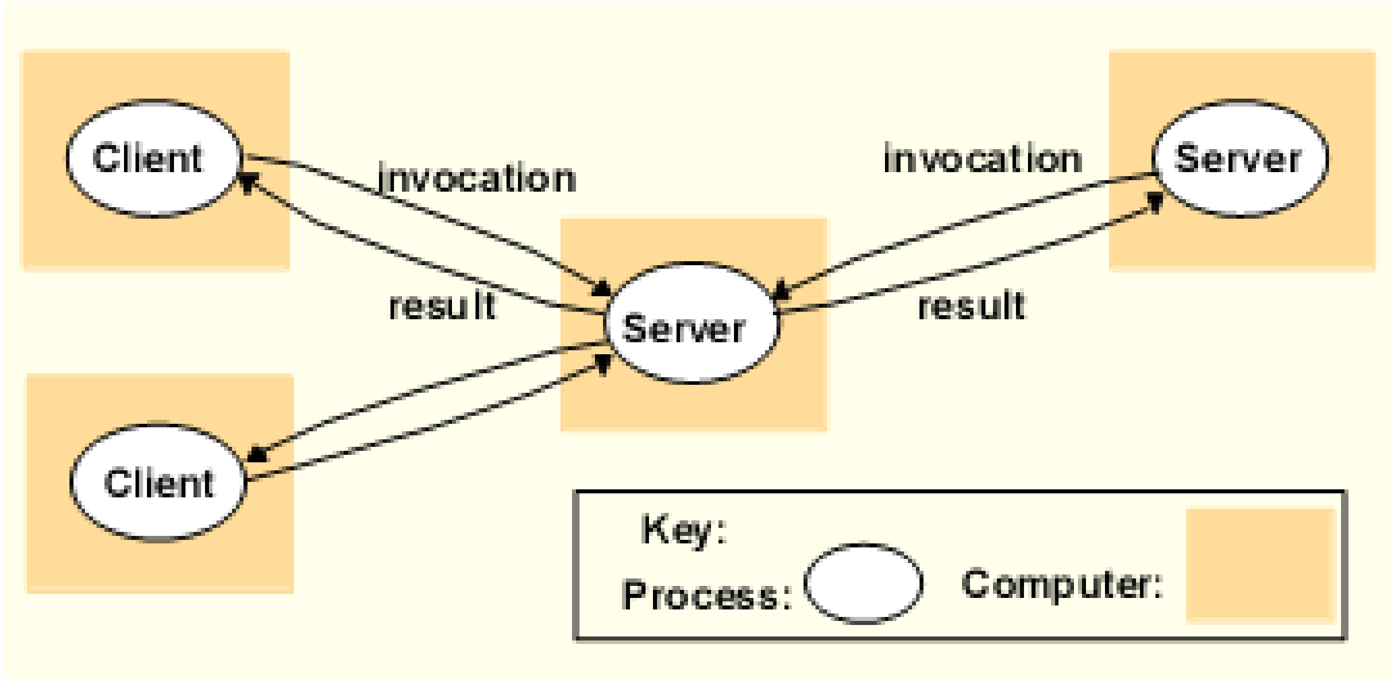
Review----体系结构元素

- 为了理解一个分布式系统的基础构建块，有必要考虑下面四个关键问题:
  - 在分布式系统中进行通信的实体是什么?
  - 它们如何通信，特别是使用什么通信范型?
  - 它们在整个体系结构中扮演什么(可能改变的)角色，承担什么责任?
  - 它们怎样被映射到物理分布式基础设施上(它们被放置在哪里)?

Review---- client/server架构

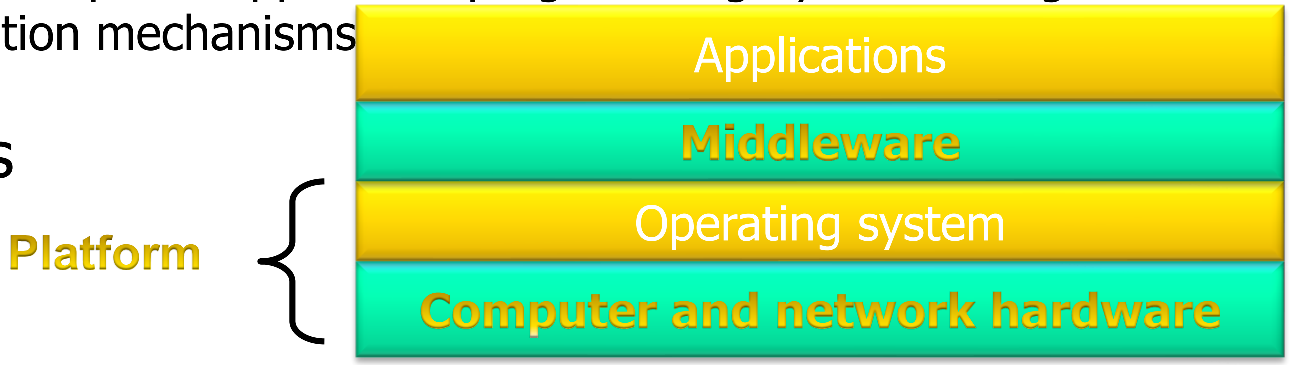
- 历史上最重要的结构之一，是**Internet**应用最常见的结构。



Review----
Layering – Platform and middleware

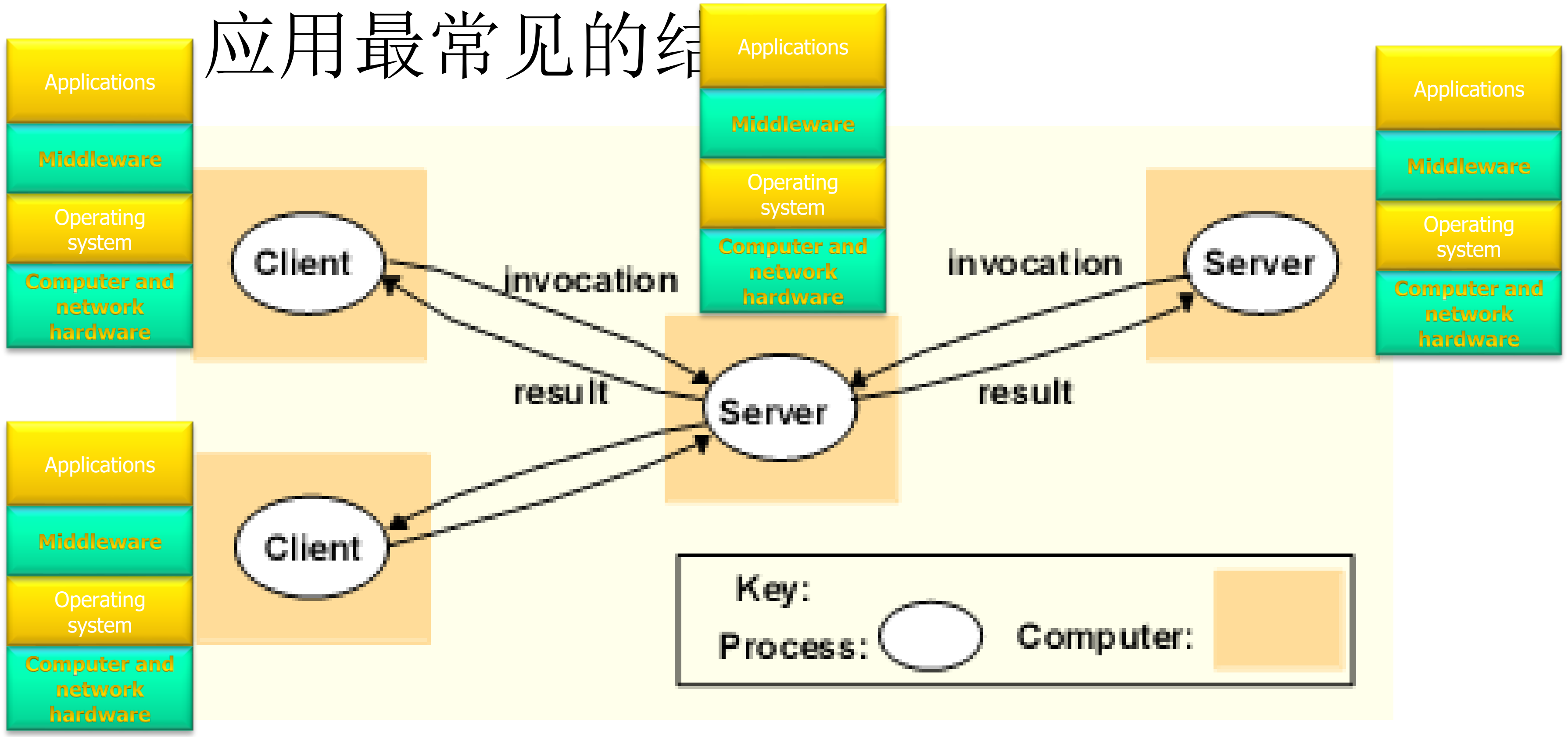
Distributed Systems can be organized into three layers

1. Platform
   - Low-level hardware and software layers
   - Provides common services for higher layers
2. Middleware
   - Mask heterogeneity and provide convenient programming models to application programmers
   - Typically, it simplifies application programming by abstracting communication mechanisms
3. Applications



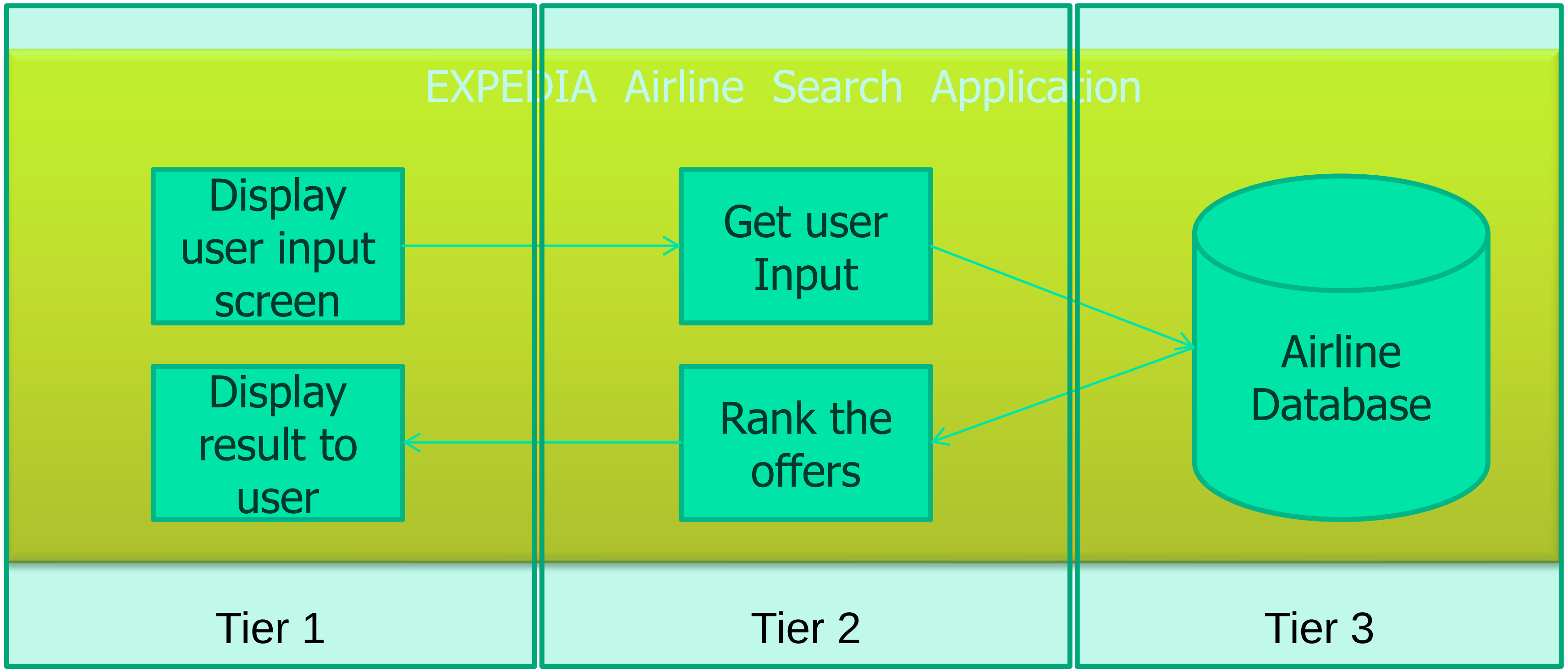
Review----client/server架构

- 历史上最重要的结构之一，是**Internet**应用最常见的结



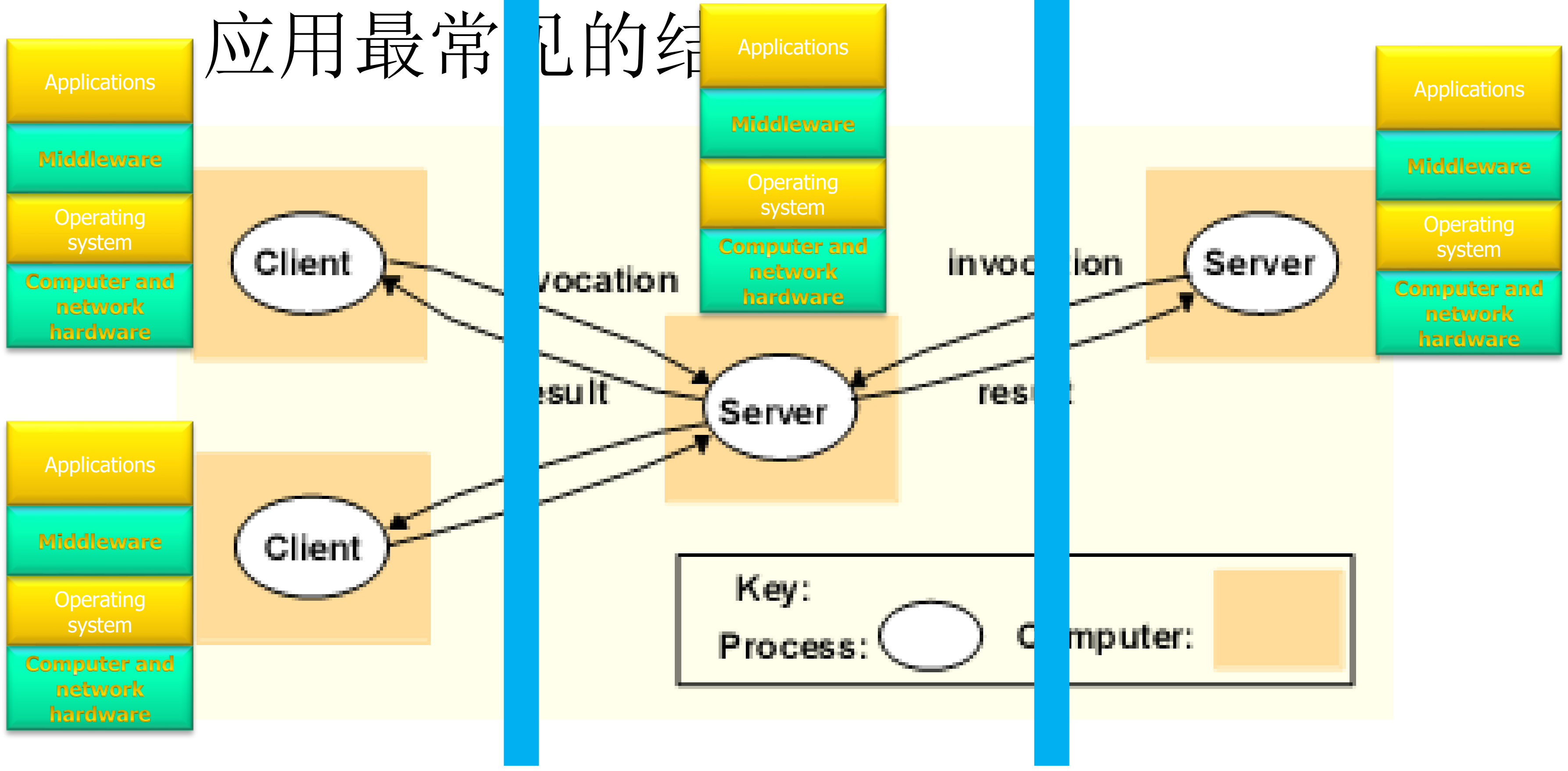
Review----
A Three-Tiered Architecture

- How do you design an airline search application:



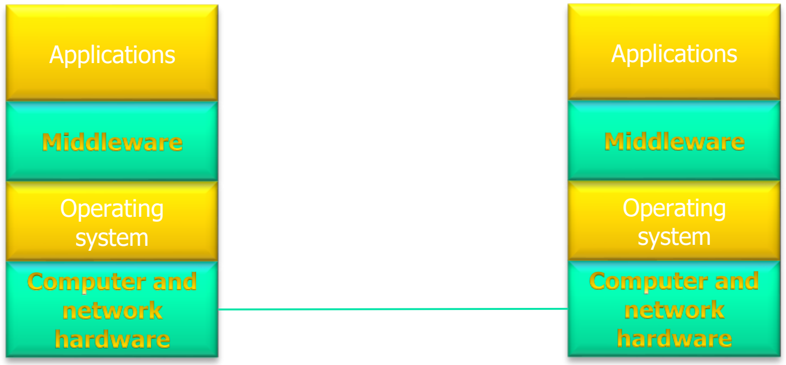
Review----client/server架构

- 历史上最重要的结构之一 是**Internet** 应用最常见的结



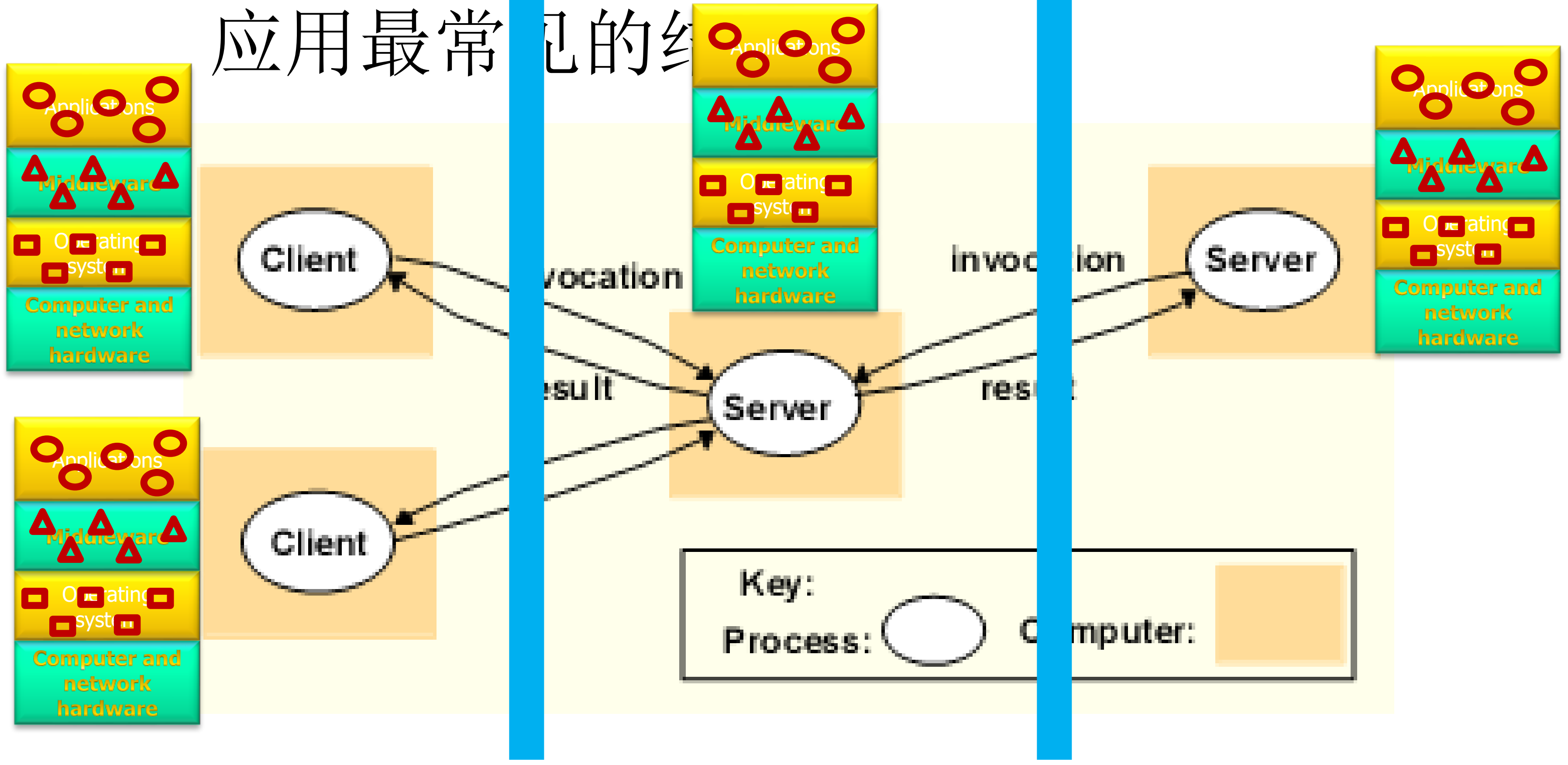
Review---- Communication Entities and Paradigms

