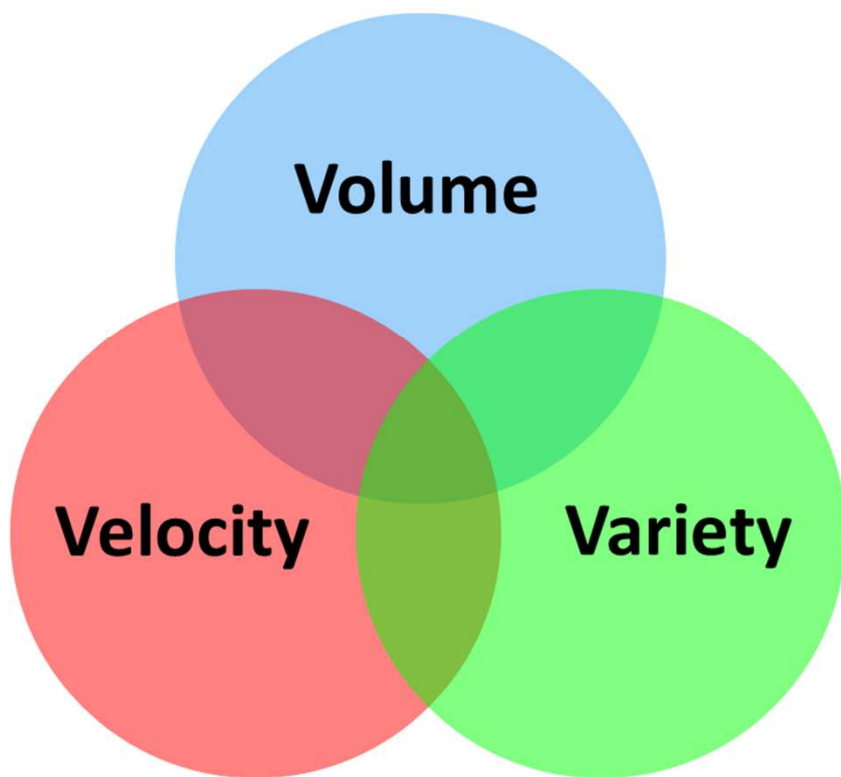


大数据系统与大规模数据分析

大数据存储系统 (1)



陈世敏

中科院计算所
计算机体系结构
国家重点实验室

©2015-2017 陈世敏

Outline

- 分布式系统基本概念
- 分布式文件系统
- Google File System和HDFS

Outline

- 分布式系统基本概念
 - 网络与协议
 - 通信方式
 - 分布式系统类型、故障类型、CAP
- 分布式文件系统
- Google File System和HDFS

Internet（互联网）

- 为什么叫Internet?

- Internet: the network of networks

- Network: network of devices

- 通过网线、无线、交换机等把设备连接起来

- 设备：计算机、平板、手机、无线传感器.....

- 局域网：实验室、家庭内网、数据中心网络、校园网、公司内网.....

