

# 计算机网络

## 第二章 应用层协议及网络编程

徐敬东 张建忠

[xujd@nankai.edu.cn](mailto:xujd@nankai.edu.cn)

[zhangjz@nankai.edu.cn](mailto:zhangjz@nankai.edu.cn)

计算机网络与信息安全研究室

**总体目标：**理解应用层协议与进程通信模型，掌握Socket编程方法，学习典型的应用层协议

理解客户/服务器模型和对等计算模型，初步了解传输层服务及对应用层的支持

掌握基于套接字的网络编程方法，理解数据报式套接字和流式套接字的功能

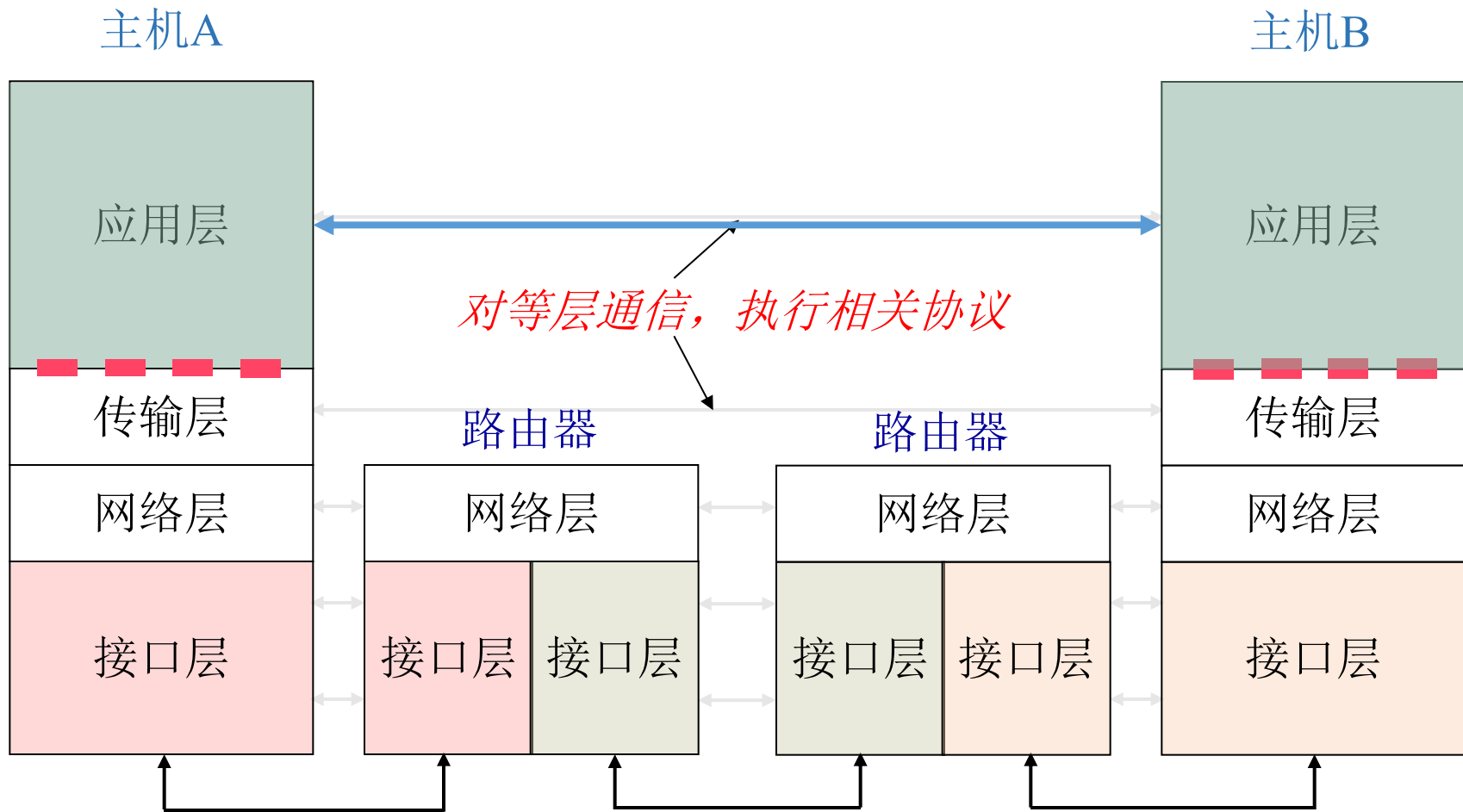
掌握域名系统构成和解析过程，理解根域名服务器在国家网络基础设施中的重要作用

掌握Web服务的特点、HTTP 1.0和1.1的工作机制及所面临的性能问题，以及HTTP/2的优化机制和解决的关键问题

掌握内容分发网络所解决的问题和基本工作机制，理解两种基本的重定向方法，了解动态自适应流媒体协议的基本思想

- 2.1 应用协议与进程通信模型
- 2.2 传输层服务对应用的支持
- 2.3 Socket编程
- 2.4 域名系统
- 2.5 传统的应用层服务与协议
- 2.6 Web服务与HTTP协议
- 2.7 内容分发网络CDN
- 2.8 动态自适应流媒体协议DASH

