090 ESTABLISHED
tcp 0 0 222.200.180.178:35376 10.96.0.1:443 ESTABLISHED
tcp 0 0 127.0.0.1:2379 127.0.0.1:43888 ESTABLISHED
tcp 0 0 222.200.180.178:6443 127.0.0.1:43946 ESTABLISHED
tcp 0 0 127.0.0.1:43990 222.200.180.178:9090 ESTABLISHED
tcp 0 0 127.0.0.1:10251 127.0.0.1:60306 TIME WAIT
tcp 0 0 127.0.0.1:43876 127.0.0.1:2379 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.184:38702 ESTABLISHED
tcp6 0 0 127.0.0.1:6443 127.0.0.1:43500 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 10.96.0.9:35037 ESTABLISHED
tcp6 0 0 222.200.180.178:3306 222.200.180.178:36878 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:35376 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.183:58776 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:42364 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:39898 ESTABLISHED
tcp6 0 0 222.200.180.178:3306 222.200.180.178:36220 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:54102 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.179:34888 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.184:38690 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:39802 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:54086 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:39942 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:39780 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:39972 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:39168 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.178:39604 ESTABLISHED
tcp6 0 0 222.200.180.178:6443 222.200.180.183:43114 ESTABLISHED

```

静态端口配置: /proc/services

动态连接查询: netstat -n