Communicating entities (what is communicating)		***Communication Paradigms (how they communicate)***		
System-oriented	*Problem-oriented*	*IPC*	*Remote Invocation*	*Indirect Communication*
• Nodes • Processes • Threads	• Objects • 对象、组件、服务	• Sockets	• RPC • RMI	• Group communication • Publish-subscribe • Message queues

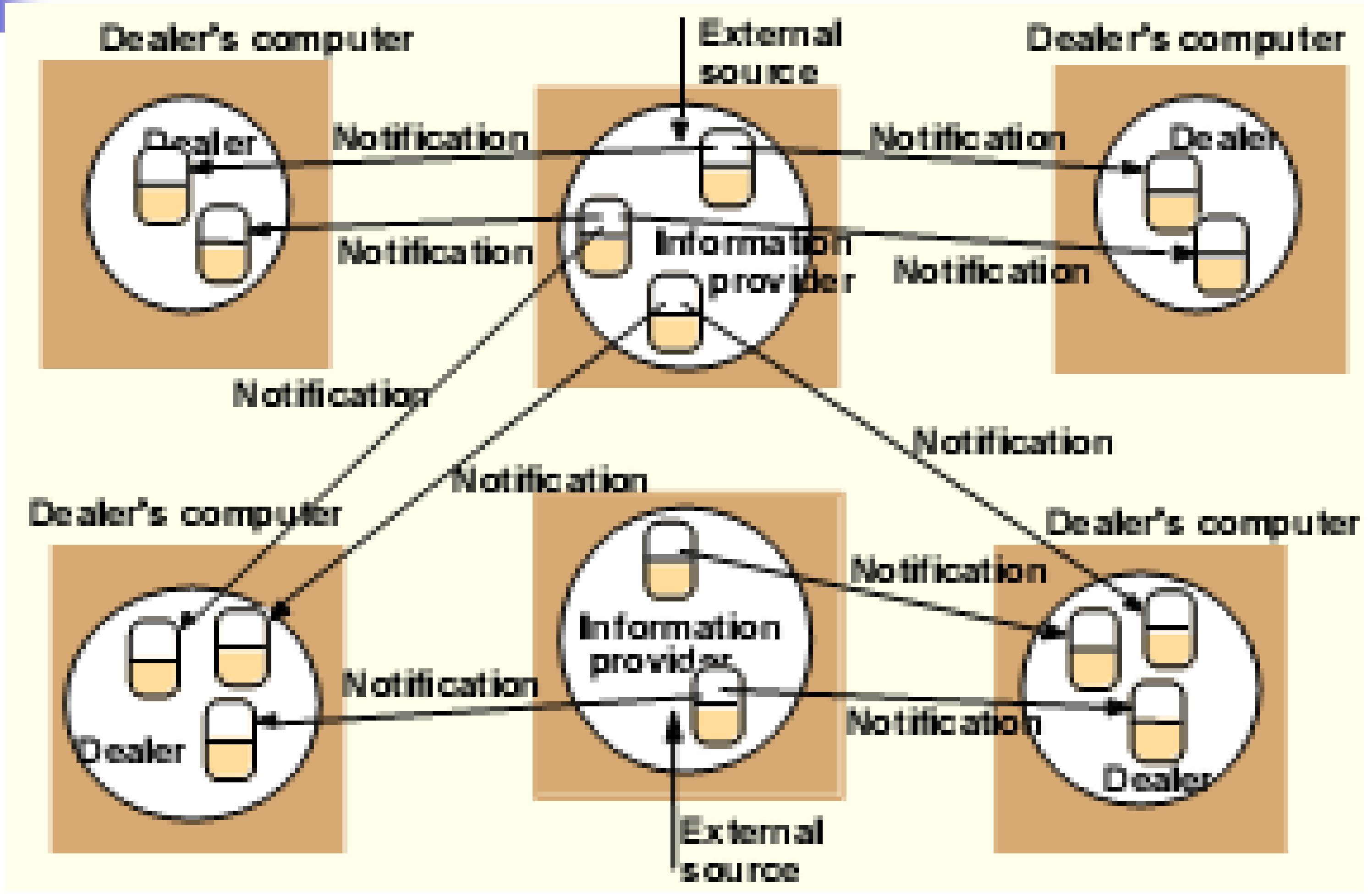


Review----client/server架构

- 历史上最重要的结构之一 是**Internet** 应用最常见的

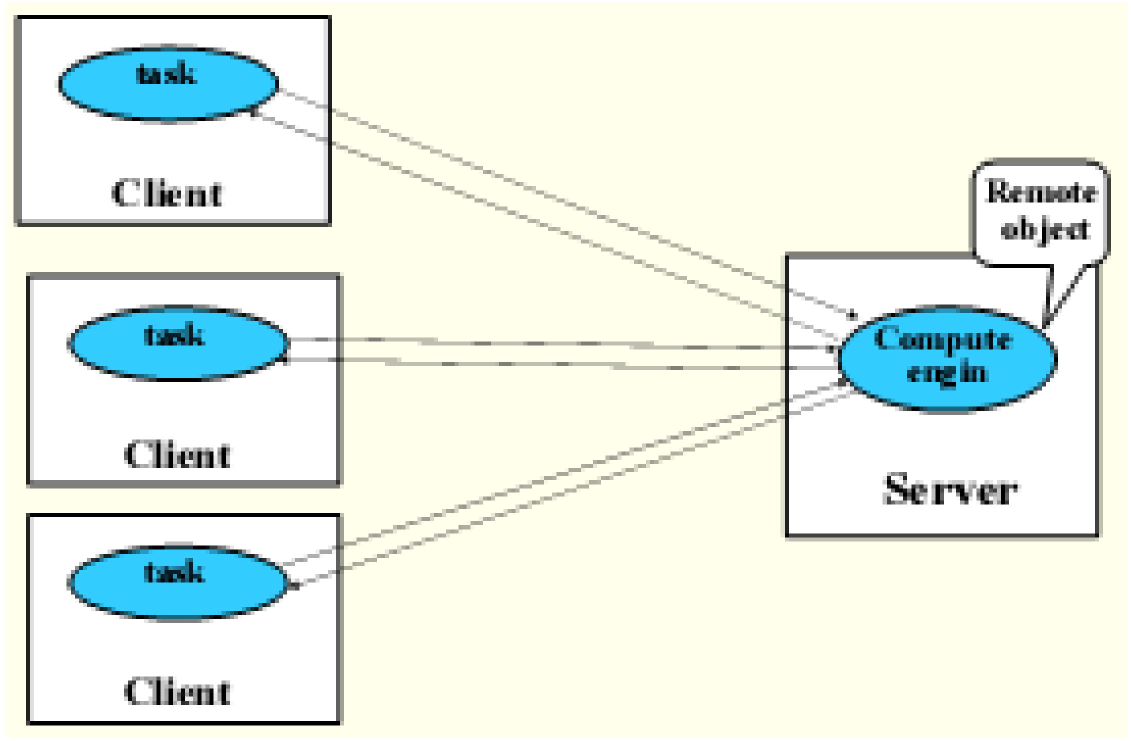


事件与通知模型


Dealer's computer
External source
Dealer's computer
Dealer
Notification
Notification
Dealer
Notification
Information provider
Notification
Notification
Notification
Notification
Dealer's computer
Dealer's computer
Notification
Information provider
Notification
Notification
Dealer
Dealer
External source

一个RMI的分布式应用的实例

- 一个分布式计算引擎提供计算能力，客户端将计算任务提交给引擎计算，提交的任务要包括:计算步骤，计算引擎将计算结果返回。



第3章 进程间通信

主要内容

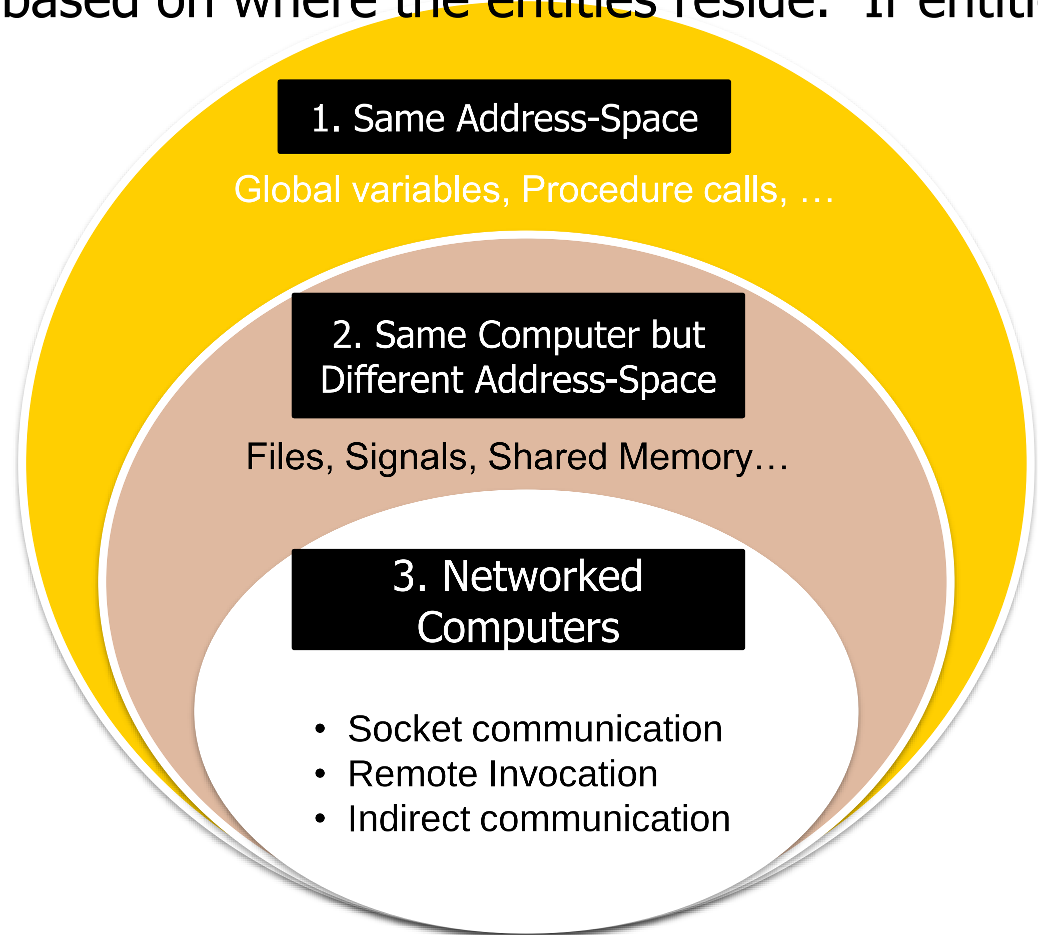
- 通信范型基础
  - 通信范型的分类
  - Internet协议的API （编程模型）
  - 外部数据的表示和编码
- 第一类：客户—服务器通信
- 第二类：远程调用
- 第三类：间接通信
- 总结

主要内容

- 通信范型基础
  - 通信范型的分类
  - Internet协议的API （编程模型）
  - 外部数据的表示和编码
- 第一类：客户—服务器通信
- 第二类：远程调用
- 第三类：间接通信
- 总结

Classification of Communication Paradigms

- Communication Paradigms can be categorized into three types based on where the entities reside. If entities are running on:


1. Same Address-Space
Global variables, Procedure calls, …
2. Same Computer but Different Address-Space
Files, Signals, Shared Memory…
3. Networked Computers
• Socket communication
• Remote Invocation
• Indirect communication

Today, we are going to study how entities that reside on networked computers communicate in Distributed Systems

- **Socket communication**
- **Remote Invocation**
- **Indirect communication**

Communication Paradigms

- Socket communication （C/S）
  - Low-level API for communication using underlying network protocols
- Remote Invocation
  - A procedure call abstraction for communicating between entities
- Indirect Communication
  - Communicating without direct coupling between sender and receiver

主要内容

- 通信范型基础
  - 通信范型的分类
  - Internet协议的API （编程模型）
  - 外部数据的表示和编码
- 第一类：客户—服务器通信
- 第二类：远程调用
- 第三类：间接通信
- 总结

Internet协议的API （编程模型）

- 通信的目的　与　socket
- UDP与TCP
  - 连接与无连接的对比
- 同步通信与异步通信
- 五种I/O模式

Internet协议的API （编程模型）

- 通信的目的　与　socket
- UDP与TCP
  - 连接与无连接的对比
- 同步通信与异步通信
- 五种I/O模式

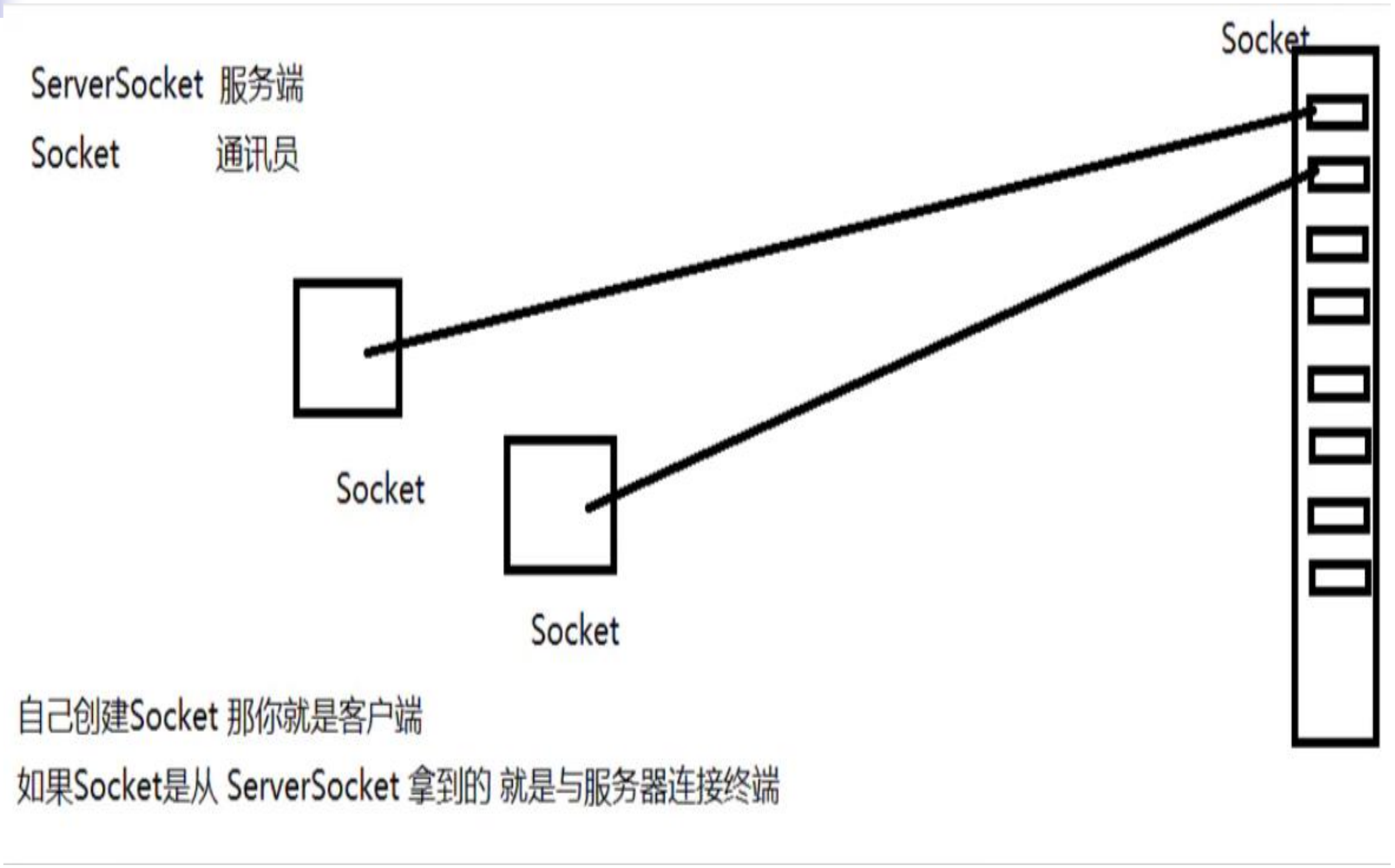
通信的目的

- 消息的目的地
  - Internet address + local port
    - 每个端口都有一个进程，一 个进程可以对应多个端口，任何一个进程只要知道端口号，都可以发送消息
  - 服务器提供服务，对外公布端口号，然后守候消息的到来
  - 服务的名字：在运行时借助于名字服务
  - 操作系统提供与位置无关的标识符对应底层的端口
- 可靠性（Reliability)
  - 有效性：消息最终被传递到相应的进程，尽管会有一些丢包现象发生。
  - 完整性：到达目的地的消息和发出的消息是完全一致的， 并且没有重复
- 有序性（Ordering）
  - 接收消息的顺序和发送的顺序相同

IP地址、协议与端口

- Sockets
- 进程间通信的端口
- 两种通信协议 (UDP and TCP)使用 socket 抽象
- 绑定到一个局部的端口 (216 possible port number) 和一个IP地址
- **同一台机器**上的进程不能共用一个端口号

