# 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

zhangjz@nankai.edu.cn

计算机网络与信息安全研究室

# 学习目标

**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，了解理解动态自适应流媒体协议的基本思想

# 内容提纲

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

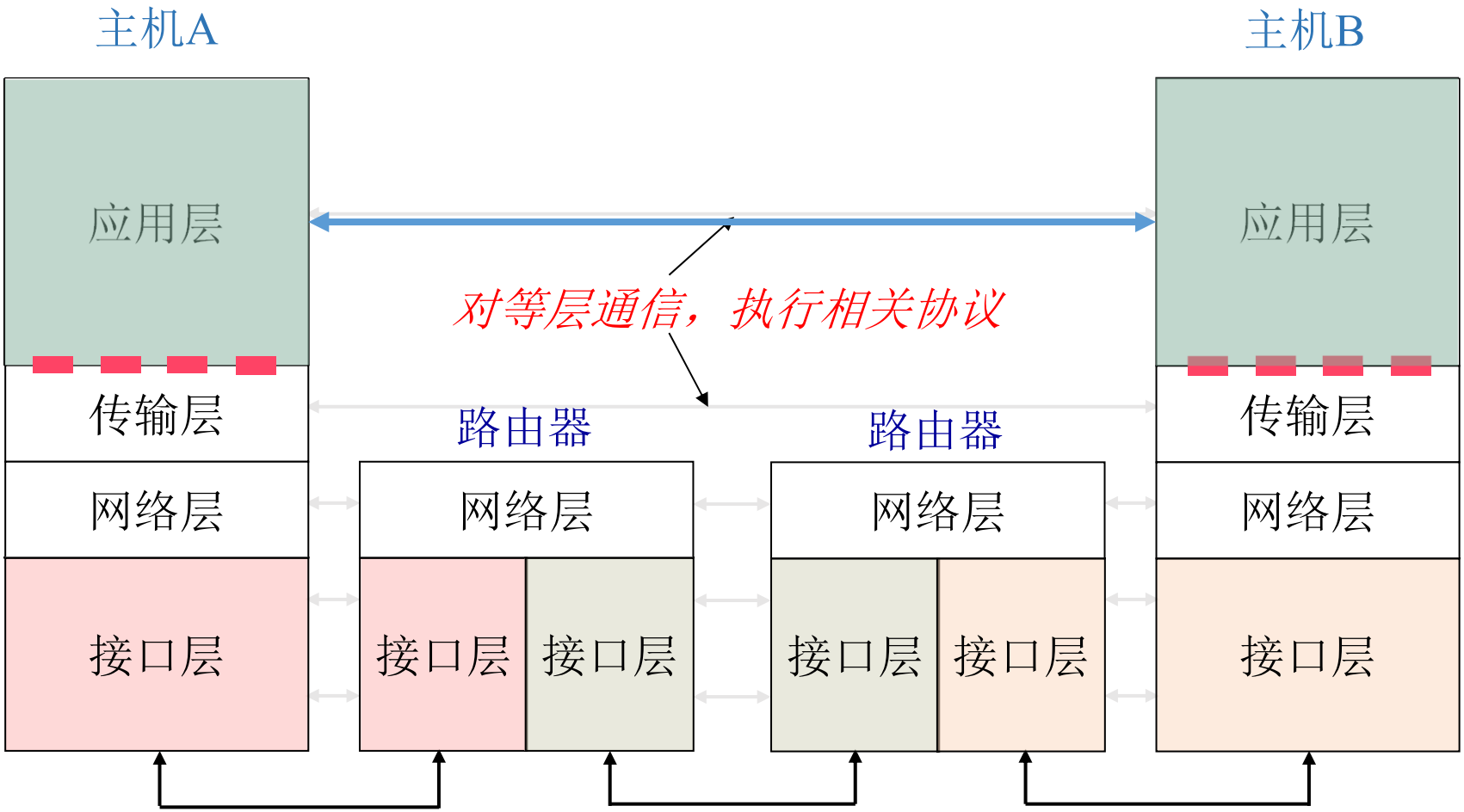
2.5 传统的应用层服务与协议

2.6 Web服务与HTTP协议

2.7 内容分发网络CDN

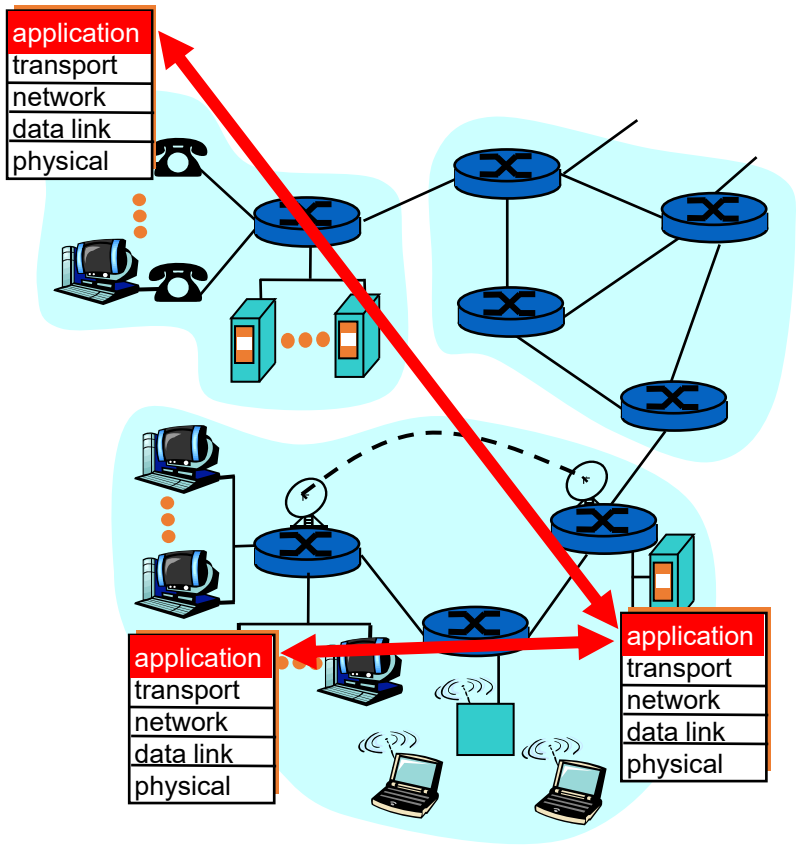
2.8 动态自适应流媒体协议DASH

## TCP/IP体系结构


主机A
主机B
应用层
应用层
对等层通信，执行相关协议
传输层
路由器
路由器
传输层
网络层
网络层
网络层
网络层
接口层
接口层
接口层
接口层
接口层
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、FTP、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