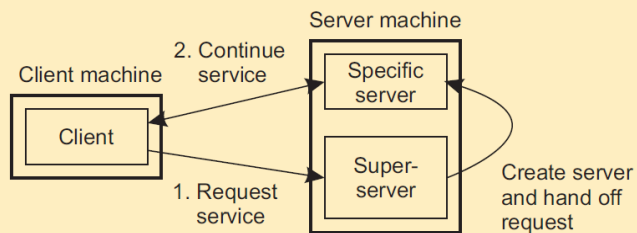
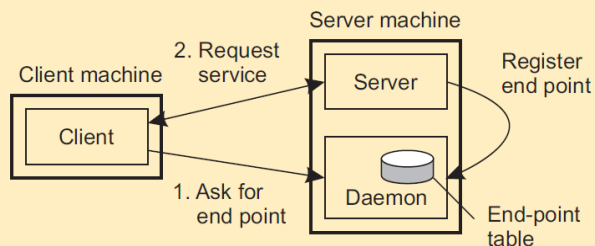


## 服务器连接

➤ 观察发现: 大多数服务都跟一个特定的端口绑定在一起

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
smtp	25	Simple Mail Transfer
www	80	Web (HTTP)

➤ 动态分配一个端口





## 带外通信

### ➤ 问题

在服务器已经接收了服务器请求后，是否有可能中断服务器的运行？

### ➤ 解决方案1： 利用一个分开的端口用于紧急数据通信

- ❑ 服务器拥有一个分离的线程或者进程用于紧急通信；
- ❑ 紧急信息到达 => 相关联的请求暂时挂起；
- ❑ 注意：我们需要 OS 支持基于优先级的调度策略；

### ➤ 解决方案2： 利用传输层的机制

- ❑ 样例：TCP 允许在同一个连接中传输紧急信息；
- ❑ 利用OS的信号机制捕捉紧急信息；



## 服务器和状态

### ➤ 无状态服务器（Stateless servers）

在处理完请求后，从来不保存关于客户端的精确的信息；

❑ 不记录一个文件是否被打开（文件请求完成就关闭）；

❑ 不保证清空客户端的cache；

❑ 不去追踪客户的信息；

### ➤ 结果

❑ 客户端和服务端完全独立；

❑ 由于客户端或者服务器的宕机导致的状态不一致减少；

❑ 由于服务器不能预测客户端的行为，可能导致性能下降；

### ➤ 问题

面向连接的通信是否适用于无状态的设计？





## 服务器和状态

### ➤ 有状态服务器 (Stateful servers)

记录客户端的状态信息：

- ❑ 记录客户端打开的文件，这样可以提前实现预取；