- Network of networks

- 把一个个局域网连接起来

- 形成覆盖全球的网络

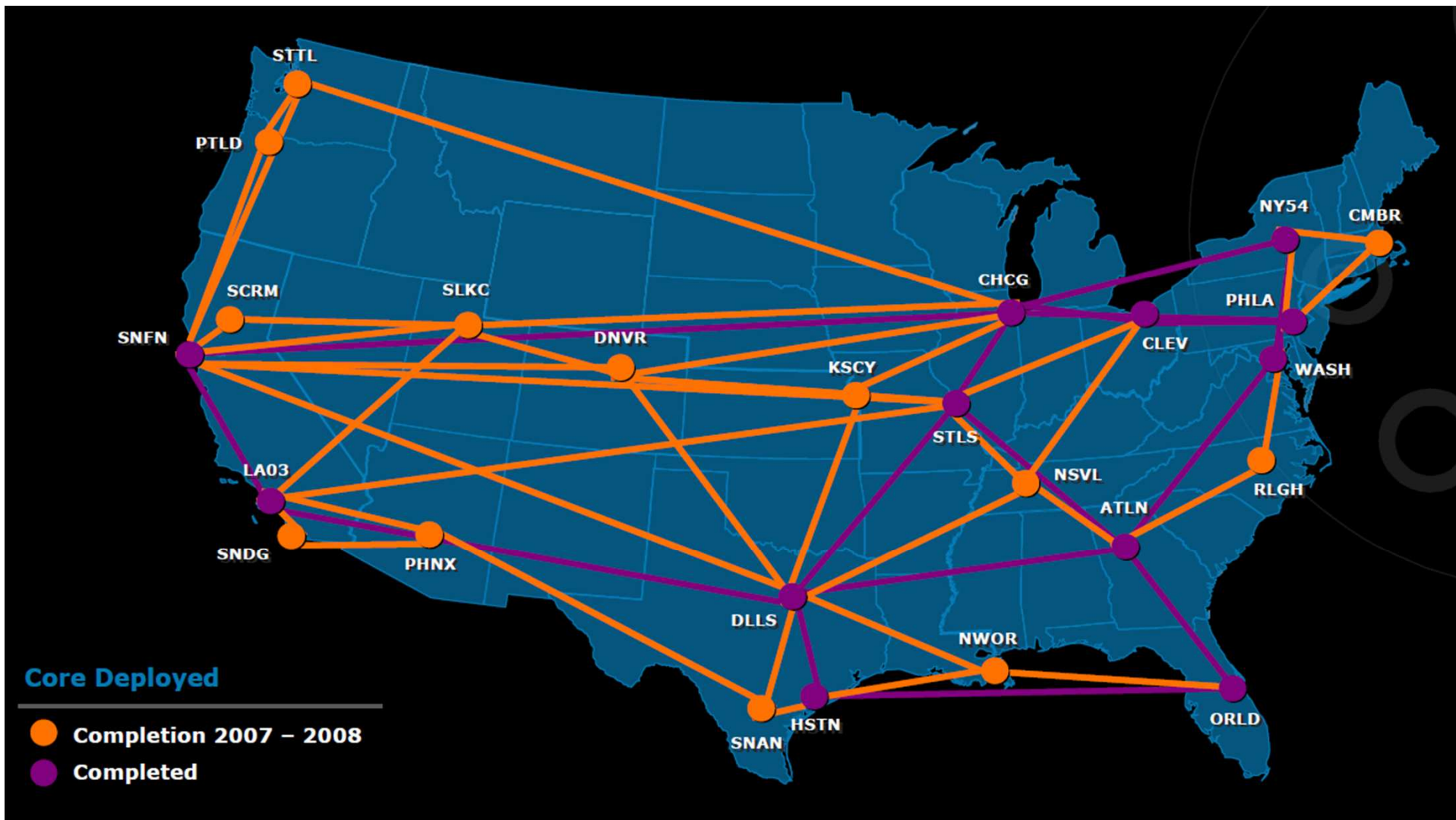
- 谁来连？

- ISP (Internet Service Provider) 建立各自的骨干网

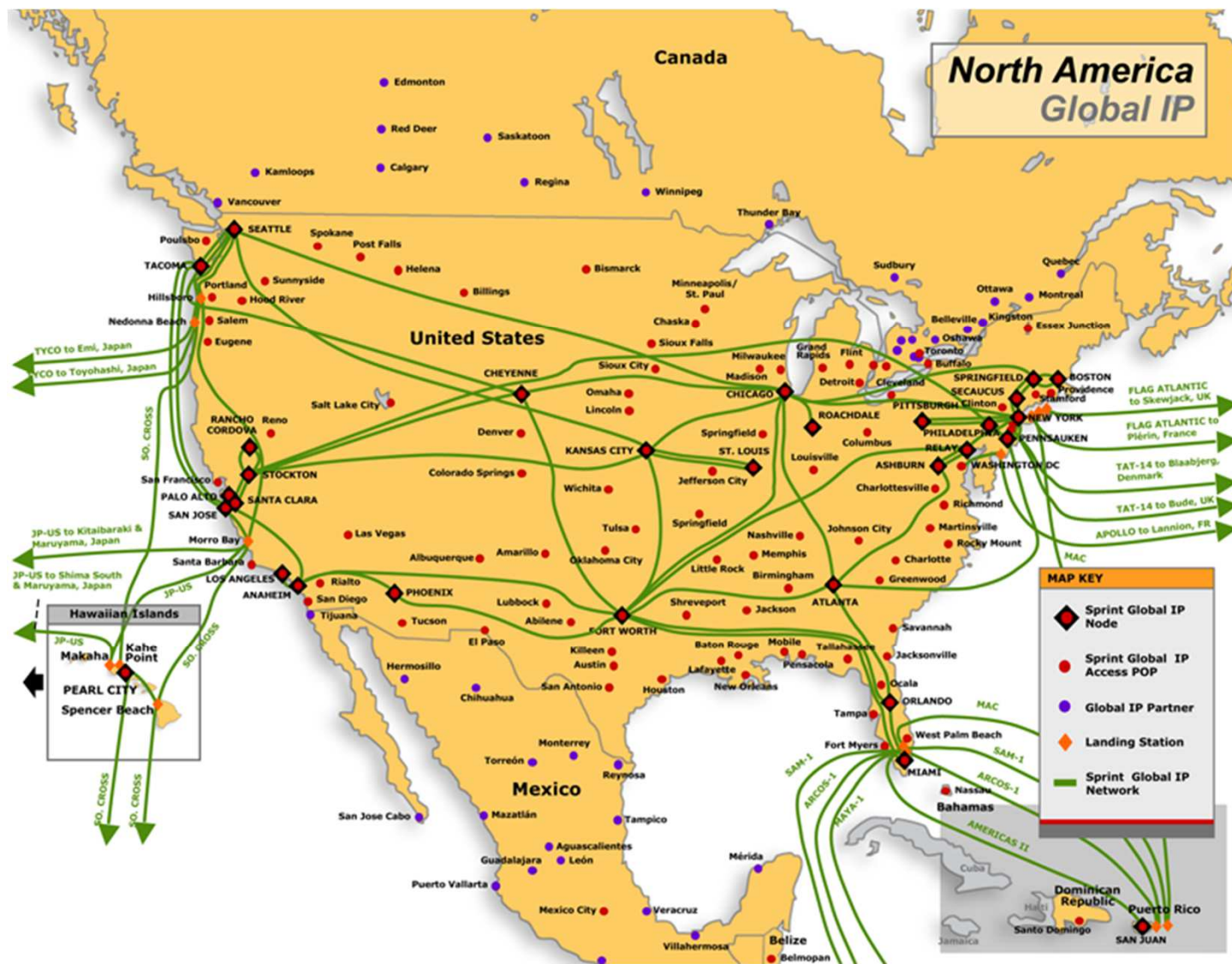
- 骨干网(Backbone network)：通常是光纤

- 是冗余的，两地之间通常存在不同ISP的不止一条线路

AT&T美国骨干网



Sprint美国骨干网



CenturyLink Fiber Network



Circuit Definitions

Fiber Route ———



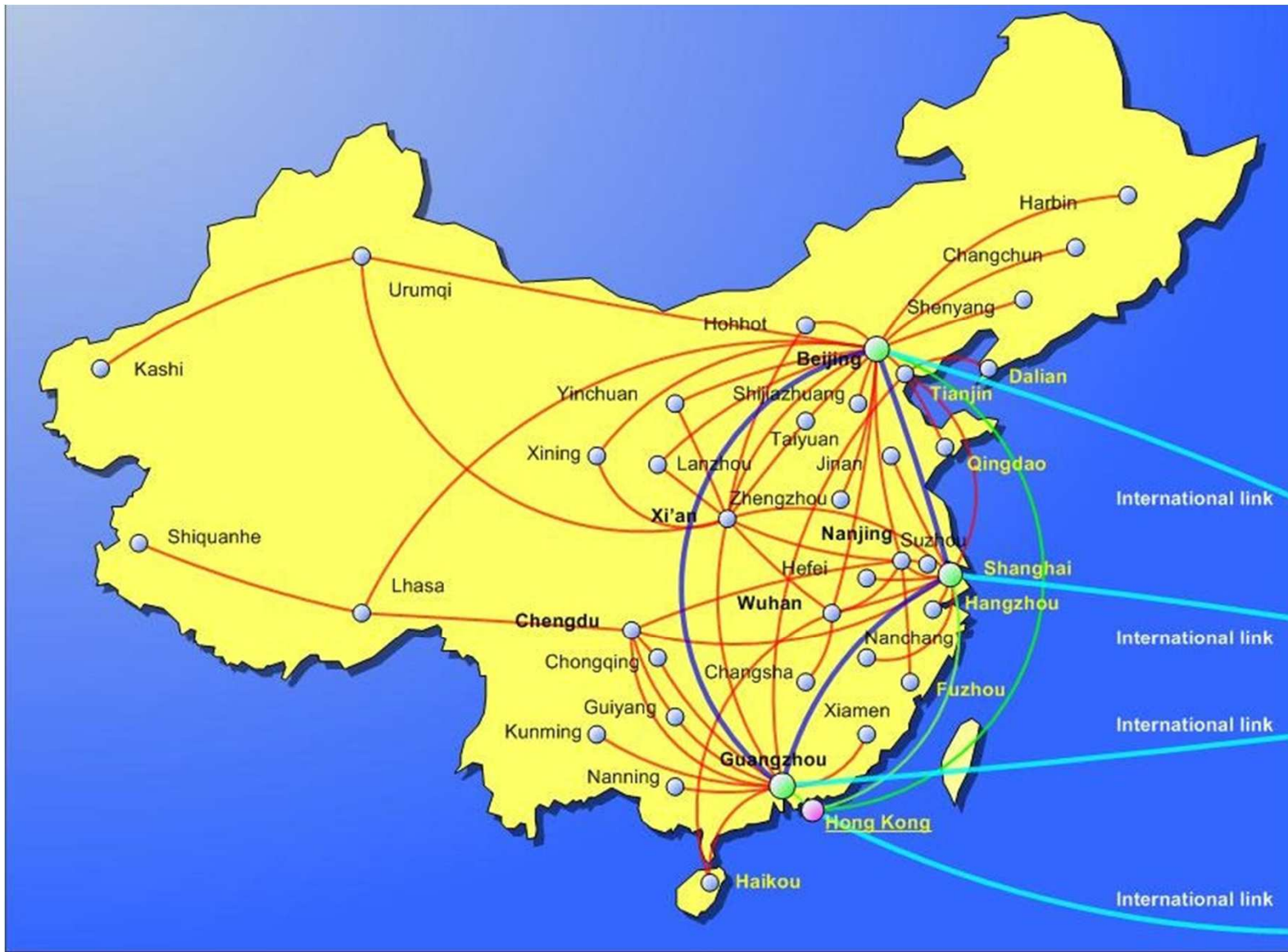
©2013 CenturyLink, Inc. All Rights Reserved. NM090928 5/13

中国教育科研网CERNET

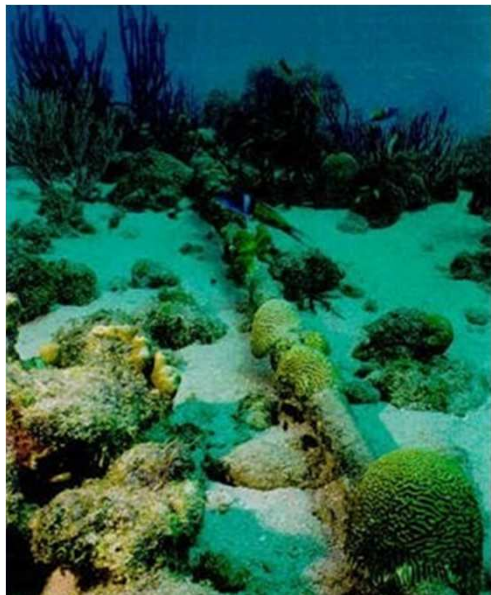
CERNET 拓扑图



ChinaNet (中国电信)



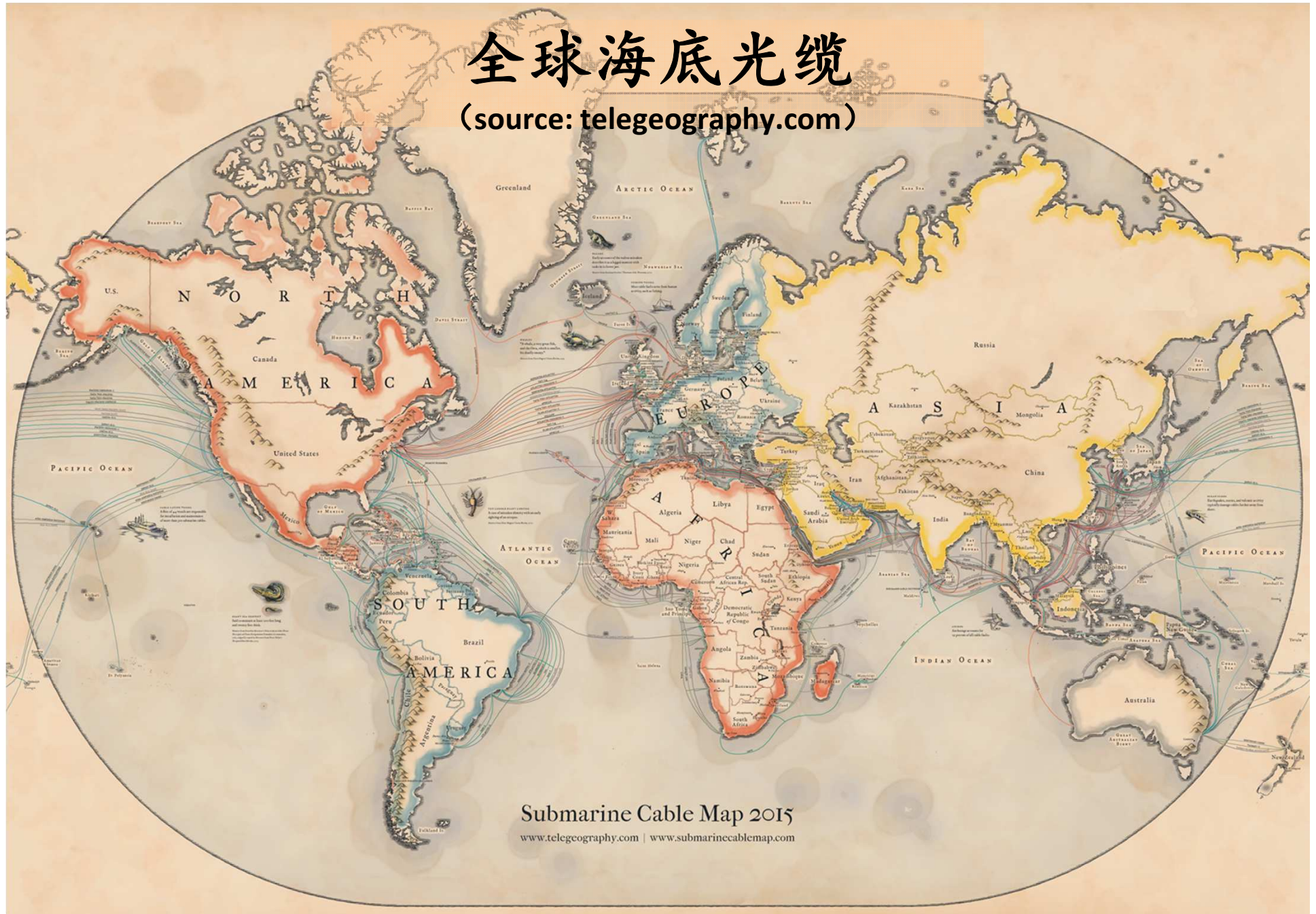
海底光缆



图片来源：百度图片

全球海底光缆

(source: telegeography.com)



海底光缆

(source: telegeography.com)



Protocol (通信协议)

- 什么是Protocol?

- 两方或多方之间
- 对通信方式的协定
 - 语法
 - 数据格式, 编码, 长度
 - 语义



Open Systems Interconnection Model (OSI) vs. Internet

OSI model		Internet	
Application		Application (dns, http, rpc, ssh, ...)	用户态 程序
Presentation			
Session			
Transport	端到端设备之间传输	TCP/UDP	OS 内核
Network	多跳全网寻址、路由	IP	
Data link	直接连接的设备之间传输	Data link	网络 设备
Physical	物理连线	Physical	

IP/TCP/UDP

- IP (Internet Protocol)

- IPv4地址：例如210.76.211.7，唯一标识一台联网的机器
- Routing(路由)
- IP packet: header, data
- Connectionless (无连接), unordered(无序), best-effort (不保证可靠)

- TCP (Transmission Control Protocol)

- 在IP基础上实现
- Port端口号：不同的进程/socket
- Reliable (可靠的), ordered (有顺序), connection-oriented (有连接), error checked (数据校验)

建立连接成本高，
但是使用方便

- UDP (User Datagram Protocol)

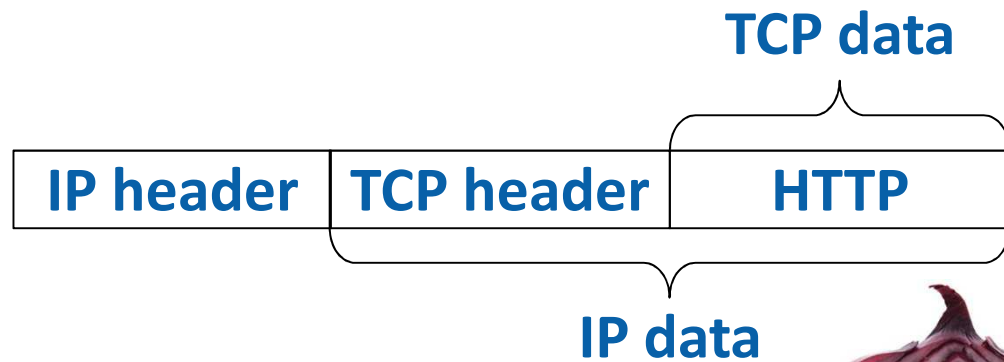
- 在IP基础上实现
- Port端口号：不同的进程
- 进行数据校验，其它与IP相同

不需建立连接，速度快
但是应用程序要自己检查
数据是否收到、顺序等

应用层协议

- Encapsulate

- 一层包一层



- DNS (Domain Name Service)

- 域名：例如 sep.ucas.ac.cn

- UDP port 53

- 全球有根域名服务，各国各自管理自己的域名分配

- HTTP (Hyper Text Transfer Protocol)

- TCP port 80

- 有的其它协议是基于HTTP实现的，以便通过防火墙等



Process/Thread

- 在OS内核中两者很相似
- Process (进程)
 - 创建: fork
 - 私有的虚存空间
 - 私有的打开文件 (files, sockets, devices, pipes ...)
- Thread (线程)
 - 创建: pthread_create → clone
 - 共享的虚存空间
 - 共享的打开文件
 - 一个进程中可以有多个线程

应用程序的通信方式

- Shared memory (共享内存)

- 在单机上

- 同一个进程内部，多个线程之间
 - 多个进程之间，把同一块物理内存映射到多个进程的虚存空间中

- 一方修改，另一方可以立即看到

- 需要并发控制

- Message passing (消息传递)

- 单机上，多进程之间

- 多机之间

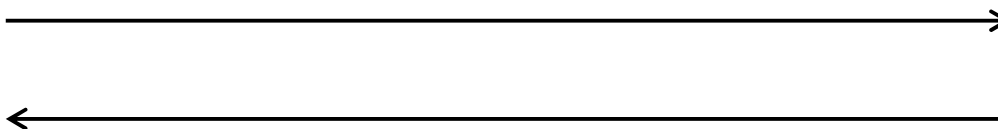
- 例如：socket (TCP/UDP)，pipe等

通常的消息传递

请求：功能+参数

- 功能：功能的ID或者功能名
 - 例如：HTTP GET, HTTP PUT
 - 接收方根据它，完成需要的操作
- 参数：输入数据
 - 编码(Encoding)，串行化(Serialization)