Socket



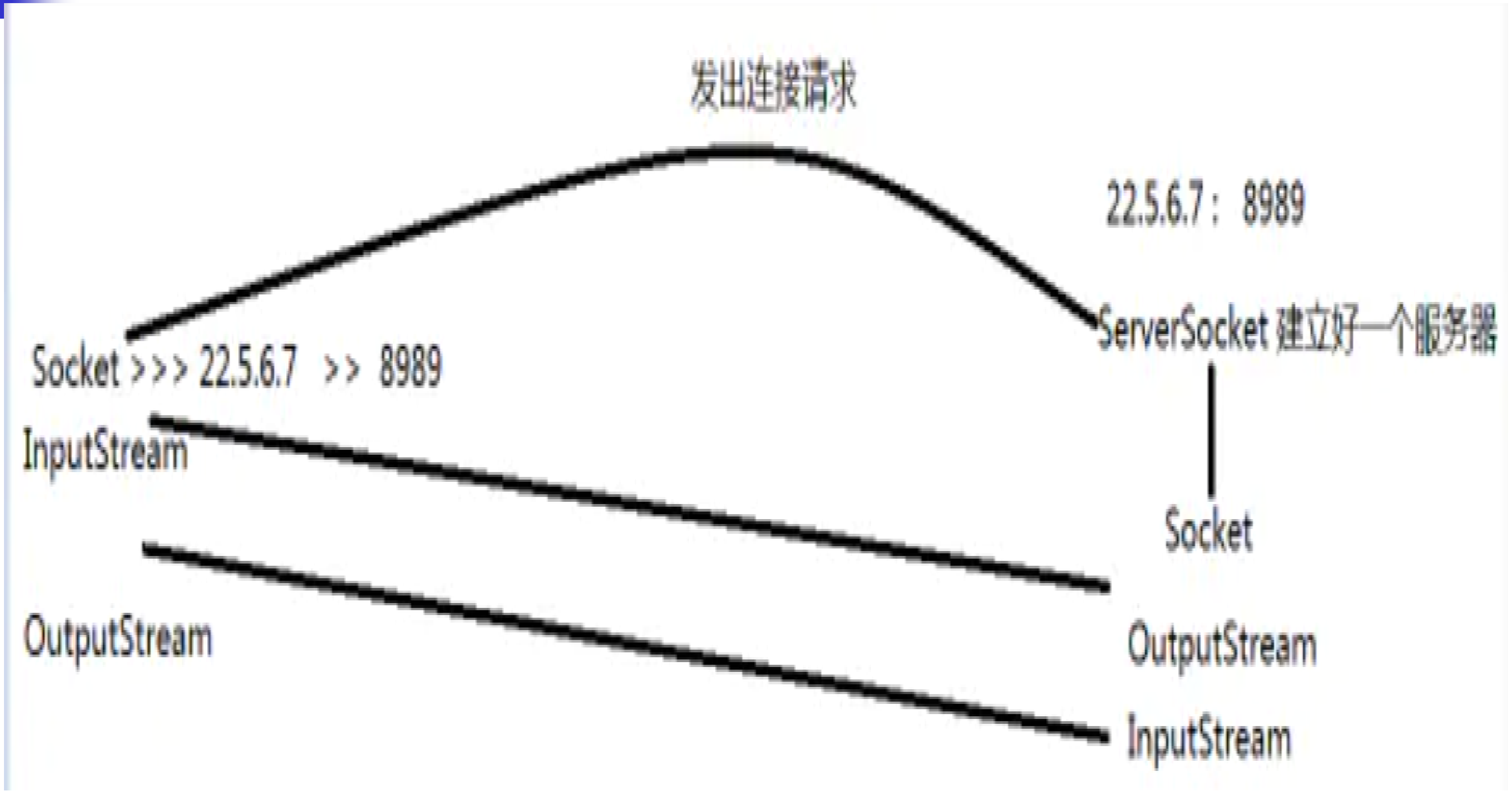
ServerSocket 服务端

Socket 通讯员

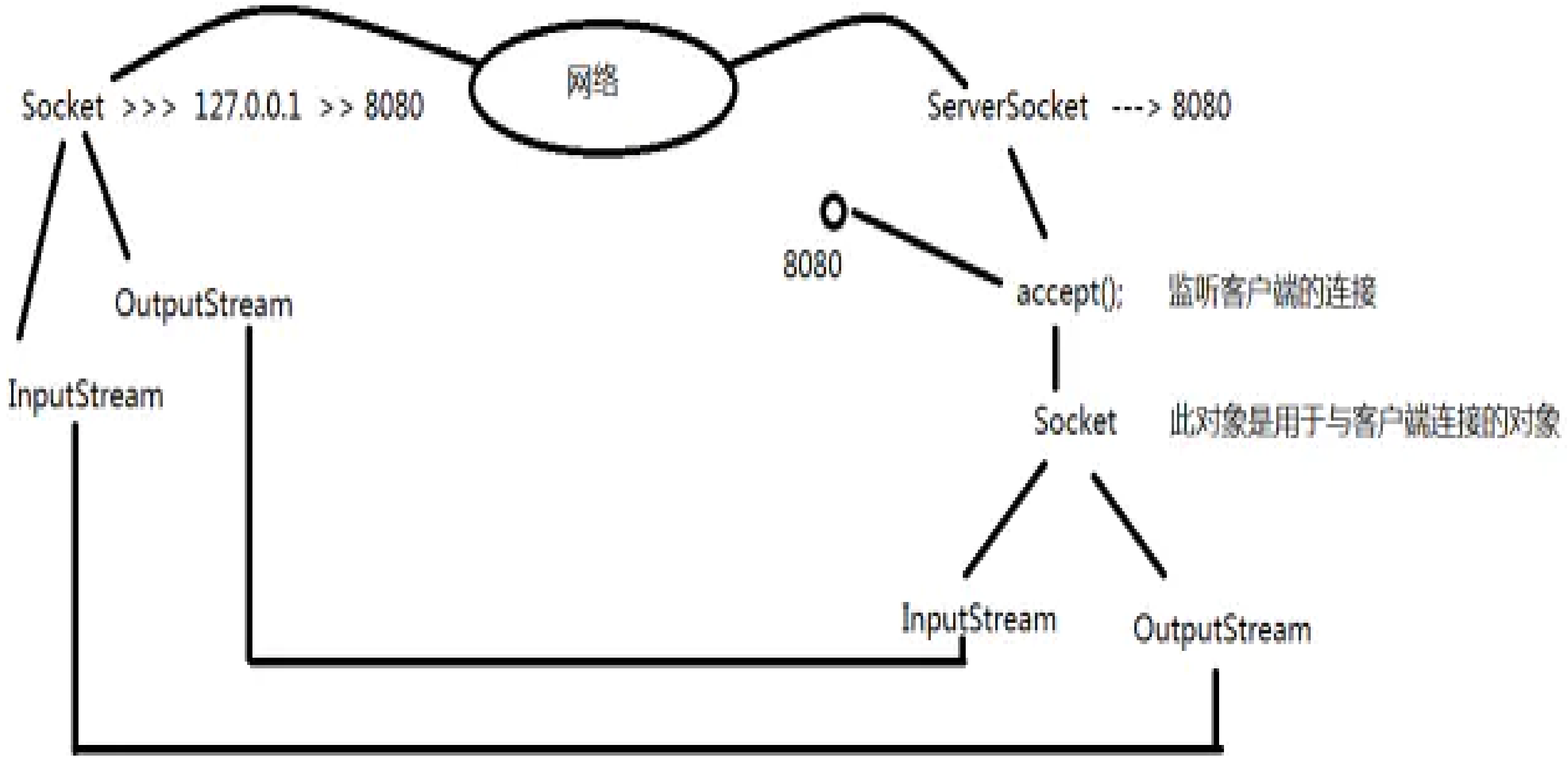
自己创建Socket 那你就是客户端

如果Socket是从 ServerSocket 拿到的 就是与服务器连接终端

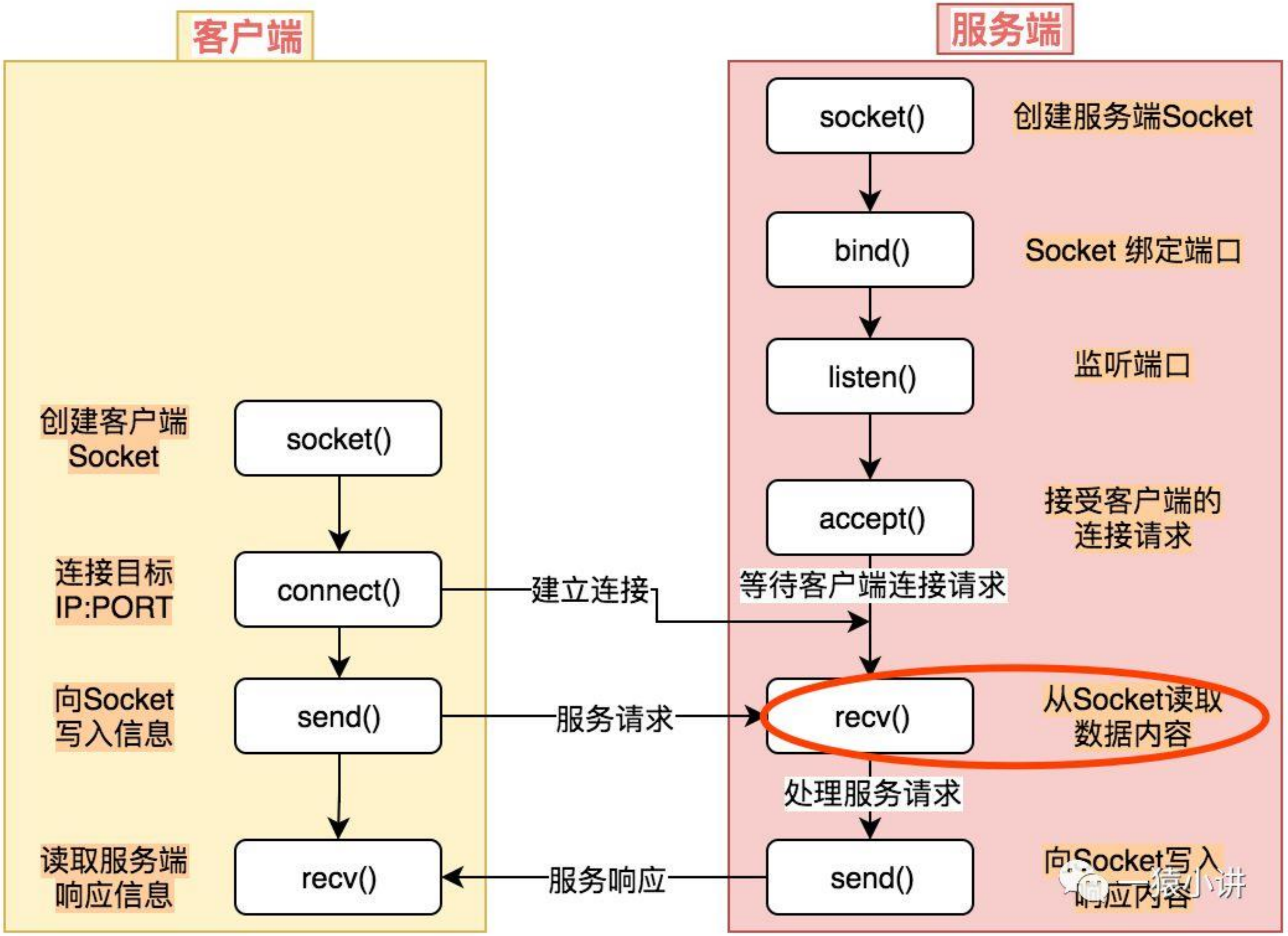
JAVA socket连接建立过程



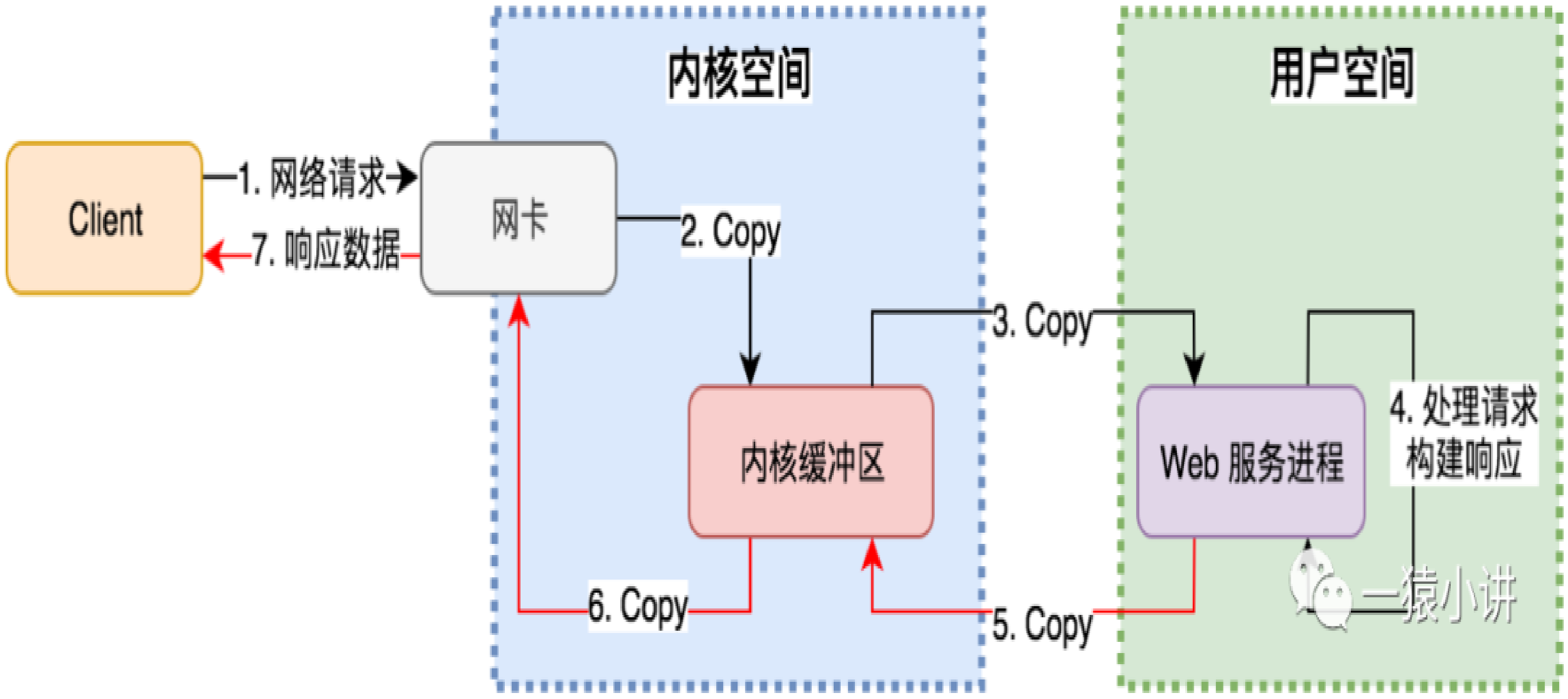
完整示意图


Socket >>> 127.0.0.1 >> 8080
网络
ServerSocket ---> 8080
8080
accept(); 监听客户端的连接
OutputStream
InputStream
Socket 此对象是用于与客户端连接的对象
InputStream
OutputStream

UNIX socket连接与通信



数据从网卡到处理的过程

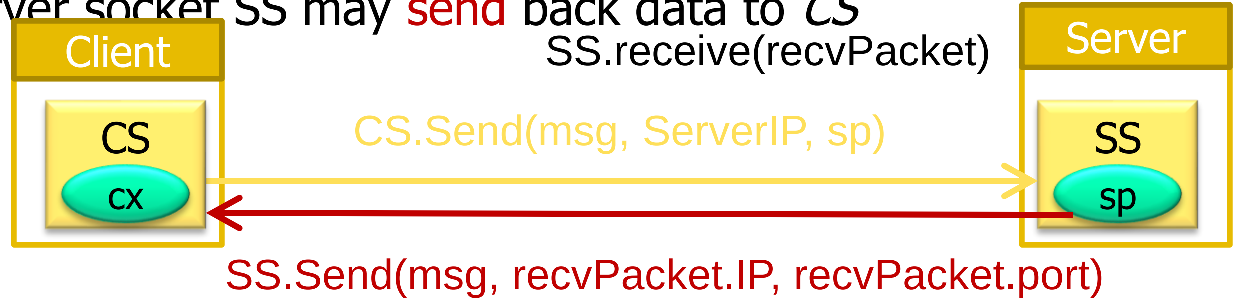


Internet协议的API （编程模型）

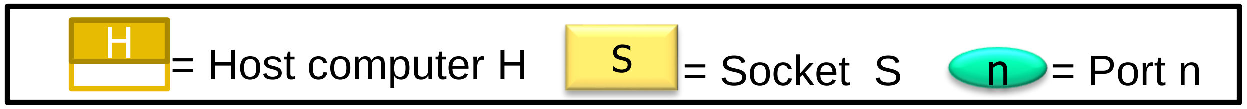
- 通信的目的　与　socket
- UDP与TCP
  - 连接与无连接的对比
- 同步通信与异步通信
- 五种I/O模式

1. UDP Sockets

- Messages are sent from sender process to receiver process using UDP protocol.
  - UDP provides connectionless communication,
  - With no acknowledgements or message transmission retries
- Communication mechanism:
  - Server opens a UDP socket *SS* at a known port *sp*,
  - Socket *SS* waits to receive a request
  - Client opens a UDP socket *CS* at a random port *cx*
  - Client socket *CS* sends a message to ServerIP and port *sp*
  - Server socket SS may send back data to *CS*


Client
CS
cx
SS.receive(recvPacket)
CS.Send(msg, ServerIP, sp)
Server
SS
sp
SS.Send(msg, recvPacket.IP, recvPacket.port)

No ACK will be sent by the receiver



UDP 报文—设计考虑

- UDP 报文通信
  - 无确认、无重发
- UDP 报文通信的特点
  - Message size: 不大于 64k，多出的部分被截取
  - blocking:
    - non-blocking sends (如果到了接收端的消 息找不到对应的端口号，消息将被丢弃。
  - Timeout: 接收端限制。
  - Receive from any: 一个socket，对发送端没有限制

UDP Sockets – Design Considerations

- Messages may be delivered out-of-order
  - If necessary, programmer must re-order packets
- Communication is not reliable
  - Messages might be dropped due to check-sum error or buffer overflows at routers
- **Sender** must explicitly fragment a long message into smaller chunks before transmitting
  - A maximum size of 548 bytes is suggested for transmission
- **Receiver** should allocate a buffer that is big enough to fit the sender's message
  - Otherwise the message will be truncated

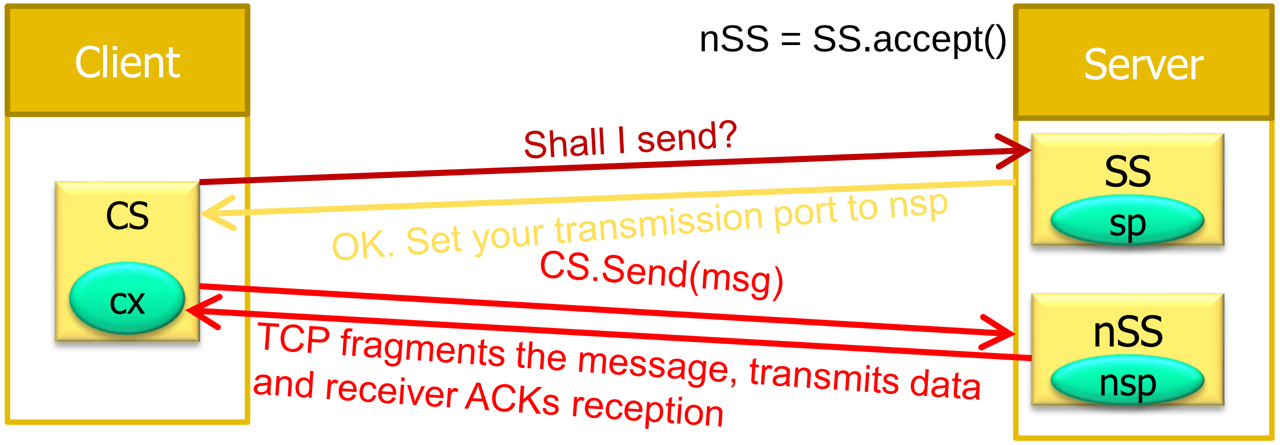
UDP 报文通信故障模型

- 缺失故障
  - 如果发现校验和不正确或和发送双方缓冲区不够，会导致消息的丢失
- 乱序故障
  - 接收顺序和发送的顺序不一致
- 没有任意故障，无法保证有效性，但是可以保证完整性
- 应用程序可以在UDP上实现可靠的通信
- 有些应用可以容忍偶发的缺失故障，因此可以得到更高的效率。
  - 例如：DNS服务，音视频文件。

2. TCP Sockets

- Messages are sent from sender to receiver using TCP protocol
  - TCP provides in-order delivery, reliability and congestion control

- Communication mechanism
  - Server opens a TCP server socket *SS* at a known port *sp*
  - Server waits to receive a request (using *accept* call)
  - Client opens a TCP socket *CS* at a random port *cx*
  - *CS* initiates a connection initiation message to ServerIP and port *sp*
  - Server socket SS allocates a new socket NSS on random port *nsp* for the client
  - *CS* can send data to *NSS*


Client
nSS = SS.accept()
Server
Shall I send?
SS
sp
CS
OK. Set your transmission port to nsp
CS.Send(msg)
cx
nSS
nsp
TCP fragments the message, transmits data and receiver ACKs reception

Advantages of TCP Sockets

- TCP Sockets ensure in-order delivery of messages
- Applications can send messages of any size
- TCP Sockets ensure reliable communication using acknowledgements and retransmissions
- Congestion control of TCP regulates sender rate, and thus prevents network overload

TCP流 通信

- The API to the TCP
  - 提供一个字节流的抽象，其中可以写数据和读数据
- 这种通信模式隐含了以下的网络特征：
  - 不考虑消息的尺寸
  - 不考虑消息丢失的问题
  - 流控制
  - 消息的重复和乱序
  - 消息的目的地

流通信相关的问题

- TCP流通信
- 与流通信相关的问题：
  - 数据项的匹配： 通信双方需要对流上传输的数据内容达成一致
  - 阻塞:
    - 发送操作后被阻塞直到数据到达接收方的缓冲区,
    - 接收操作被阻塞直到数据到达缓冲区
  - 线程：服务器每当收到一个连结的请求，它创建一个线程。

TCP流通信的故障模型

- TCP协议保证可靠通信所要求的完整性和有效性。
- 连结可能会由于一些未知的故障遭到破坏。
  - 无法区分是网络故障还是进程的故障
  - 无法知道最近发出去的消息是否被接收端收到

Internet协议的API （编程模型）

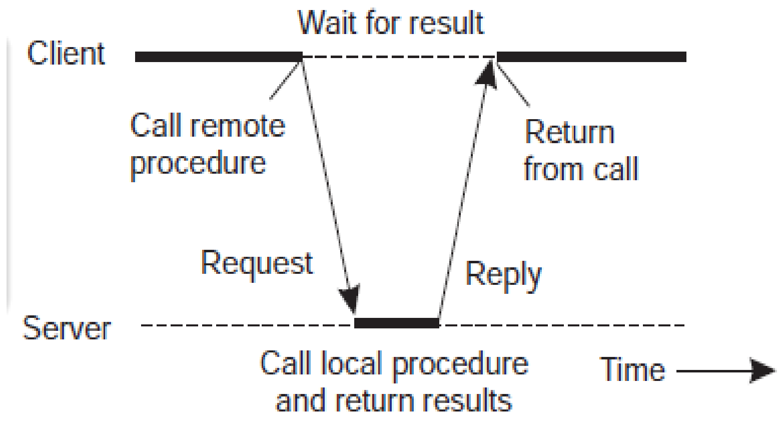
- 通信的目的　与　socket
- UDP与TCP
  - 连接与无连接的对比
- 同步通信与异步通信
- 五种I/O模式