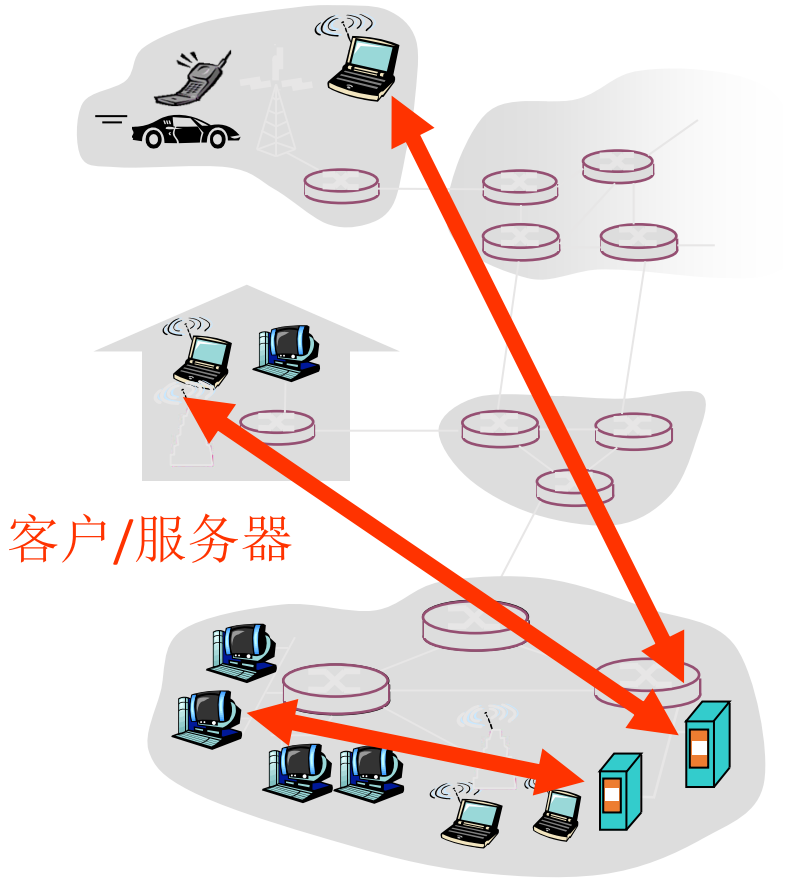
### ❑ 客户/服务器（C/S）模型

- ❖ 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

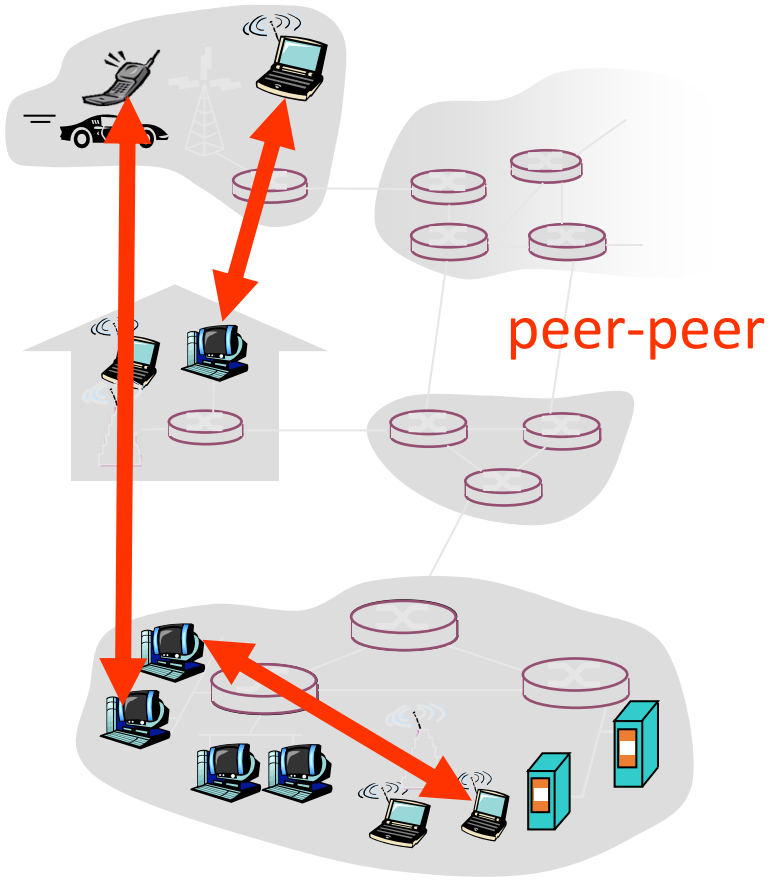
- 被动等待
- 长久在线
- 固定IP地址
- 利用集群/云提供可扩展性

❑客户进程

- 发起与服务器的通信
- 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
- 任意的主机之间都可能进行直接通信
- 主机之间可能间歇性地进行连接
- 主机可能使用动态的IP地址
- 高可扩展性但维护困难



## 客户/服务器与P2P混合模型

### Skype

- VoIP P2P应用
- 中心服务器：远端客户的地址发现
- 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：用户探测与位置发现
  - 当在线时，用户向中心服务器注册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）
- *Q:* 利用主机的IP地址表示主机中的进程是否可以？
  - *A:* 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**
  - 如报文包含哪些字段、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**
  - 进程何时发送报文、收到报文后的动作等

**公共协议**

- RFC中定义的协议
- 实现之间可相互兼容
- 例如：HTTP、SMTP、DNS、FTP等

**专有协议**

- 公司或组织自己设计的协议
- 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较高的要求
  - 文件传输能够容忍较长的时延和时延抖动
- **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<br>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 |

