

《通信与网络》实验一

Socket网络编程

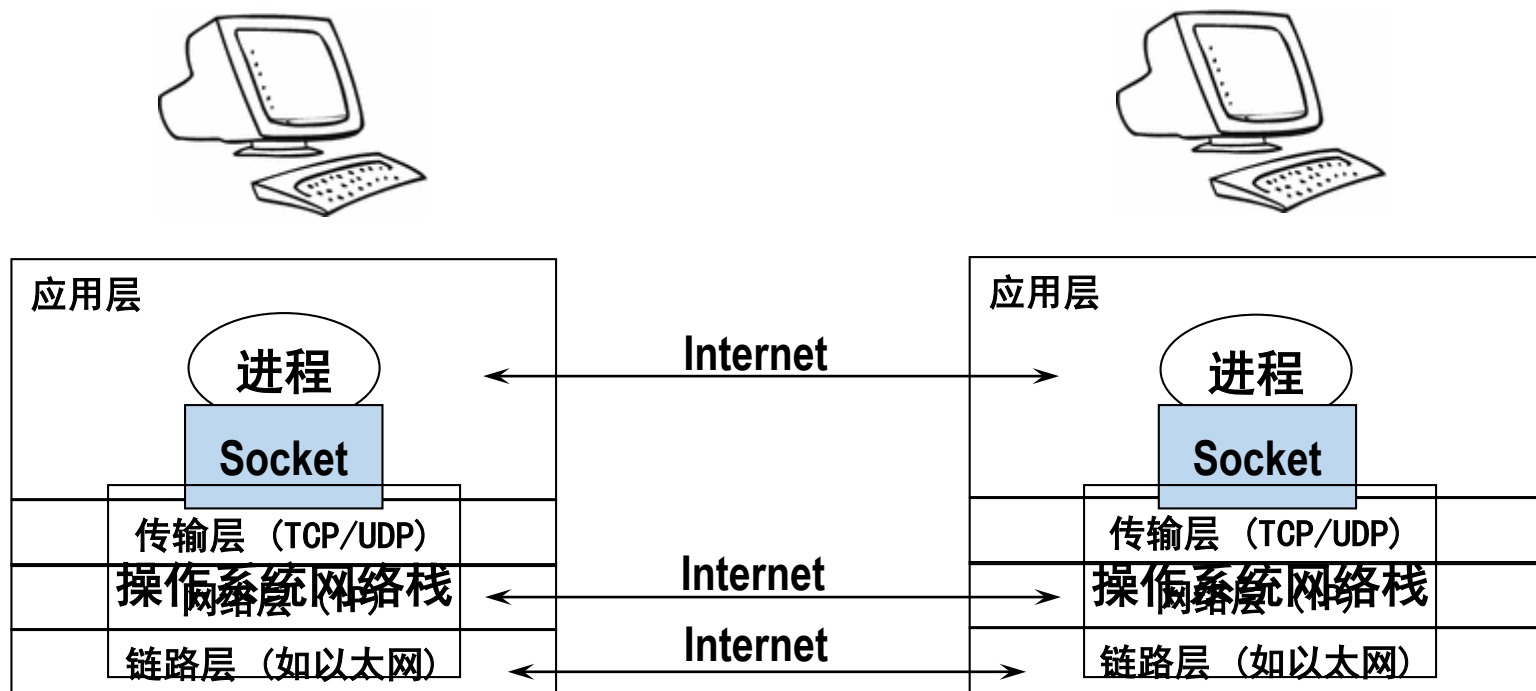
2022年9月

目录

- 理解Socket与进程通信
- Socket中的客户端与服务端
- UNIX下的Socket API编程
- 实验内容介绍

一、理解Socket与 进程通信

Socket与进程通信



Socket是操作系统提供的网络编程接口

传输数据过程中的分工

- **网络**

- 将数据包转发给目标主机
- 根据目标IP地址进行转发

- **操作系统**

- 将数据包传送到目标Socket
- 传送过程依据目标端口号（如80）

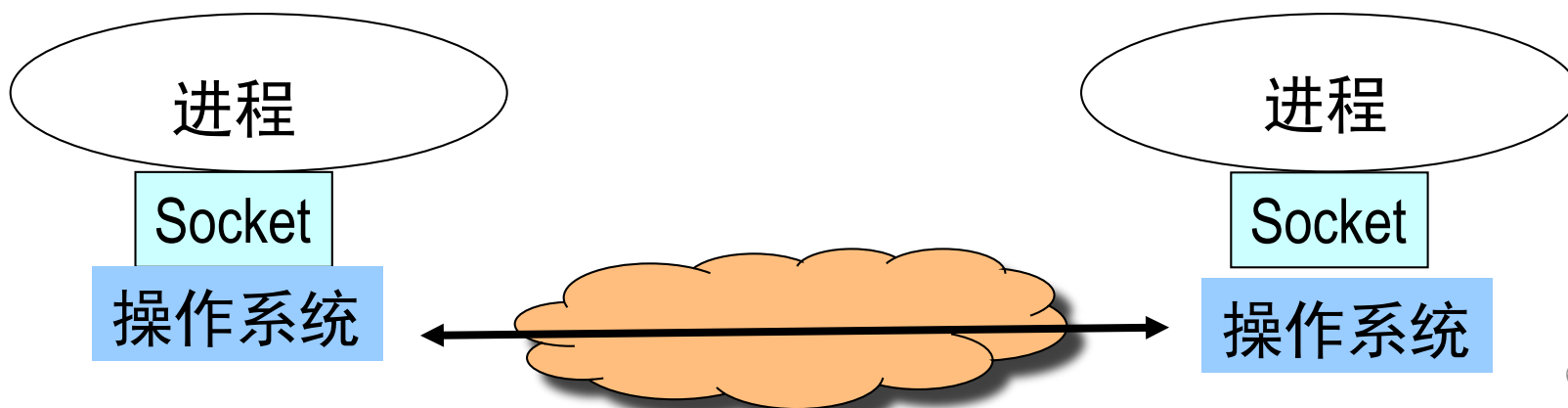
- **应用**

- 向Socket读写数据
- 处理数据（如渲染网页）



Socket是进程间通信的接口

- 从一个进程到另一进程
 - 报文（Message）必须通过底层的网络传递
- 进程通过Socket发送与接收报文（Message）
 - Socket可以类比于进出“进程屋子”的“大门”
- Socket同时是应用程序接口（API）
 - 支持着网络应用的创建



两类进程可使用的通信方式

- **Datagram Socket (UDP)**
 - “尽力而为” 的传输
 - 无连接
- **Stream Socket (TCP)**
 - 字节流
 - 可靠的传输
 - 面向连接

Datagram Socket: 基于UDP协议

UDP

- 单一Socket接收报文
- 不保证一定送达
- 不一定按顺序传送
- 数据报是独立的包
- 每个包都必须包含地址

邮政通信

- 单一邮箱接受信件
- 不可靠
- 不一定按顺序投递
- 信件独立发送
- 每封邮件都必须有地址

UDP应用示例:

多媒体、voice over IP通信（网络电话）

Stream Socket: 基于TCP协议

TCP

- 通信可靠，保证送达
- 以字节流形式按序传送
- 面向连接的通信
- 首先建立连接，再是数据传输

电话通信

- 保证送达
- 按顺序发送信号
- 面向连接的通信
- 先建立连接（接通电话），再数据传输（通话）

TCP应用示例：
网页浏览、电子邮件

Socket的标识

- **传输协议**

- TCP (Stream Socket): 流式传输, 可靠
- UDP (Datagram Socket): 数据包, 尽力而为

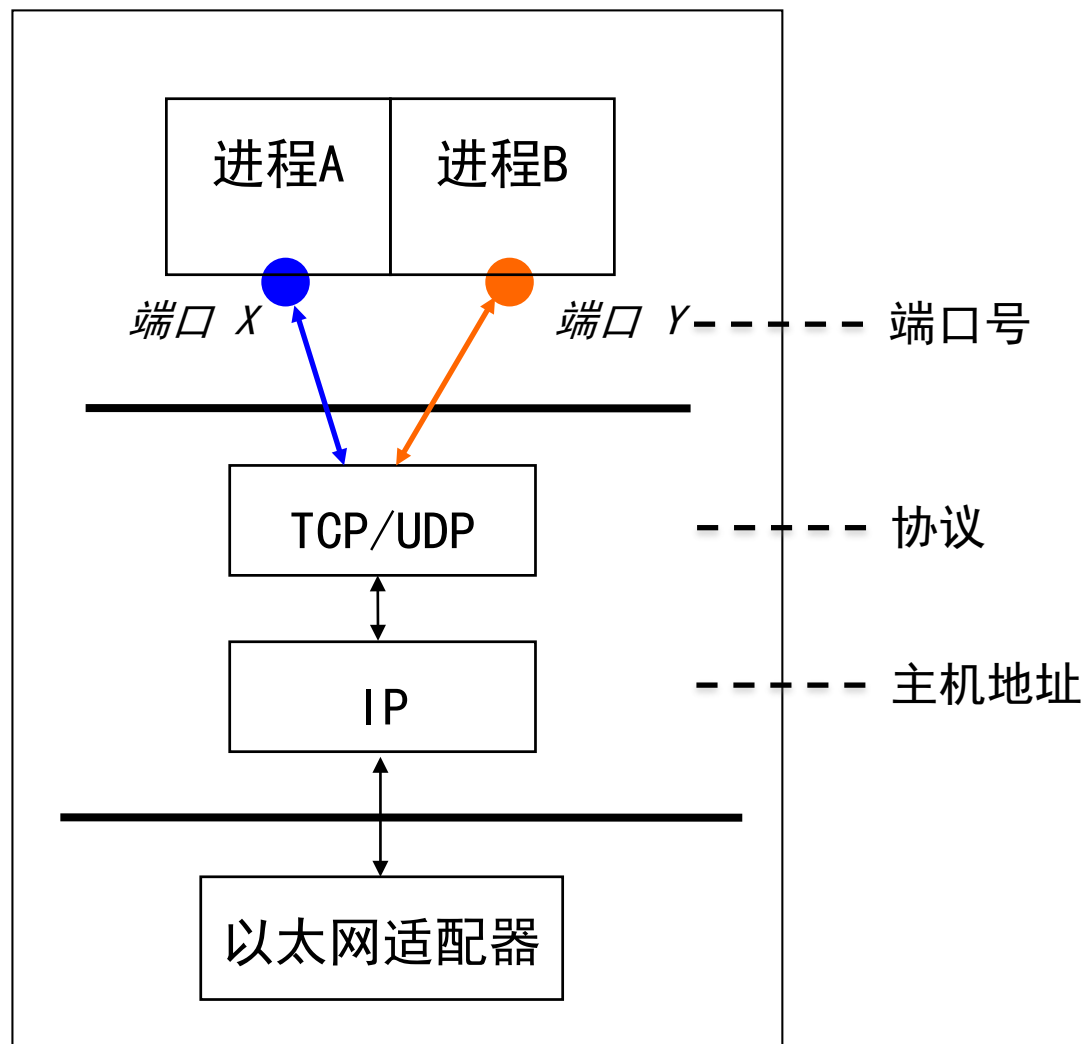
- **接收主机**

- 主机地址唯一标志着主机
- IP 地址是32bit的量

- **接收Socket**

- 同一个主机可能运行着很多不同的进程
- 目的端口唯一标志着这些进程的socket
- 端口号是16bit的量 (0-65535)