发送方

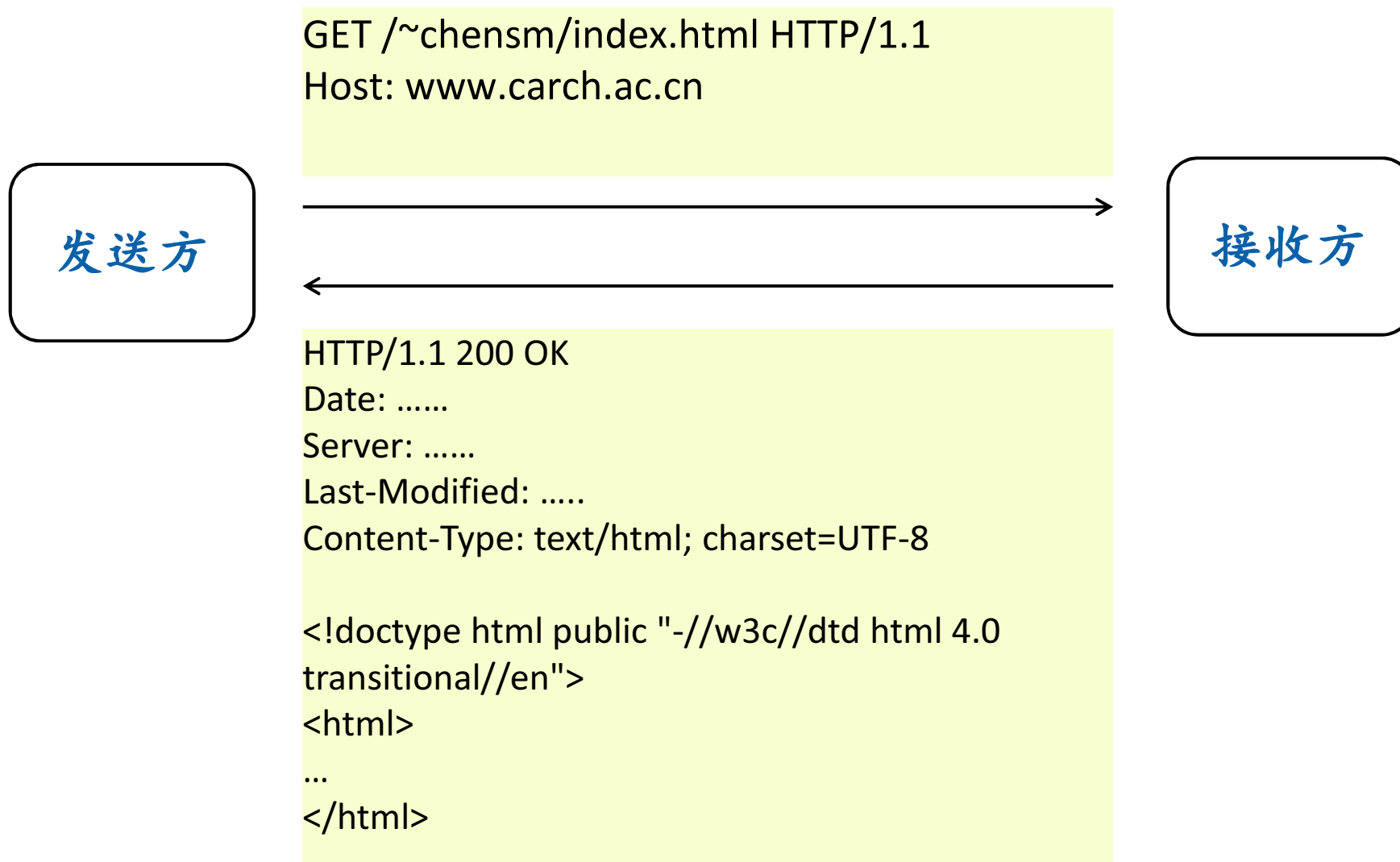


接收方

回答：结果+结果数据

- 结果：结果ID或者结果名
 - 例如：OK, ERROR, SUCCESS
- 结果数据：也需要编码和串行化

举例：访问一个网页



分布式系统类型

- Client / Server

- 客户端发送请求，服务器完成操作，发回响应
- 例如：3-tier web architecture
 - Presentation: web server
 - Business Logic: application server
 - Data: database server

- P2P (Peer-to-peer)

- 分布式系统中每个节点都执行相似的功能
- 整个系统功能完全是分布式完成的
- 没有中心控制节点

- Master / workers

- 有一个/一组节点为主，进行中心控制协调
- 其它多个节点为workers，完成具体工作

故障模型(Failure Model)

- Fail stop

- 当出现故障时，进程停止/崩溃

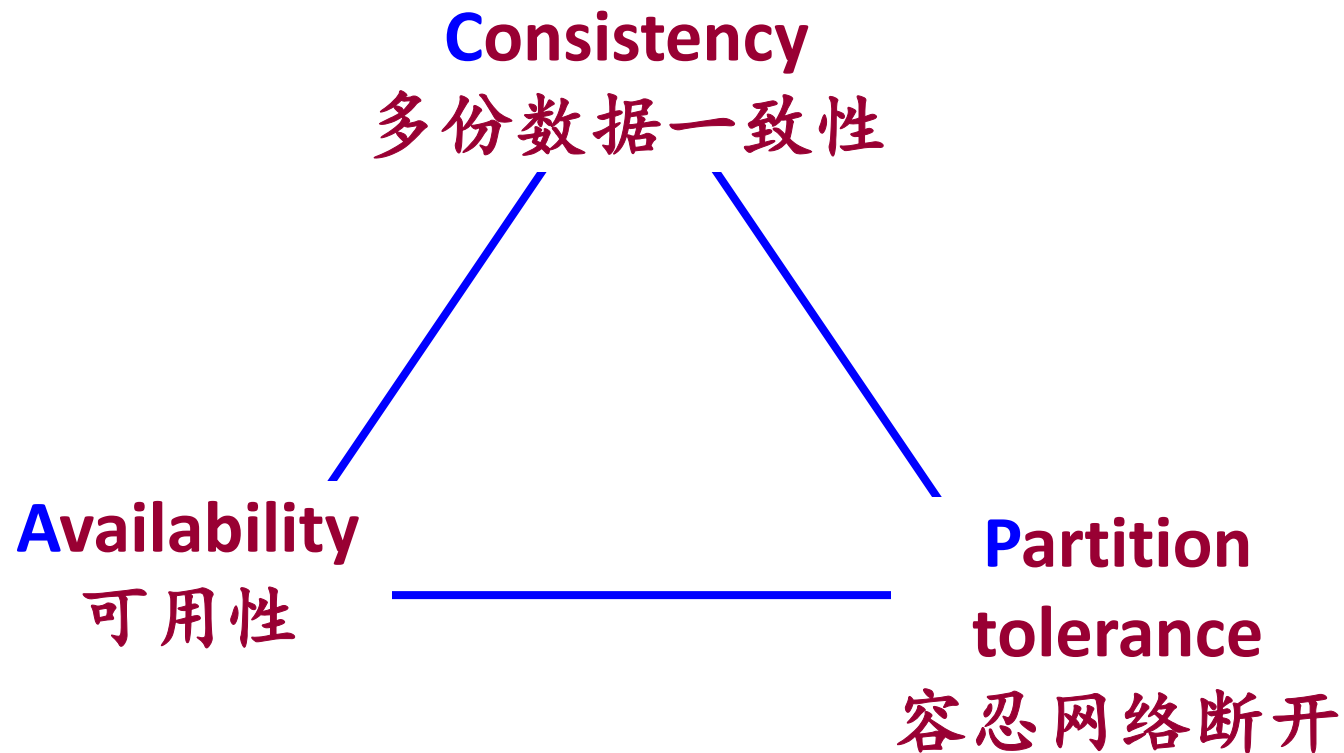
- Fail slow

- 当出现故障时，运行速度变得很慢

- Byzantine failure

- 包含恶意攻击

CAP定理



三者不可得兼

系统设计的理念

- Simple is beauty

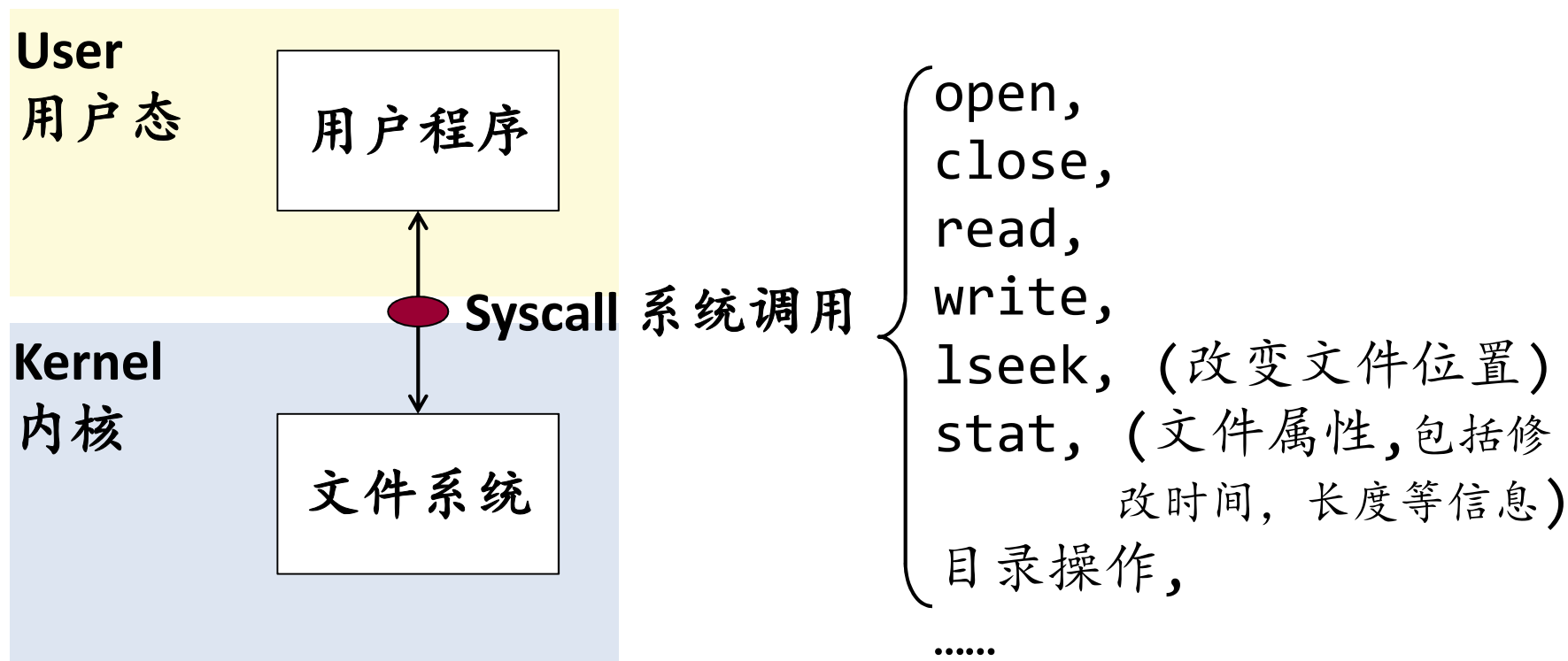
- 本质的往往是简单的
- 越复杂的越难以正确地实现，可能引入各种漏洞问题
- “Simplicity is the ultimate sophistication.”
— Leonardo da Vinci
- “Nature is pleased with simplicity.”
— Isaac Newton
- “世上本无事，庸人自扰之。”
— 孔子

Outline

- 分布式系统基本概念
- 分布式文件系统
 - NFS
 - AFS
- Google File System和HDFS

本地文件系统 (Local File System)

- 例子：Linux ext4, Windows ntfs, Mac OS hfs ...



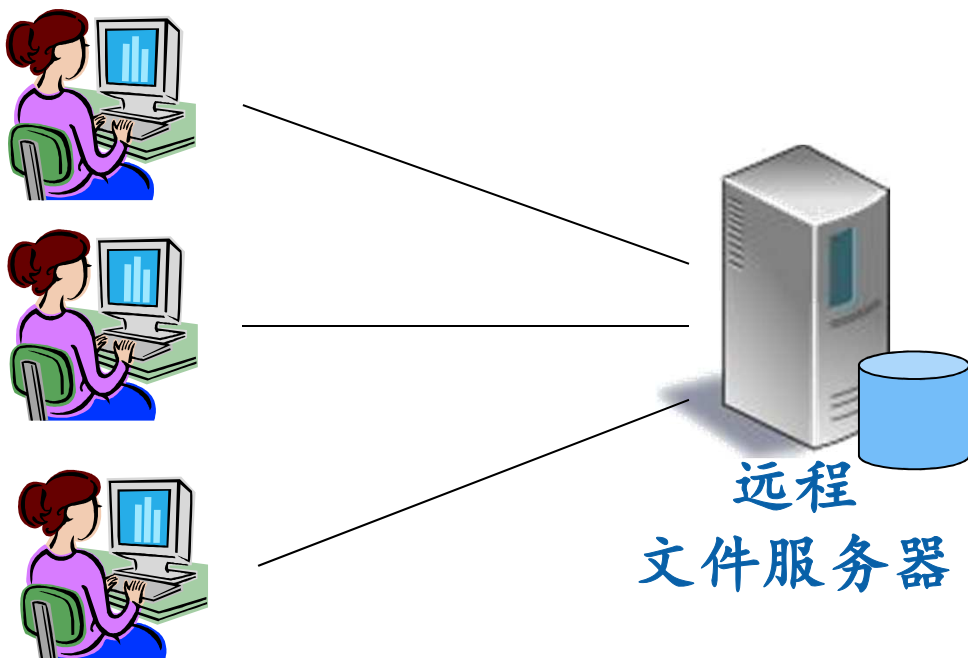
举例

// 把/myfile文件的开始4KB的数据拷贝到文件尾部

```
char buf[4096];  
int fd = open ("/myfile", 0600, O_RDWR);  
read(fd, buf, 4096);  
lseek(fd, 0, SEEK_END);  
write(fd, buf, 4096);  
close(fd);
```