同步和异步IO 阻塞和非阻塞IO

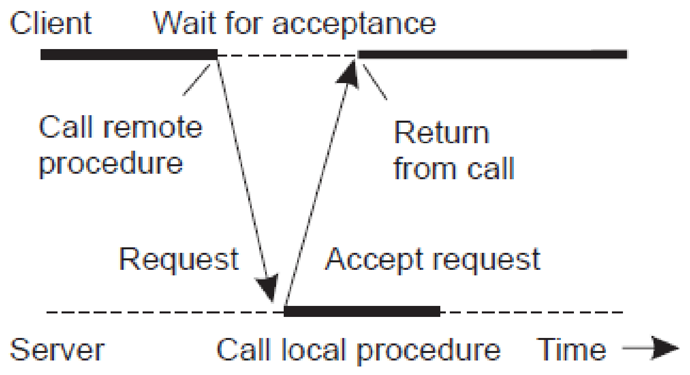
- 同步和异步IO的概念：
  - 同步是用户线程发起I/O请求后需要等待或者轮询内核I/O操作完成后才能继续执行
  - 异步是用户线程发起I/O请求后仍继续执行，当内核I/O操作完成后会通知用户线程，或者调用用户线程注册的回调函数
- 阻塞和非阻塞IO的概念：
  - 阻塞是指I/O操作需要彻底完成后才能返回用户空间
  - 非阻塞是指I/O操作被调用后立即返回一个状态值，无需等I/O操作彻底完成

举例：Synchronous vs. Asynchronous RPCs

- An RPC with strict request-reply blocks the client until the server returns
  - Blocking wastes resources at the client
- Asynchronous RPCs are used if the client does not need the result from server
  - The server immediately sends an ACK back to client
  - The client continues the execution after an ACK from the server

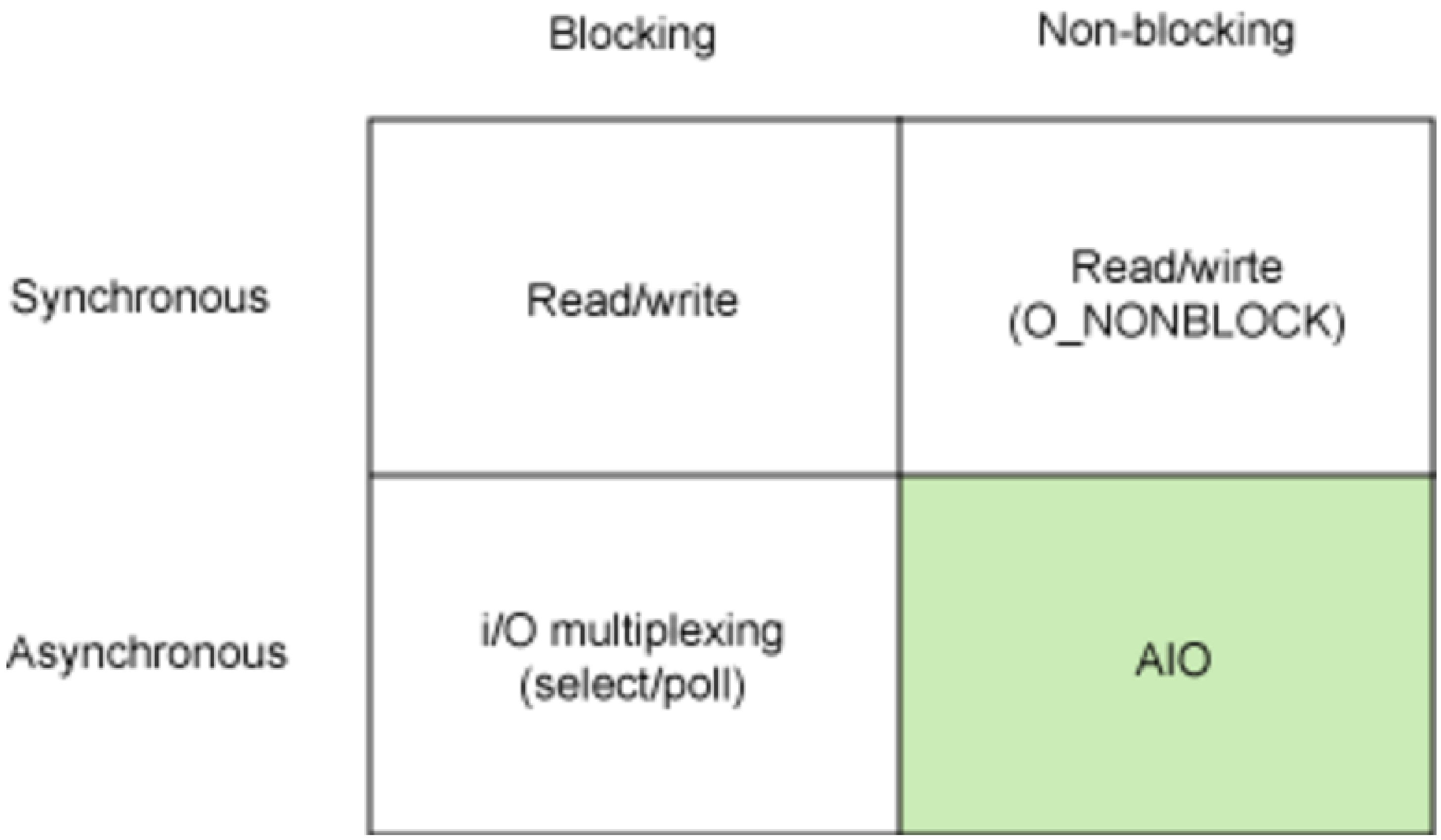


Synchronous RPCs



Asynchronous RPCs

四种通信方式



	Blocking	Non-blocking
Synchronous	Read/write	Read/wirte (O_NONBLOCK)
Asynchronous	i/O multiplexing (select/poll)	AIO

进程间通信的特点

两种通信模式：

- 同步通信：
- 发送(send)操作和接收 (receive)操作是<u>阻塞</u>的。
- 异步通信：
  - <u>发送</u>操作是<u>非阻塞</u>的，接收可以是阻塞的或者不是阻塞的。
  - 编程复杂，运行效率高。

什么情况下采用同步通信？

- 如果下一条指令一定等IO完成，则必须使用blocking的方式
- 在要求CPU占用率较低，以及响应时间较短的情况下，必须使用blocking方式。
  - 比如在VoIP软件电话中，接收语音数据包的IO操作就需要使用blocking的方式，因为VoIP应用对IO操作的时延很敏感。

什么情况下采用异步通信?

- 如果程序同时处理多个IO操作，或者除了IO操作之外，还有其它的工作可做。那么使用nonblocking的方式较好，这样使用的线程数较少。
- 在高性能的服务器中，由于需要同时响应数目非常大的连接，无法为每一个连接都启动一个线程。这种情况下，也需要使用nonblocking的方式。
  - 例如，在嵌入式应用中，内存的容量十分有限，对线程的数量也有严格要求的场合下， 就需要使用nonblocking的方式，让一个线程处理多个IO工作。

一些常用的异步通信的方法(1/2)

1. 将需要等待的部分写在底层的类里面：

- 对于send，底层向上提供调用的send接口，**上层**调用了send后立即返回，实际消息在底层类的发送队列中，由底层类中的专门线程负责逐个发送消息。
- 对receive,可以使用windows的消息机制，如果单纯使用C++实现，则做成一个观察者模式，上层模块把消息的处理类注册到底层类中，消息到来后，由底层类调用高层类的相应处理消息的方法。

一些常用的异步通信的方法(2/2)

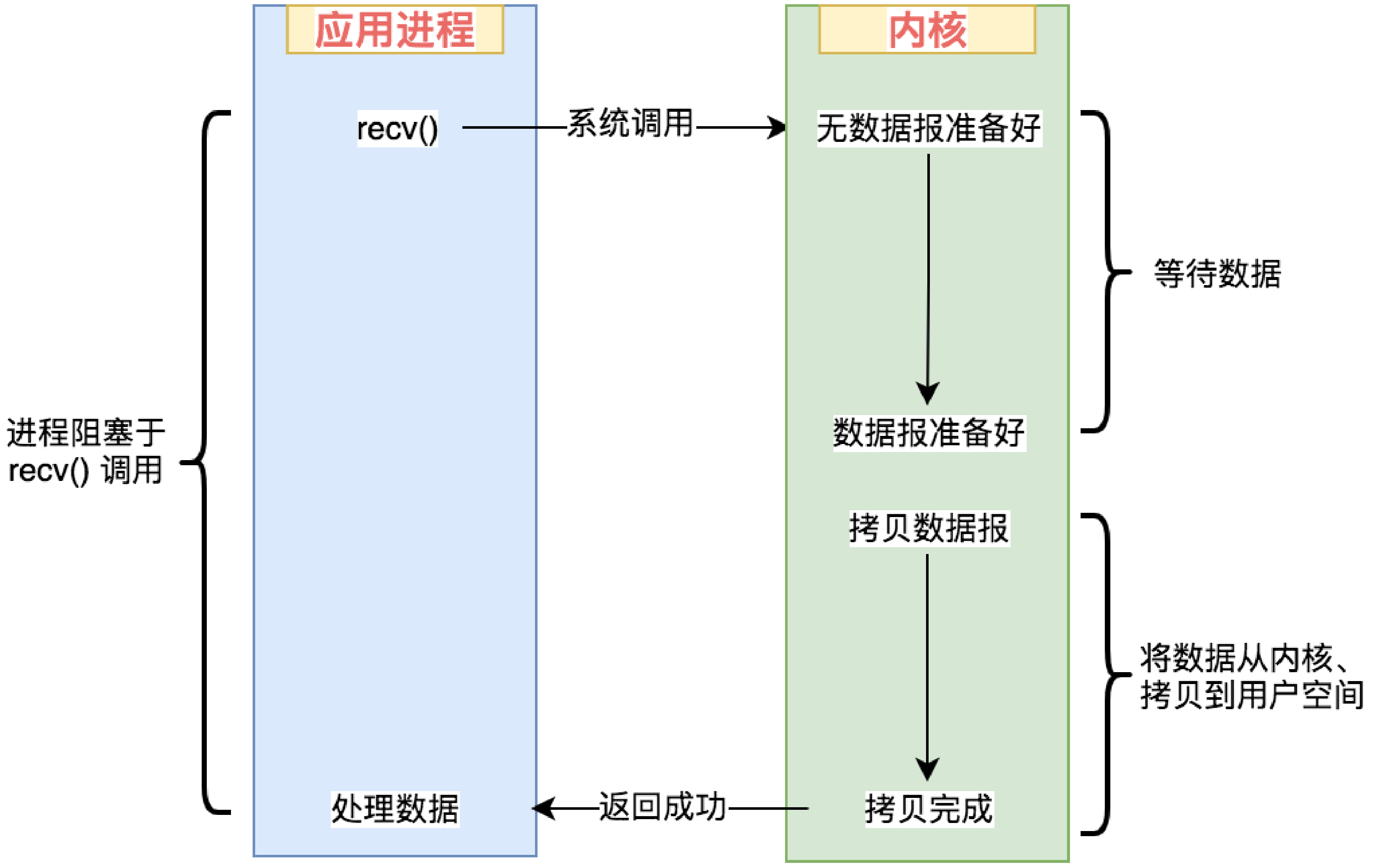
2. 在UNIX Networking Programming Vol 1中介绍了几种模式的I/O，有：

- 使用select和poll调用来判断是否有数据可用，然后再做receive
- Signal-Driven I/O Model：用户注册一个信号处理的函数，每当有数据来时，系统都会发送这个信号，用户进程在该信号处理函数中进行接收数据的操作
- Asynchronous I/O Model：用户进行 receive调用之后，直接返回做别的事情。系统将指定的数据接收到以后，发信号通知用户进程进行处理

Internet协议的API （编程模型）

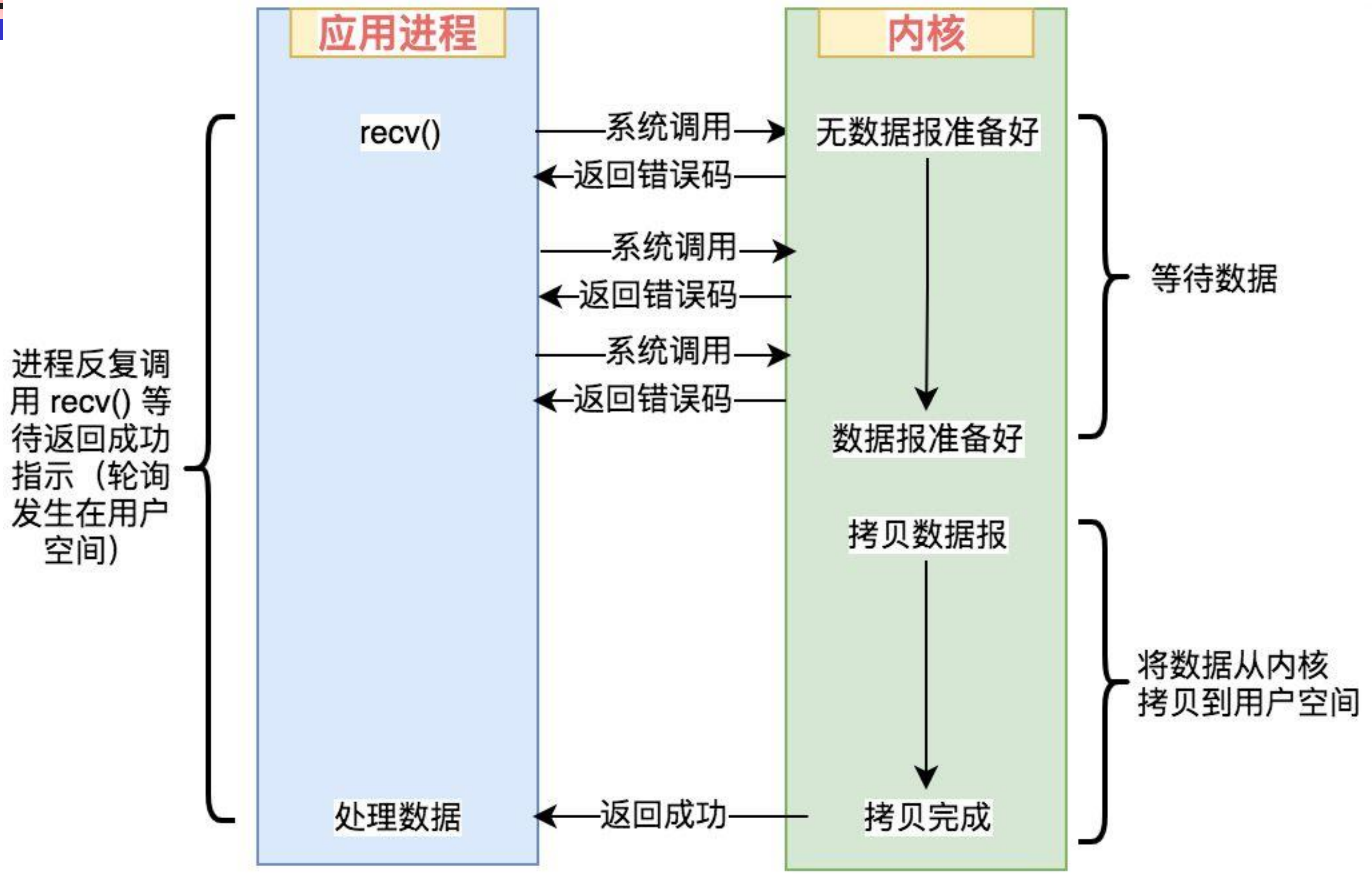
- 通信的目的　与　socket
- UDP与TCP
  - 连接与无连接的对比
- 同步通信与异步通信
- 五种I/O模式

阻塞式IO



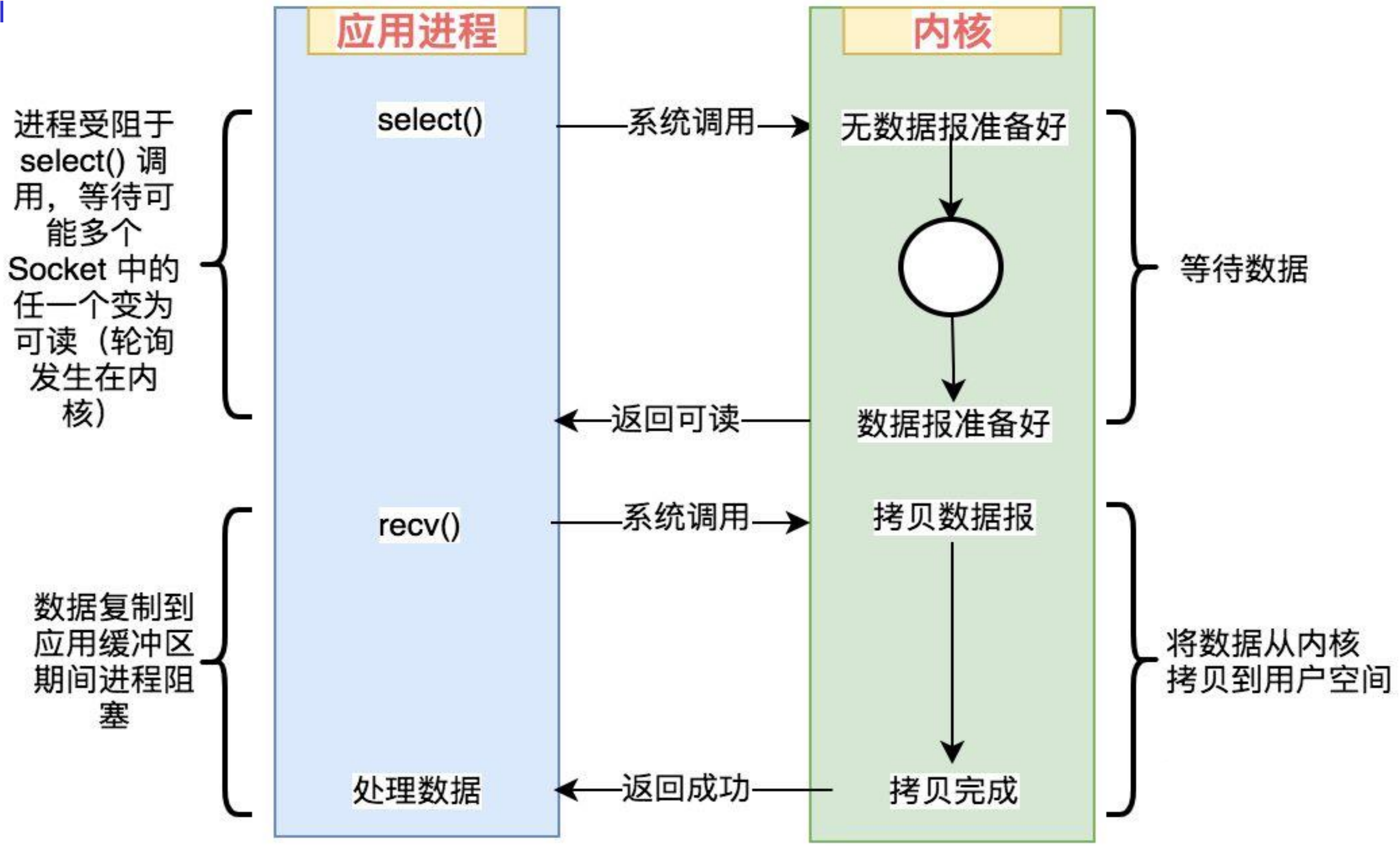
应用进程
内核
recv()
系统调用
无数据报准备好
等待数据
数据报准备好
进程阻塞于
recv() 调用
拷贝数据报
将数据从内核、
拷贝到用户空间
处理数据
返回成功
拷贝完成