Socket的标识



二、Socket中的客户端 与服务端

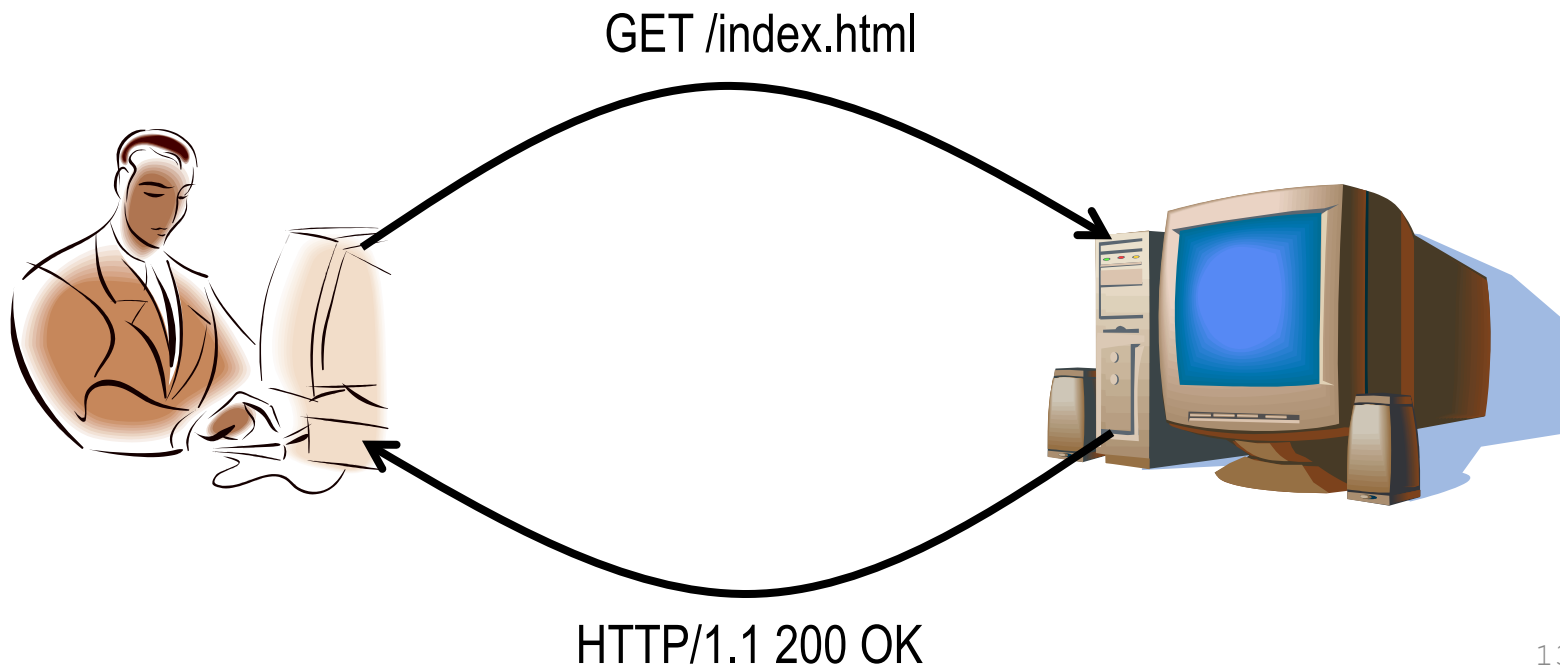
客户端与服务端

• 客户端程序

- 在终端主机运行
- 请求某项服务
- 例子：网络浏览器

• 服务端程序

- 在终端主机运行
- 提供某项服务
- 例子：网页服务器



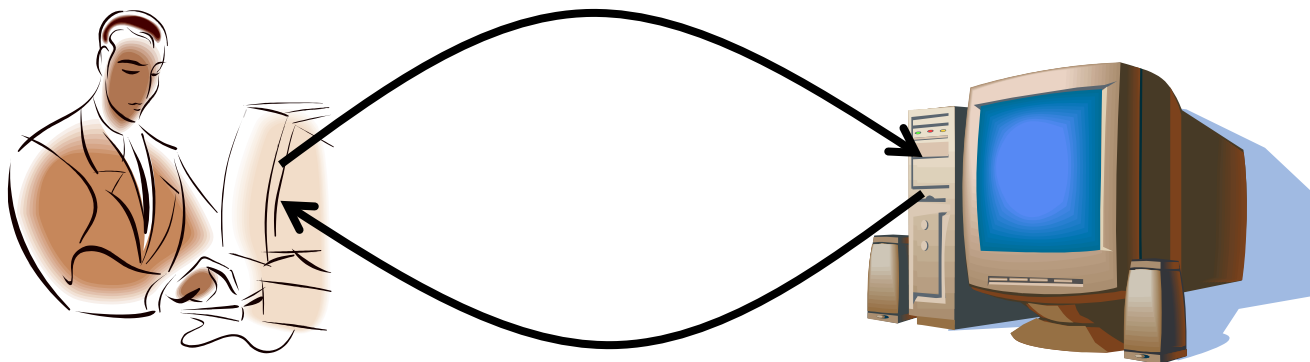
服务端与客户端的通信

- 客户端并非随时在线

- 用户需要时向服务器发起请求
- 不与其他客户端直接建立连接
- 需要知道服务器的地址
- 例：电脑上的网页浏览器

- 服务端一直在线

- 处理大量客户端请求
- 不主动联系客户端
- 需要固定的、可被查询到的地址
- 例：清华主页服务器



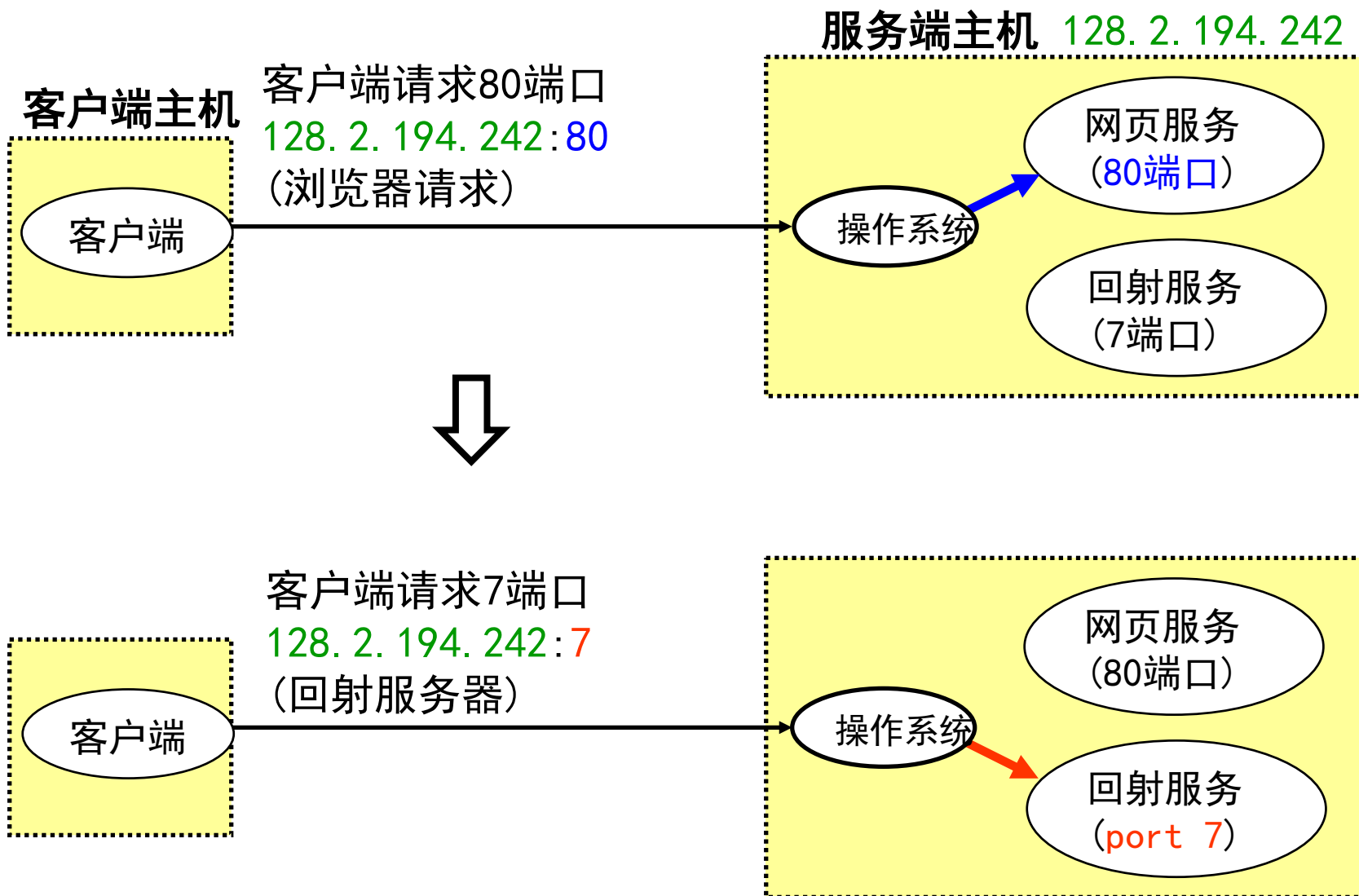