NFS (Sun's Network File System)

- Sun公司1985发布了NFSv2，定义了开放的client/server之间的通信协议标准
- 非常流行，很多数据存储产品都提供NFS



主要目的

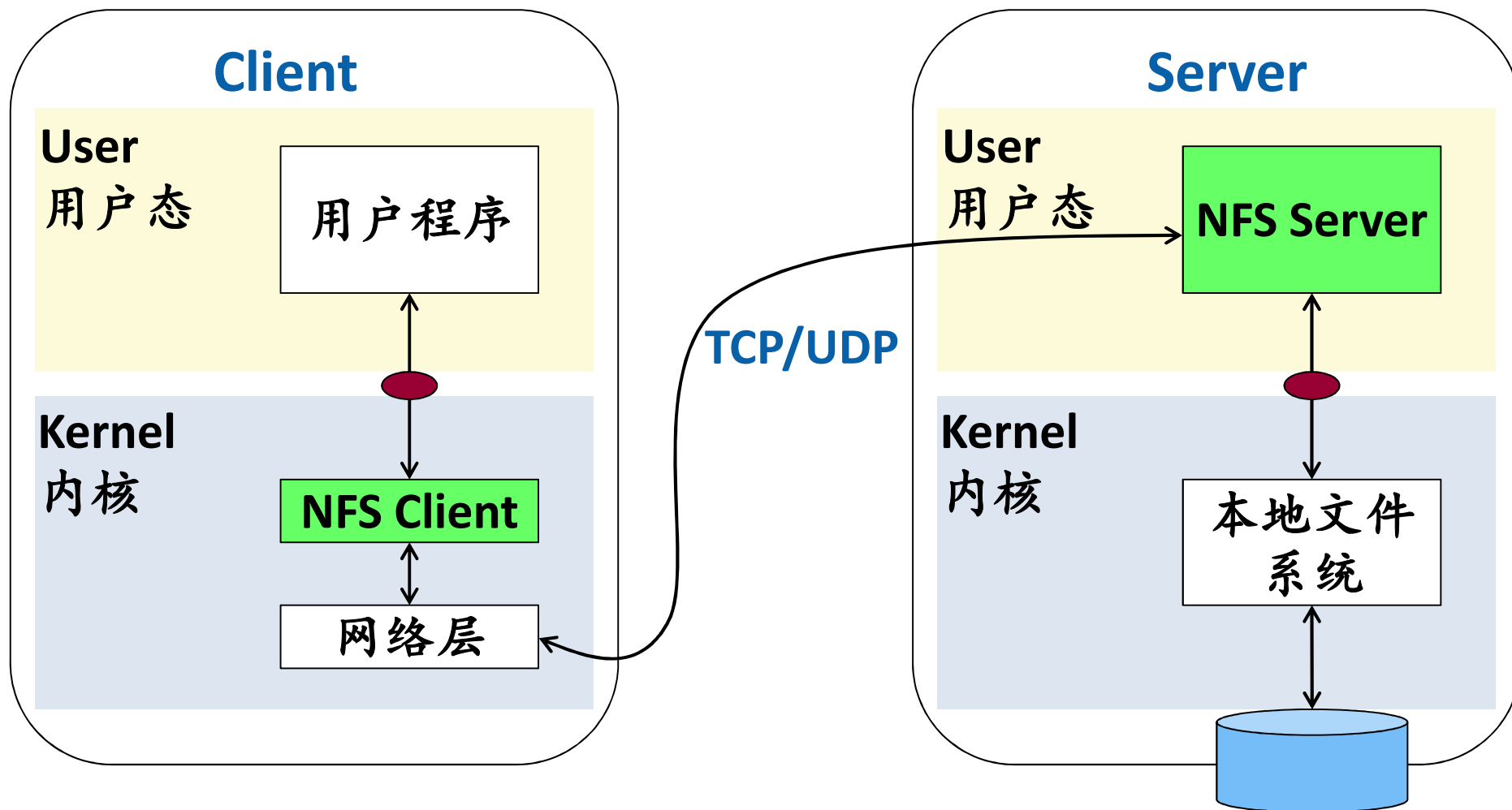
- 从不同的终端都可以访问同一个目录
- 多用户共享数据
- 集中管理

参考: <http://pages.cs.wisc.edu/~remzi/OSTEP/>

访问的文件

- 主要是用户目录下的文件
- 文档编辑、编程、编译、运行
- 不是：处理大规模数据

系统架构



如何设计为好？

- NFSv2设计目标1

- Simple and Fast server crash recovery

服务器出现故障，可以简单快速地恢复
(Fail-stop 模型)

- 怎么样最简单？

- 什么都不用做，最简单

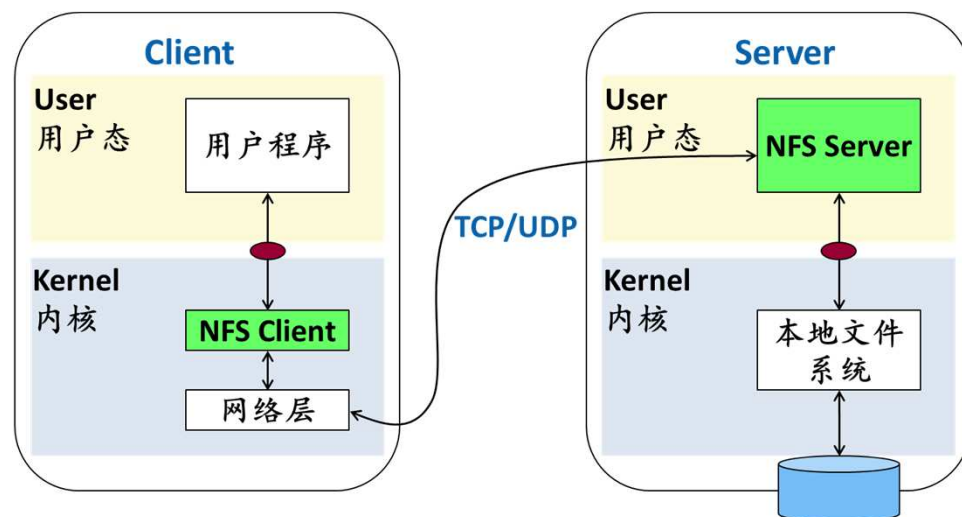
- 希望：当出现故障，NFS Server重新启动，不用做其它操作

- 解决方案

- Stateless

- Idempotent

保存State的问题



- 例如：客户端打开文件的文件位置信息，如果存在NFS Server里，会怎么样？
- 故障重启后，内存状态都丢失了，客户端再次操作就会出错！

```
int fd = open ("/myfile", 0600, O_RDWR);  
read(fd, buf, 4096);  
lseek(fd, 0, SEEK_END);  
write(fd, buf, 4096);  
close(fd);
```



**NFS Server
Restart**

还会写到文件尾吗？

Stateless 操作

- NFS Server不保持任何状态，每个操作都是无状态的
- NFSPROC_READ
 - 输入参数: file handle, offset, count
 - 返回结果: data, attributes
- NFSPROC_WRITE
 - 输入参数: file handle, offset, count, data
 - 返回结果: attributes
- NFSPROC_LOOKUP
 - 输入参数: directory file handle, name of file/directory to look up
 - 返回结果: file handle
- NFSPROC_GETATTR
 - 输入参数: file handle
 - 返回结果: attributes
- 等等

举例

客户端

```
int fd = open (“/myfile”, ...);
```

发送LOOKUP(根的FH, “myfile”)

接收LOOKUP结果
在内核file table中分配file desc
记录myfile的FH
记录文件位置为0
返回fd

服务器端

处理LOOKUP, 找到
“myfile”, 返回唯一标识FH

举例

客户端

```
read(fd, buf, 4096);
```

发送READ(FH, 0, 4096)

接收READ结果
记录文件位置为 $0+4096=4096$
拷贝数据到buf
记录文件属性(修改时间, 长度等)
返回0 (成功)

服务器端

处理READ, 读文件, 返回数据
和文件属性

举例

客户端

服务器端

```
lseek(fd, 0, SEEK_END);
```

记录文件位置为文件长度1048576
返回0（成功）

假设文件原始长度为1MB

举例

客户端

服务器端

```
write(fd, buf, 4096);
```

拷贝buf中的数据

发送WRITE(FH, 1048576, 4096, 数据)

处理WRITE, 写文件, 返回
文件属性

接收WRITE结果

记录文件位置为 $1048576+4096=1052672$

记录文件属性(修改时间, 长度等)

返回0 (成功)

举例

客户端

服务器端

```
close(fd);
```

释放file desc
返回0（成功）

Idempotent

- READ操作是Idempotent

- 在没有其它操作前提下，重复多次结果是一样的
- 为什么？
不改变数据

- WRITE操作是Idempotent

- 在没有其它操作前提下，重复多次结果是一样的
- 为什么？
在相同位置写
相同的数据

Server Crash Recovery

- NFS Server

- 只用重启，什么额外操作都不用
- 因为Stateless

- NFS Client

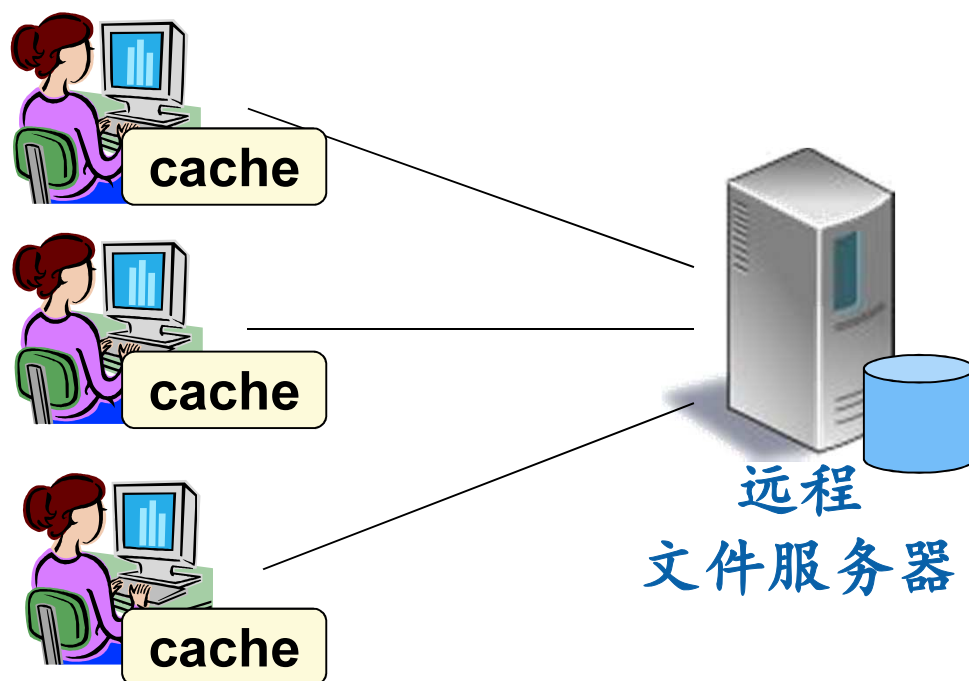
- 如果一个请求没有响应，那么就不断重试
- 因为Idempotent

如何设计为好？

- NFSv2设计目标2

- 远程文件操作性能高

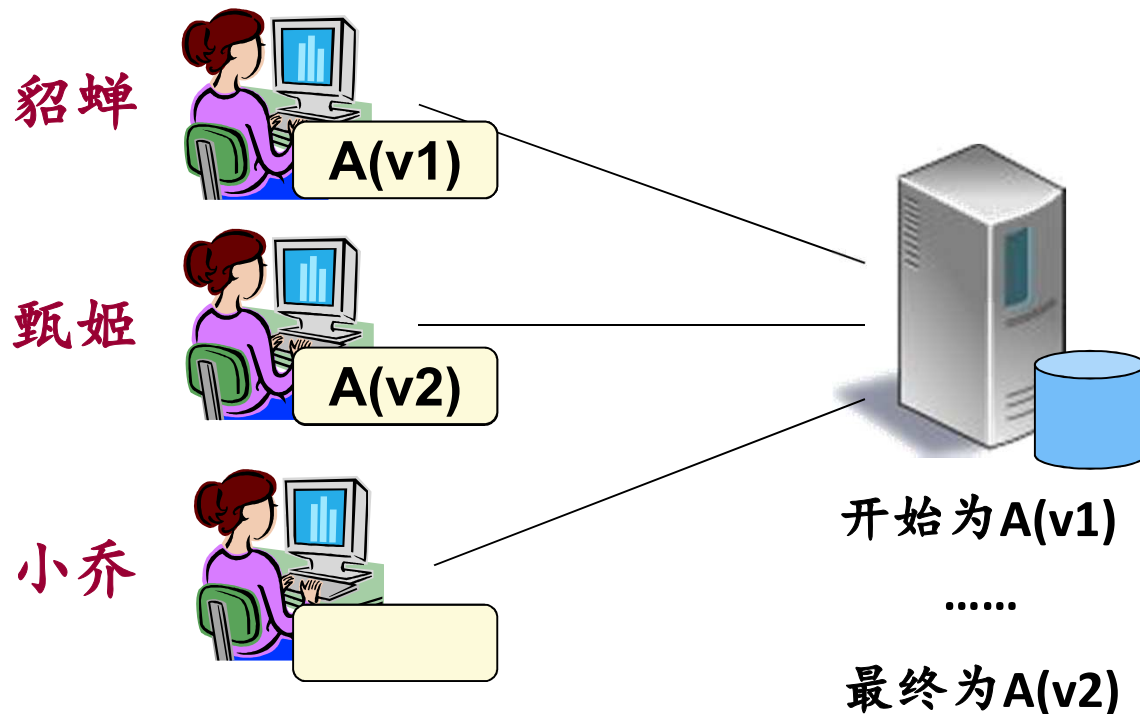
- 解决思路：Client cache (在内存中)，缓存读写的数据



- 尽量利用cache数据
- 避免远程操作

问题：Cache Consistency

- 对于同一个文件，并发访问冲突问题的表现
- 例如：文件A(版本)