非阻塞式IO



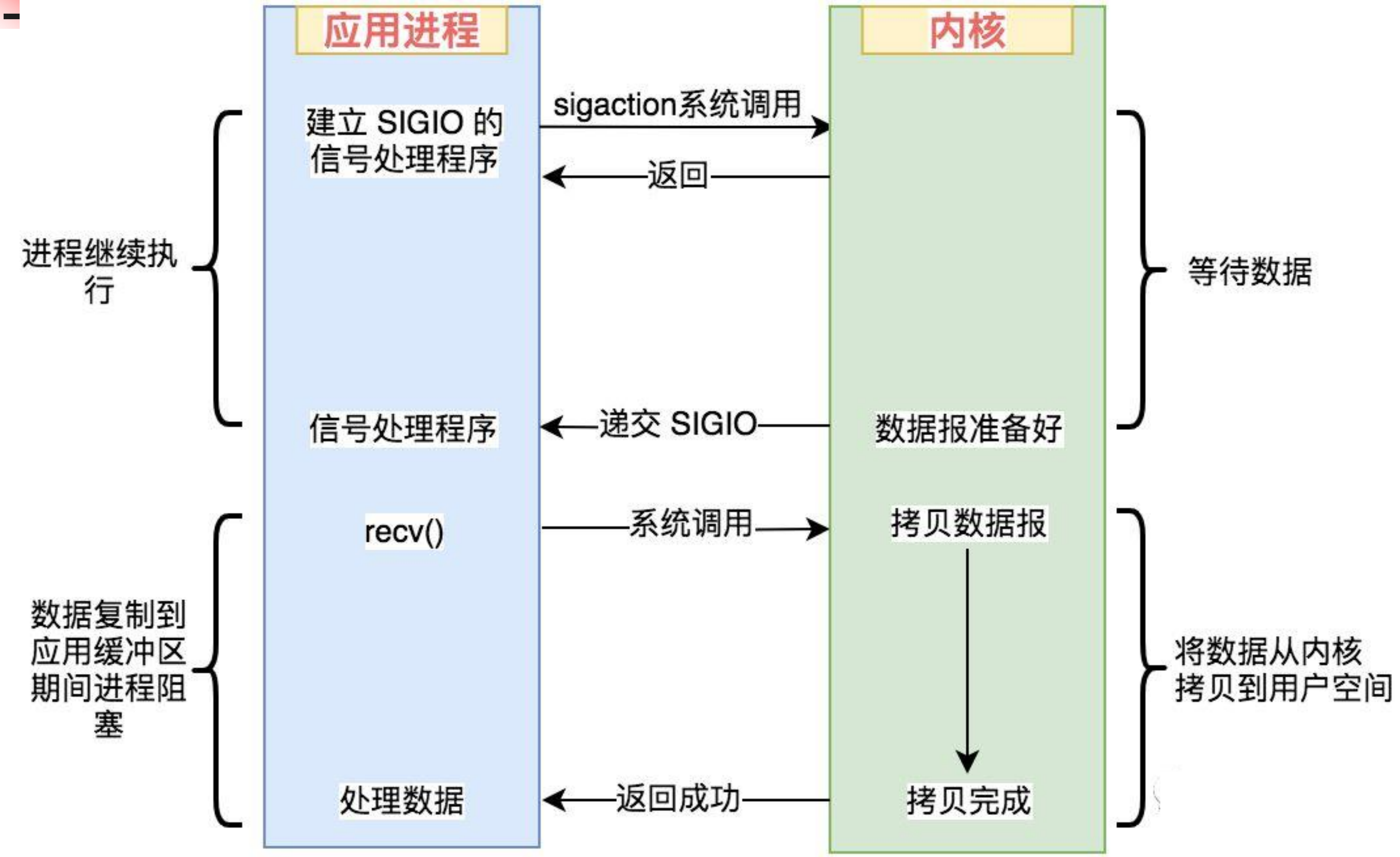
应用进程
内核
recv()
系统调用
无数据报准备好
返回错误码
系统调用
返回错误码
系统调用
返回错误码
等待数据
数据报准备好
进程反复调用 recv() 等待返回成功指示（轮询发生在用户空间）
拷贝数据报
将数据从内核拷贝到用户空间
处理数据
返回成功
拷贝完成

IO 多路复用



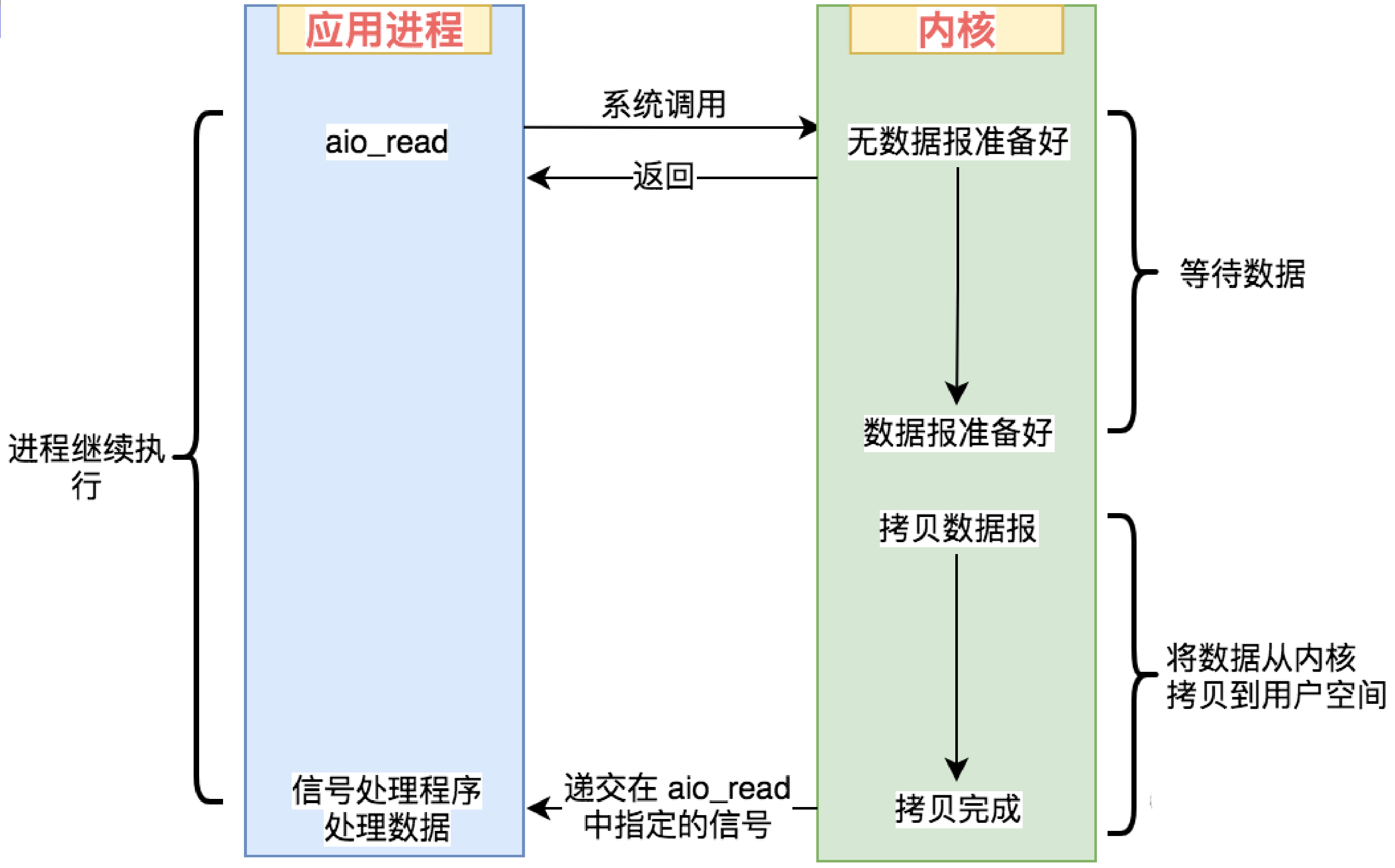
应用进程
内核
select()
系统调用
无数据报准备好
返回可读
数据报准备好
recv()
系统调用
拷贝数据报
处理数据
返回成功
拷贝完成
进程受阻于 select() 调用，等待可能多个 Socket 中的任一个变为可读（轮询发生在内核）
数据复制到应用缓冲区期间进程阻塞
等待数据
将数据从内核拷贝到用户空间

信号驱动式IO

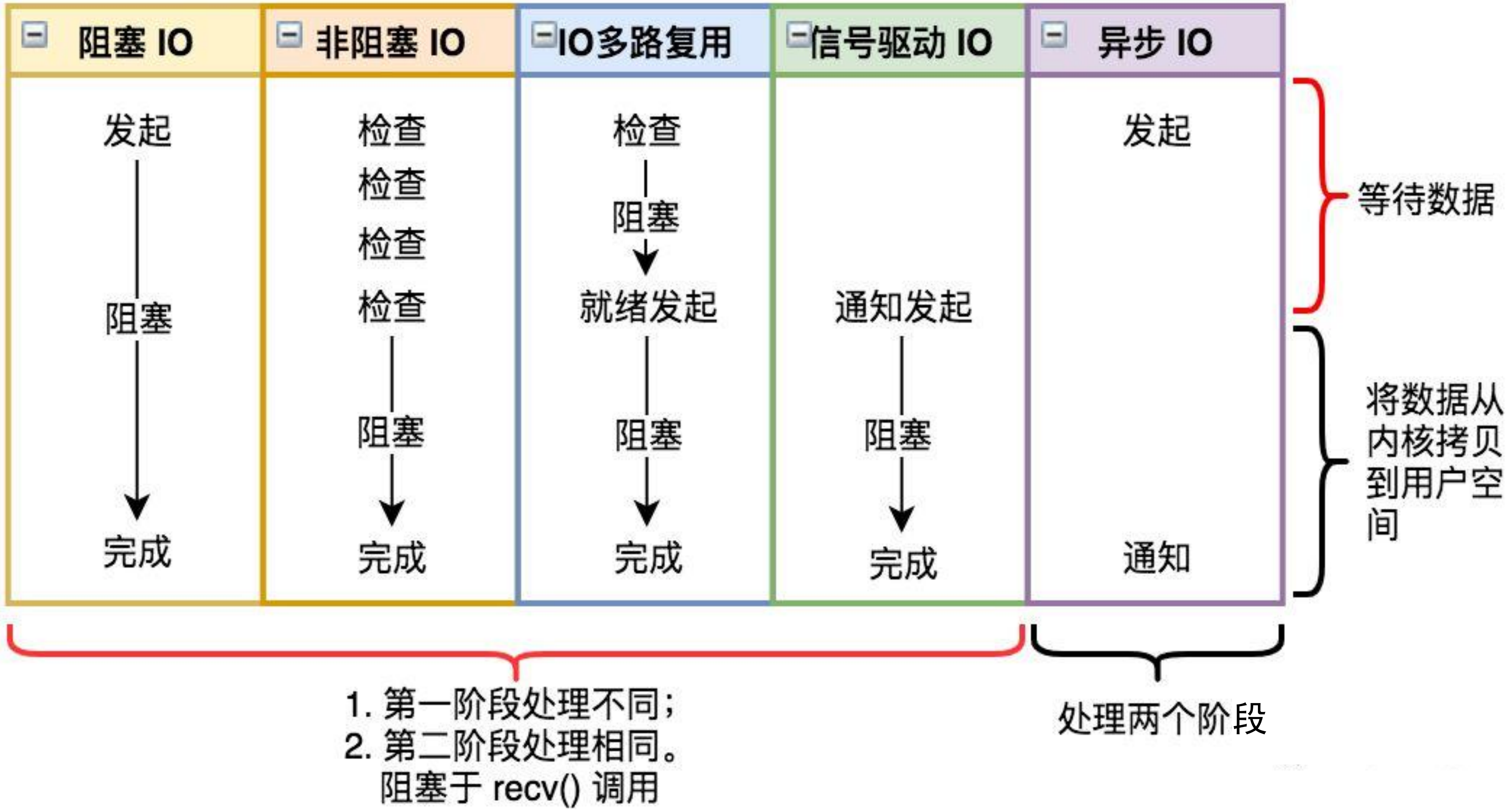


应用进程
内核
建立 SIGIO 的信号处理程序
sigaction系统调用
返回
进程继续执行
等待数据
信号处理程序
递交 SIGIO
数据报准备好
recv()
系统调用
拷贝数据报
数据复制到应用缓冲区期间进程阻塞
将数据从内核拷贝到用户空间
处理数据
返回成功
拷贝完成

异步非阻塞 IO


应用进程
内核
系统调用
aio_read
无数据报准备好
返回
等待数据
数据报准备好
进程继续执行
拷贝数据报
将数据从内核拷贝到用户空间
信号处理程序处理数据
递交在 aio_read 中指定的信号
拷贝完成

5种I/O模式总结



阻塞 IO
非阻塞 IO
IO多路复用
信号驱动 IO
异步 IO
发起
阻塞
完成
检查
检查
检查
检查
阻塞
完成
检查
阻塞
就绪发起
阻塞
完成
通知发起
阻塞
完成
发起
通知
等待数据
将数据从内核拷贝到用户空间
1. 第一阶段处理不同；
2. 第二阶段处理相同。
阻塞于 recv() 调用
处理两个阶段

主要内容

- 通信范型基础
  - 通信范型的分类
  - Internet协议的API （编程模型）
  - 外部数据的表示和编码
- 第一类：客户—服务器通信
- 第二类：远程调用
- 第三类：间接通信
- 总结

外部数据的表示和编码

- 为什么需要外部数据表示和编码？
  - 不同计算机上的数据格式不一样
    - e.g., big-endian/little-endian integer order, ASCII (Unix) / Unicode character coding
- 两台计算机如何交换数据？
  - 有一个统一的外部形式
    - 发送时：本地表示->外部表示；
    - 接收时：外部形式->本地表示专程
  - 数据按本地格式发送，并附有格式说明，接收端如果发现和自己的格式不一致，则做相应的转换。

外部数据的表示和编码

- 外部数据表示
  - 一个统一的标准，规定数据结构和基本数据的表示形式
  - 编码/解码（Marshalling/unmarshalling）
  - 用途：数据传输和保存
- 三种外部编码方法
  - CORBA’s common data representation
  - Java’s object serialization
  - XML

外部数据的表示和编码

- CORBA's Common Data Representation (CDR)
- 规定了所有在CORBA**远程调用中**可能用到的参数和返回值类型的数据的表示格式
  - 15 primitive types
    - Short (16bit), long(32bit), unsigned short, unsigned long, float, char, ...

CORBA公共数据表示

type	Representation
Sequence	Length(unsigned) **followed** by elements in order
string	Length(unsigned long) **followed** by characters in order(can also have wide characters)
array	Array elements in order(no length specified because it is fixed)
struct	In the order of declaration of the components
enumerated	Unsigned long(the value are **specified by the order** declared)
union	Type tag followed by the selected member

CORBA的一条消息

```
Struct Person {
        string  name
        string  place
        int      year;
};
```

======================================

index	sequence of bytes	4 bytes on representation in notes
0–3	5	length of string
4–7	"Smit"	'Smith'
8–11	"h___"	
12–15	6	length of string
16–19	"Lond"	'London'
20-23	"on__"	
24–27	1984	unsigned long

======================================

- flattened form represents a Person
- struct with value: {'Smith', 'London', 1984}

外部数据的表示和编码

- Java对象的序列化Serialization(Deserialization)
  - 将一个对象表示成扁平的形式（flattening）或将对各对象串行化，便于存储和传输
  - 包含对象的类的信息以及版本信息
  - 句柄（Handles）
    - 对象的指针序列化时处理成句柄
  - 每个对象只被编码一次

外部数据的表示和编码

```
Public class Person implements Serializable {
        private String name;
        private String place;
        private int year;
        public Person (String aName, String aPlace, int aYear){
                name = aName;
                place = aPlace;
                year = aYear;
        }  // followed by methods for accessing the instance variables
} Person

p = new Person("Smith", "London", 1984);
```

- Serialized values Explanation

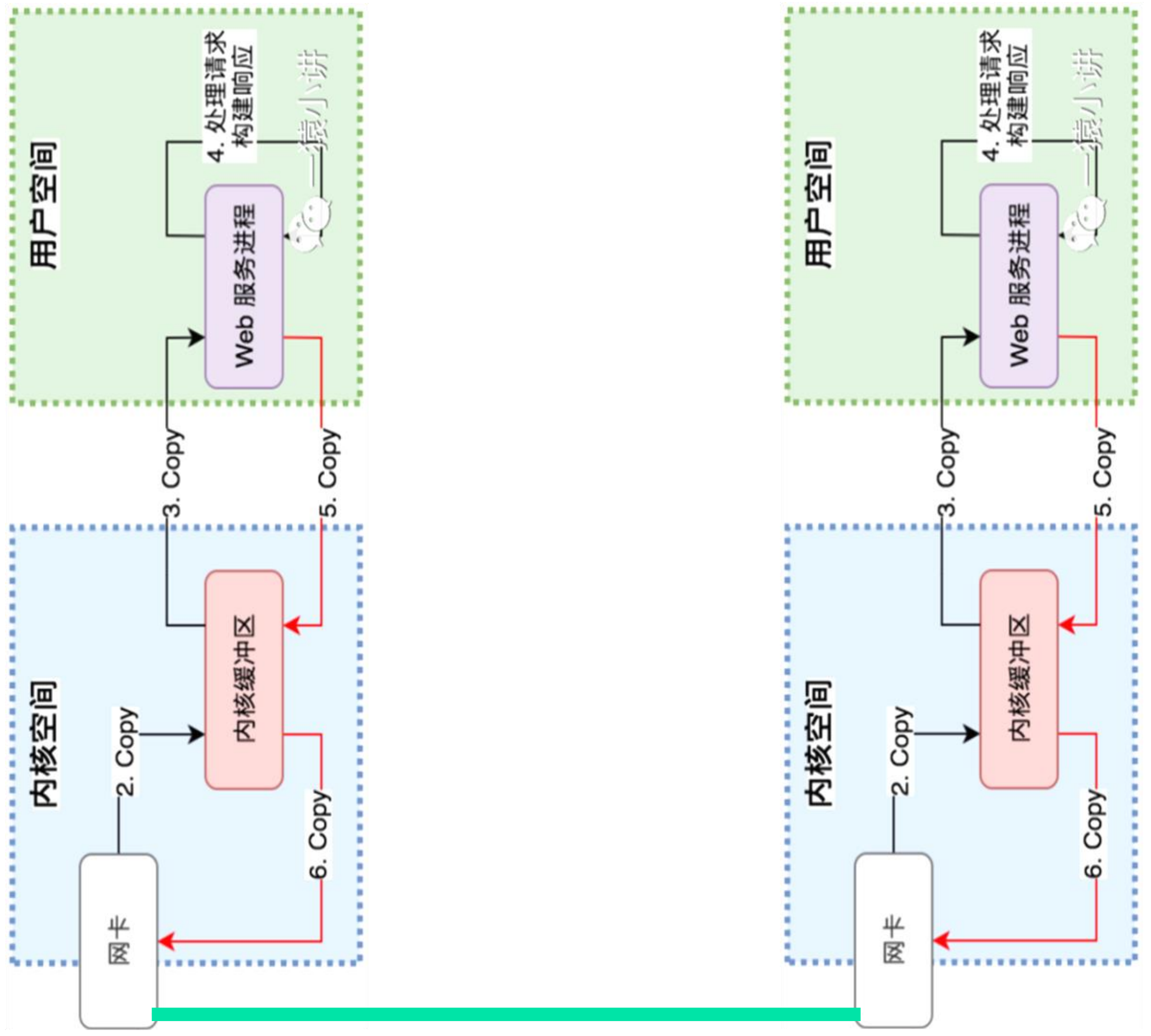
Person	8-byte	version number	h0	class name, version number
3	int year	java.lang.String name	java.lang.String place	instance variables number,
1984	5 Smith	6 London	h1	values of instance variables

- The true serialized form contains additional type markers;
- h0 and h1 are handles