服务端进程与客户端进程

- **客户端进程**
 - 客户端进程发起一个会话
- **服务端进程**
 - 服务端进程等待客户端的连接

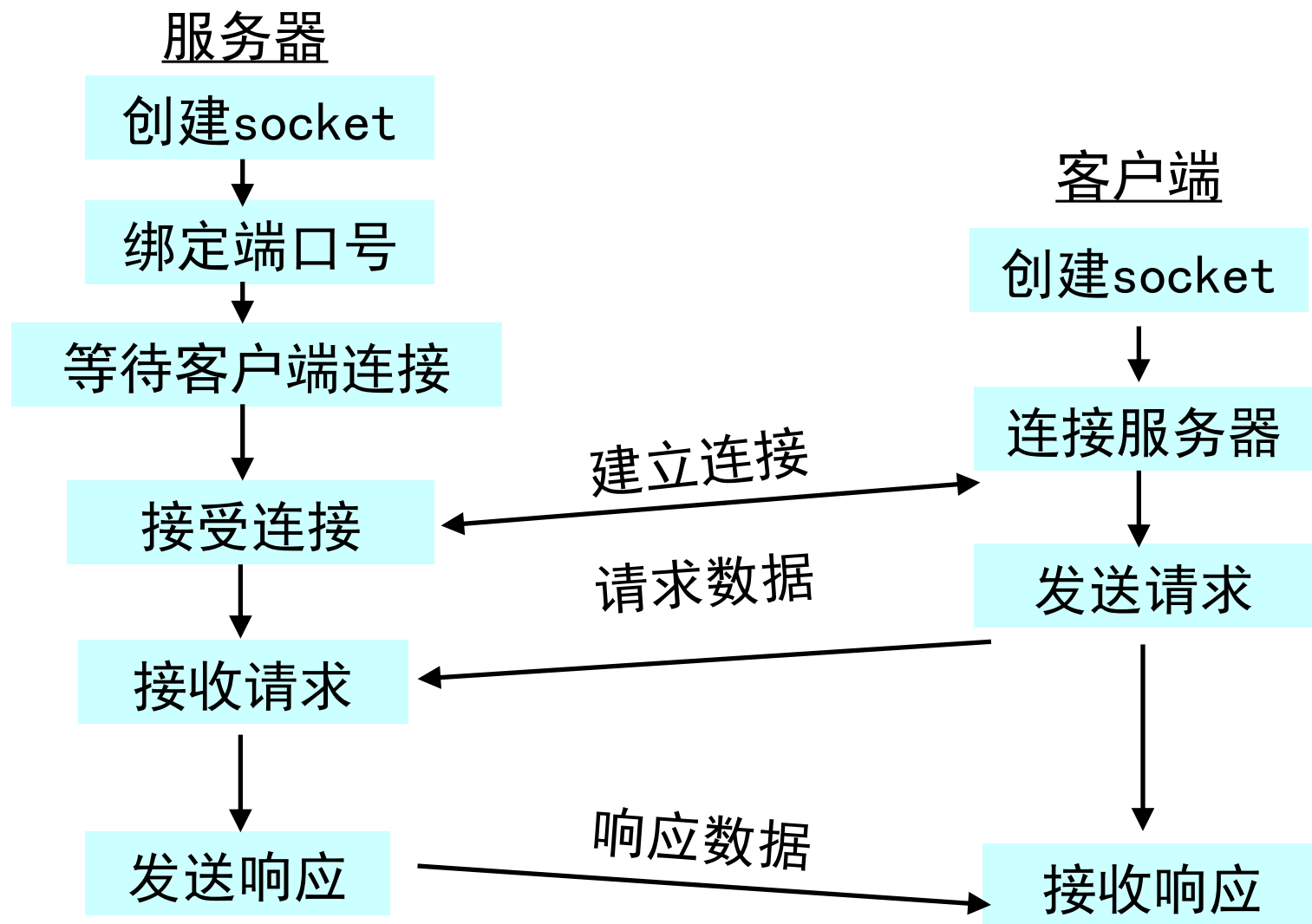
端口号的选择

- 常见应用一般有定义好的通用端口
 - 例子：网页服务采用80端口；电子邮件采用25端口
 - <http://www.iana.org/assignments/port-numbers>
- 通用端口与暂时端口
 - 服务器一般采用通用的端口（如80端口）
 - 一般取值为0到1023 (需要root权限启用这些端口)
 - 客户端一般采用未被占用的临时端口
 - 一般取值为1024到65535
- 如何唯一确定主机之间的流量
 - 两个IP与两个端口号
 - 传输协议（TCP or UDP）