## TCP/IP体系结构



接口层通常包括数据链路层和物理层

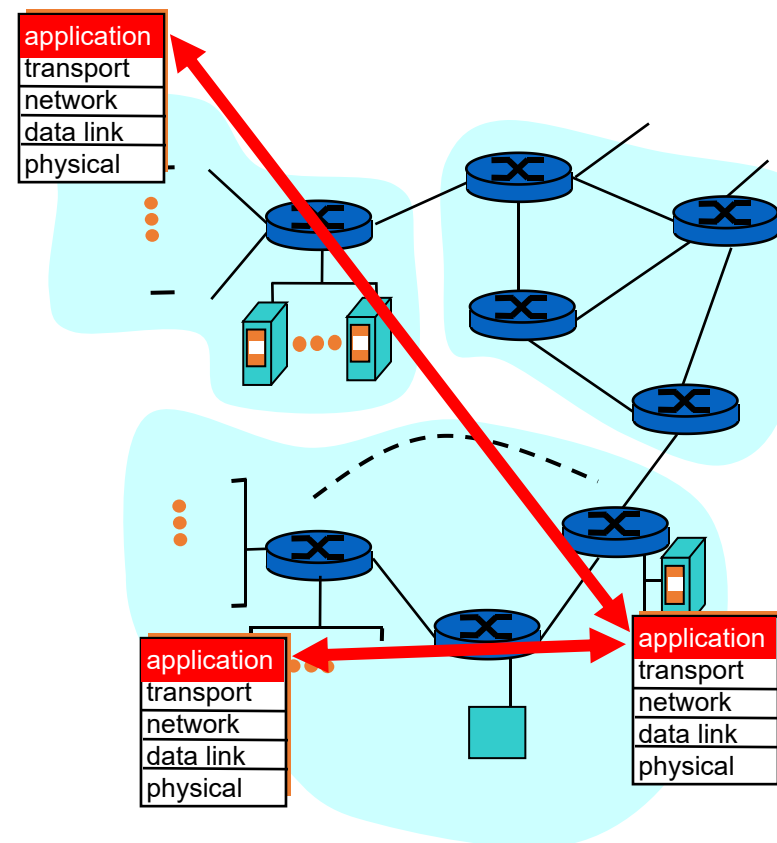
## 应用与应用层协议

■ **应用：** 可进行通信的、分布式进程

- 运行于主机的**用户空间**
- 通过交换**消息**（**Messages**）实现应用之间的交互
- 例如：Email、Web等

■ **应用层协议：** 应用层实体之间的通信规范

- 定义应用**交换的消息**和收到消息后**采取的行动**
- 使用下层协议（TCP、UDP）提供的通信服务



# 进程间通信

**进程：**主机中运行的程序

- 在同一台主机中，两个进程之间按照**进程间通信方式**进行交互通信（操作系统中定义）
- 不同主机上的进程通信，需要通过**交换消息**来完成

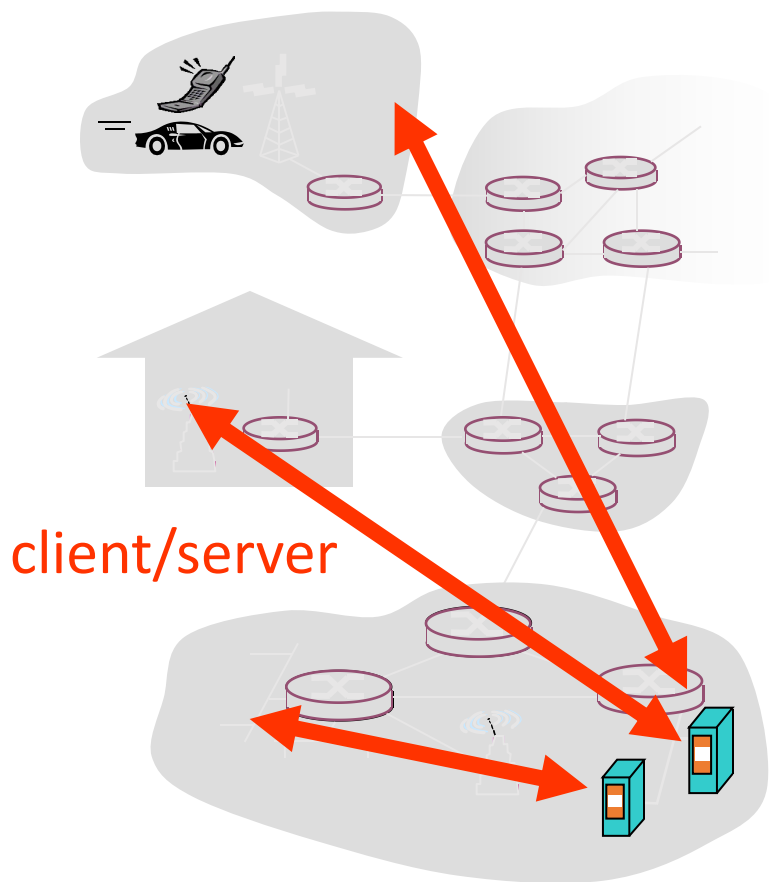
### □ 客户/服务器（C/S）模型

- ❖ 客户向服务器发出服务请求，并接收服务器的响应；服务器等待客户的请求并为客户提供服务
- ❖ 例如：Web浏览器/Web服务器；Email客户端/Email服务器