## TCP/IP体系结构中的传输层服务

**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

Q: 为什么需要UDP？

## 互联网应用：常用应用使用的传输层服务

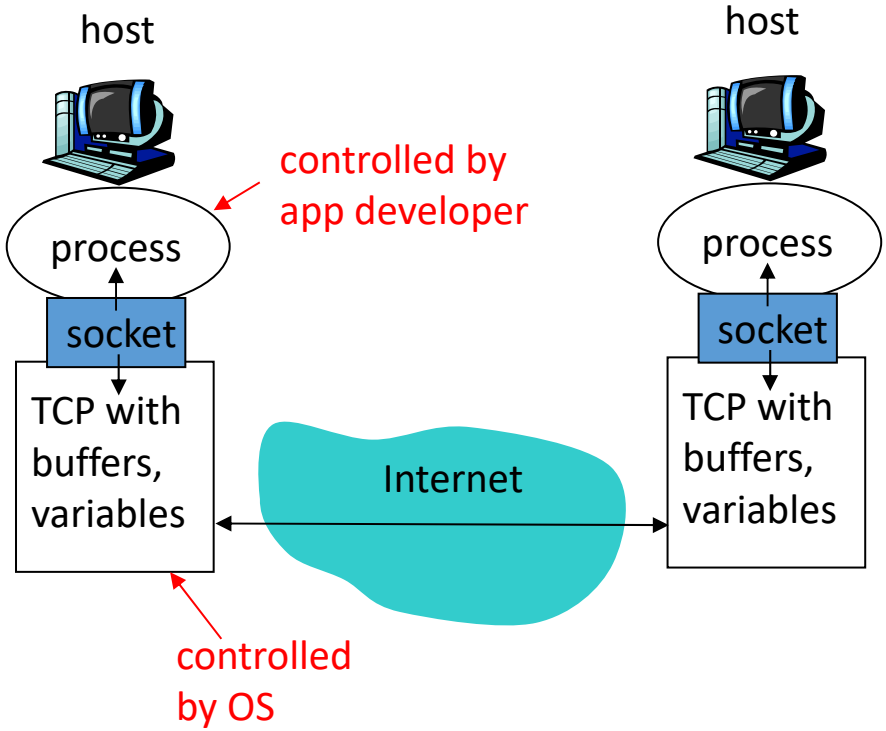
| 应用 | 应用层协议 | 传输层协议 |
|---|---|---|
| 电子邮件 | SMTP [RFC 2821] | TCP |
| Web服务 | HTTP [RFC 2616] | TCP |
| 文件传输 | FTP [RFC 959] | TCP |
| 流媒体 | HTTP、RTP | TCP or UDP |
| 网络电话 | RTP [RFC 1889]<br>SIP、专有协议 | 典型为UDP |

## 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

# 回顾：传输层提供的TCP和UDP服务

**TCP服务：**

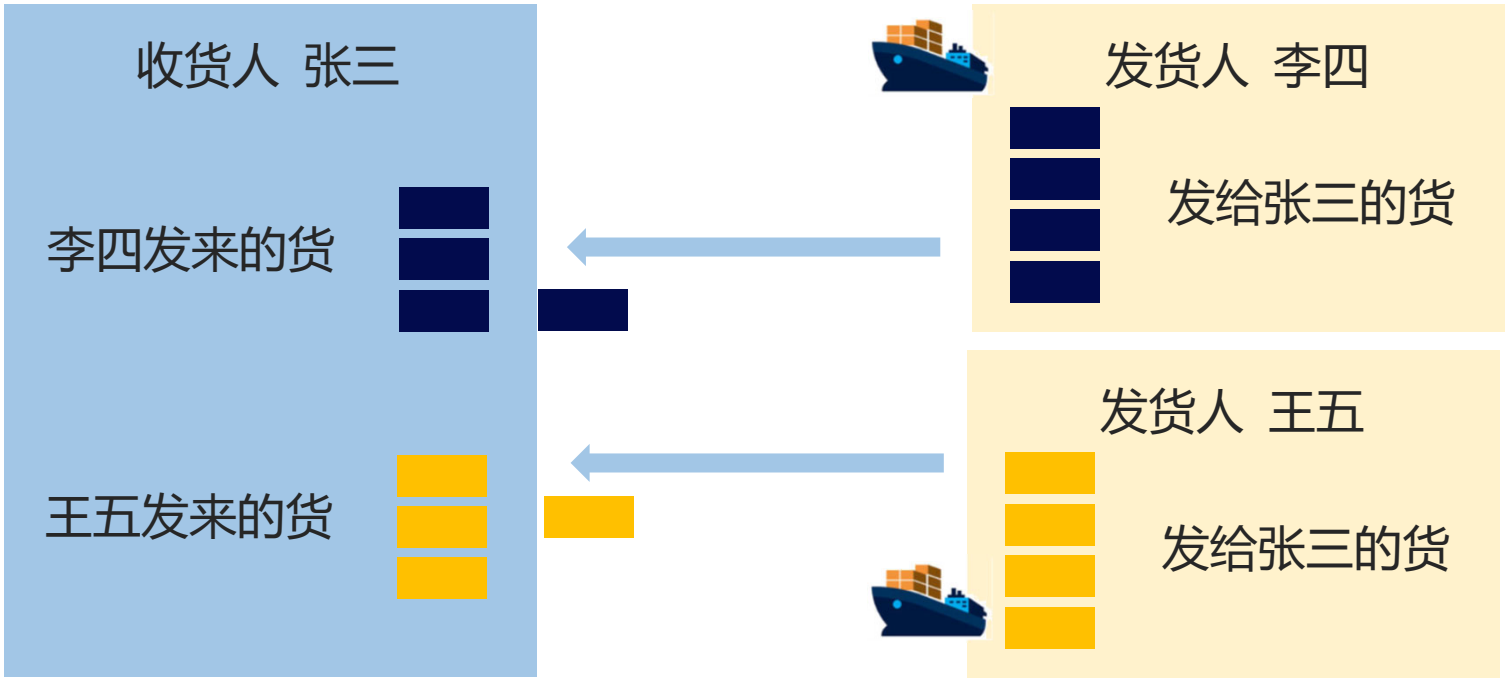
- 面向连接：客户与服务器之间需要建立连接
- 可靠传输：可保证传递数据无差错
- 流量控制：发送数据不会超过接收端的容纳容量
- 拥塞控制：提供拥塞解决方案
- 不能提供：时延和带宽保证