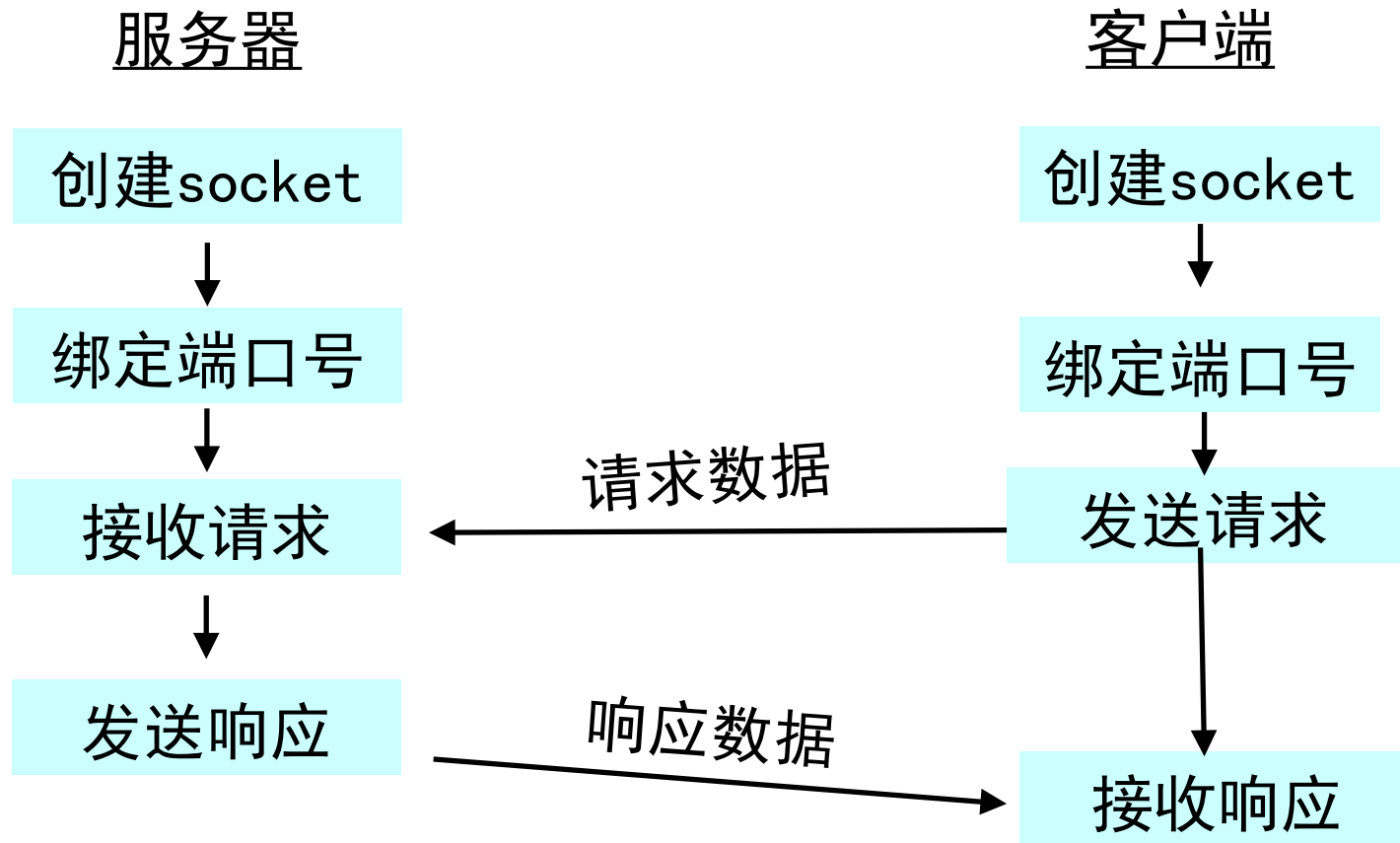
使用端口号确定具体服务



服务端与客户端的通信：面向连接的TCP Socket



服务端与客户端的通信：无连接的UDP Socket

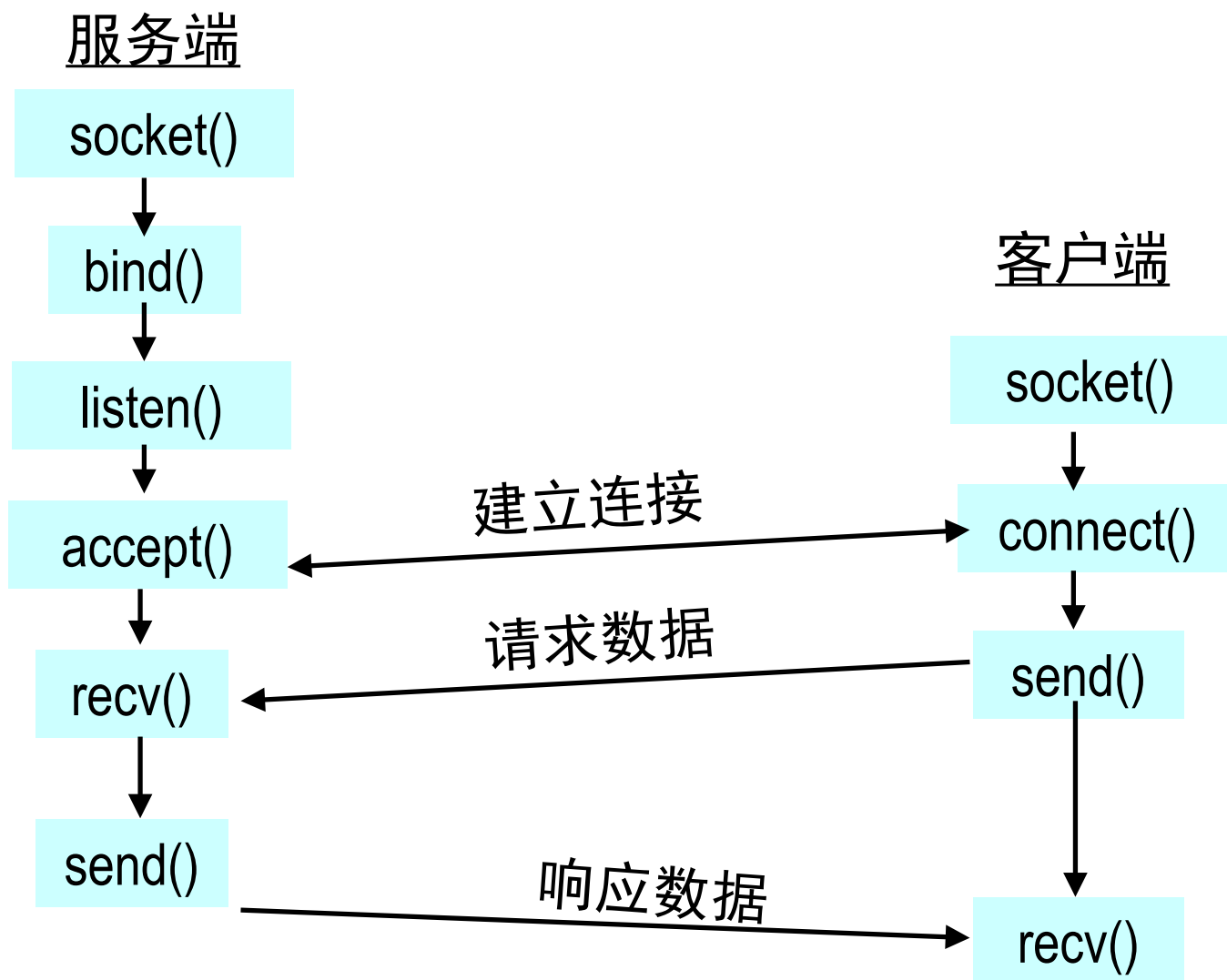


三、UNIX下的 Socket API编程

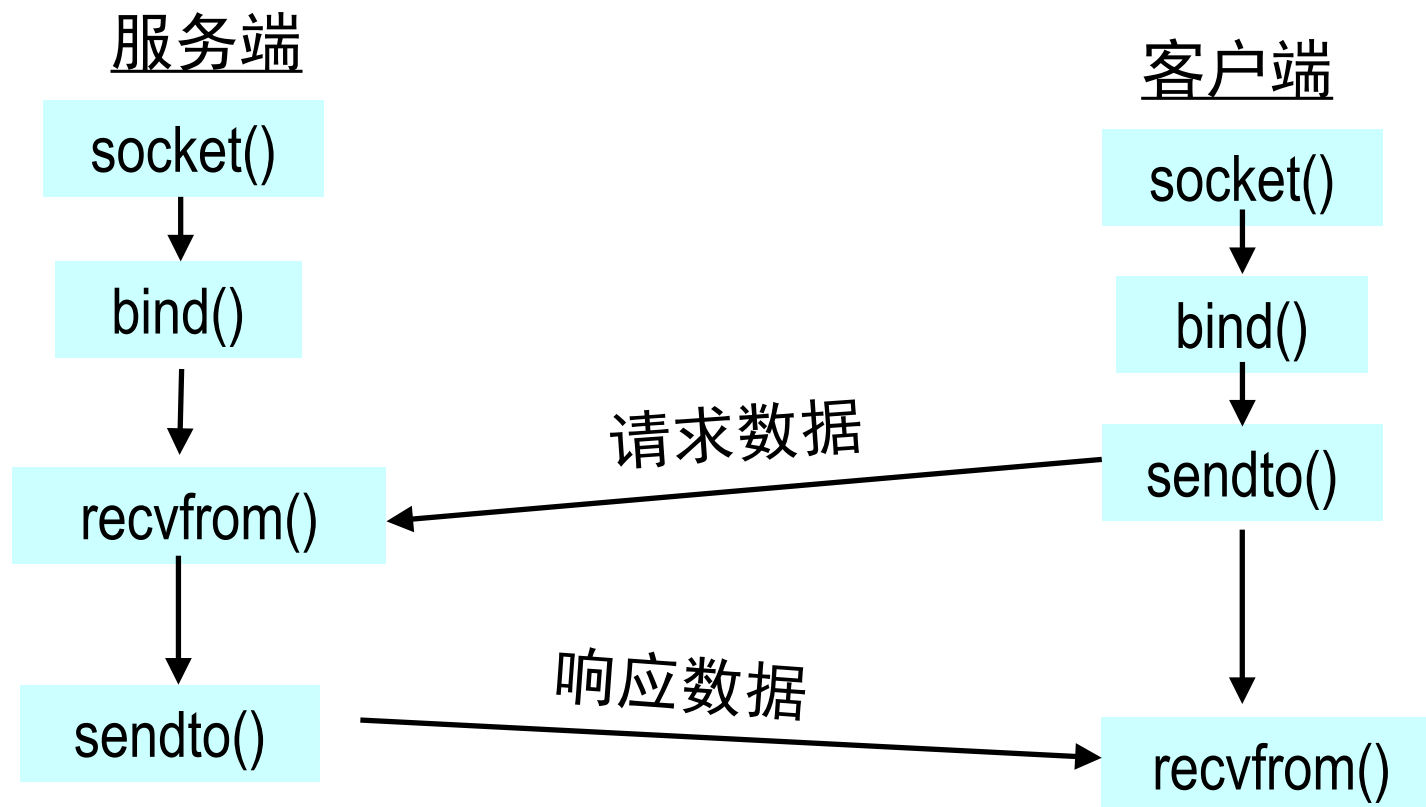
UNIX下的Socket API

- **Socket接口**
 - 最初通过Berkeley UNIX提供实现
 - 后来被所有主流操作系统使用
- **UNIX下，一切即文件**
 - 所有输入都是读文件
 - 所有输出都是写文件
- **Socket API 作为系统调用被实现**
 - 例如connect, send, recv, close, ...

TCP Socket例子



UDP Socket例子



客户端：获得服务器的地址与端口

- 一般服务端以名字与服务被人们所认知
 - 例如 “info.tsinghua.edu.cn” 和 “http”
- 需要将其翻译为IP地址与端口号
 - 例如 “166. 111. 4. 98” 和 “80”
- `Int getaddrinfo(char *node, char *service, struct addrinfo *hints, struct addrinfo **result)`
 - `*node`: 主机名（如 “info.tsinghua.edu.cn” ）或IP地址
 - `*service`: 端口号或服务名称（位于`/etc/services`中， 如ftp）
 - `hints`: 指向已知信息的结构体`addrinfo`

客户端：获得服务器的地址与端口

- 主机地址信息数据结构体定义

```
struct addrinfo {  
    int ai_flags;  
    int ai_family; //e.g. AF_INET for IPv4  
    int ai_socktype; //e.g. SOCK_STREAM for TCP  
    int ai_protocol; //e.g. IPPROTO_TCP  
    size_t ai_addrlen;  
    char *ai_canonname;  
    struct sockaddr *ai_addr; // point to sockaddr struct  
    struct addrinfo *ai_next;  
}
```

客户端：获得服务器的地址与端口

- 主机地址信息数据结构用例

```
hints.ai_family = AF_UNSPEC;    // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
int status = getaddrinfo("info.tsinghua.edu.cn", "80", &hints, &result);

// result now points to a linked list of 1 or more addrinfos, etc.
```

客户端：获得服务器的地址与端口

- **创建socket**

- `int socket(int domain, int type, int protocol)`
- 返回一个文件提示符表示socket (一切即文件)

- **Domain: 协议族**