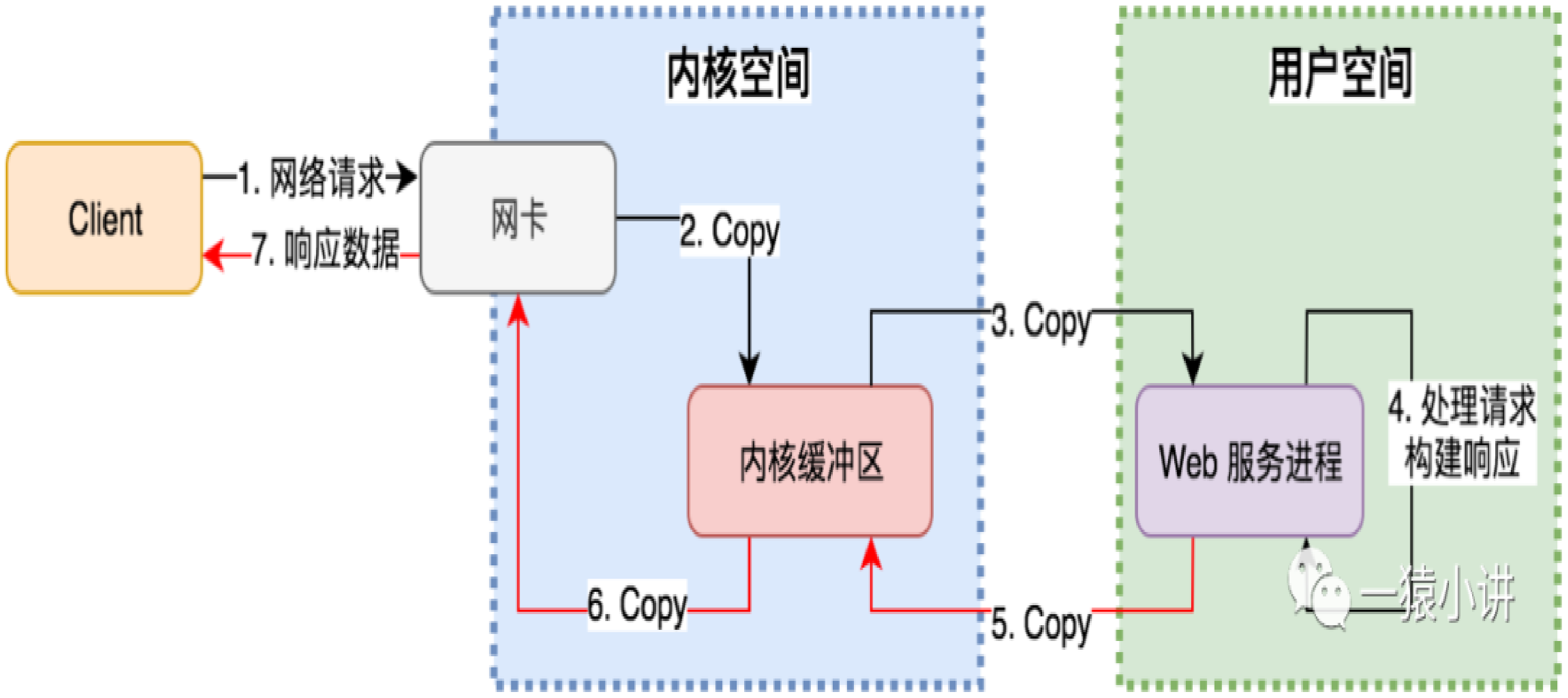
主要内容

- 通信范型基础
  - 通信范型的分类
  - Internet协议的API （编程模型）
  - 外部数据的表示和编码
- 第一类：客户—服务器通信
- 第二类：远程调用
- 第三类：间接通信
- 总结

Review：两个机器上的通信模式？



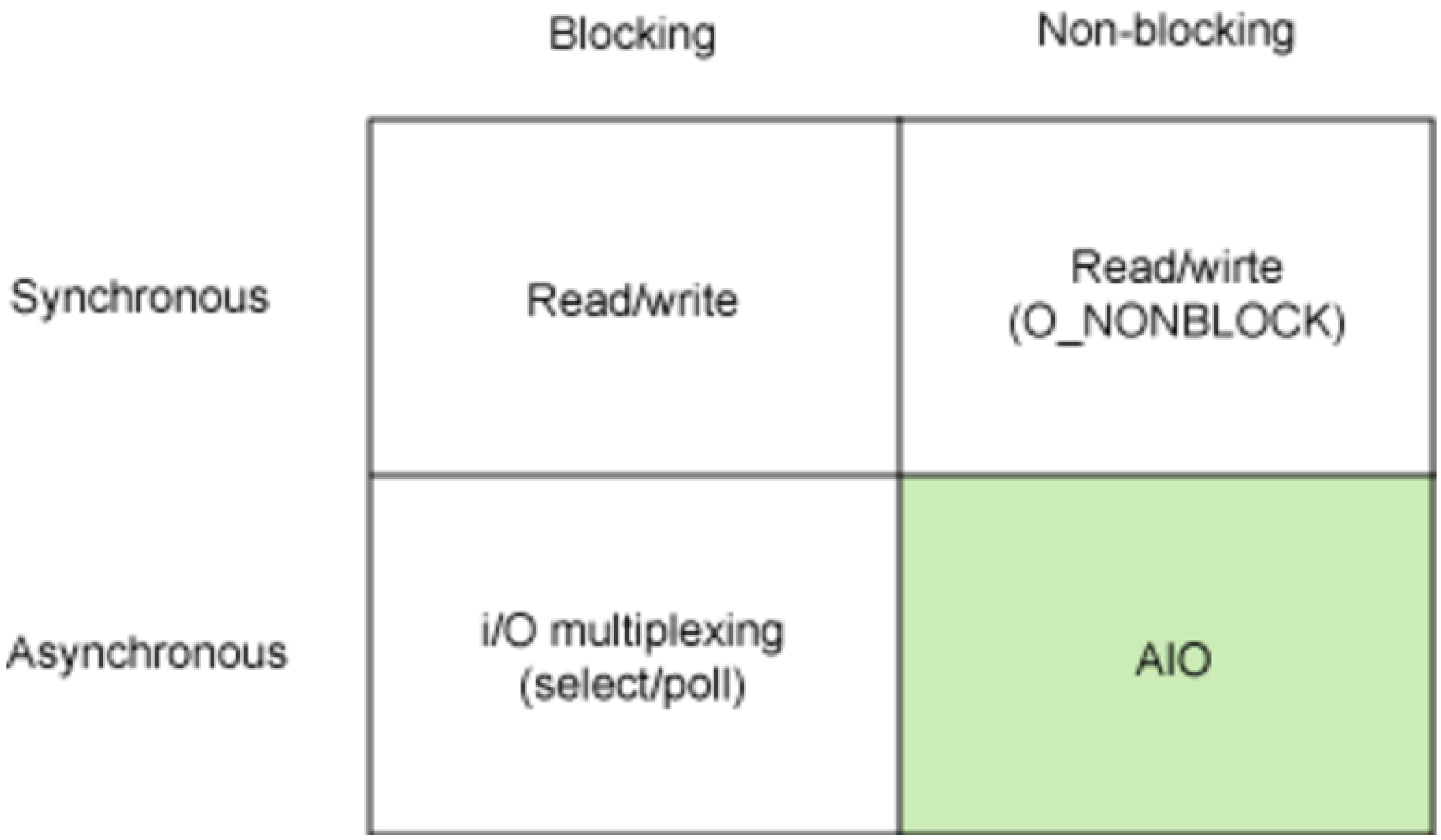
Review：数据从网卡到处理的过程



Review：同步和异步IO 阻塞和非阻塞IO

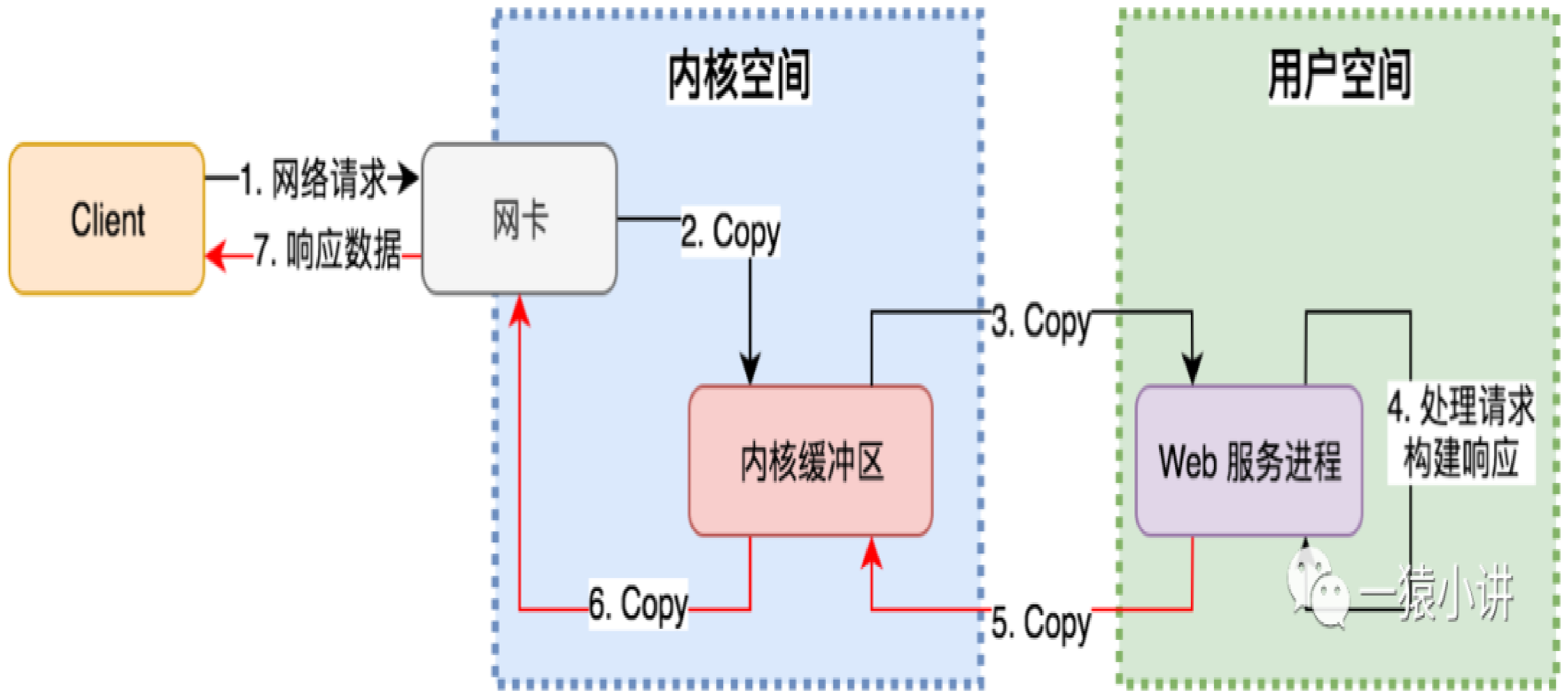
- 同步和异步IO的概念：
  - 同步是用户线程发起I/O请求后需要等待或者轮询内核I/O操作完成后才能继续执行
  - 异步是用户线程发起I/O请求后仍需要继续执行，当内核I/O操作完成后会通知用户线程，或者调用用户线程注册的回调函数
- 阻塞和非阻塞IO的概念：
  - 阻塞是指I/O操作需要彻底完成后才能返回用户空间
  - 非阻塞是指I/O操作被调用后立即返回一个状态值，无需等I/O操作彻底完成

Review：四种通信方式

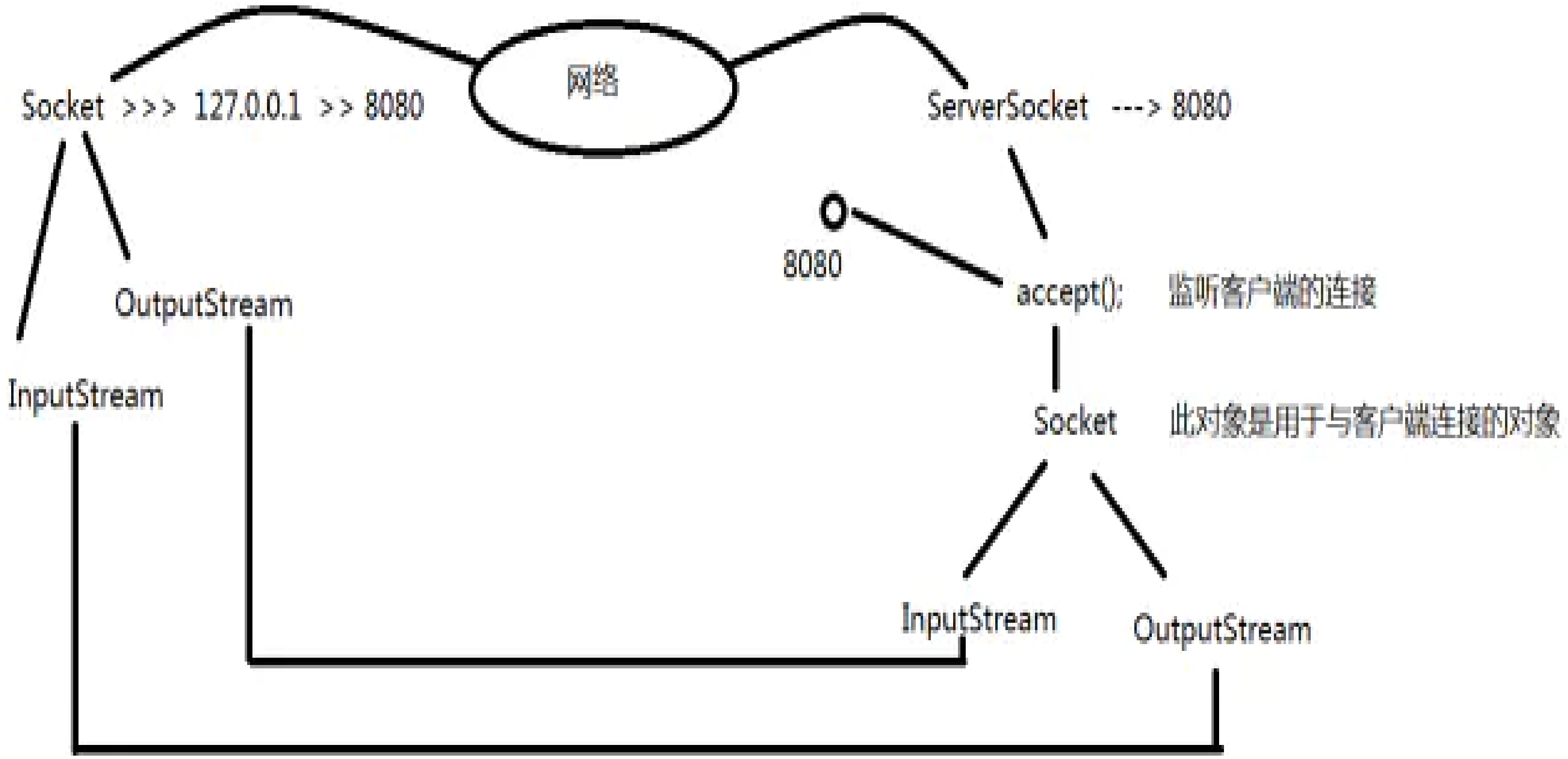


	Blocking	Non-blocking
Synchronous	Read/write	Read/wirte (O_NONBLOCK)
Asynchronous	i/O multiplexing (select/poll)	🟩 AIO

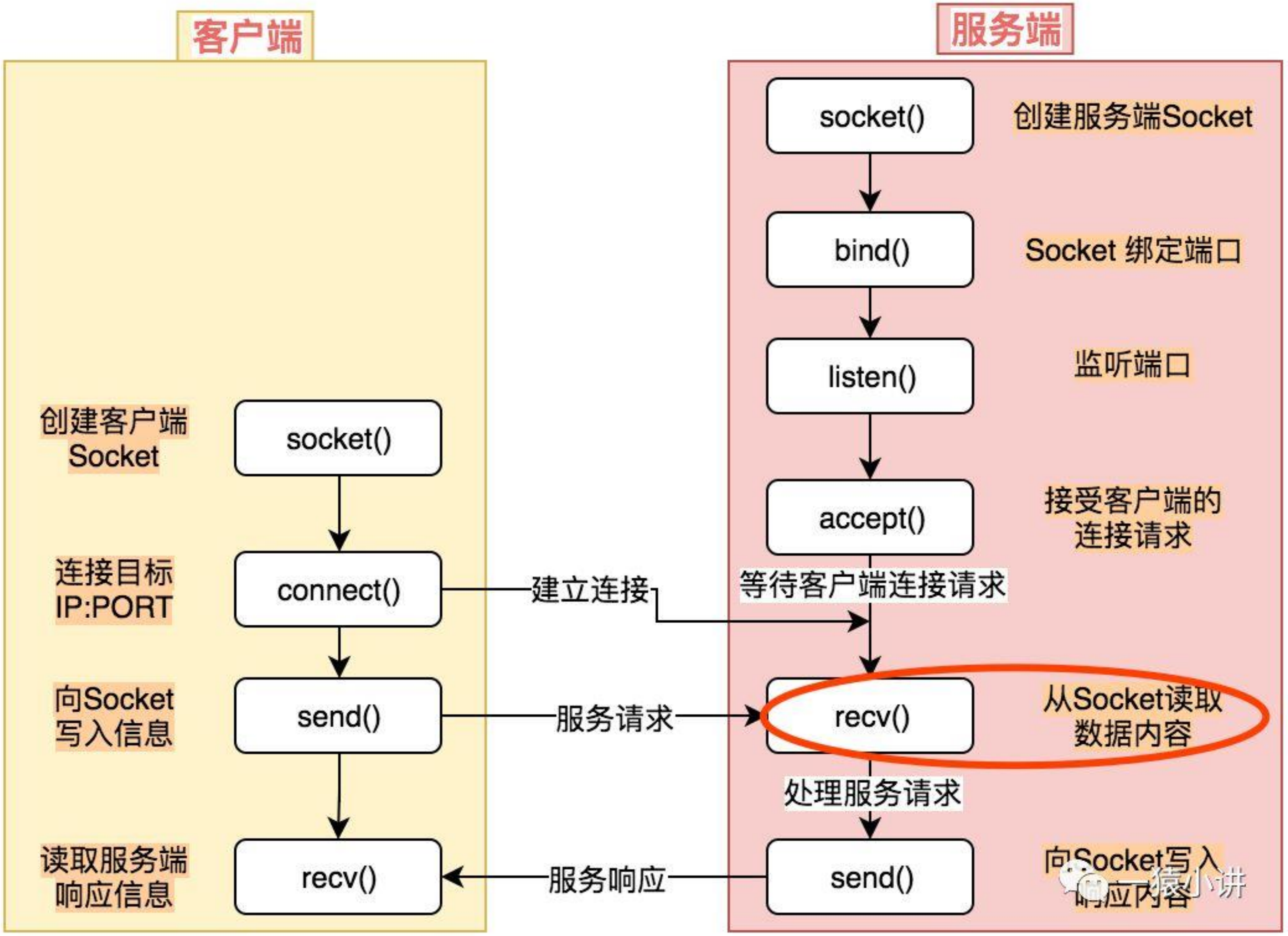
Review：五种IO模式解决了什么问题？



Review：JAVA Socket示意图


网络
Socket >>> 127.0.0.1 >> 8080
OutputStream
InputStream
ServerSocket ---> 8080
8080
accept(); 监听客户端的连接
Socket 此对象是用于与客户端连接的对象
InputStream
OutputStream

Review：UNIX socket连接与通信

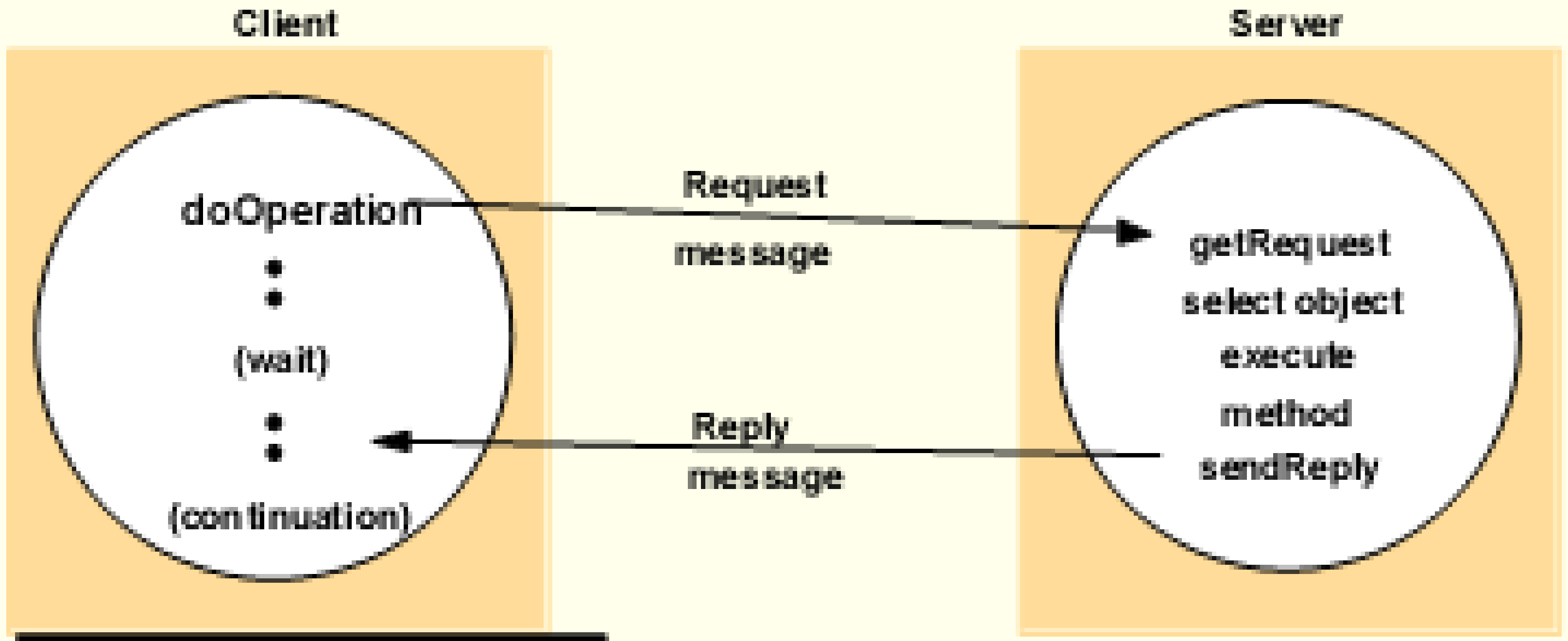


主要内容

- 通信范型基础----通信范型的分类
- Internet协议的API
- 外部数据的表示和编码
- 客户—服务器通信
- 远程调用
- 间接通信
- 总结

客户—服务器通信

- The request-reply protocol (RRP)



messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

客户—服务器通信

- public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
  - 向远程对象发送请求消息，返回应答.
  - 参数：远程对象引用，方法的ID，方法所需要的参数
- public byte[] getRequest ();
  - 在服务器的端口获得客户的请求.
- public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
  - 将结果返回到客户端的端口上。