- ❑ 知道客户端缓存了哪些数据，允许客户端在本地保存共享数据的备份；

### ➤ 观察发现

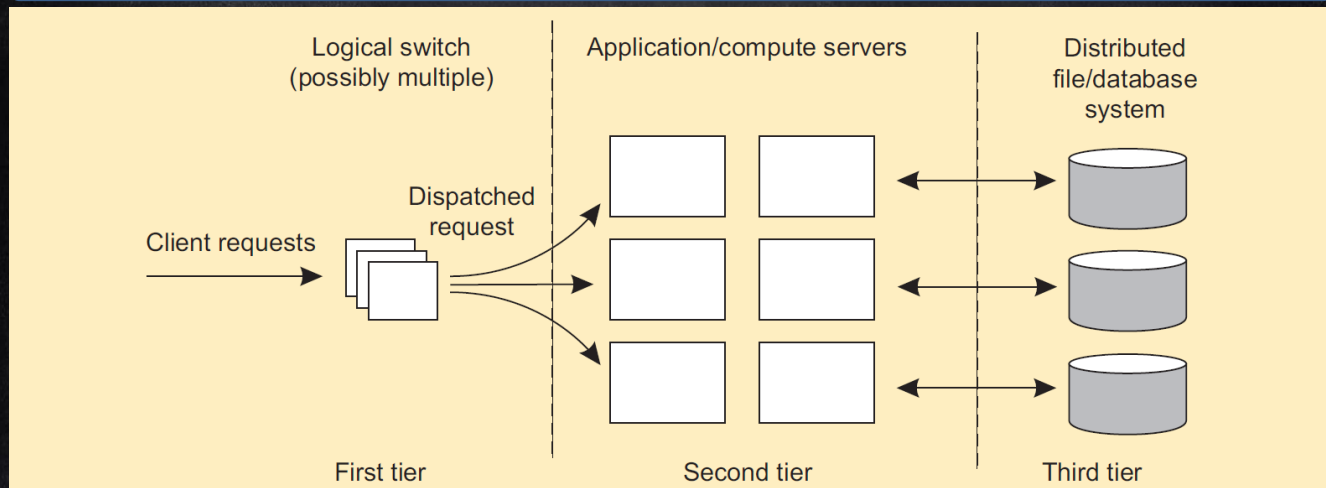
- ❑ 有状态的服务器的性能非常高；
- ❑ 可靠性问题是一个主要的问题；





# 服务器集群

## ➤ 常见的组织结构



## ➤ 关键单元

- 第一层主要用于请求分发（掩盖分布式系统的分布式特性）

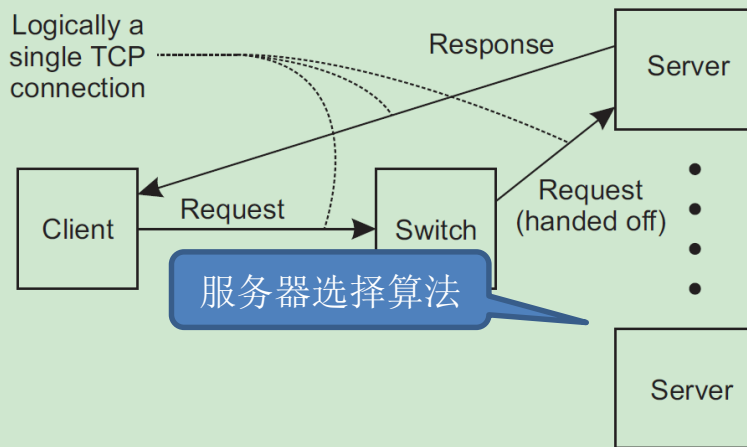


## 请求处理

### ➤ 观察

□ 让第一层处理所有的出入集群的通信会导致性能瓶颈：

### ➤ 观察



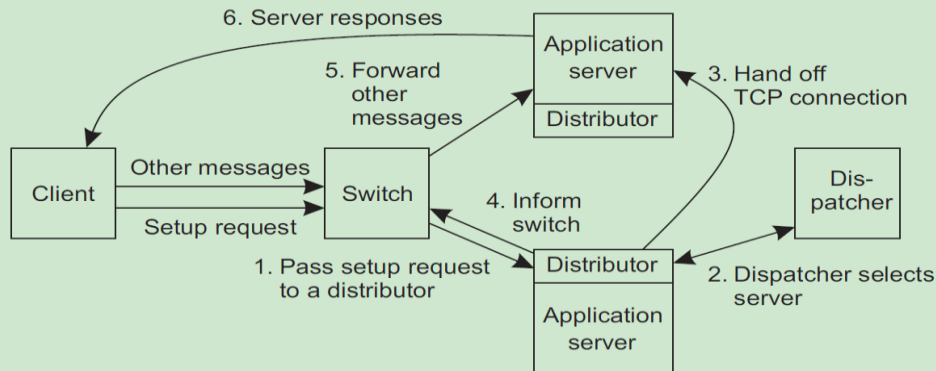


## 服务器集群

前端容易出现过载的情况：需要一种新的度量方法

- ❑ 基于传输层交换方法：前端简单地将TCP请求传递到服务器，只是会考虑某些简单的性能指标如CPU利用率等；
- ❑ 基于内容可感知的分派方法：前端会读取请求的内容，然后选择最合适的服务器；

将基于传输层交换和基于内容可感知的分派方法相结合





## 当服务器扩展到整个互联网

### ➤ 观察发现

将服务器集群扩展到互联网规模可能会引入管理问题。可以通过采用单个云计算商提供的数据中心避免这些问题。

### ➤ 请求分派：如果局部性比较重要的话

常用的方法是利用DNS：

- ❑ 客户端通过DNS查询特定的服务-客户端的IP地址是请求的一部分；
- ❑ DNS服务器记录请求服务的副本服务器信息，并且返回大部分本地服务的地址；

### ➤ 客户透明性

为了保障透明性，DNS resolver会代替客户端，问题是有可能距离远。

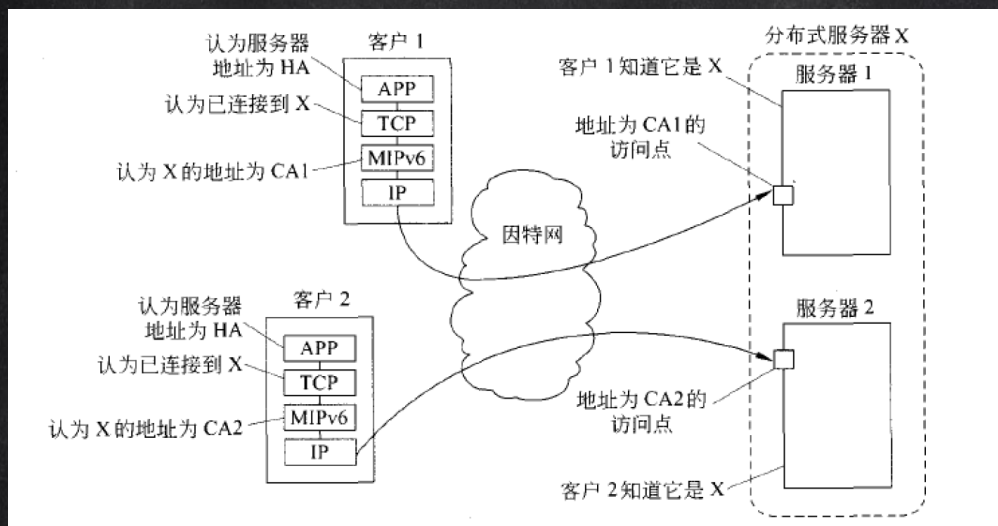