**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

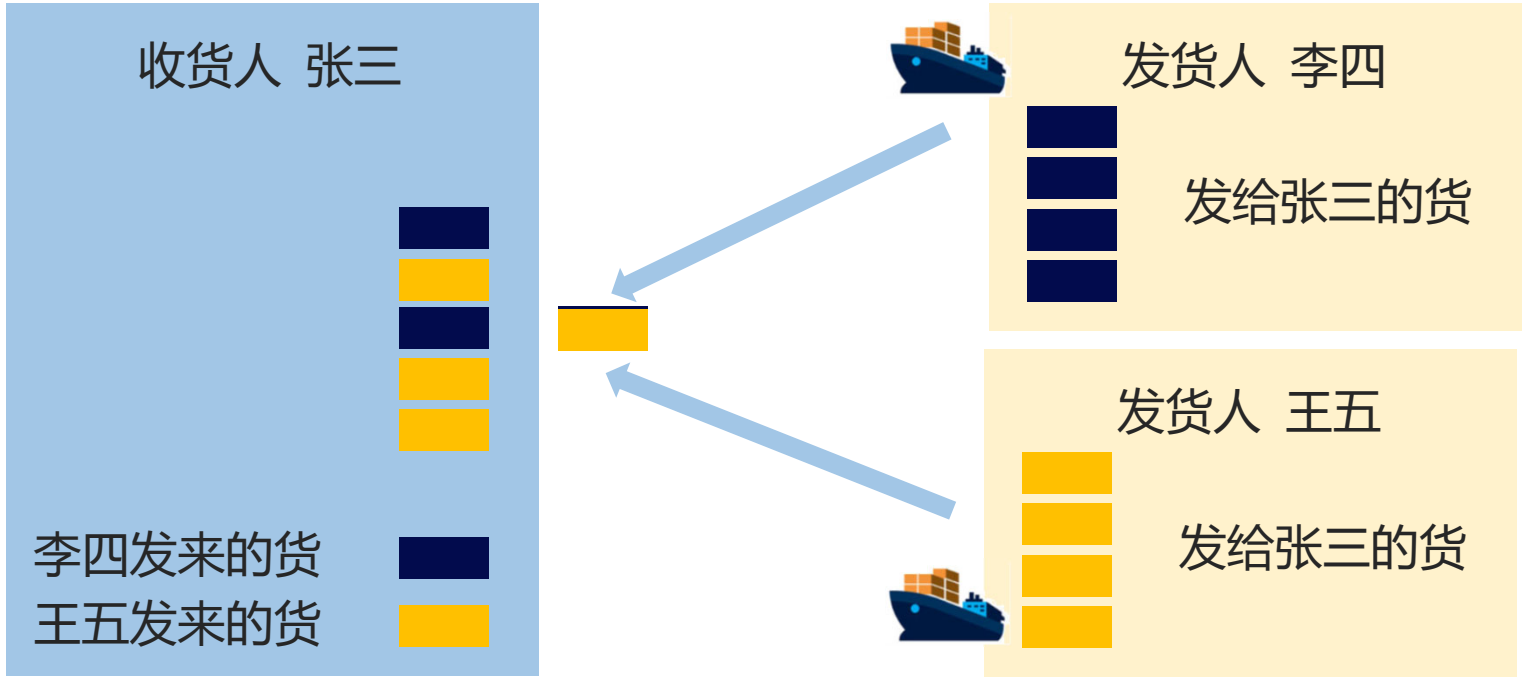
# 提前了解：TCP和UDP的区别

## TCP传输示意图

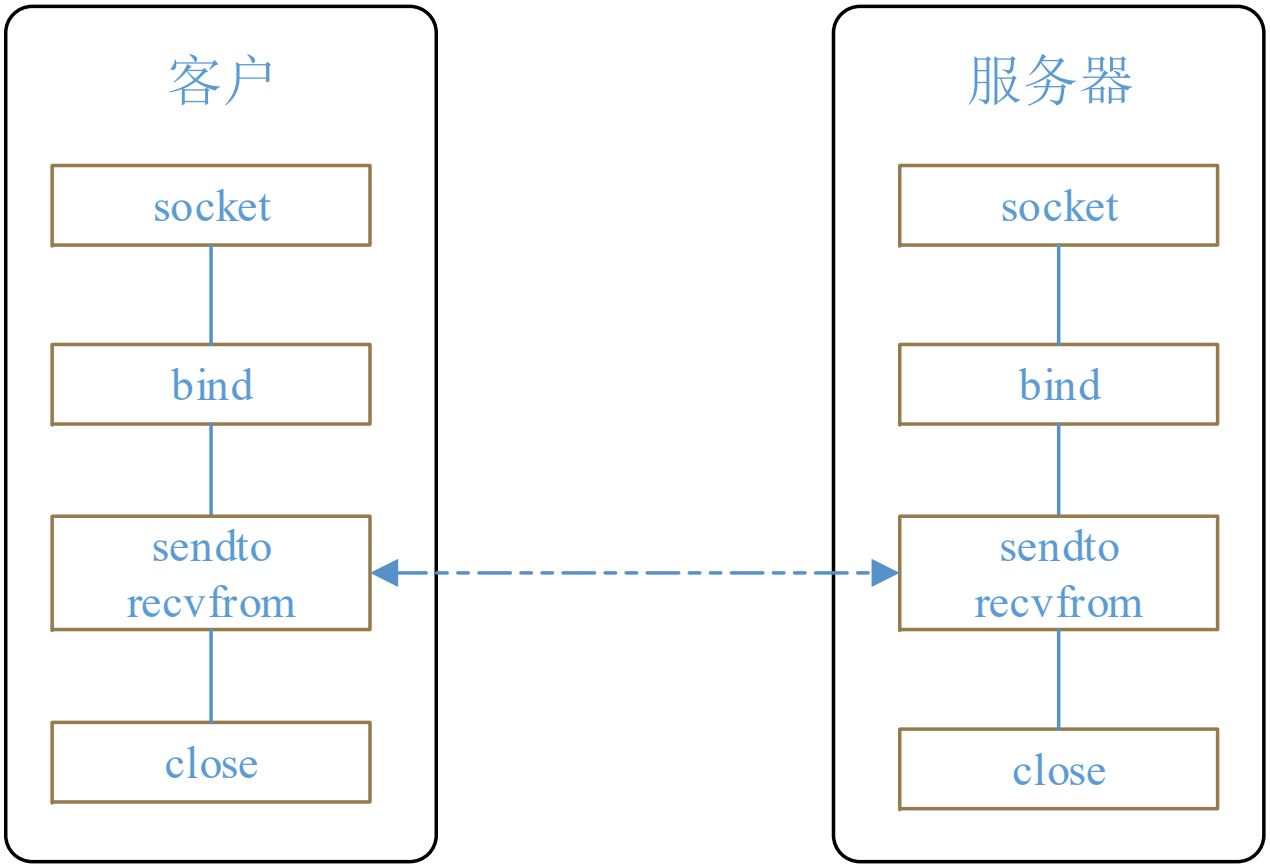


# 提前了解：TCP和UDP提供服务的区别

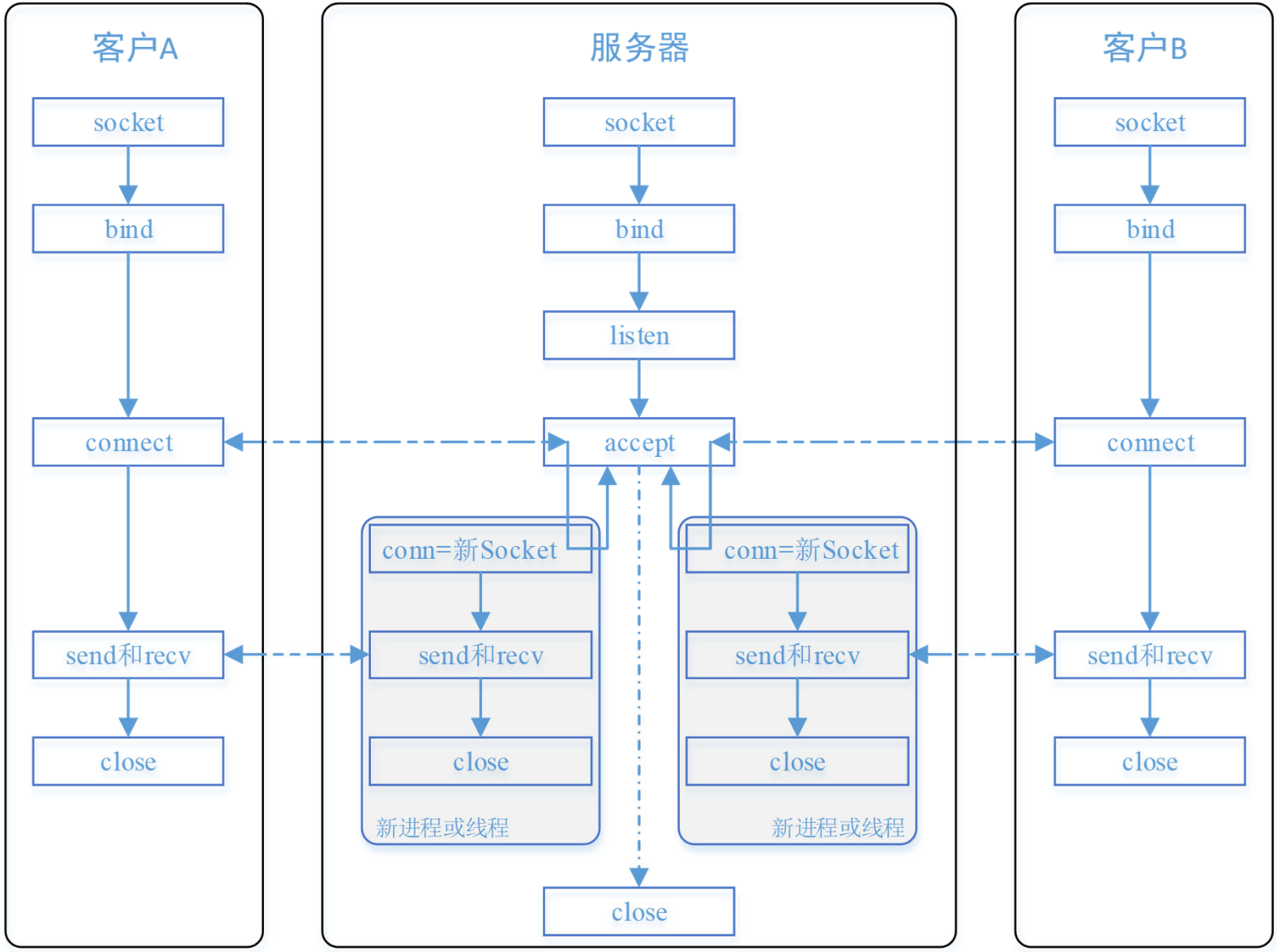
## UDP传输示意图



# 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



# 2.3 Socket编程 – C中常用的Socket函数

## 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

# 2.3 Socket编程 – C中常用的Socket函数

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

# 2.3 Socket编程 – C中常用的Socket函数

## bind

```
int bind(
  SOCKET                s,
  const 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

# 2.3 Socket编程 – C中常用的Socket函数

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

# 2.3 Socket编程 – C中常用的Socket函数

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

# 2.3 Socket编程 – C中常用的Socket函数

## 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

# 2.3 Socket编程 – C中常用的Socket函数

## sendto

```
int WSAAPI sendto(
        SOCKET              s,
        const char          *buf,
        int                 len,
        int                 flags,
        const 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。

# 2.3 Socket编程 – C中常用的Socket函数

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。

# 2.3 Socket编程 – C中常用的Socket函数

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。

# 2.3 Socket编程 – C中常用的Socket函数

## 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

# 2.3 Socket编程 – 编程中应注意的问题

## 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

- ***Q***：计算机中，整数是如何存储的？
- ***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