### □ 对等计算（P2P）模型

- ❖ 最小化（或根本不用）专用服务器
- ❖ 例如：Skype, BitTorrent等

## 客户/服务器进程交互模型



### 服务器进程:

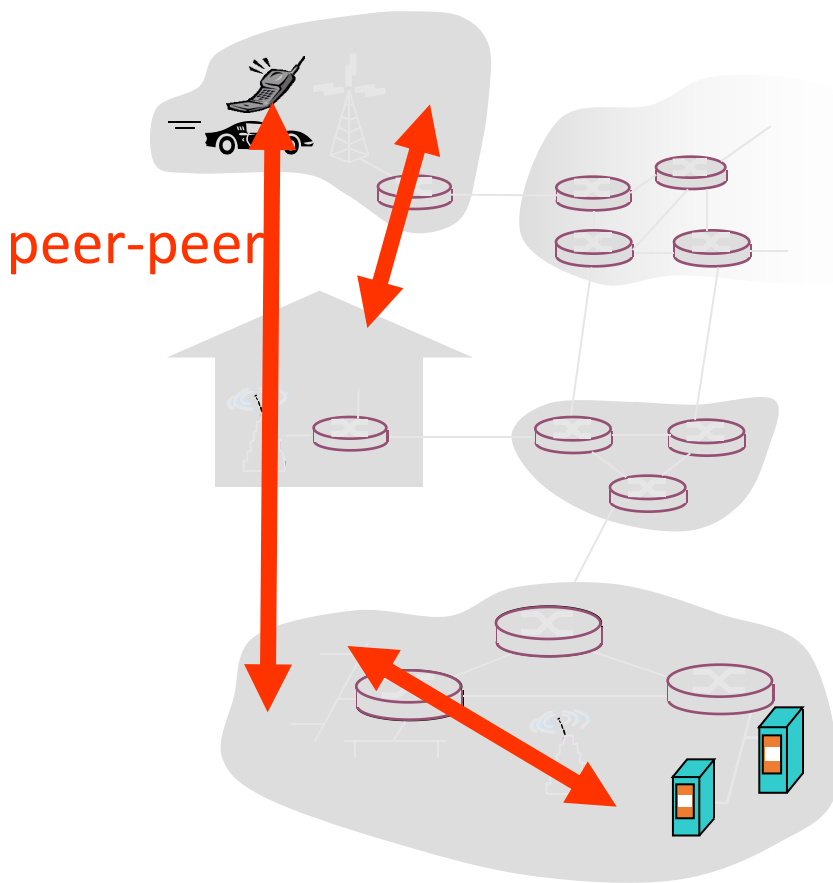
- 被动等待
- 长久在线
- 固定IP地址
- 利用集群/云提供扩展性

### 客户进程:

- 启动与服务器的通信
- 可能为间歇性连接
- 可能使用动态IP地址
- 不与其他客户进行直接通信

### 纯P2P进程交互模型

- 无长久在线的服务器
- 任意的终端系统之间都可能进行直接通信
- 端系统之间可能间歇性地进行连接
- 端系统可能使用动态的IP地址





### 客户/服务器与P2P的混合

#### Skype

- VoIP P2P应用
- 中心服务器：远程客户的地址发现
- 端-端连接：直连（不通过服务器）

#### 即时消息

- 用户间利用P2P模式进行消息发送与聊天
- 中心服务器：用户呈现探测与位置发现
  - 当在线时，用户向中心服务器注册他的IP地址
  - 用户利用中心服务器搜索对方用户的IP地址

### 进程的地址标识

- 为了发送和接收消息，进程必须具有一个标识符
- 主机拥有一个唯一的32位的IPv4地址（或128位的IPv6地址）
- 利用主机的IP地址表示主机中的进程是否可以？
  - 不可以。一个主机中可能同时运行着多个进程。

### 进程的地址标识（续）

- **进程标识符**：包括**IP地址**和**端口号**
- **端口号举例**：
  - Web服务器进程: 80
  - Email（SMTP）服务器进程: 25
- 为了向www.nankai.edu.cn的Web服务器发送消息，Web服务器需要使用的进程标识符为
  - IP地址：222.30.45.190（主机的IP地址）
  - 端口号：80

## 应用层协议定义的内容

### ■ 消息的类型

- 如请求request、响应response

### ■ 消息的语法

- 如消息包含哪些字段、字段之间如何分割等

### ■ 消息的语义

- 字段中信息代表的具体含义

### ■ 消息的处理

- 进程何时发送消息、收到消息后的动作等

### 公共协议

- RFC中定义的协议
- 可相互兼容
- 例如：HTTP、SMTP等

### 专有协议

- 公司或组织专有
- 例如：Skype、QQ等

### 应用需要怎么的传输层服务？

#### ■ 数据丢失率

- 音视频等应用可以容忍一定的数据丢失
- 文件传输、远程登录等应用要求100%的数据可靠

#### ■ 时延

- 网络电话、交互游戏等应用对时延有一定的要求

#### ■ 带宽

- 多媒体等应用需要一定的带宽保证
- 有些应用则是弹性的

### 常用应用对传输层的要求

应用	数据丢失	带宽	时延
文件传输	否	弹性	否
电子邮件	否	弹性	否
Web文档	可容忍	弹性	否
实时音视频	可容忍	音频: 5kbps-1Mbps 视频: 10kbps-5Mbps	是
可缓存音视频	可容忍	同上	是
交互游戏	可容忍	高于几kbps	是
即时消息	否	弹性	是或否

### 互联网传输层提供的服务

#### TCP服务:

- **面向连接:** 客户与服务器之间需要建立连接
- **可靠传输:** 可保证传递数据无差错
- **流量控制:** 发送数据不会超过接收端的容纳容量
- **拥塞控制:** 提供拥塞解决方案
- **不能提供:** 时延和带宽保证

#### UDP服务:

- **不可靠:** 不可靠的数据投递
- **不能提供:** 连接建立、可靠性、流量控制、拥塞控制、时延和带宽保证

### 互联网应用：常用应用使用的传输层服务

应用	应用层协议	传输层协议
电子邮件	SMTP [RFC 2821]	TCP
Web服务	HTTP [RFC 2616]	TCP
文件传输	FTP [RFC 959]	TCP
流媒体	HTTP、RTP	TCP or UDP
网络电话	RTP [RFC 1889] SIP、专有协议	典型为UDP



### 网络编程界面

- TCP/IP协议通常在操作系统的**内核**中实现
- **编程界面**：由操作系统提供的功能调用，可以使应用程序方便地使用内核的功能
- **socket（套接字）**：支持TCP/IP的操作系统为网络程序开发提供的**典型**网络编程界面

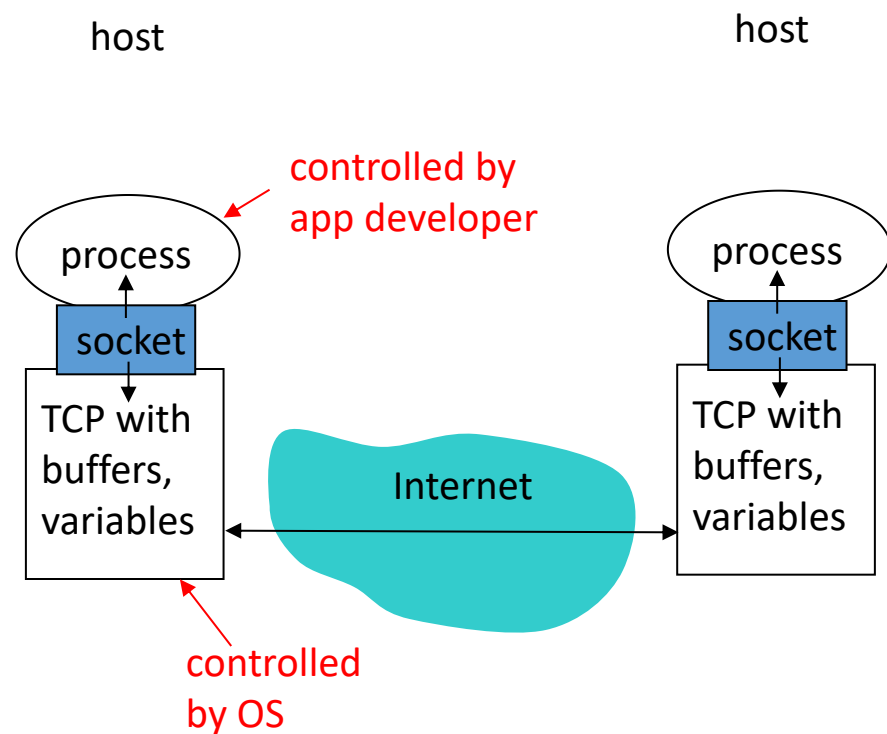
### 套接字sockets

■ 进程通过**套接字**发送消息和接收消息

■ 套接字可以看成一道“门”