## 分布式服务器访问的稳定性

分布式服务器的基本思想是构造一个可靠的、高性能的、稳定的服务器即实现一个稳定的访问点。

➤ 具有稳定IPv6地址的分布式服务器





## 分布式服务器：处理细节

- 本质：客户端具有**Mobile IPV6**能够透明地跟任意其他节点简历连接
  - ❑ 客户端 C 通过IPv6 与宿主网络 HA 简历连接；
  - ❑ HA 被一个宿主代理维护着，HA 可以将连接传递给注册的需要地址 CA；
  - ❑ C 可以应用路由优化方法，通过将网络包直接转发到地址 CA（不用传递给宿主agent）；
- 协作分布式系统
  - ❑ 原始的服务器维护一个宿主地址，但是会将连接信息传递给协作节点；
  - ❑ 原始节点和协作节点看起来是一个服务器；



## 管理服务器：PlanetLab

### ➤ 本质

- ❑ 不同的组织贡献机器， 这些机器被共享用于进行各种实验；

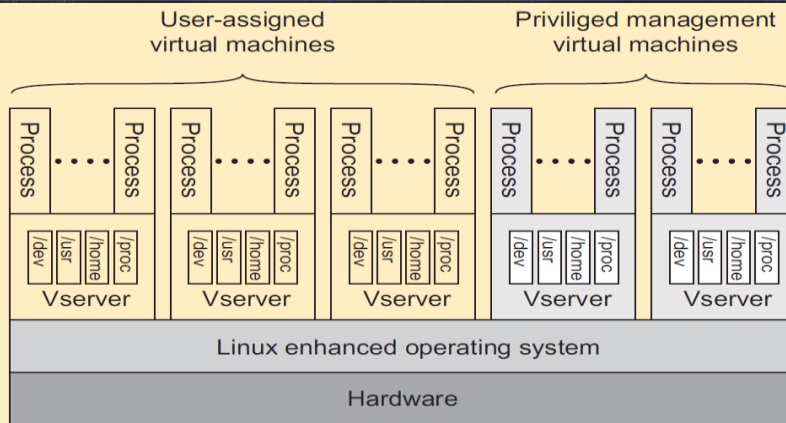
### ➤ 本质

- ❑ 我们需要保证不同的分布式应用相互不影响 => 虚拟化



# PlanetLab基本的组织结构

## ➤ 总览



## ➤ Vserver

- ❑ 具有自己的软件库、服务器版本等的独立受保护环境。分布式应用会分发到多个这样的机器上。



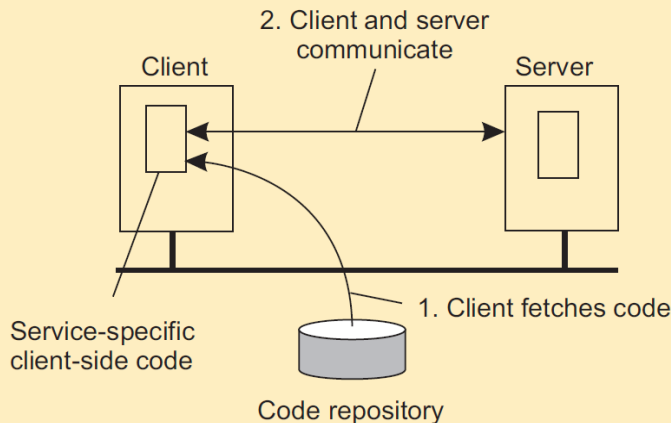


# 代码迁移

## ➤ 负载分布

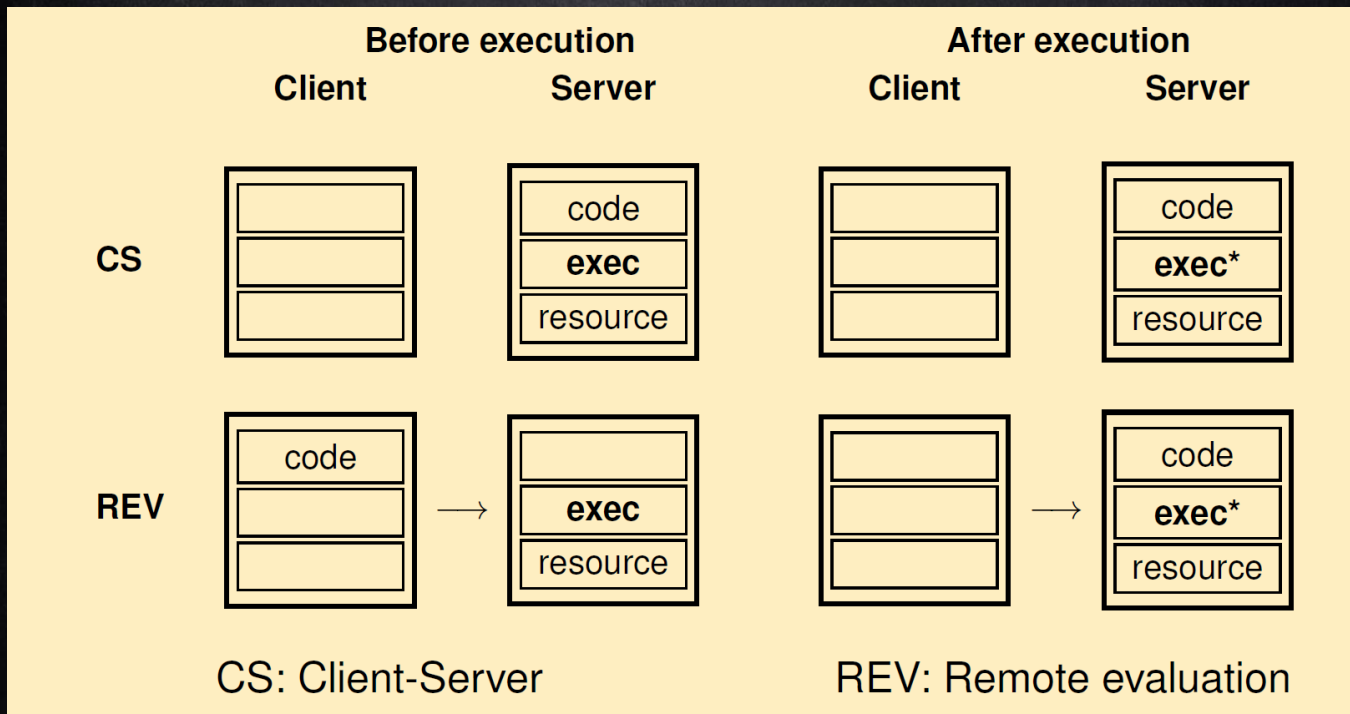
- ❑ 确保数据中心中的服务器的复杂运行更加充分（防止浪费能源）
- ❑ 让计算更加贴近数据端，最小化通信代价（例如移动计算）

## ➤ 灵活性：当需要的时候讲代码迁移到客户端



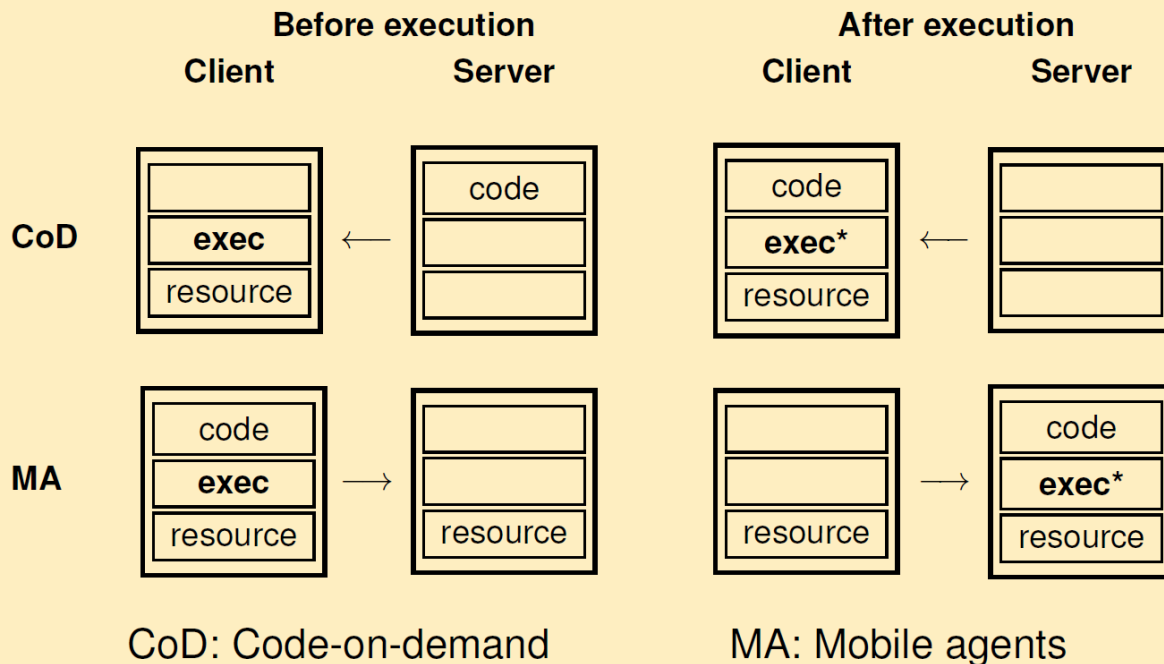