# 2.3 Socket编程 – 编程中应注意的问题

## 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()
{
WSAStartup(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()
{
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);
            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;
}
```

# 2.3 Socket编程 – Python对Socket的封装

- **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

# 2.3 Socket编程 – Python Socket编程举例

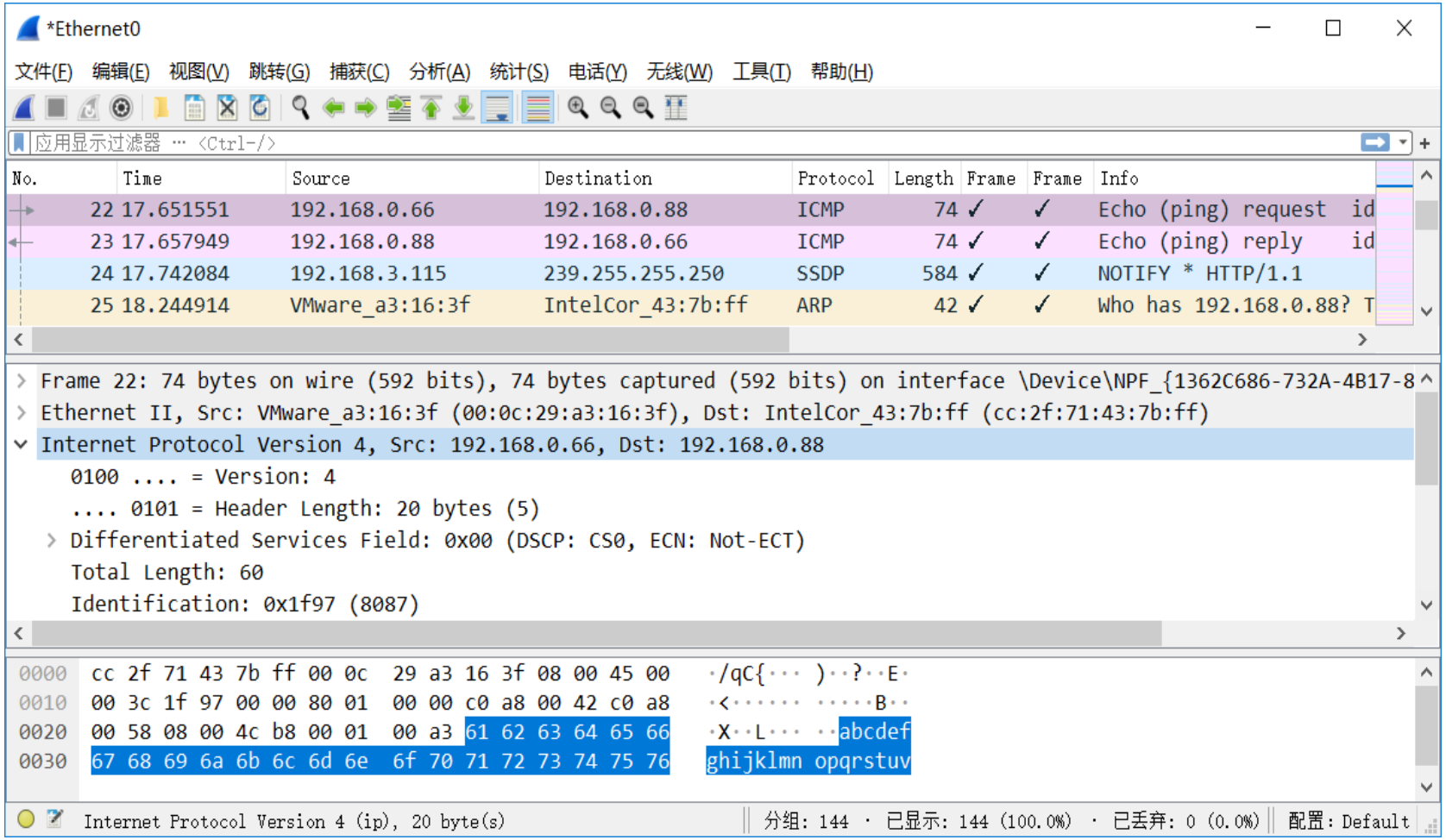
```
1       class MyTCPHandler(socketserver.BaseRequestHandler):
2       #重写基类中的handle函数
3       def handle(self):
4               #收发数据使用的socket存储在self.request中。
5               self.sock=self.request
6               ……
7               #接收数据。
8                       self.recv_data=self.sock.recv(1024)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

# 实用：网络数据包捕获与分析

## Wireshark




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

## 域名系统概述

- 互联网中使用IP地址寻址主机（例如：IP数据包转发）
- 为了方便记忆，每台提供服务的主机通常会有一个或多个名字
  - 例如：访问学院网站，输入名字cc.nankai.edu.cn
    - 对应的IPv4地址：222.30.45.190
    - 对应的IPv6地址：2001:250:401:d450::190

查询命令：nslookup cc.nankai.edu.cn

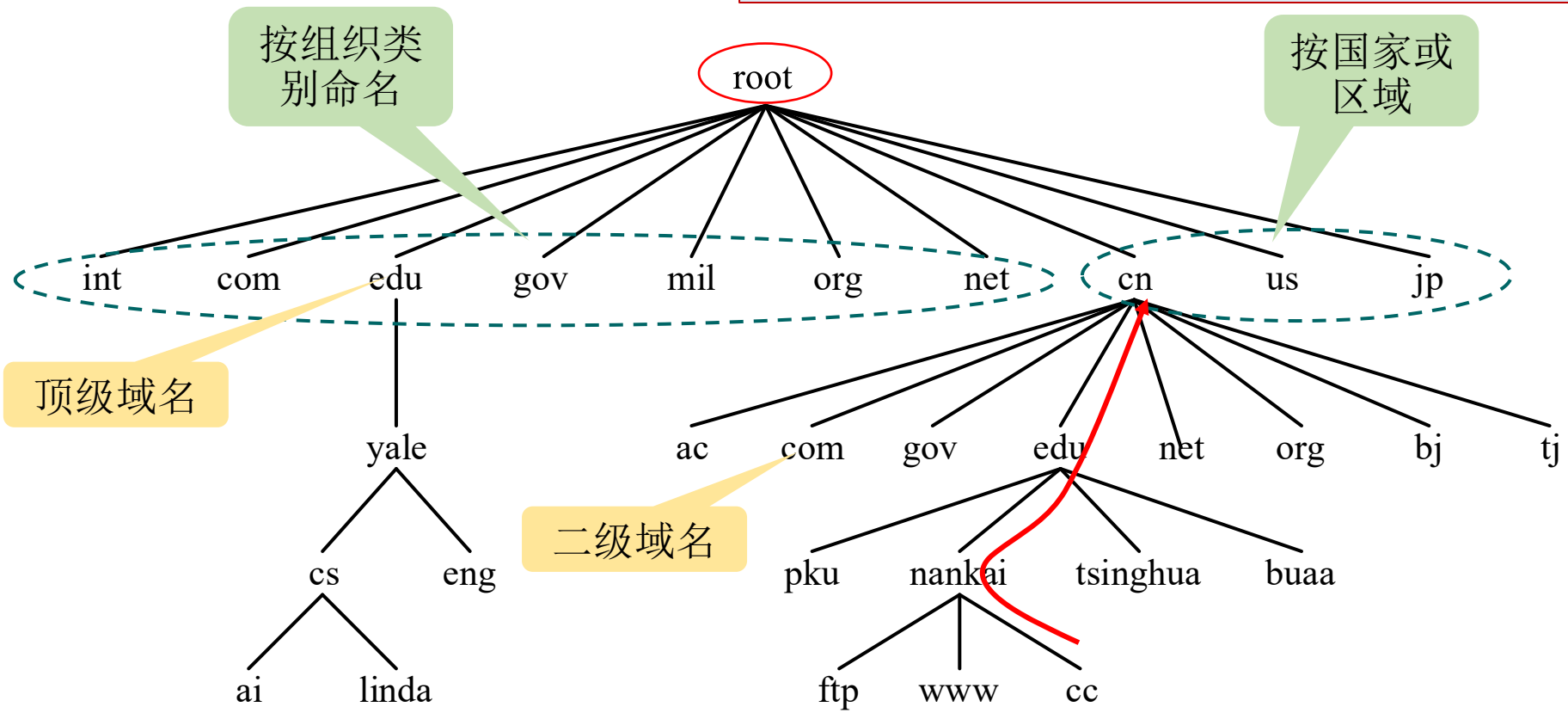
- 如何将名字映射到地址？
  - 早期的集中式管理和发布
    - 本地存储Hosts文件，实现名字到地址的静态映射
    - 可以通过FTP服务为连入Internet的主机提供域名的发布和下载
  - **存在的问题？**