- 发送进程把消息从“门”推出去
- 发送进程推出去的消息利用下层的通信设施传递到接收进程所在的“门”
- 接收进程再从“门”把消息拉进去

■ **API:** (1) 选择使用的传输层协议; (2) 对套接字的一些参数进行修改



### 套接字sockets

- **数据报套接字**（datagram sockets）：使用UDP协议，支持主机之间面向非连接、不可靠的数据传输
- **流式套接字**（stream sockets）：使用TCP协议，支持主机之间面向连接的、顺序的、可靠的、全双工字节流传输
- 支持socket的**操作系统**：Windows、UNIX、Linux、iOS、Android等
- 支持socket的**编程语言**：C、C++、Python、Java、VB等

## TCP服务：

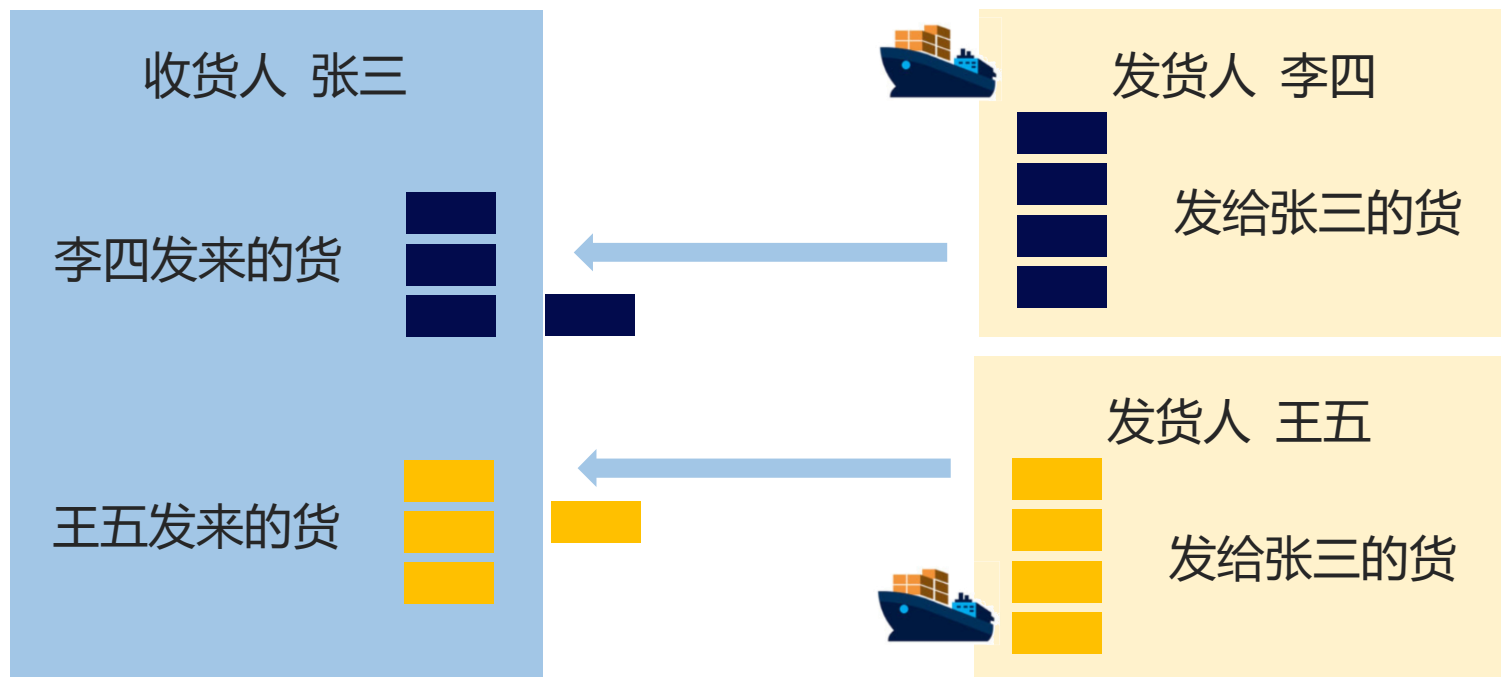
- **面向连接：** 客户与服务器之间需要建立连接
- **可靠传输：** 可保证传递数据无差错
- **流量控制：** 发送数据不会超过接收端的容纳容量
- **拥塞控制：** 提供拥塞解决方案
- **不能提供：** 时延和带宽保证