客户—服务器通信

RRP协议的提交保证：

- 在使用UDP的条件下，满足提交保证，reply信息起到确认的作用。
- 通常情况下，RRP是同步的，请求被阻塞，等待。
- 用UDP而不用TCP的理由：
  - TCP的确认机制和RRP中的reply作用重复。
  - 建立一次TCP连结，除了请求和应答消息之外，还要多一对消息
  - RRP主要用于传递参数和结果，消息量少，不需要流控制。

客户—服务器通信

RRP协议的故障模型（for UDP）

- 缺失错误
  - 消息不能保证被送到
- 乱序
  - 消息不能保证按顺序到达
- 进程崩溃
  - 假设不是拜占庭错误
  - 超时处理用来应付这些故障，doOperation在等待返回时，设定一个等待时间。超过这个等待时间采取什么对策，取决于系统提供什么级别上的提交保证。

客户—服务器通信

- 超时
  - 一旦超时，返回服务器故障，很少这样处理。
  - doOperation发出多次请求都超时，则返回异常
- 重复的请求
  - 根据超时处理的约定，服务器可能接到重复的请求，因此协议设计成通过检查请求ID，过滤来自同一个client的重复请求。
  - 如果服务器还没有发送应答，则完成操作后发送应答。

客户—服务器通信

- 应答信息丢失
  - 如果已经发送应答，再次执行操作，获得结果。
    - 注意，在这种情况下需要实现幂等操作，例如，往一个集合中增加元素 ，相反，在一个队列后面增加数据项增不是幂等操作。
- 保留历史
  - 服务器保留已发送的应答消息的记录，避免重复操作
  - 保留多少历史？如果服务器只想保留一个结果，只要客户端一次发一个请求即可。
  - 如果为大量的客户服务，即使保留一个结果也很大负担。
  - 定期清理。

RPC Exchange Protocol

- The request protocol（RP）
  - 不需要返回，不需要确认的情况
- The request-reply protocol（RRP）
  - 大多数客户服务器模式使用，不需要确认，因为服务器应答可作对客户请求的确认，而客户的下一请求被看作是对应 答的确认。
- The request-reply-acknowledge replay protocol（RRAP）
  - 确认可以使server端丢弃一些保留的记录，节省存储开销。

客户—服务器通信

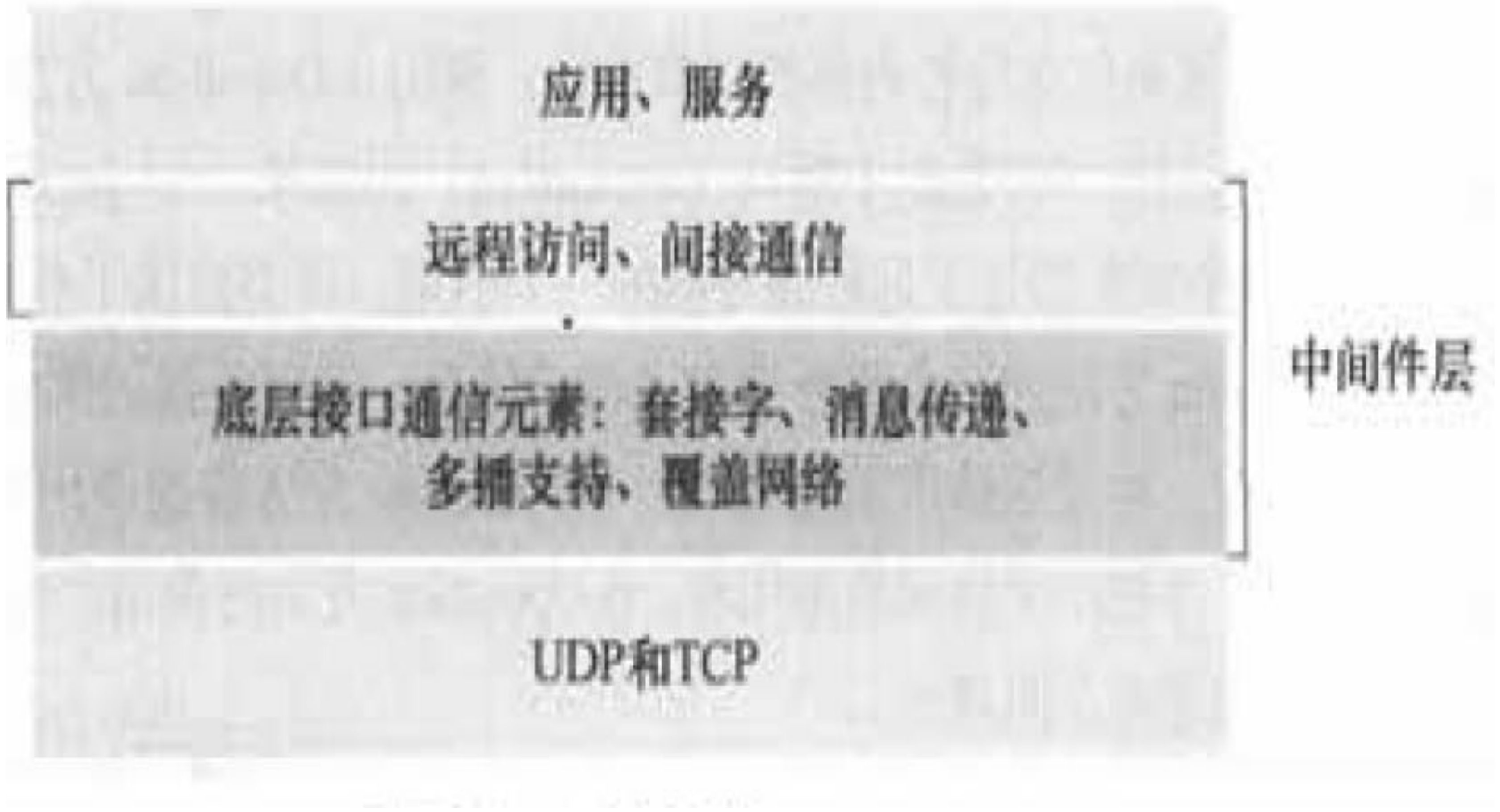
在TCP上实现RRP

- 成本高，但是不需要处理重发和过滤的情况
- 连续的请求和应答可以使用同一个流，以减少建立连结的成本

主要内容

- 通信范型基础
  - 通信范型的分类
  - Internet协议的API （编程模型）
  - 外部数据的表示和编码
- 第一类：客户—服务器通信
- 第二类：远程调用
- 第三类：间接通信
- 总结

内容结构关系

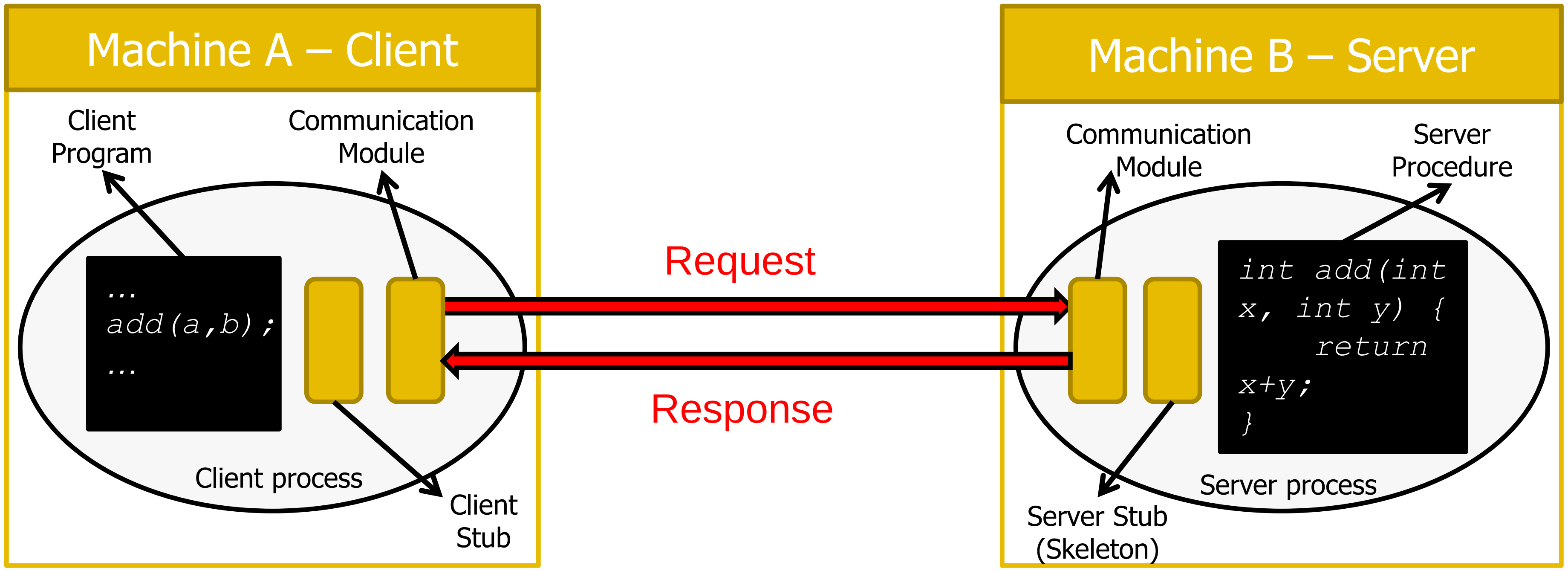


Remote Invocation

- Remote invocation enables an entity to call a procedure that typically executes on an another computer without the programmer explicitly coding the details of communication
  - The underlying middleware will take care of raw-communication
  - Programmer can transparently communicate with remote entity
- We will study two types of remote invocations:
  a. Remote Procedure Calls (RPC)
  b. Remote Method Invocation (RMI)

Remote Procedure Calls (RPC)

- RPC enables a sender to communicate with a receiver using a simple procedure call
  - No communication or message-passing is visible to the programmer
- Basic RPC Approach



Challenges in RPC

- Parameter passing via Marshaling
  - Procedure parameters and results have to be transferred over the network as bits
- Data representation
  - Data representation has to be uniform
    - Architecture of the sender and receiver machines may differ

Challenges in RPC

- Parameter passing via Marshaling
  - Procedure parameters and results have to be transferred over the network as bits
- Data representation
  - Data representation has to be uniform
    - Architecture of the sender and receiver machines may differ

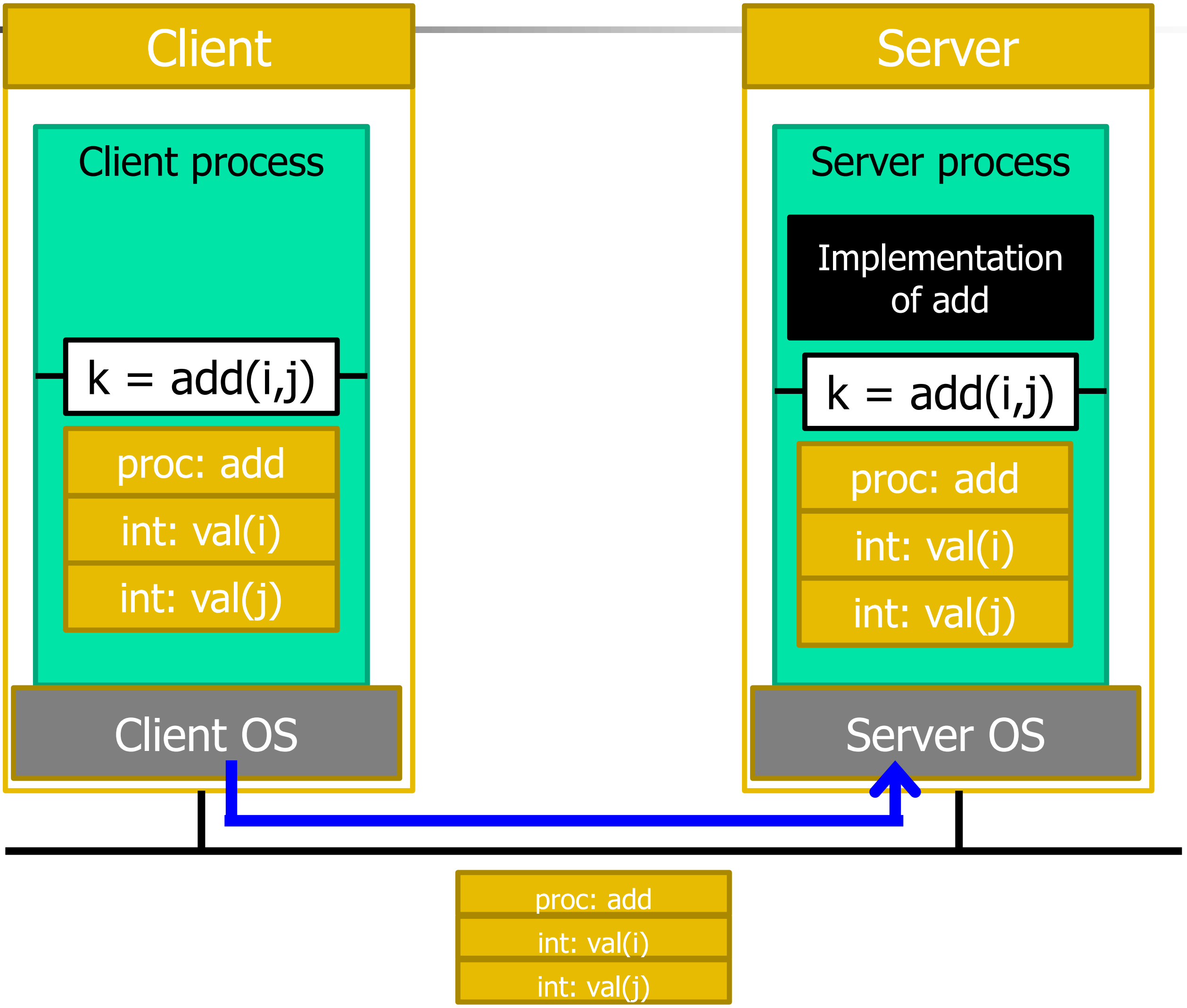
Parameter Passing via Marshaling

- Packing parameters into a message that will be transmitted over the network is called parameter marshalling
- The parameters to the procedure and the result have to be marshaled before transmitting them over the network
- Two types of parameters can passed
    1. Value parameters
    2. Reference parameters

1. Passing Value Parameters

- Value parameters have complete information about the variable, and can be directly encoded into the message
  - e.g., integer, float, character
- Values passed are passed through call-by-value
  - The changes made by the callee procedure are not reflected in the caller procedure

Example of Passing Value Parameters



2. Passing Reference Parameters

- Passing reference parameters like value parameters in RPC leads to incorrect results due to two reasons:

  a. Invalidity of reference parameters at the server
     - Reference parameters are valid only within client’s address space
     - Solution: Pass the reference parameter by copying the data that is referenced

  b. Changes to reference parameters are not reflected back at the client
     - Solution: “Copy/Restore” the data
       - Copy the data that is referenced by the parameter.
       - Copy-back the value at server to the client.

Challenges in RPC

- Parameter passing via Marshaling
  - Procedure parameters and results have to be transferred over the network as bits

- Data representation
  - Data representation has to be uniform
    - Architecture of the sender and receiver machines may differ

Data Representation

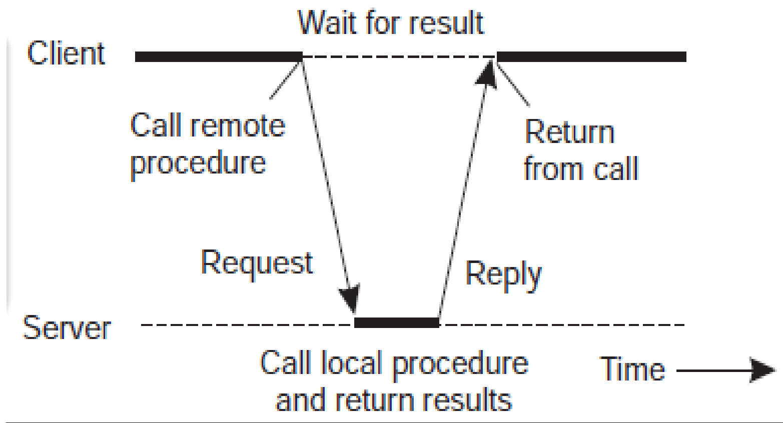
- Computers in DS often have different architectures and operating systems
  - The size of the data-type differ
    - e.g., A *long* data-type is 4-bytes in 32-bit Unix, while it is 8-bytes in 64-bit Unix systems
  - The format in which the data is stored differ
    - e.g., Intel stores data in little-endian format, while SPARC stores in big-endian format
- The client and server have to agree on how simple data is represented in the message
  - e.g., format and size of data-types such as integer, char and float

Remote Procedure Call Types

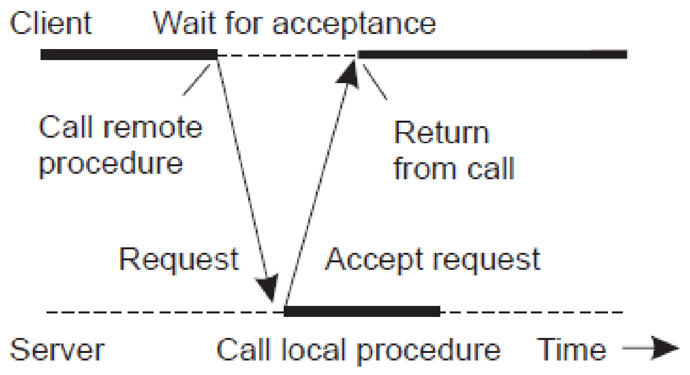
- Remote procedure calls can be:
  - Synchronous
  - Asynchronous (or Deferred Synchronous)

Synchronous vs. Asynchronous RPCs

- An RPC with strict request-reply blocks the client until the server returns
  - Blocking wastes resources at the client
- Asynchronous RPCs are used if the client does not need the result from server
  - The server immediately sends an ACK back to client
  - The client continues the execution after an ACK from the server



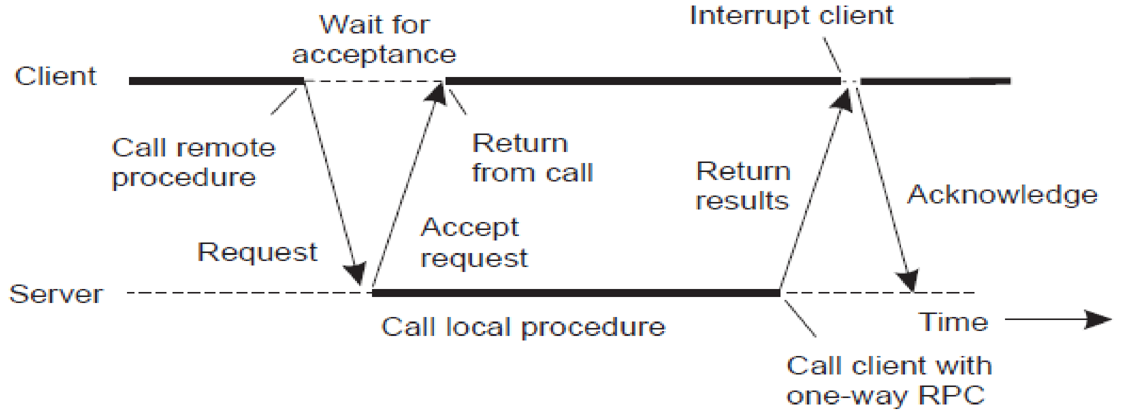
Synchronous RPCs



Asynchronous RPCs

Deferred Synchronous RPCs

- Asynchronous RPC is also useful when a client wants the results, but does not want to be blocked until the call finishes
- Client uses deferred synchronous RPCs
  - Single request-response RPC is split into two RPCs
  - First, client triggers an asynchronous RPC on server
  - Second, on completion, server calls-back client to deliver the results