- PF_INET 为 IPv4
- PF_INET6 为 IPv6

- **Type: 通信类型**

- SOCK_STREAM: TCP
- SOCK_DGRAM: UDP

客户端：获得服务器的地址与端口

- **创建socket**

- `int socket(int domain, int type, int protocol)`
- 返回一个文件提示符表示socket (一切即文件)

- **Protocol: 指明协议**

- UNSPEC: 未指明
- (PF_INET 与 SOCK_STREAM 已经表明使用TCP协议)

- **例子**

```
sockfd = socket(result->ai_family,result->ai_socktype,  
                result->ai_protocol);
```

客户端：获得服务器的地址与端口

- **客户端联系服务器以建立连接**

- 将创建的socket指向服务端地址与端口
- 获得本地端口号（由操作系统指定）
- 向服务器发起请求
- 连接被阻塞（blocking）

- **建立连接**

- `int connect(int sockfd, struct sockaddr *server_address, socketlen_t addrlen)`
- 参数：socket描述符，服务端地址，地址长度
- 返回值0表示成功，-1表示出错
- 例子：`connect(sockfd, result->ai_addr, result->ai_addrlen);`

客户端：发送数据

- 发送数据

`int send(int sockfd, void *msg, size_t len, int flags)`

- 参数：socket描述符，发送数据缓存区指针，缓存区长度
- 返回写入的字节数；返回-1表示错误
- send函数是阻塞的，仅当数据被发送后才给出返回值
- 需要将较短的信息预先写入缓存区，再一次发送

客户端：接收数据

- 接收数据

- `int recv(int sockfd, void *buf, size_t len, int flags)`

- 参数：socket描述符，接收数据缓存区指针，缓存区长度
 - 返回读取的字节数；0表示end of file，-1表示错误
 - 思考：为什么需要len这个参数？如果缓存区长度小于len会如何？
 - `recv`函数是阻塞的，仅当数据接收到以后才给出返回值

服务器：准备socket

- **服务器创建socket并且绑定IP地址与端口**
 - 服务器创建socket（与client一样）
 - 服务器为socket分配端口号（IP为主机IP）
- **创建socket**
 - `int socket(int domain, int type, int protocol)`
- **绑定IP与端口号**
 - `int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)`

服务器：允许客户端等待

- 客户端请求数量往往较大
 - 服务端不能同时处理所有客户端请求
 - 服务端可拒绝请求，也可令请求等待
- 定义多少请求可被挂起
 - `int listen(int sockfd, int backlog)`
 - 参数：socket描述符，允许等待的客户端数量
 - 返回0成功，-1表示错误
 - listen是非阻塞的：立刻提供返回值
- 当过多客户端到来时
 - 一些请求会被拒绝！尽力而为的服务