## 代码迁移的模型





## 代码迁移的模型





## 强/弱可移动性

### ➤ 对象组件

- ❑ 代码片段：包含实际执行的代码；
- ❑ 数据片段：包含状态
- ❑ 执行状态：包含线程执行对象代码的上下文；

### ➤ 弱移动性：仅仅移动代码和数据片段（重启执行）

- ❑ 相对简单，特别是如果代码是可移植的；
- ❑ 需要区分两种模式：代码推送（Push）和代码拉取（Pull）

### ➤ 强移动性：移动组件，包括执行状态

- ❑ 迁移（Migration）：将整个对象从一个机器移动到另外一个机器；
- ❑ 克隆（Cloning）：开始克隆，将其设置为相同的执行状态；





# 异构系统中的迁移

## ➤ 主要问题

- ❑ 目标机器可能不适合执行迁移后的代码；
- ❑ 进程/线程/处理器的上下文的定义比较依赖于硬件、操作系统和运行时系统；

## ➤ 仅有的解决方案：在不同的平台上抽象机器的实现

- ❑ 解释型语言，拥有自己的VM；
- ❑ 虚拟机监控器；



# 迁移虚拟机

## ➤ 迁移镜像：三个不同的方案

- ❑ 将内存特免推送到新的机器，在迁移过程中重新发送被修改过的页面；
- ❑ 停止当前的虚拟机；迁移内存，然后重新启动虚拟机；
- ❑ 让新的虚拟机按需拉取内存页面：在新的虚拟机上立即创建进程，并且按需拷贝内存页面；

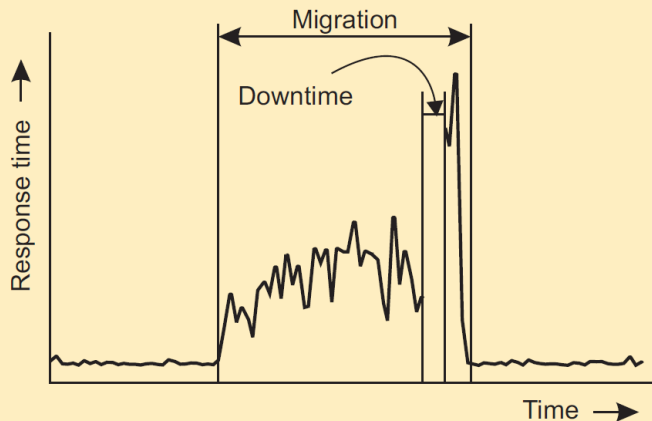


## 虚拟机迁移的性能

### ➤ 问题

一次完整的虚拟机迁移可能需要几十秒。在迁移期间，服务可能处于几秒钟的完全不可用的状态。

### ➤ 在虚拟机迁移过程中虚拟机响应时间的度量





## 作业

虚拟机迁移的实现和迁移代价量化如迁移时间，迁移导致服务的响应延迟，服务不可用的时间，额外的CPU、网络IO、磁盘IO、以及内存等。