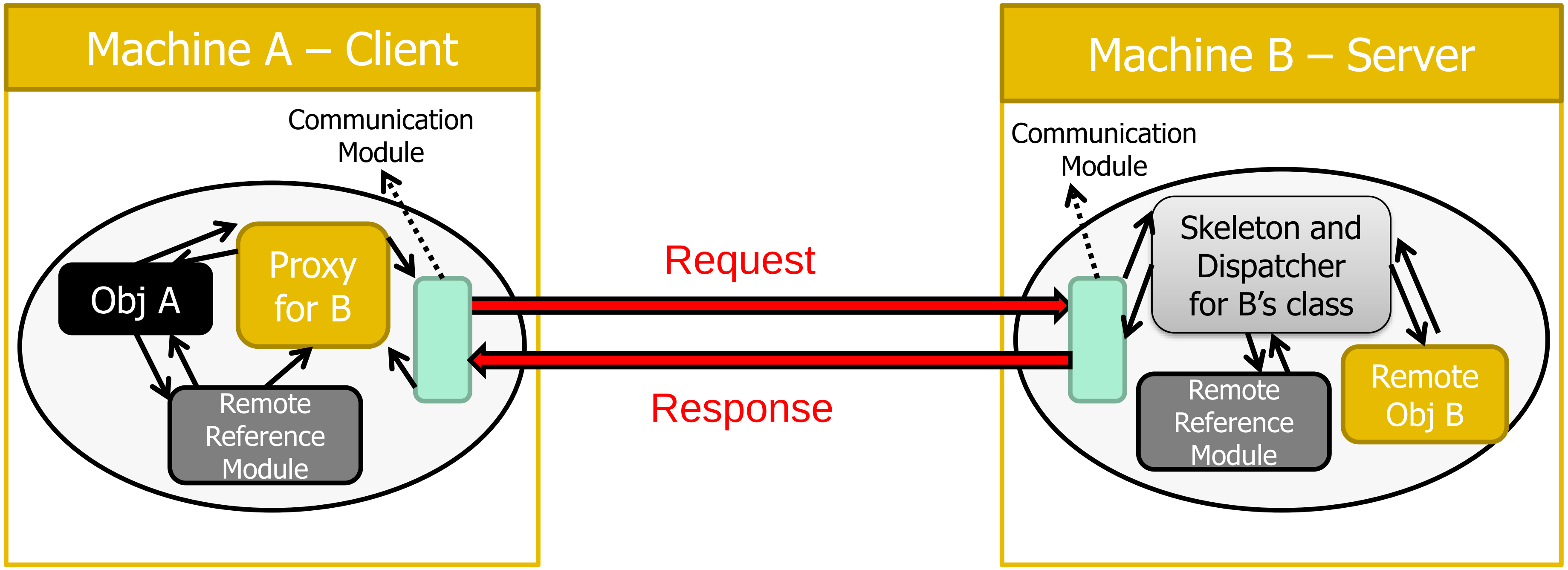
Remote Method Invocation (RMI)

- In RMI, a calling object can invoke a method on a potentially remote object
- RMI is similar to RPC, but in a world of distributed objects
  - The programmer can use the full expressive power of object-oriented programming
  - RMI not only allows to pass value parameters, but also pass object references

Remote Objects and Supporting Modules

- In RMI, objects whose methods can be invoked remotely are known as "*remote objects*"
  - Remote objects implement remote interface
- During any method call, the system has to resolve whether the method is being called on a local or a remote object
  - Local calls should be called on a local object
  - Remote calls should be called via remote method invocation
- Remote Reference Module is responsible for translating between local and remote object references

RMI Control Flow



主要内容

- 通信范型基础
  - 通信范型的分类
  - Internet协议的API （编程模型）
  - 外部数据的表示和编码
- 第一类：客户—服务器通信
- 第二类：远程调用
- 第三类：间接通信
- 总结

Indirect Communication

- Recall: Indirect communication uses middleware to
  - Provide one-to-many communication
  - Mechanisms eliminate ***space and time coupling***
    - Space coupling: Sender and receiver should know each other’s identities
    - Time coupling: Sender and receiver should be explicitly listening to each other during communication

- Approach used: Indirection
  - Sender → A Middle-Man → Receiver

Middleware for Indirect Communication

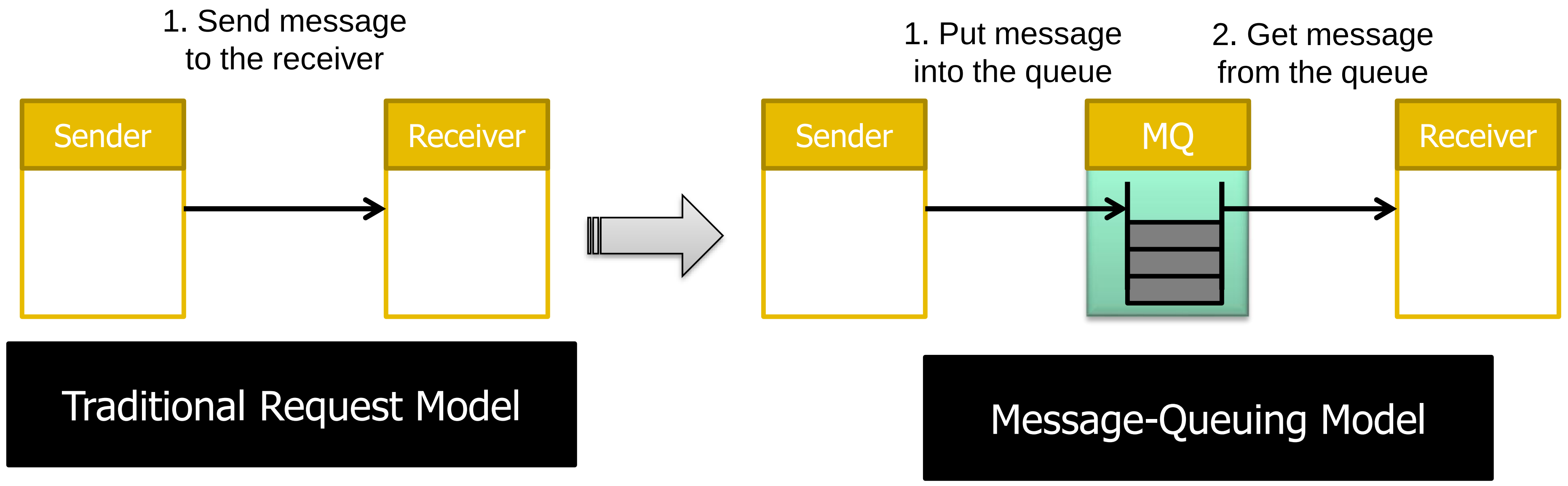
- Indirect communication can be achieved through:
  1. Message-Queuing Systems
  2. Group Communication Systems

Middleware for Indirect Communication

- Indirect communication can be achieved through:
  1. Message-Queuing Systems
  2. Group Communication Systems

Message-Queuing (MQ) Systems

- Message Queuing (MQ) systems provide space and time decoupling between sender and receiver
  - They provide intermediate-term storage capacity for messages (in the form of Queues), without requiring sender or receiver to be active during communication



1. Send message to the receiver
Sender
Receiver
Traditional Request Model
1. Put message into the queue
2. Get message from the queue
Sender
MQ
Receiver
Message-Queuing Model

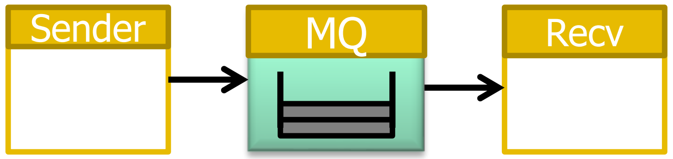
Space and Time Decoupling

- MQ enables space and time decoupling between sender and receivers
  - Sender and receiver can be *passive* during communication

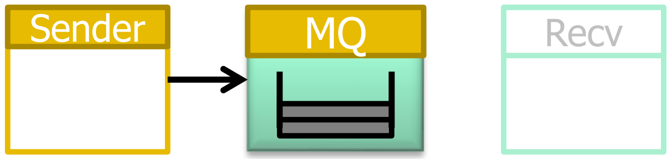
- However, MQ has other types of coupling
  - Sender and receiver have to know the identity of the queue
  - The middleware (queue) should be always *active*

Space and Time Decoupling (cont'd)

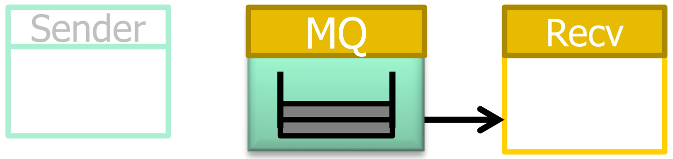
- Four combination of loosely-coupled communications are possible in MQ:



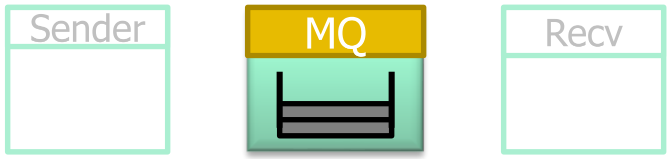
1. Sender active; Receiver active



2. Sender active; Receiver passive



3. Sender passive; Receiver active



4. Sender passive; Receiver passive

Interfaces Provided by the MQ System

- Message Queues enable asynchronous communication by providing the following primitives

Primitive	Meaning
PUT	Append a message to a specified queue
GET	Block until the specified queue is nonempty, and remove the first message
POLL	Check a specified queue for messages, and remove the first. Never block
NOTIFY	Install a handler (call-back function) to be called when a message is put into the specified queue

Architecture of an MQ System

- The architecture of an MQ system has to address the following challenges:

  a. Placement of the Queue
     - Is the queue placed near to the sender or receiver?

  b. Identity of the Queue
     - How can sender and receiver identify the queue location?

  c. Intermediate Queue Managers
     - Can MQ be scaled to a large-scale distributed system?

a. Placement of the Queue

- Each application has a specific pattern of inserting and receiving the messages
- MQ system is optimized by placing the queue at a location that improves performance
- Typically, a queue is placed in one of the two locations
  - Source queues: Queue is placed near the source
  - Destination queues: Queue is placed near the destination
- Examples:
  - “Email Messages” is optimized by the use of destination queues
  - “RSS Feeds” requires source queuing

b. Identity of the Queue

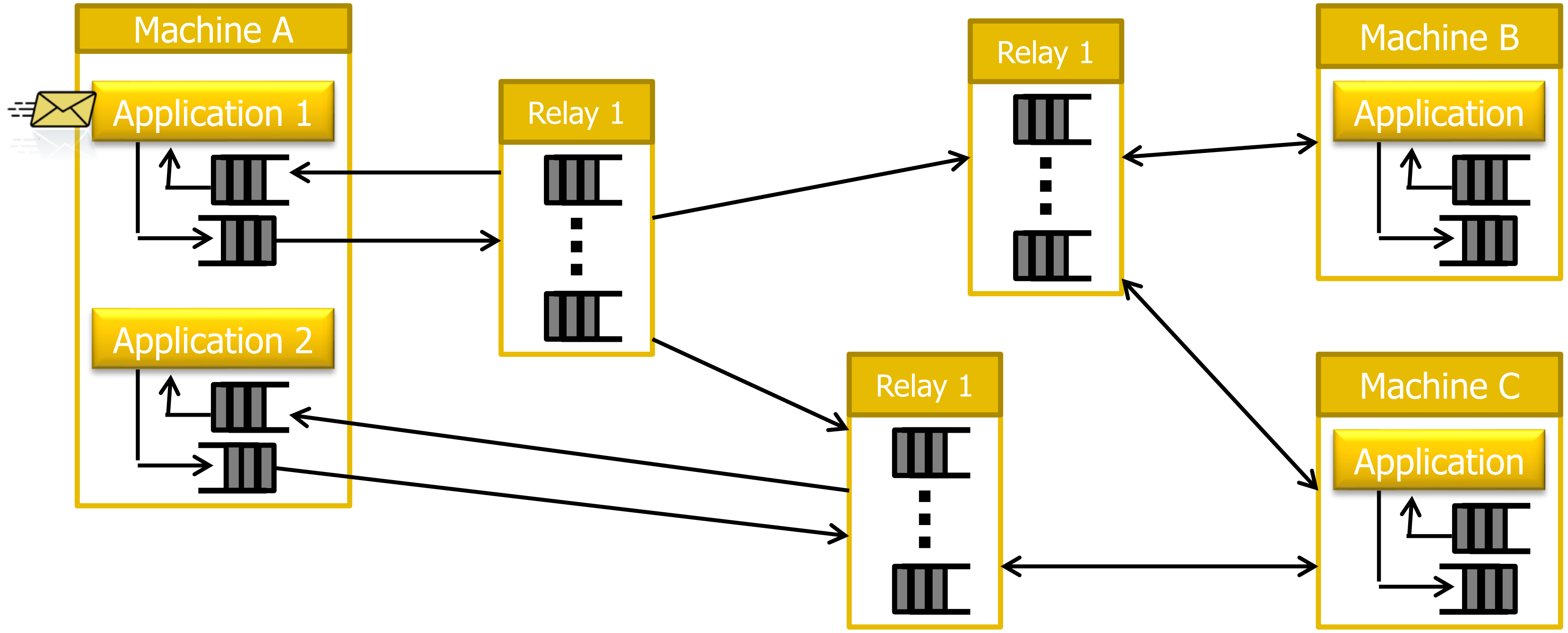
- In MQ systems, queues are generally <u>addressed by names</u>
- However, the sender and the receiver should be aware of <u>the network location of the queue</u>
- A <u>naming service </u>for queues is necessary
  - Database of queue names to network locations is maintained
  - Database can be distributed (similar to DNS)

c. Intermediate Queue Managers

- Queues are managed by Queue Managers
  - Queue Managers directly interact with sending and receiving processes
- However, Queue Managers are not scalable in dynamic large-scale Distributed Systems (DSs)
  - Computers participating in a DS may change (thus changing the topology of the DS)
  - There is no general naming service available to dynamically map queue names to network locations
- Solution: To build an overlay network (e.g., Relays)

c. Intermediate Queue Managers (Cont'd)

- Relay queue managers (or relays) assist in building dynamic scalable MQ systems
  - Relays act as "routers" for routing the messages from sender to the queue manager



Middleware for Indirect Communication

- Indirect communication can be achieved through:
  1. Message-Queuing Systems
  2. Group Communication Systems

Group Communication Systems

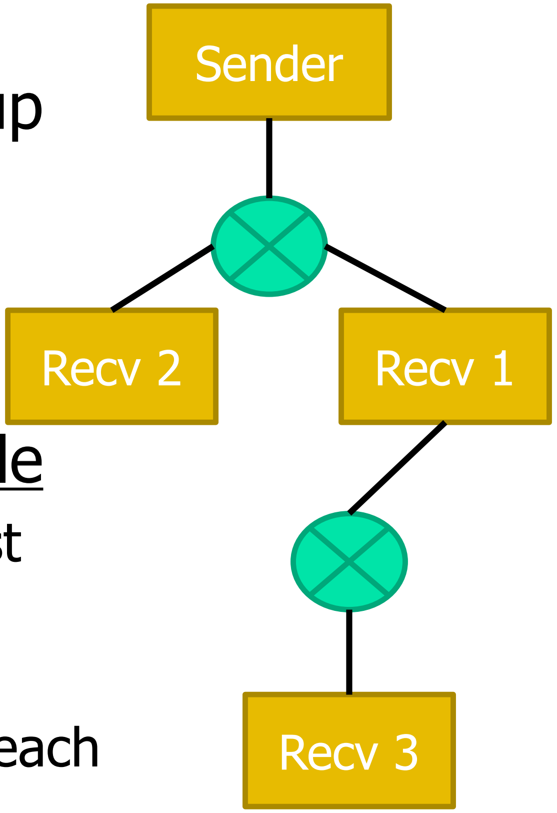
- Group Communication systems enable <u>one-to-many</u> communication
  - A prime example is multicast communication support for applications

- Multicast can be supported using two approaches
  1. Network-level multicasting
  2. Application-level multicasting

1. Network-Level Multicast

- Each multicast group is assigned a unique IP address
- Applications "join" the multicast group
- Multicast tree is built by connecting routers and computers in the group

- Network-level multicast is not scalable
  - Each DS may have a number of multicast groups
  - Each router on the network has to store information for multicast IP address for each group for each DS

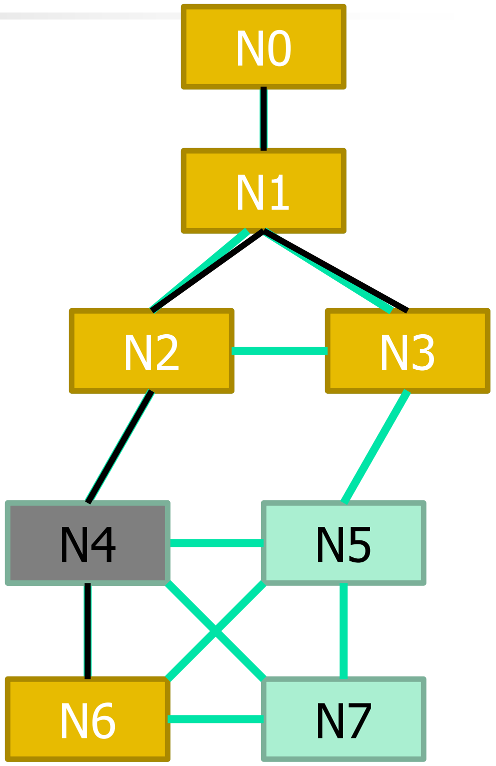


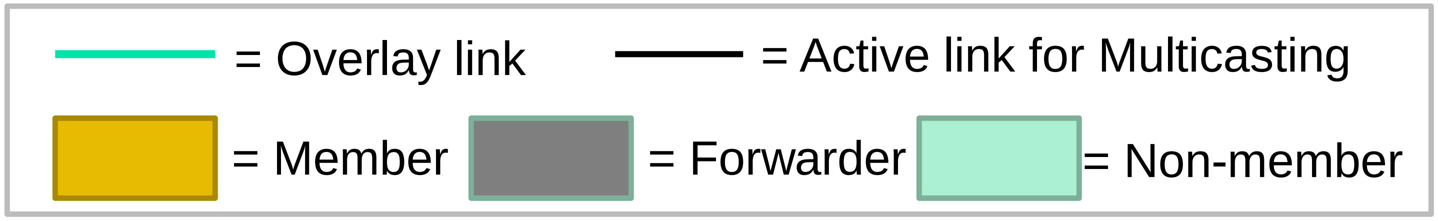
2. Application-Level Multicast (ALM)

- ALM organizes the computers involved in a DS into an overlay network
  - The computers in the overlay network *cooperate* to deliver messages to other computers in the network
- Network **routers** do not directly **participate in** the group **communication**
  - The overhead of maintaining information at all the Internet routers is eliminated
  - Connections between computers in an overlay network may cross several physical links. Hence, ALM may not be optimal

Building Multicast Tree in ALM

- Initiator (root) generates a multicast identifier *mid*
- If node P wants to join, it sends a join request to the root
- When request arrives at Q (any node):
  - If Q has not seen a join request before, it becomes a forwarder
  - P becomes child of Q. Join request continues to be forwarded





主要内容

- 通信范型基础
  - 通信范型的分类
  - Internet协议的API （编程模型）
  - 外部数据的表示和编码
- 第一类：客户—服务器通信
- 第二类：远程调用
- 第三类：间接通信
- 总结

Summary

- Several powerful and flexible paradigms to communicate between entities in a DS
  - Inter-Process Communication (IPC)
    - IPC provides a low-level communication API
    - e.g., Socket API
  - Remote Invocation
    - Programmer can transparently invoke a remote function by using a local procedure-call syntax
    - e.g., RPC and RMI
  - Indirect Communication
    - Allows one-to-many communication paradigm
    - Enables space and time decoupling
    - e.g., Multicasting and Message-Queue systems

总结

- 两种不同的通信模块
  - Datagram Socket: based on UDP, efficient but suffer from failures
  - Stream Socket: based on TCP, reliable but expensive
- 编码解码
  - CORBA’s CDR(common data representation) and Java serialization
- RRP协议
  - Base on UDP or TCP
- 多播
  - IP 多播是简单的多播协议

思考题

- 简述Socket的I/O通信模式
- 选择一种平台，比如JAVA，简单分析其Client/Server结构通信时，两者的I/O及相互通信模式
- （可选）如果服务器要多线程并发呢？

END