服务器：接受客户端请求

- 服务器需一直等待客户端请求
 - 被动地等待客户端请求到来
 - 在客户端请求到来之前一直阻塞
 - 之后继续接受新的请求



- 接受客户端的新连接

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)

- 参数：socket描述符，提供客户端地址和端口的数据结构体，结构体长度
- 返回新连接的socket描述符

客户端与服务器：关闭连接

- 当连接仍启用时
 - 两边均可读写
 - 两条单向数据流
 - 在实践中，客户端先发起写请求，服务端读请求；之后服务端写、客户端读，以此类推
- 关闭连接
 - 两边均可关闭连接
 - `int close(int sockfd)`
- 关闭连接时，仍在传输中的数据会怎样？
 - 关闭连接后，仍在传输的数据依然会到达对方
 - 因此，服务端可以在客户端完成读取之前关闭socket

服务器：一次服务一个客户端？

- **串行服务用户请求效率低下**
 - 服务端可以一次只处理一个请求
 - 但所有其他客户必须排队等待
- **服务器需要复用：**
 - 轮换执行不同请求
 - 做一部分请求的准备，之后切换到正在等待其他资源的请求（如读取磁盘文件）
 - 非阻塞进行IO读写
 - 为每个请求分配不同的进程或线程
 - 允许操作系统将CPU计算资源分配给不同进程
 - 融合上述两种方法

利用fork()服务多个客户端

- 处理多客户端请求的流程
 - 创建循环，使用accept()等待客户端请求
 - 客户端连接后，调用fork()创建子进程处理请求请求
 - 主进程继续等待客户端请求
 - 子进程处理结束后close()
- 可进一步参考 *Beej's guide to network programming*

现实中客户端与服务器例子

- **Apache Web server**
 - 在1995年开源的web服务器程序
 - 名字由来：“a patchy server”
 - <http://www.apache.org>
- **Mozilla Web browser**
 - <http://www.mozilla.org/developer/>
- **Sendmail**
 - <http://www.sendmail.org/>
- **BIND 域名系统**
 - Client resolver and DNS server
 - <http://www.isc.org/index.pl?/sw/bind/>
- ...

四、实验内容介绍

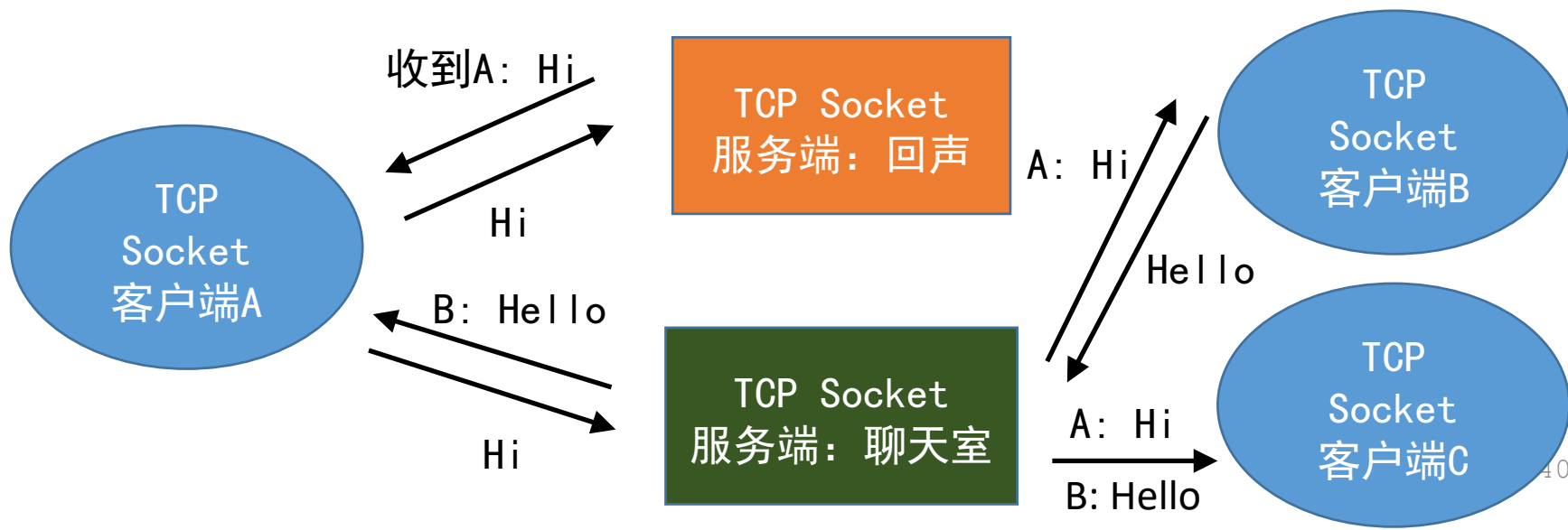
1. 实现客户端聊天程序

• 功能设计

- 实现TCP Socket客户端，连接至服务端正常发送/接收消息
- 输入q后能够正常退出聊天

• 实验验证

- 连接至助教提供的服务端进行测试（回声\聊天室）



1. 实现客户端聊天程序

• 功能设计

- 实现TCP Socket客户端，连接至服务端正常发送/接收消息
- 输入q后能够正常退出聊天

• 实验验证

- 连接至助教提供的服务端进行测试（回声\聊天室）

```
Default (python)
Last login: Tue Sep  6 16:17:50 on ttys002
(base) hanzhenyu@mbp13 ~ % cd ~/code_base/Mine/通信与网络助教/1/完整答案
(base) hanzhenyu@mbp13 ~/code_base/Mine/通信与网络助教/1/完整答案 % python chat_client.py
请输入聊天服务器IP
127.0.0.1
请输入聊天服务器端口
1234
与 127.0.0.1 连接建立成功，可以开始聊天了！（输入q断开连接）
input from client 1
('127.0.0.1', 61743):input from client 2
('127.0.0.1', 61743)离开了
```