## UDP服务：

- **不可靠：** 不可靠的数据投递
- **不能提供：** 连接建立、可靠性、流量控制、拥塞控制、时延和带宽保证

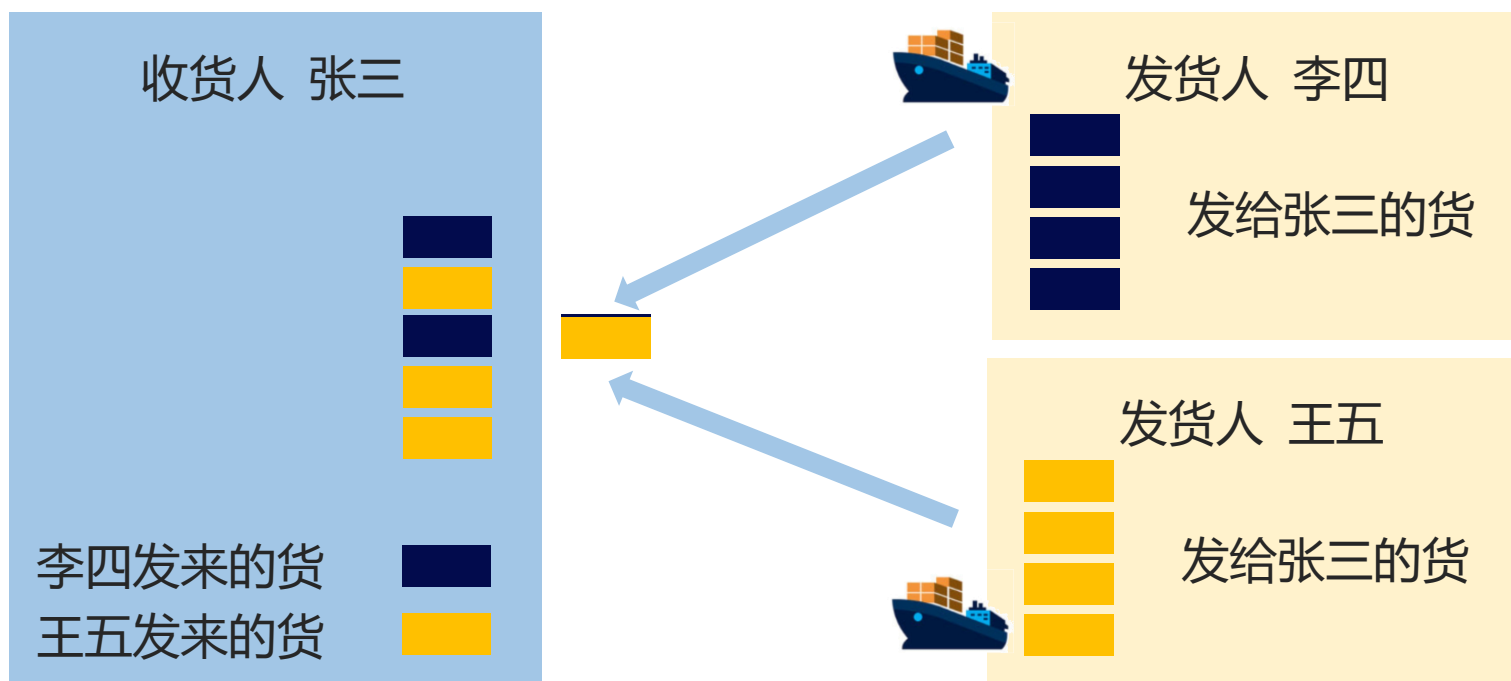


## TCP传输示意图

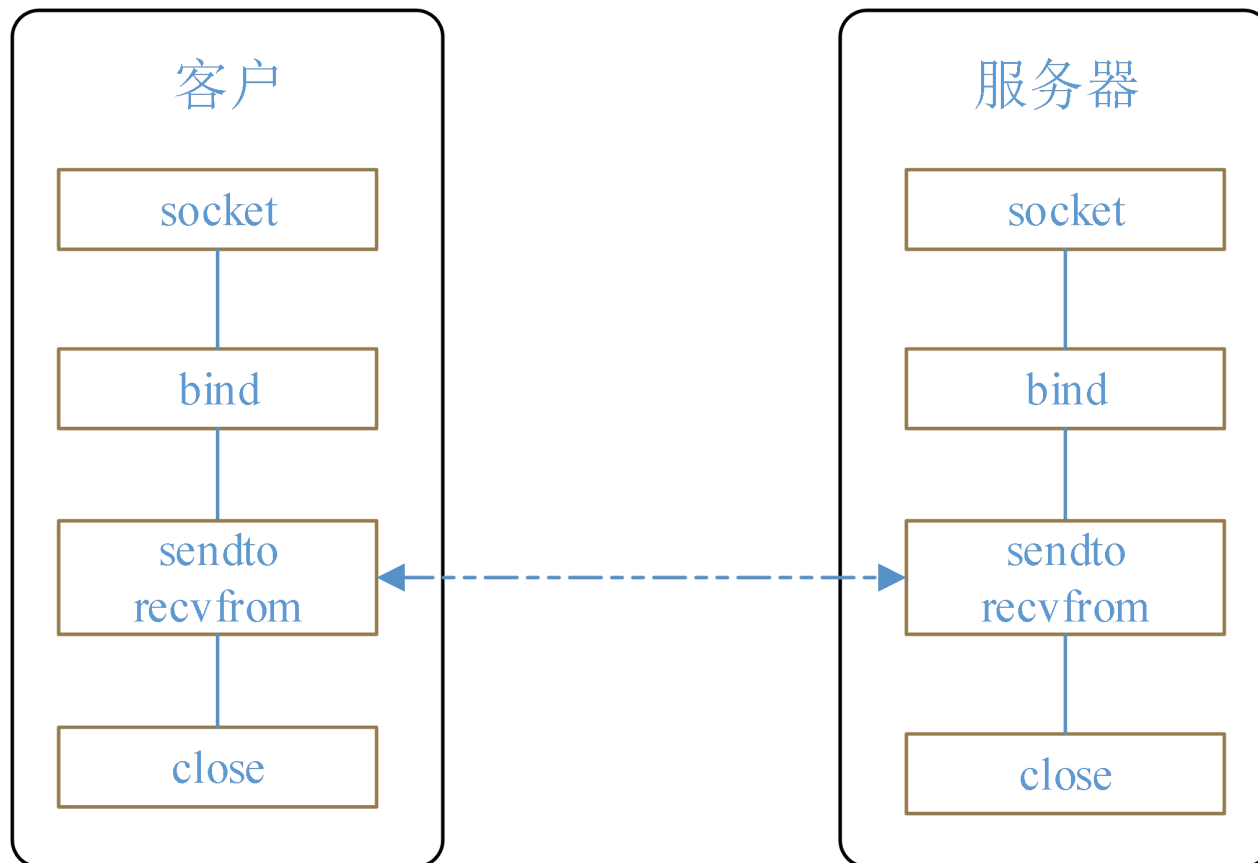




## UDP传输示意图



## 2.3 利用UDP服务的应用程序编写步骤



## 2.3 利用TCP服务的应用程序编写步骤





### WSAStartup

```
int WSAAPI WSAStartup (                //成功返回0， 否则为错误代码
    WORD wVersionRequested,            //调用者希望使用的最高版本
    LPWSADATA lpWSADATA                //可用Socket的详细信息
);
```