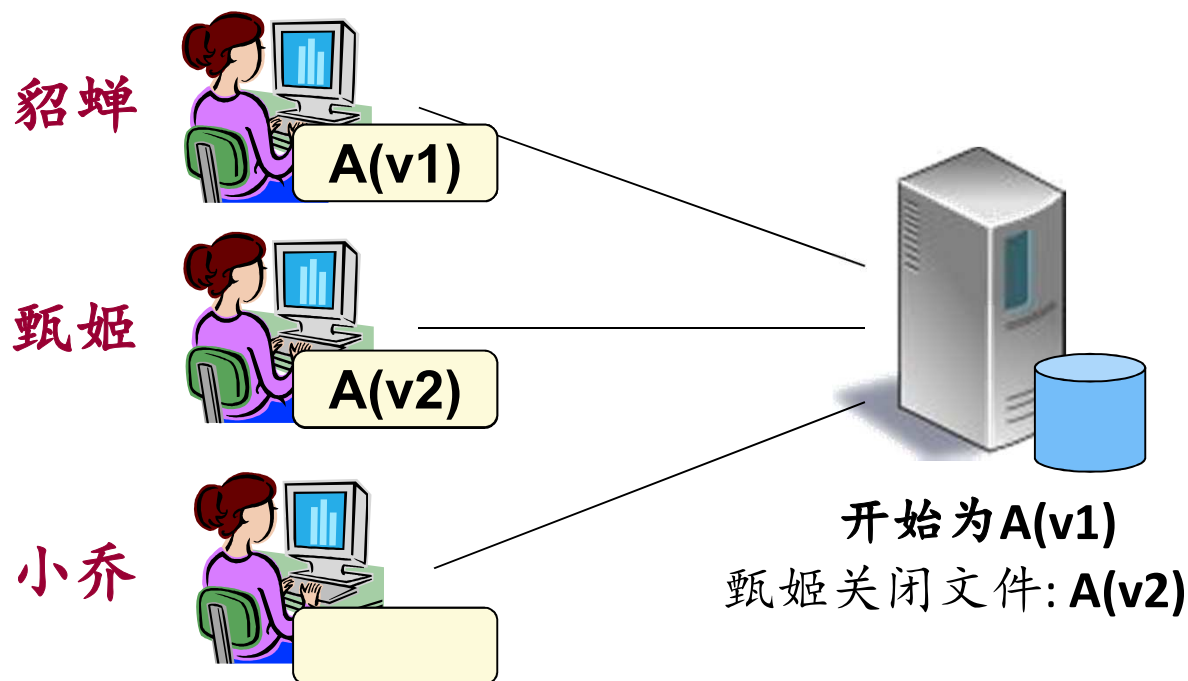
- Client貂蝉读了文件A(v1), 放入缓存
- Client甄姬读了文件A(v1), 然后写, 成为A(v2)
- 问题1: 貂蝉缓存的数据变成**过时的**
- 问题2: 其它client小乔如何能读到最新的数据?

注意：这里和transaction ACID不同，可以理解为每个文件操作都是已经commit的！

NFSv2对于Cache Consistency的解决方法

- Flush-on-close (又称作close-to-open) consistency

□ 在文件关闭时，必须把缓存的已修改的文件数据，写回 NFS Server



- Client貂蝉读了文件A(v1), 放入缓存
- Client甄姬读了文件A(v1), 然后写, 成为A(v2)
- 甄姬关闭文件时, 将最新版本发回NFS Server

问题: 貂蝉怎么知道服务器上文件变化了, 需要丢弃旧版本呢?

NFSv2对于Cache Consistency的解决方法

- Flush-on-close (又称作close-to-open) consistency

- 在文件关闭时，必须把缓存的已修改的文件数据，写回NFS Server

- 每次在使用缓存的数据前，必须检查是否过时

- 用GETATTR请求去poll（轮询），获得最新的文件属性
- 比较文件修改时间

- 性能问题

- 大量的GETATTR（即使文件只被一个client缓存）
- 关闭文件的写回性能

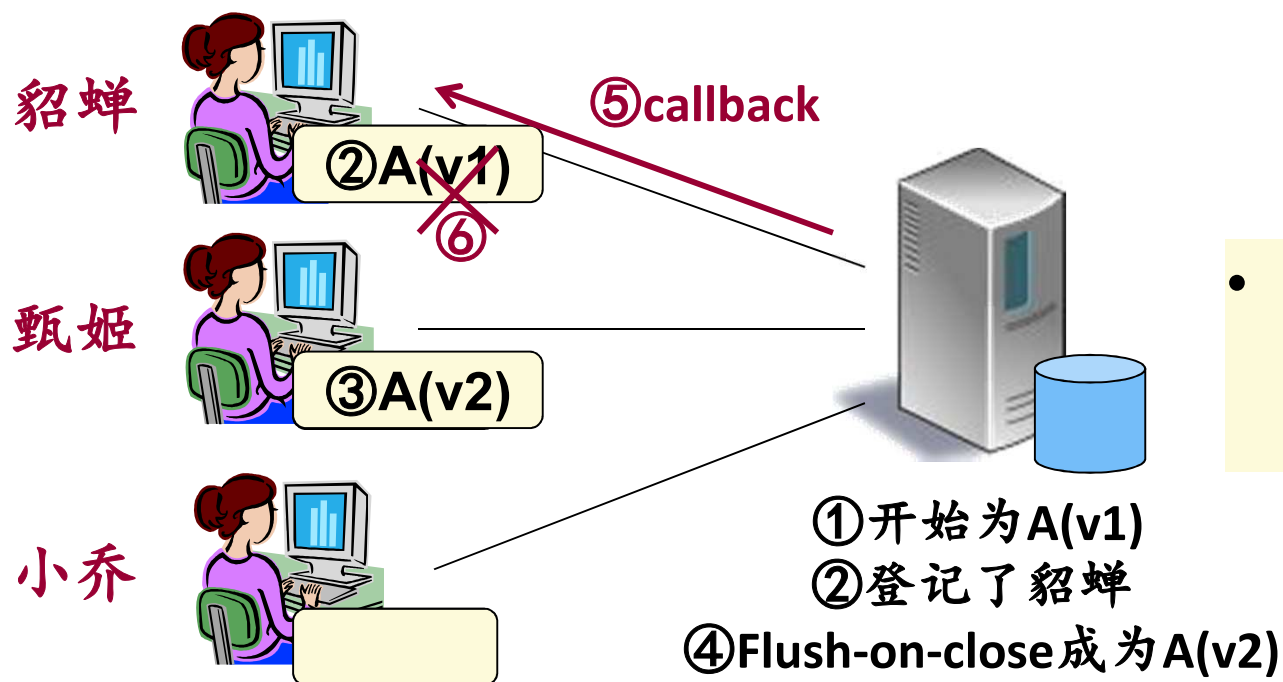
AFS (Andrew File System)

- CMU M. Satyanarayanan (Satya) in 1980s
- 在美国大学中很流行
- 设计目标: Scalability
 - 一个服务器支持尽可能多的客户端
 - 解决NFS polling状态的问题

解决polling状态的问题

• Invalidation

- Client 获得一个文件时，在server上登记
- 当server发现文件修改时，向已登记的client发一个callback
- Client收到callback，则删除缓存的文件



- 有效地避免了polling的代价，减轻了Server的负担

其它不同点：AFS vs. NFSv2

- AFS缓存整个文件

- 而NFS是以数据页为单位的
- AFS open: 将把整个文件从Server读到Client
- 多次操作：就像本地文件一样
- 单次对一个大文件进行随机读/写：比较慢

- AFS缓存在本地硬盘中

- 而NFS的缓存是在内存中的
- 所以AFS可以缓存大文件

- AFS

- 有统一的名字空间，而NFS可以mount到任何地方
- 有详细权限管理等

Outline

- 分布式系统基本概念
- 分布式文件系统
- Google File System和HDFS

GFS/HDFS

- Google File System

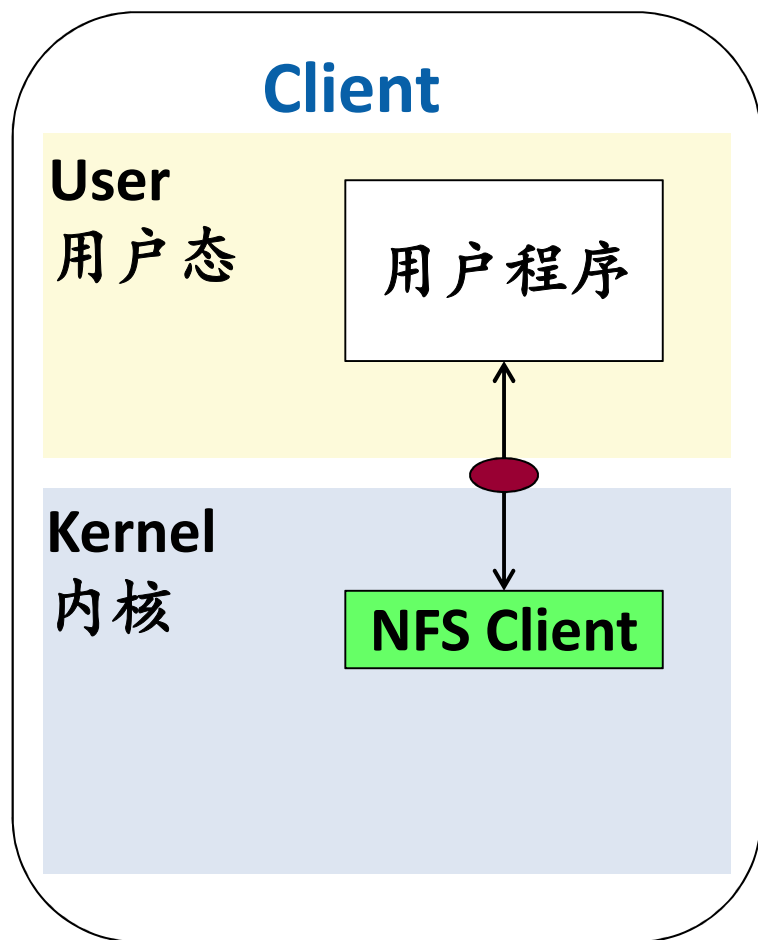
- SOSP 2003, C/C++实现
- Google MapReduce系统的基础