## 域名系统概述（续）

- DNS（Domain Name System）
  - 自动实现名字到地址映射的系统
- DNS基本思想：
  - 名字和地址映射关系分布式存放，形成具有层次结构的分布式数据库系统（分布式管理）
  - 通过查询分布式数据库，获得名字到地址的映射，或相反
- 关键：
  - 如何组织分布式数据库？
  - 如何在分布式数据库中查找？

## DNS域名体系

**优点**：可以支持大规模、快速扩展，不需要中心节点支持；通过部分名字空间的授权实现非集中式管理；名字到地址的映射可以通过分布方式实现




- 层级命名、逐级授权、多级管理
  - 例如：cc.nankai.edu.cn

**域名**已成为互联网的重要基础性资源

## DNS域名体系（续）

- 顶级域名：由互联网名称与数字地址分配机构(ICANN)负责管理
  - ICANN 与域名注册商签订合同， 准许其受理顶级域名.com、.net、.org 下的域名注册

| 顶级域名 | 分配 |
| --- | --- |
| com | 商业组织 |
| edu | 教育机构 |
| gov | 政府机构 |
| mil | 军队机构 |
| net | 主要的网络支持中心 |
| org | 其他组织 |
| Int | 国际组织 |
| 国家地区代码 | 国家或地区 |

## DNS域名体系（续）

- 中国互联网络信息中心（CNNIC）管理.cn域名

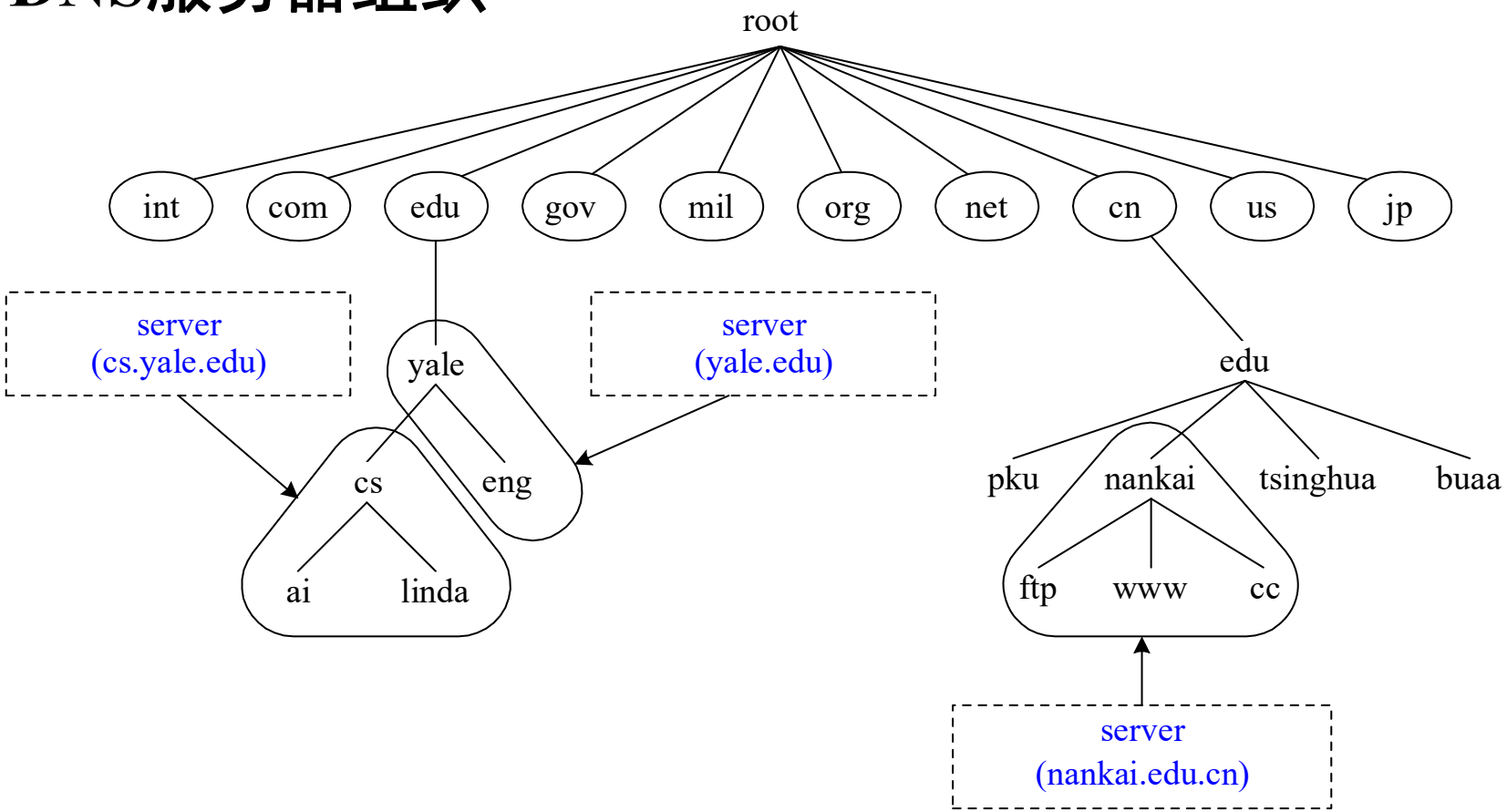
| 划分模式 | 我国二级域名 | 分配 |
|---|---|---|
| 类别域名 | ac | 科研机构 |
| | com | 工、商、金融等企业 |
| | edu | 教育机构 |
| | gov | 政府部门 |
| | net | 互联网络、接入网络信息中心和运行中心 |
| | org | 各种非盈利性的组织 |
| 行政区域域名 | bj | 北京市 |
| | sh | 上海市 |
| | tj | 天津市 |
| | cq | 重庆市 |
| | …… | |