- **功能：** 初始化Socket DLL， 协商使用的Socket版本
- **WSADATA：**
  - wVersion: 推荐调用者使用的Socket版本号
  - wHighVersion: 系统实现的Socket最高版本号
- 如果调用成功， 不再使用时需要调用WSACleanup释放Socket DLL资源

## WSAStartup

Caller version support	Winsock DLL version support	wVersion requested	wVersion returned	wHighVersion returned	End result
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	Application fails
1.0	1.1	1.0	—	—	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.0 1.1	2.0	1.1	1.1	use 1.1
2.0	1.0 1.1 2.0	2.0	2.0	2.0	use 2.0
2.0 2.2	1.0 1.1 2.0	2.2	2.0	2.0	use 2.0
2.2	1.0 1.1 2.0 2.1 2.2	2.2	2.2	2.2	use 2.2

### WSACleanup

```
int WSAAPI WSAStartup(  
    //成功返回0  
);
```

- **功能：** 结束使用Socket，释放Socket DLL资源
- 调用失败后可利用WSAGetLastError获取详细错误信息

```
socket          SOCKET WSAAPI socket(  
                int af,  
                int type,  
                int protocol  
                );
```

- **功能：** 创建一个Socket，并绑定到一个特定的传输层服务
- **参数：**
  - af: 地址类型。AF\_INET、AF\_INET6等
  - type: 服务类型。SOCK\_STREAM、SOCK\_DGRAM等
  - Protocol: 协议。IPPROTO\_TCP、IPPROTO\_UDP、IPPROTO\_ICMP等。  
如为0，则由系统自动选择
- **返回：**
  - 正确: Socket描述符
  - 错误: INVALID\_SOCKET。可通过WSAGetLastError获取错误详情

bind

```
int bind(  
    SOCKET s,  
    const struct sockaddr *addr,  
    int namelen  
);
```

- **功能：** 将一个本地地址绑定到指定的Socket
- **参数：**
  - s: socket描述符。
  - addr: 地址。包括IP地址和端口号。如为INADDR\_ANY和in6addr\_any, 则由系统自动分配。
  - namelen: 地址长度。通常为sockaddr结构的长度
- **返回：**
  - 正确: 0
  - 错误: SOCKET\_ERROR。可通过WSAGetLastError获取错误详情

```
listen          int WSAAPI listen(  
                SOCKET          s,  
                int              backlog  
                );
```

- **功能：** 使socket进入监听状态，监听远程连接是否到来
- **参数：**
  - s: socket描述符。
  - backlog: 连接等待队列的最大长度
- **返回：**
  - 正确: 0
  - 错误: SOCKET\_ERROR。可通过WSAGetLastError获取错误详情
- **使用：** 流方式，Server端

```
connect      int WSAAPI connect(  
              SOCKET          s,  
              const sockaddr   *name,  
              int              namelen  
            );
```

- **功能：** 向一个特定的socket发出建连请求
- **参数：**
  - s: socket描述符。
  - addr: 地址。包括IP地址和端口号。
  - namelen: 地址长度。通常为sockaddr结构的长度
- **返回：**
  - 正确: 0
  - 错误: SOCKET\_ERROR。可通过WSAGetLastError获取错误详情
- **使用：** 流方式、Client端

```
accept          SOCKET WSAAPI accept(  
                SOCKET          s,  
                sockaddr         *addr,  
                int              *addrlen  
                );
```

- **功能：** 接受一个特定socket请求等待队列中的连接请求
- **参数：**
  - s: socket描述符。
  - addr: 返回远程端地址。
  - namelen: 返回地址长度。
- **返回：**
  - 正确: 返回新连接的socket描述符。
  - 错误: INVALID\_SOCKET。可通过WSAGetLastError获取错误详情
- **使用：** 流方式、Server端。通常运行后进入阻塞状态，直到连接请求到来



### sendto

```
int WSAAPI sendto(  
    SOCKET s,  
    const char *buf,  
    int len,  
    int flags,  
    const struct sockaddr *to,  
    int tolen );
```

- **功能：** 向特定的目的地发送数据

- **参数：**

- s: socket描述符。                      buf: 发送数据缓存区。
- len: 发送缓冲区的长度      flags: 对调用的处理方式, 如OOB等
- to: 目标socket的地址      tolen: 目标地址的长度

- **返回：**

- 正确: 返回实际发送的字节数。
- 错误: SOCKET\_ERROR。可通过WSAGetLastError获取错误详情

- **使用：** 数据报方式。

recvfrom

```
int WSAAPI recvfrom(  
    SOCKET    s,  
    char      *buf,  
    int       len,  
    int       flags,  
    sockaddr  *from,  
    int       *fromlen );
```

- **功能：** 从特定的目的地接收数据

- **参数：**

- s: socket描述符。                      buf: 接收数据的缓存区。
- len: 接收缓冲区的长度              flags: 对调用的处理方式, 如OOB等
- from: 源socket的地址              fromlen: 源地址的长度

- **返回：**

- 正确: 接收到的字节数。
- 错误: SOCKET\_ERROR。可通过WSAGetLastError获取错误详情

- **使用：** 数据报方式。

send

```
int WSAAPI send(  
    SOCKET      s,  
    const char *buf,  
    int         len,  
    int         flags );
```

- **功能：** 向远程socket发送数据
- **参数：**
  - s: socket描述符。                      buf: 发送数据缓存区。
  - len: 发送缓冲区的长度      flags: 对调用的处理方式, 如OOB等
- **返回：**
  - 正确: 返回实际发送的字节数。
  - 错误: SOCKET\_ERROR。可通过WSAGetLastError获取错误详情
- **使用：** 流方式。

### closesocket

```
int WSAAPI closesocket(  
    SOCKET s  
);
```

- **功能：** 关闭一个存在的socket
- **参数：**
  - s: socket描述符。
- **返回：**
  - 正确： 0。
  - 错误： SOCKET\_ERROR。可通过WSAGetLastError获取错误详情

### SOCKADDR 结构

```
typedef struct sockaddr {  
    u_short  sa_family;  
    char      sa_data[14];  
} SOCKADDR, *PSOCKADDR, *LPSOCKADDR;
```

### SOCKADDR\_IN 结构

```
typedef struct sockaddr_in {  
    short      sin_family;  
    u_short    sin_port;  
    struct in_addr sin_addr;  
    char       sin_zero[8];  
} SOCKADDR_IN, *PSOCKADDR_IN, *LPSOCKADDR_IN;
```