- Hadoop Distributed File System

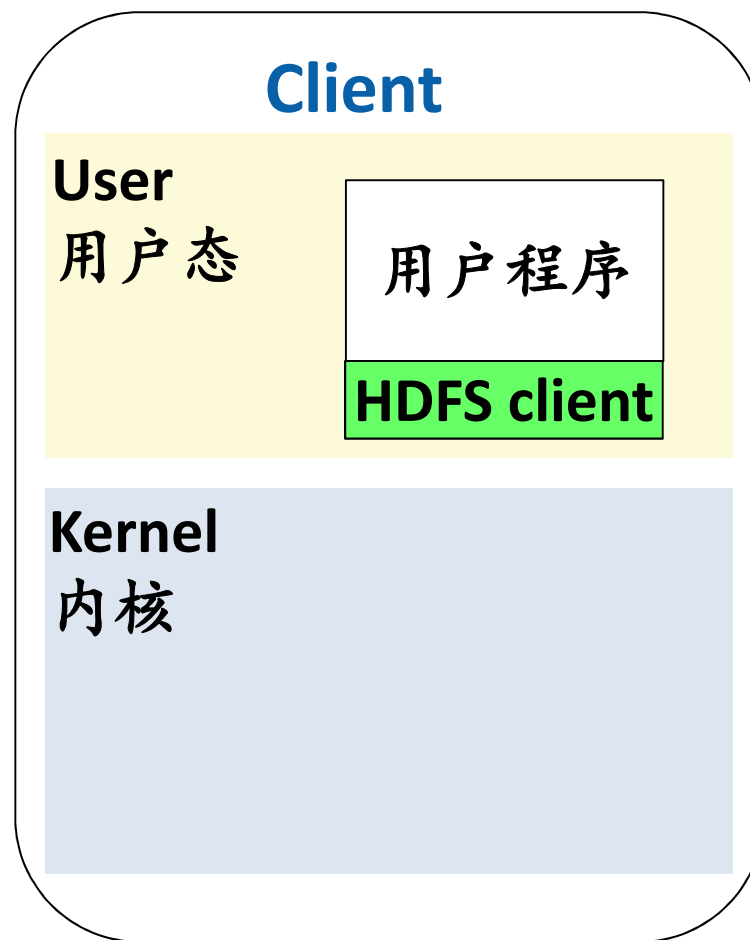
- Google File System的开源实现
- 基于Java
- 应用层的文件系统
- 与Hadoop捆绑在一起



POSIX文件系统 vs. 应用层文件系统



所有程序不改变就可以用



必须链接HDFS client库才能使用

GFS设计目标

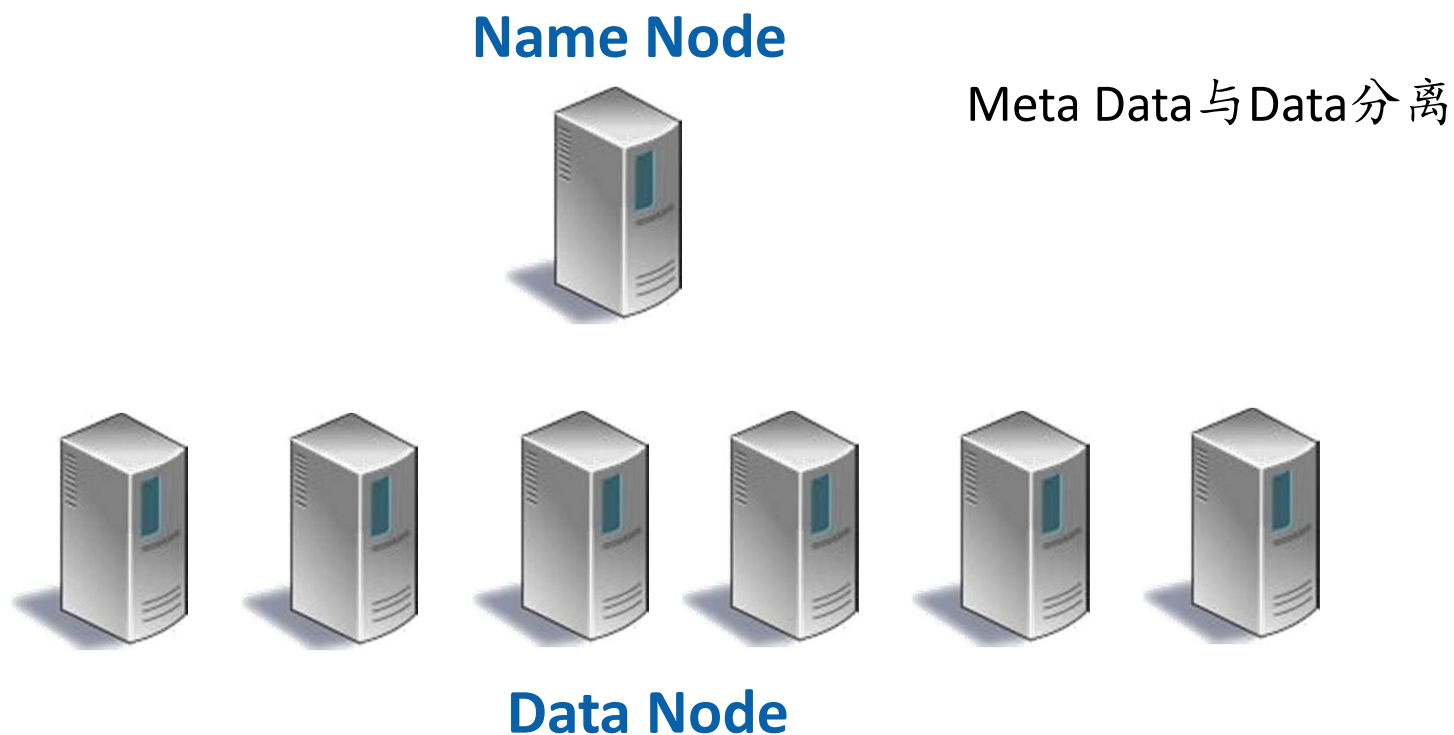
- 优化

- 大块数据的顺序读
- 并行追加(append)

- 不支持

- 文件修改(overwrite)操作
- 所以，consistency的实现可以大大简化！

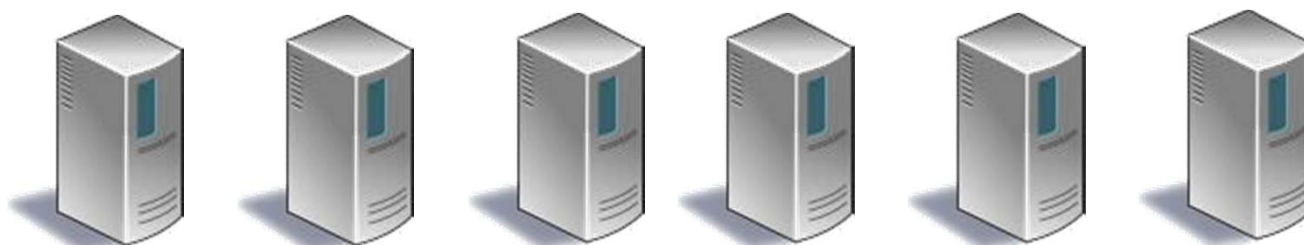
HDFS/GFS 系统架构



- Name Node: 存储文件的metadata(元数据)
 - 文件名, 长度, 分成多少数据块, 每个数据块分布在哪些Data Node上
- Data Node: 存储数据块

HDFS/GFS 系统架构

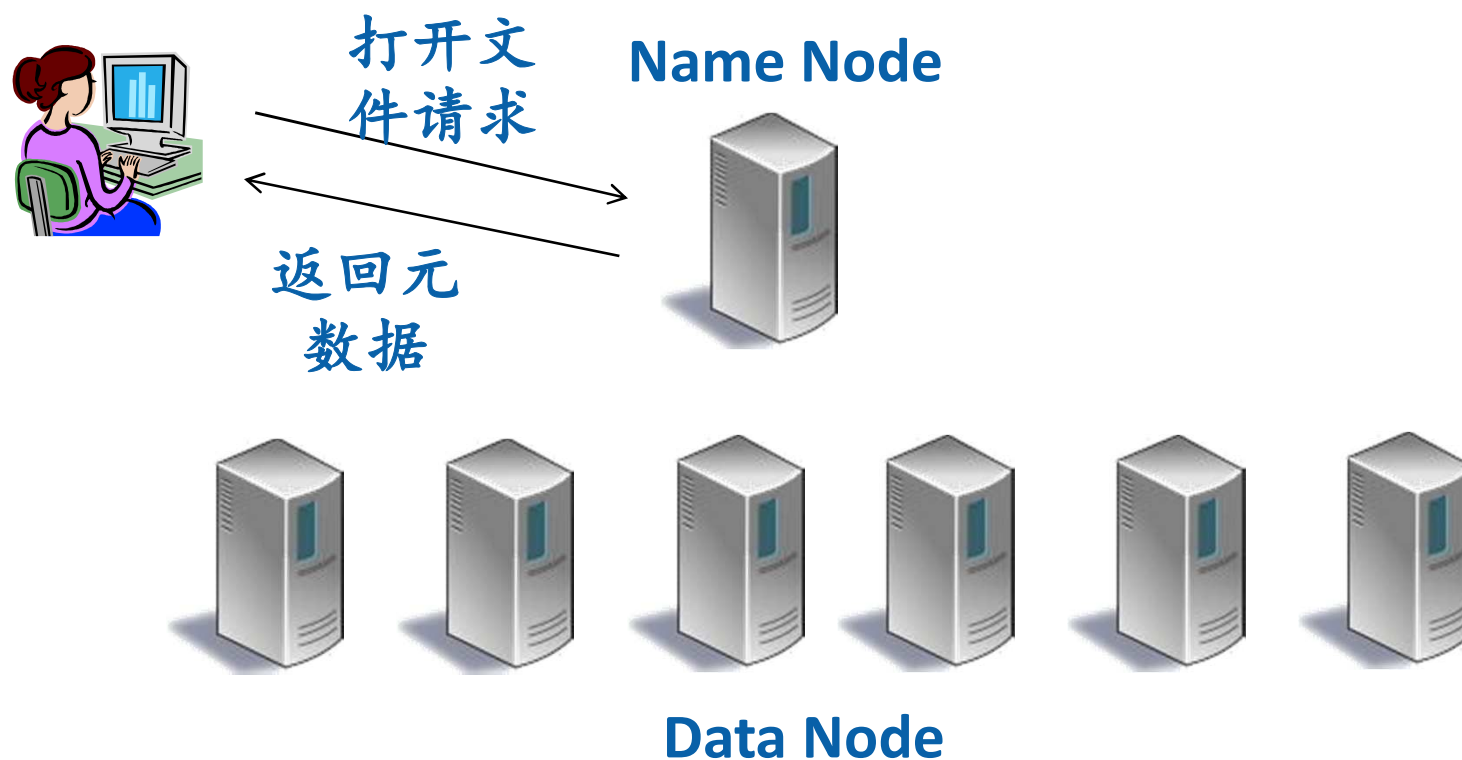
Name Node



Data Node

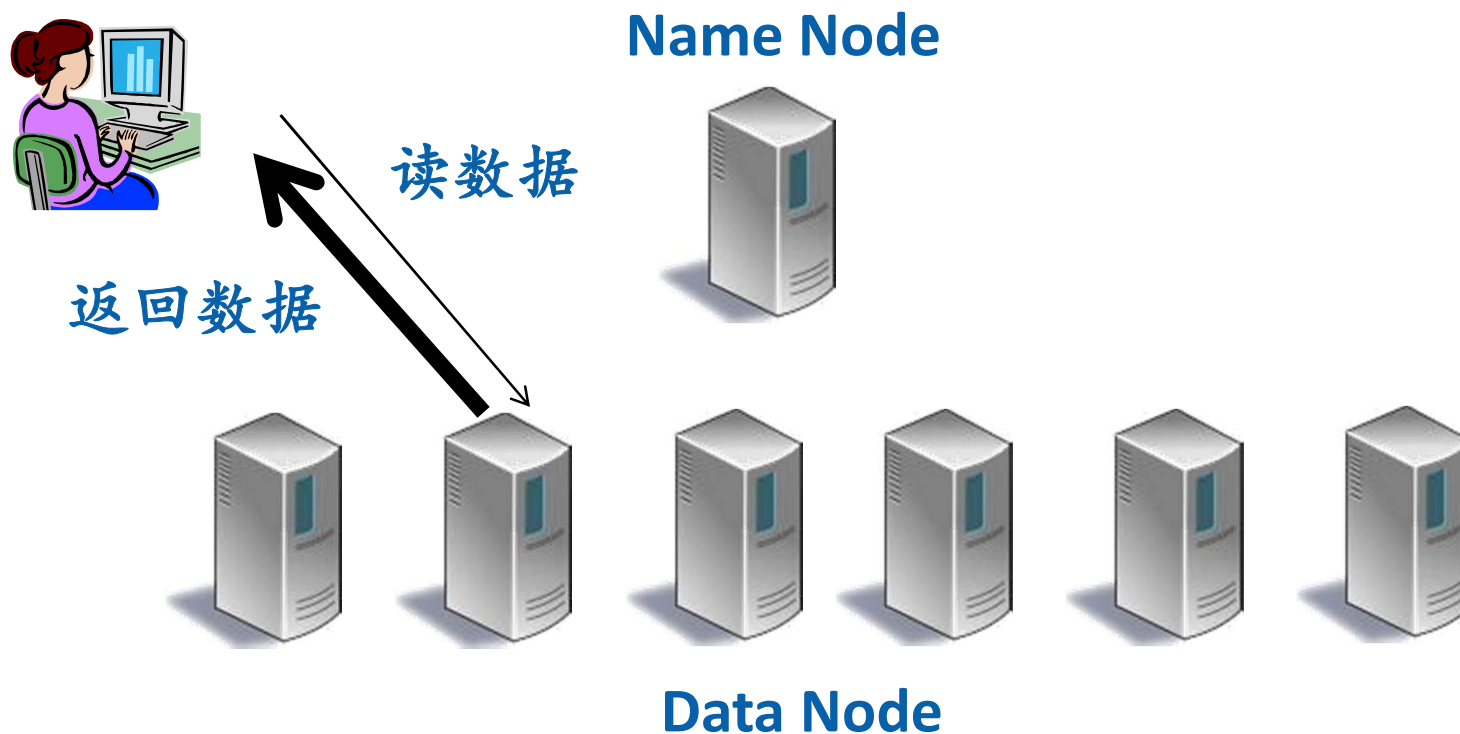
- 文件切分成定长的数据块（默认为64MB大小的数据块）
- 每个数据块独立地分布存储在Data Node上
- 默认每个数据块存储3份，在3个不同的data node上
 - Rack-aware