能够发送消息到其他 client
能够实现退出功能

连接至聊天室服务端示例

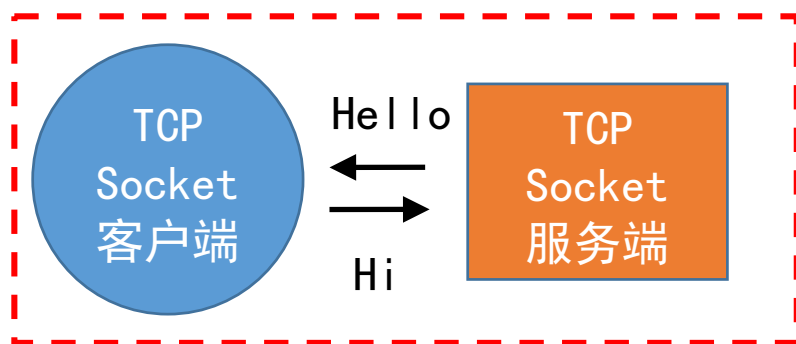
2. 实现一对一服务端聊天程序

- 功能设计

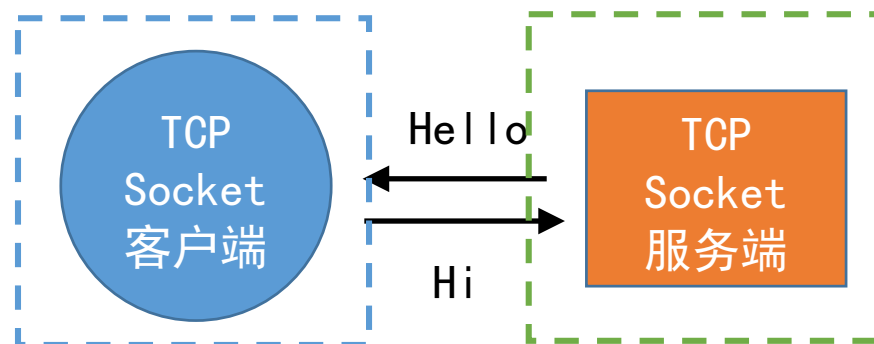
- 实现TCP Socket服务端，等待客户端连接后一对一聊天

- 实验验证

- 同一主机客户端与服务端连接
 - 不同主机客户端与服务端连接（与其他同学组队）



同一主机

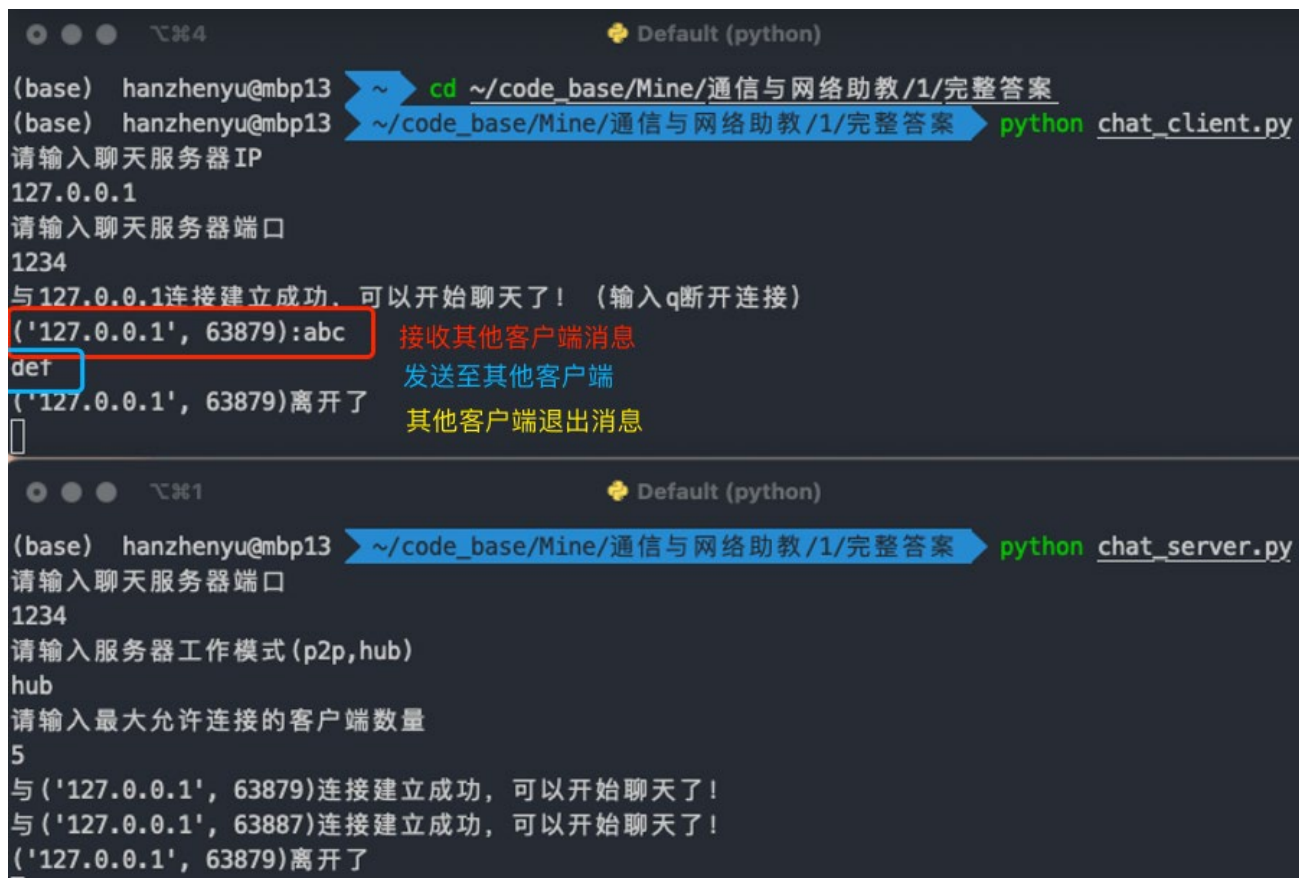


不同主机

3. 【选做】实现聊天室服务端

• 功能设计

- 实现TCP Socket服务端，将客户端消息广播给连接的其他所有其他客户端



The image displays two terminal windows side-by-side, illustrating the setup and operation of a chat server and client.

Top Terminal (chat_client.py):

- Environment: (base) hanzhenyu@mbp13
- Command: `cd ~/code_base/Mine/通信与网络助教/1/完整答案`
- Command: `python chat_client.py`
- Prompt: 请输入聊天服务器IP
- Input: 127.0.0.1
- Prompt: 请输入聊天服务器端口
- Input: 1234
- Status: 与 127.0.0.1 连接建立成功，可以开始聊天了！（输入q断开连接）
- Message: ('127.0.0.1', 63879):abc (highlighted in red)
- Message: def (highlighted in blue)
- Message: ('127.0.0.1', 63879)离开了
- Annotations on the right:
 - 接收其他客户端消息 (corresponding to 'abc')
 - 发送至其他客户端 (corresponding to 'def')
 - 其他客户端退出消息 (corresponding to the disconnect message)

Bottom Terminal (chat_server.py):

- Environment: (base) hanzhenyu@mbp13
- Command: `~/code_base/Mine/通信与网络助教/1/完整答案 python chat_server.py`
- Prompt: 请输入聊天服务器端口
- Input: 1234
- Prompt: 请输入服务器工作模式 (p2p, hub)
- Input: hub
- Prompt: 请输入最大允许连接的客户端数量
- Input: 5
- Status: 与 ('127.0.0.1', 63879) 连接建立成功，可以开始聊天了！
- Status: 与 ('127.0.0.1', 63887) 连接建立成功，可以开始聊天了！
- Status: ('127.0.0.1', 63879) 离开了

4. 注意事项

- 编程语言

- 提供Python与C++两种代码框架
- Python版本有较好的跨平台特性，而C++在不同平台有不同的socket实现；提供的C++代码基于Unix实现

- 实验考核

- 提交实验报告+代码至网络学堂
- 实验报告需包括实验中的重要现象、思考题回答