### in\_addr 结构

```
struct in_addr {  
    union {  
        struct {  
            u_char s_b1;  
            u_char s_b2;  
            u_char s_b3;  
            u_char s_b4;  
        } S_un_b;  
        struct {  
            u_short s_w1;  
            u_short s_w2;  
        } S_un_w;  
        u_long S_addr;  
    } S_un;  
};
```

### Big-Endian和Little-Endian

- **Little-Endian**: 低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。
- **Big-Endian**: 高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。
- 例如：0x12 34 56 78在内存中的存放形式为：
  - 内存                      低地址 -----> 高地址
  - Big-Endian:            0x12 | 0x34 | 0x56 | 0x78
  - Little-Endian:        0x78 | 0x56 | 0x34 | 0x12

### Big-Endian和Little-Endian

- 常见CPU的字节序

- Big-Endian : PowerPC、IBM、Sun
- Little-Endian : **x86**、DEC
- ARM既可工作在Big-Endian, 也可工作在Little-endian

- 网络使用的字节序: 网络通信协议都使用*Big-Endian*编码序

- 主机序与网络序的转换

- htons: host to net -- short
- htonl: host to net -- long
- ntohs: net to host -- short
- ntohl: net to host -- long



### CreateThread

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES    lpThreadAttributes,  
    SIZE_T                   dwStackSize,  
    LPTHREAD_START_ROUTINE   lpStartAddress,  
    __drv_aliasesMem LPVOID  lpParameter,  
    DWORD                    dwCreationFlags,  
    LPDWORD                  lpThreadId );
```

- **功能：** 创建线程

- **参数：**

- lpThreadAttributes: 返回句柄能否被继承。NULL为不能继承。
- dwStackSize: 堆栈的初始大小。0为缺省大小。
- lpStartAddress: 新线程的开始执行地址。自己线程函数的开始地址。
- lpParameter: 传递给线程的参数。
- dwCreationFlags: 控制线程的标志。0为立即执行。
- lpThreadId: 指向进程标识符的指针。NULL为不返回该指针。

- **返回：**

- 正确: 新创建线程的句柄。
- 错误: NULL。可通过GetLastError获取错误详情

## 2.3 Socket编程 – 举例（数据报客户端）



```
void main()
{
    WSStartup(wVersionRequested, &wsaData);

    SOCKET sockClient = socket(AF_INET, SOCK_DGRAM, 0);

    SOCKADDR_IN addrSrv;

    sendto(sockClient, sendBuf, sendlen, 0, (SOCKADDR *)&addrSrv, len);

    recvfrom(sockClient, recvBuf, 50, 0, (SOCKADDR *)&addrSrv, &len);

    closesocket(sockClient);
    WSACleanup();
}
```

## 2.3 Socket编程 – 举例（数据报服务端）



```
void main()
{
    WSAStartup(wVersionRequested, &wsaData);

    SOCKET sockSrv = socket(AF_INET, SOCK_DGRAM, 0);

    SOCKADDR_IN addrSrv;
    bind(sockSrv, (SOCKADDR*)&addrSrv, sizeof(SOCKADDR));

    SOCKADDR_IN addrClient;    //远程IP地址
    loop {
        recvfrom(sockSrv, recvBuf, 50, 0, (SOCKADDR *)&addrClient, &len);
        sendto(sockSrv, sendBuf, sendlen, 0, (SOCKADDR *) &addrClient, len);
    }

    closesocket(sockClient);
    WSACleanup();
}
```

## 2.3 Socket编程 – 举例（流式客户端）



```
void main()
{
    WSAStartup(wVersionRequested, &wsaData);

    SOCKET sockClient = socket(AF_INET, SOCK_STREAM, 0);

    connect(sockClient, (SOCKADDR*)&addrSrv, sizeof(SOCKADDR));

    recv(sockClient, recvBuf, 50, 0);
    send(sockClient, "hello", sendlen, 0);

    closesocket(sockClient);
    WSACleanup();
}
```

## 2.3 Socket编程 – 举例（流式服务端）



```
void main()
{
    WSAStartup(wVersionRequested, &wsaData);

    SOCKET sockSrv = socket(AF_INET, SOCK_STREAM, 0);

    bind(sockSrv, (SOCKADDR*)&addrSrv, sizeof(SOCKADDR));

    listen(sockSrv, 5);

    while (1)
    {
        SOCKET sockConn = accept(sockSrv, (SOCKADDR*)&addrClient, &len);
        send(sockConn, sendBuf, strlen, 0);
        recv(sockConn, recvBuf, 50, 0);
        closesocket(sockConn);
    }

    closesocket(sockSrv);
    WSACleanup();
}
```

## 2.3 Socket编程 – 举例（多线程流式服务端）



```
void main()
{
    WSASStartup(wVersionRequested, &wsaData);
    SOCKET sockSrv = socket(AF_INET, SOCK_STREAM, 0);
    bind(sockSrv, (SOCKADDR*)&addrSrv, sizeof(SOCKADDR));
    listen(sockSrv, 5);
    while (1) {
        SOCKET sockConn = accept(sockSrv, (SOCKADDR*)&addrClient, &len);
        hThread = CreateThread(NULL, NULL, handlerRequest, LPVOID(sockConn), 0, &dwThreadId);
        CloseHandle(hThread);
    }
    closesocket(sockSrv);
    WSACleanup();
}