HDFS/GFS文件操作：open



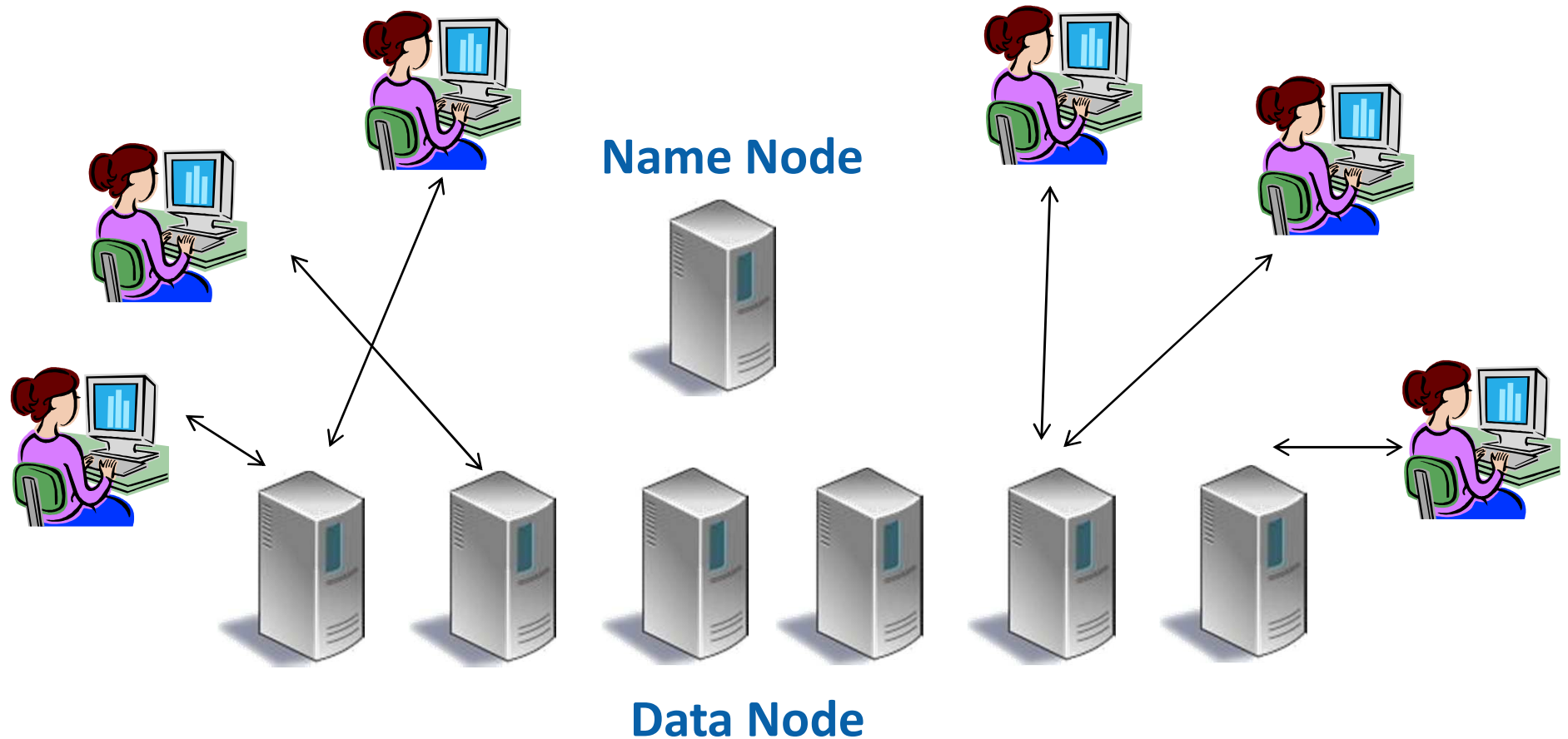
- 打开文件时，与Name Node通信一次

HDFS/GFS文件操作：read



- 之后的读操作，直接与Data Node通信，绕过了Name Node
- 可以从多个副本中选择最佳的Data Node读取数据

HDFS/GFS文件操作：read

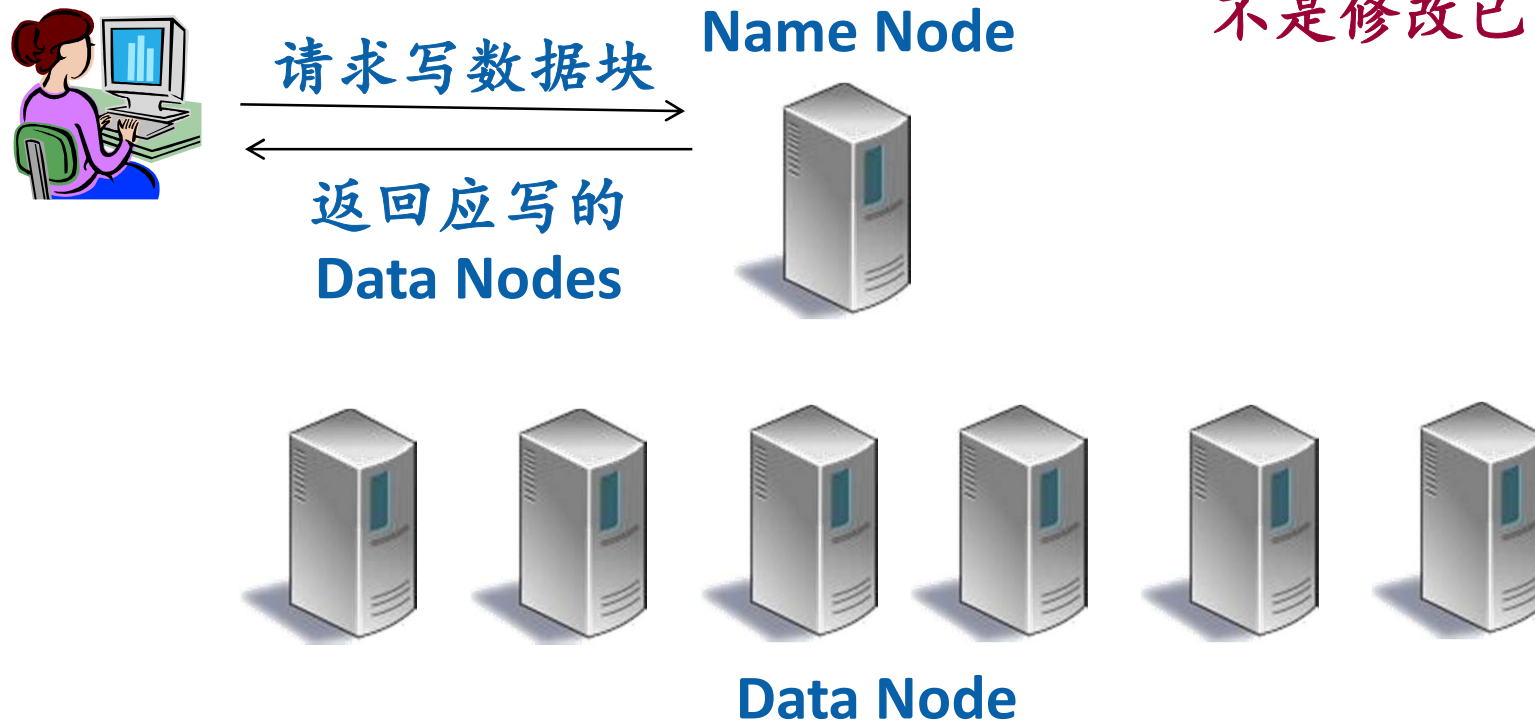


- 可以支持很多并发的读请求

HDFS/GFS文件操作：write(1)

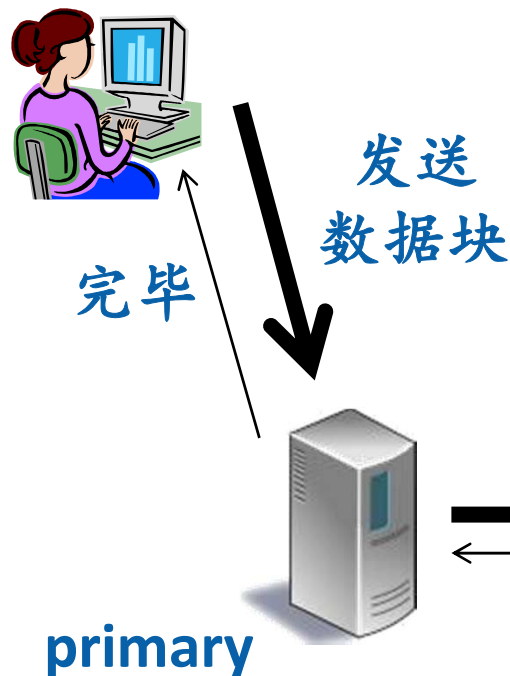
Client 可能是在一个
Data Node 所在的机器上

写新创建的文件
不是修改已有的文件！



- Name Node决定应该写到哪些Data Nodes
 - Rack-aware, load balancing
 - 3个副本：本机、本机柜、其它机柜

HDFS/GFS文件操作：write(2)