DWORD WINAPI handlerRequest(LPVOID lparam)
{
    SOCKET ClientSocket = (SOCKET) LPVOID lparam;
    send(ClientSocket, sendBuf, strlen, 0);
    recv(ClientSocket, recvBuf, 50, 0);
    closesocket(ClientSocket);
    return 0;
}
```

- **socketserver模块**：网络服务器编程架构
  - TCPServer类：针对TCP服务器编程
  - UDPServer类：针对UDP服务器编程
- **TCPServer类与UDPServer类的使用**
  - 编写请求处理类
  - 创建TCPServer或UDPServer对象
  - 利用TCPServer类或UDPServer类的server\_forever()方法进行多次循环处理

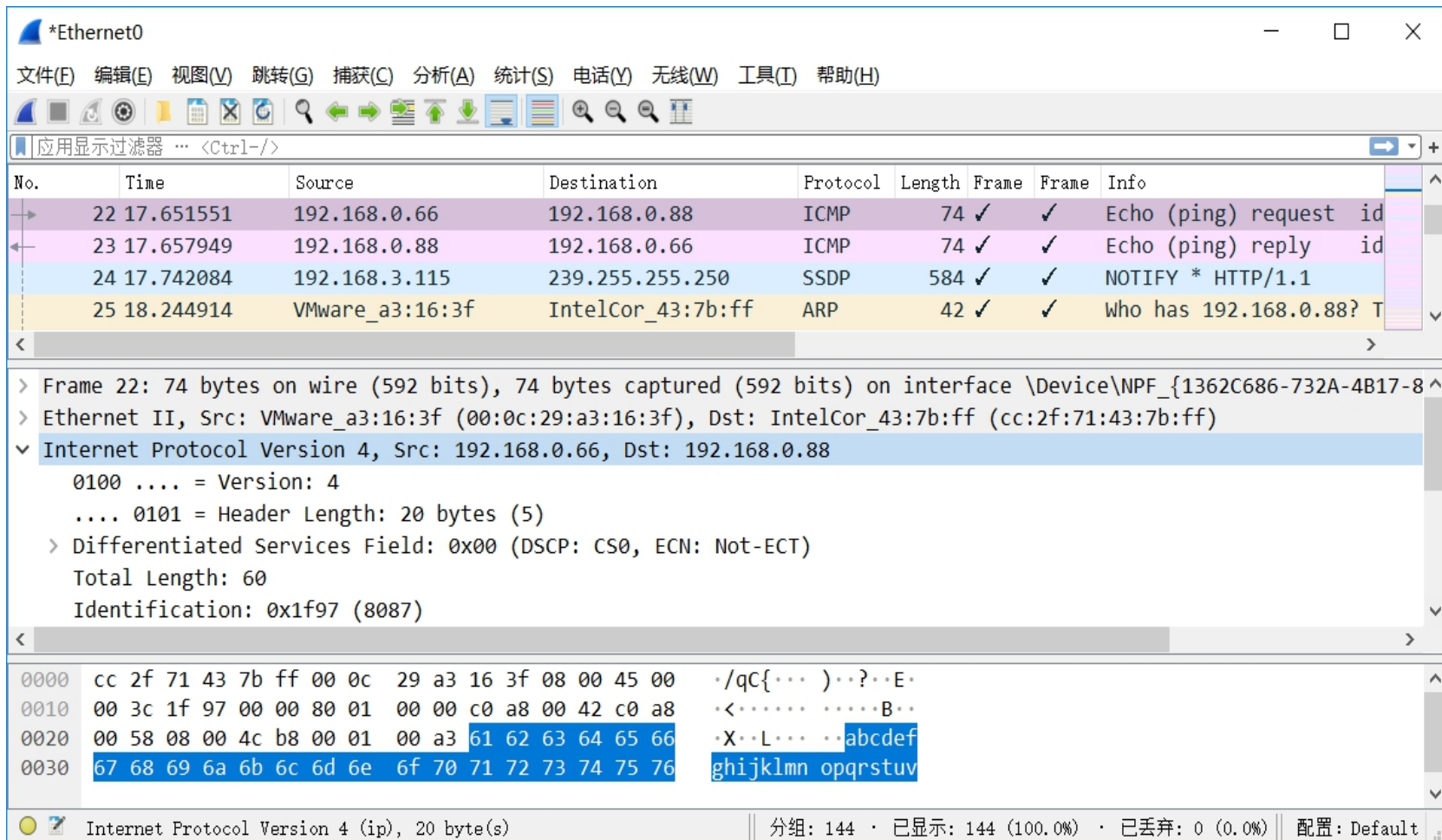
```
1 class MyTCPHandler(socketserver.BaseRequestHandler):
2     #重写基类中的handle函数
3     def handle(self):
4         #收发数据使用的socket存储在self.request中。
5         self.sock=self.request
6         .....
7         #接收数据。
8         self.recv_data=self.sock.recv(1024)
9         .....
10        #发送数据
11        self.sock.sendall(self.send_data)
12    .....
```

### 主程序

```
1 # 创建TCP服务器，为该服务器绑定的IP地址为host，端口号为port
2 with socketserver.TCPServer((host,int(port)), MyTCPHandler) as server:
3     # 循环进行收发处理，并在不用时自动关闭
4     server.serve_forever()
```



## Wireshark



\*Ethernet0

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(I) 帮助(H)

应用显示过滤器 ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Frame	Frame	Info
22	17.651551	192.168.0.66	192.168.0.88	ICMP	74	✓	✓	Echo (ping) request id
23	17.657949	192.168.0.88	192.168.0.66	ICMP	74	✓	✓	Echo (ping) reply id
24	17.742084	192.168.3.115	239.255.255.250	SSDP	584	✓	✓	NOTIFY * HTTP/1.1
25	18.244914	VMware_a3:16:3f	IntelCor_43:7b:ff	ARP	42	✓	✓	Who has 192.168.0.88? T

> Frame 22: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface \Device\NPF\_{1362C686-732A-4B17-8...}

> Ethernet II, Src: VMware\_a3:16:3f (00:0c:29:a3:16:3f), Dst: IntelCor\_43:7b:ff (cc:2f:71:43:7b:ff)

▼ Internet Protocol Version 4, Src: 192.168.0.66, Dst: 192.168.0.88

- 0100 .... = Version: 4
- .... 0101 = Header Length: 20 bytes (5)
- > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
- Total Length: 60
- Identification: 0x1f97 (8087)

< >

Offset	Hex	ASCII
0000	cc 2f 71 43 7b ff 00 0c 29 a3 16 3f 08 00 45 00	./qC{... )..?..E.
0010	00 3c 1f 97 00 00 80 01 00 00 c0 a8 00 42 c0 a8	.<.....B..
0020	00 58 08 00 4c b8 00 01 00 a3 61 62 63 64 65 66	·X·L... ..abcdef
0030	67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76	ghijklmn opqrstuv

Internet Protocol Version 4 (ip), 20 byte(s) | 分组: 144 · 已显示: 144 (100.0%) · 已丢弃: 0 (0.0%) | 配置: Default

捕获数据包、设置显示过滤规则、设置捕获范围、查看统计信息