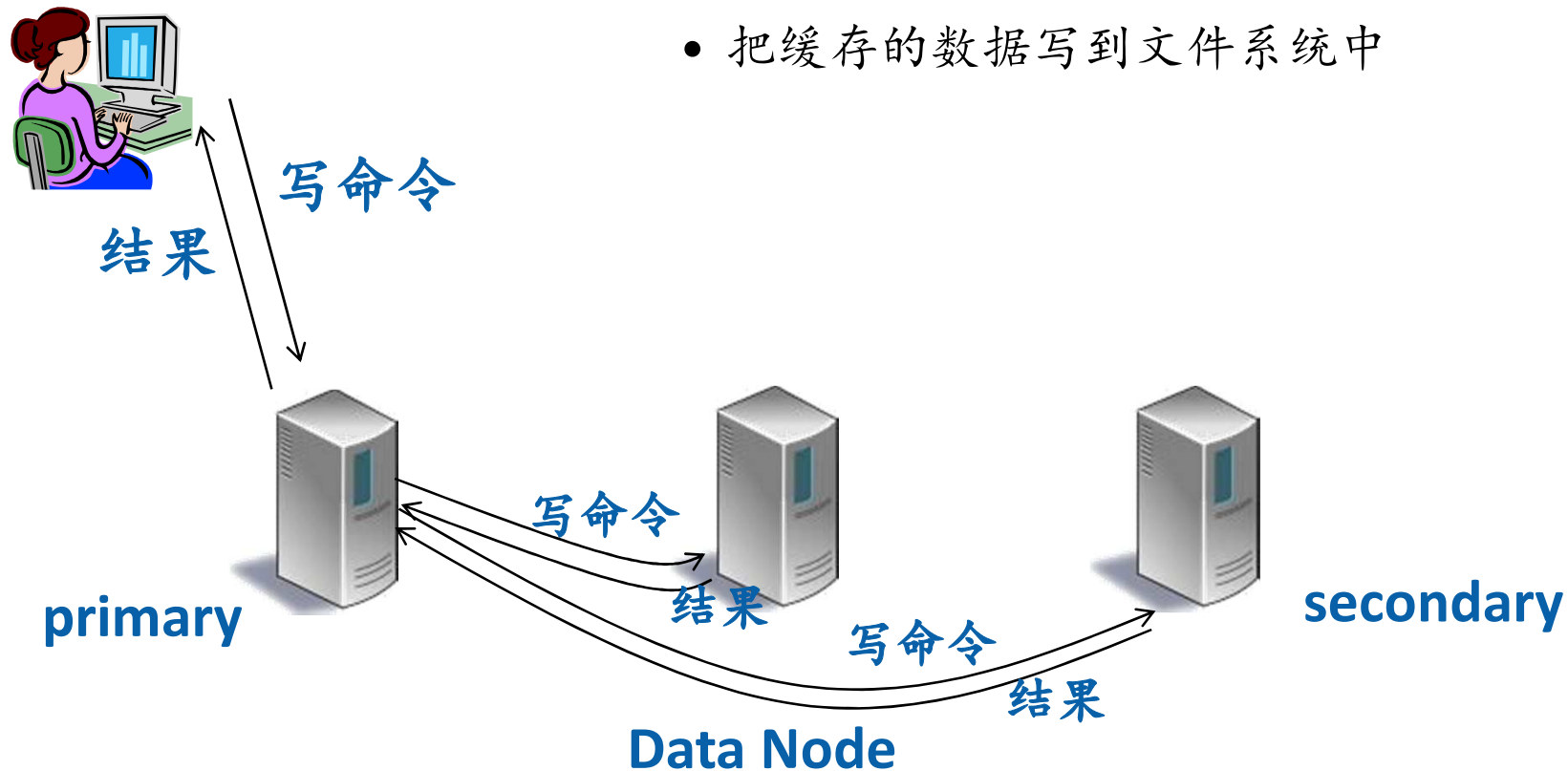


- 形成一个数据传递的pipeline
 - 数据依次沿流水线传递到Primary和secondary data node
 - 最大限度地利用网络带宽
- Data node 在内存中缓存数据
- 注意：数据到此时还没有写进HDFS

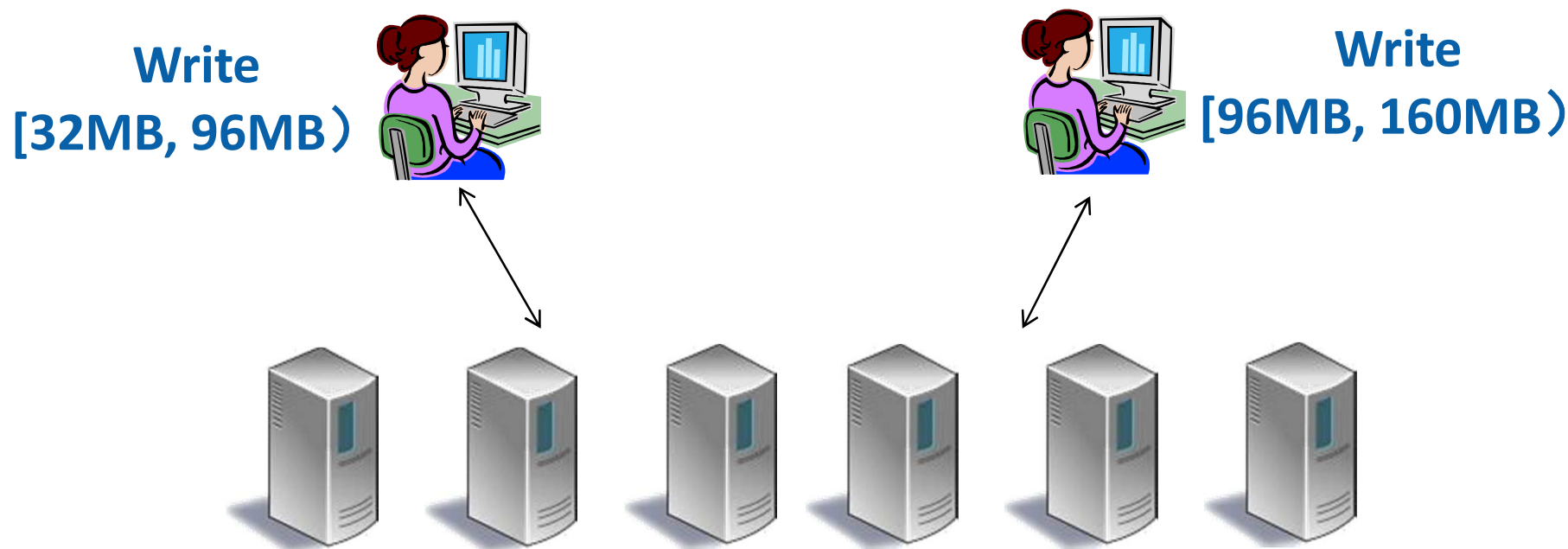
Data Node

HDFS/GFS文件操作：write(3)

- 收到写命令时才进行真正地写操作
- 把缓存的数据写到文件系统中

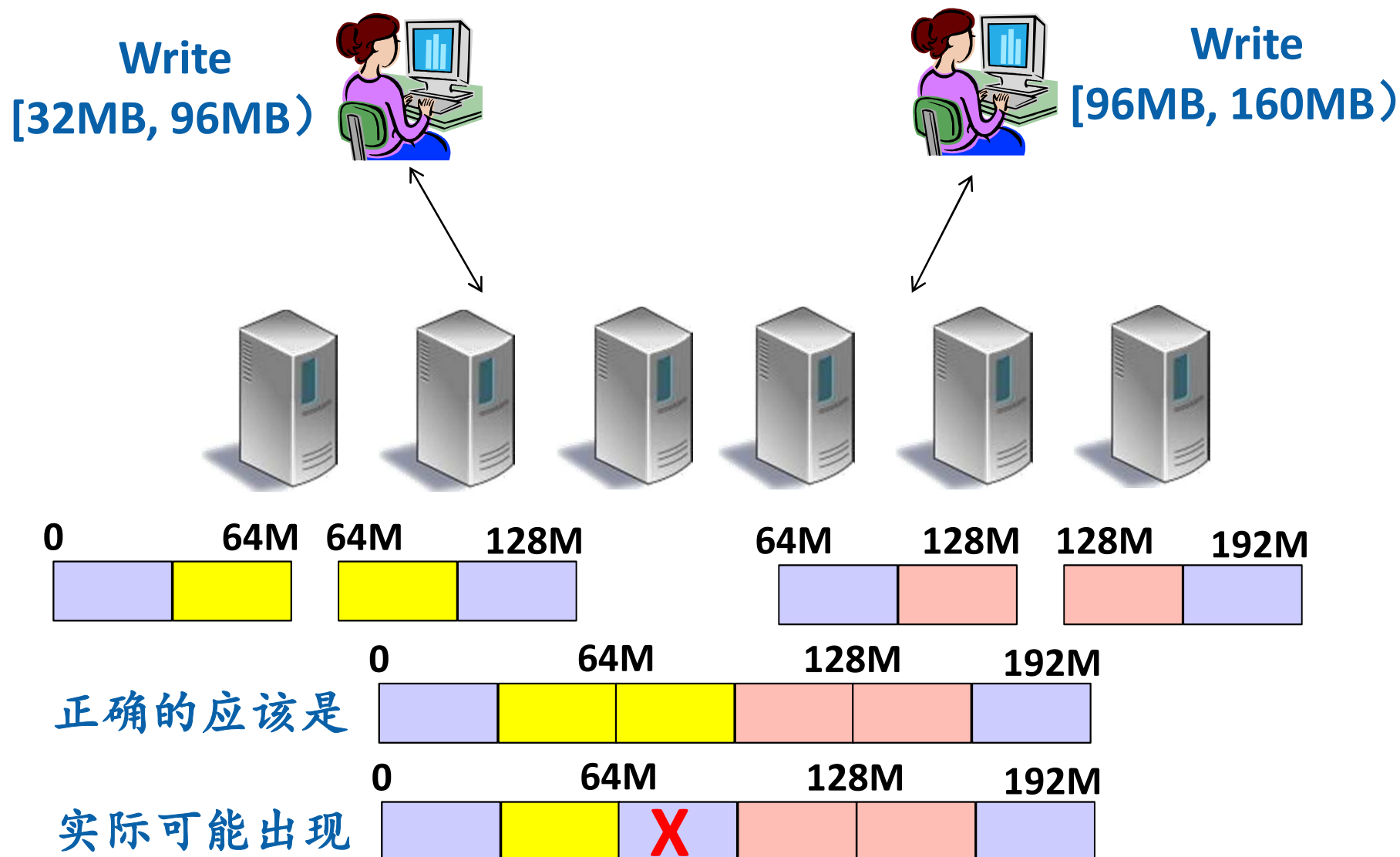


HDFS/GFS文件操作：并发写的问题

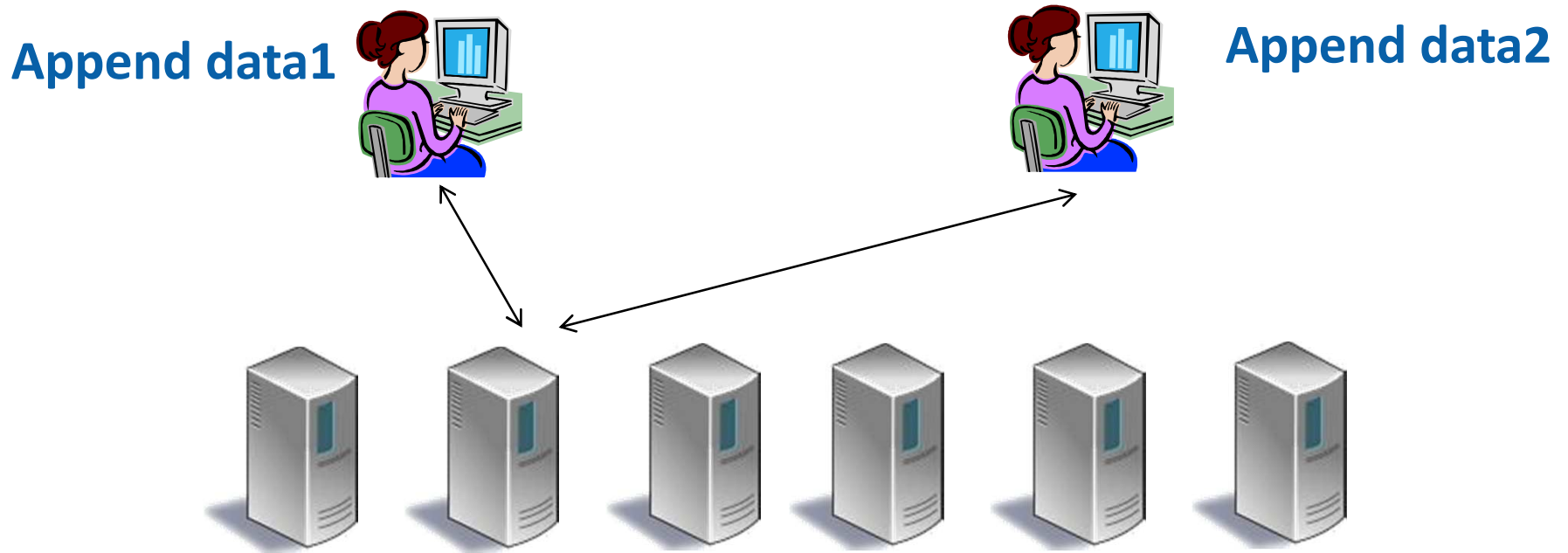


- 如果一个写操作跨了两个数据块
 - 那么就被分解成为两个独立的写数据块操作
 - 完全没有任何关于两个独立数据块写顺序的规定
 - 实际上是distributed transaction，有问题！

HDFS/GFS文件操作：并发写的问题



HDFS/GFS文件操作：并发Append



- 文件最后一个数据块在同一个primary data node
- 可以在单机上完成concurrency control
- 保证并行append成功，但是不保证append的顺序

HDFS/GFS小结

- 分布式文件系统
- 很好的顺序读性能
 - 为大块数据的顺序读优化
- 不支持并行的写操作：不需要distributed transaction
- 支持并行的append

总结

- 分布式系统基本概念

- 网络与协议
- 通信方式
- 分布式系统类型、故障类型、CAP

- 分布式文件系统

- NFS
- AFS

- Google File System和HDFS