**Whois查询服务（TCP 43端口）**：中国万网（www.zw.cn）、站长之家（whois.chinaz.com）等

## DNS域名解析

- **域名解析**：名字到地址映射（通过名字查地址）
  - 分布式：层级的服务器组织，协同实现解析
  - 有效性：大多数解析可以在本地完成，一部分会产生互联网流量
  - 可靠性：通过冗余设置，避免单点失效
- **客户-服务器模式**
  - 域名服务器：
    - 保存名字到地址映射关系（数据库）
    - 接收客户端请求，并给出响应
  - 域名解析器（客户端）：
    - 请求域名解析的客户进程
    - 向域名服务器发起解析请求，并等待服务器的响应

## DNS服务器组织




- 每台域名服务器包含一个或多个区域的信息
- 父节点服务器已知子节点服务器的地址

# DNS服务器组织

根域名服务器对互联网发展至关重要，除IPv4时代的13个根服务器，目前在16个国家设立了25台IPv6根服务器，我国部署了4台，打破了我国无根服务器的困境

- 根域名服务器：
  - 全球13个逻辑根域名服务器（a~m），每个根服务器都有多个镜像，实际的服务器数量目前达1326个（中国：大陆13个，台湾6个，香港9个）


根服务器：*letter.root-servers.net*

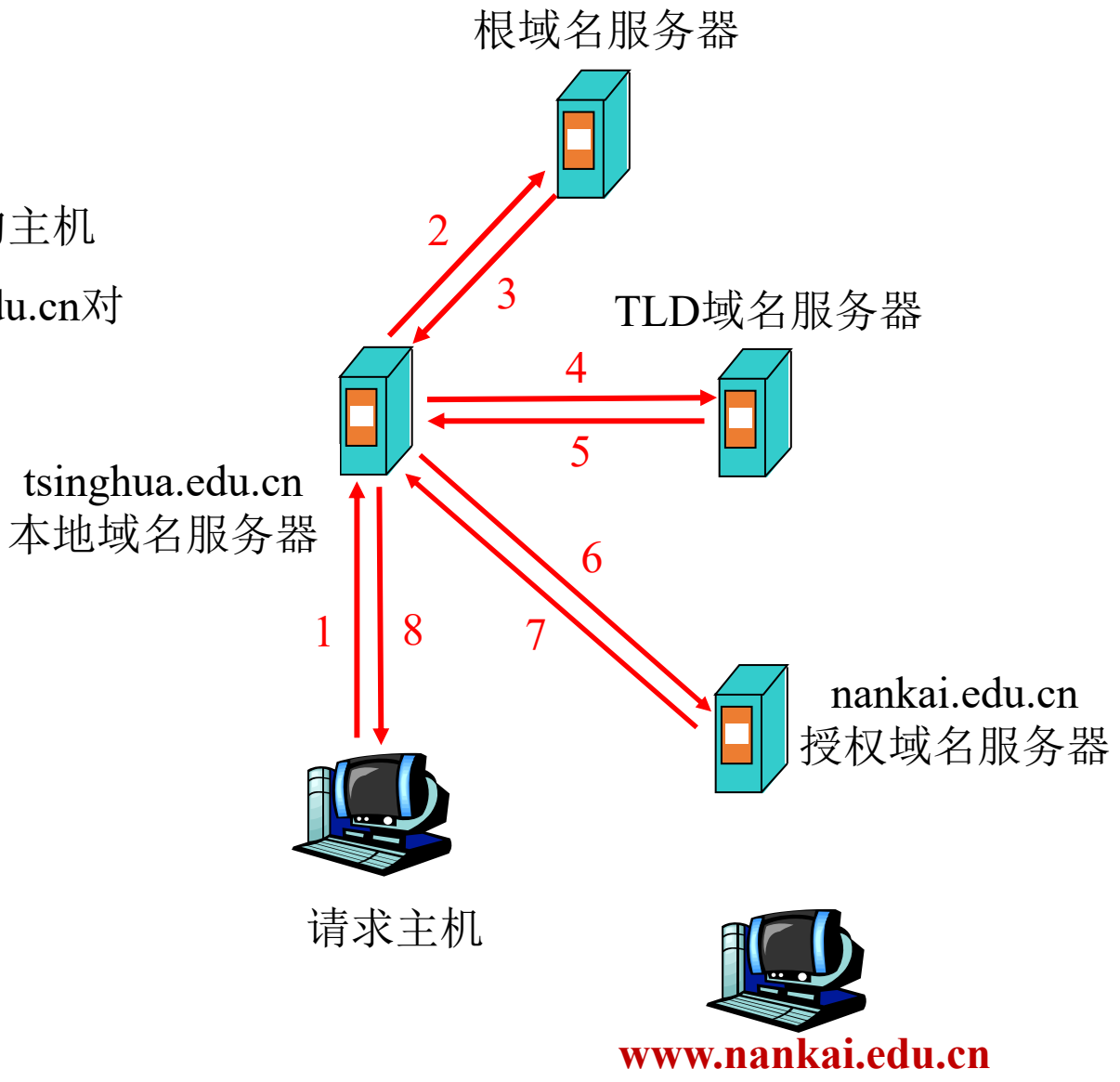
https://root-servers.org/

## DNS服务器组织

- **顶级域名服务器**（Top-Level Domain, TLD）
  - 负责顶级域名的解析
- **授权域名服务器**
  - 对于名字与地址映射，保留其初始数据来源的服务器
  - 主要区分名字与地址映射是原始的还是被缓存的（非授权）
- **本地域名服务器**（或称默认域名服务器）
  - 一般每个ISP都部署有域名服务器，其用户可将该服务器设置成本地域名服务器（或默认域名服务器）
  - 当进行域名解析时，查询请求首先发送到本地域名服务器（即查询的起点）

## DNS域名解析示例

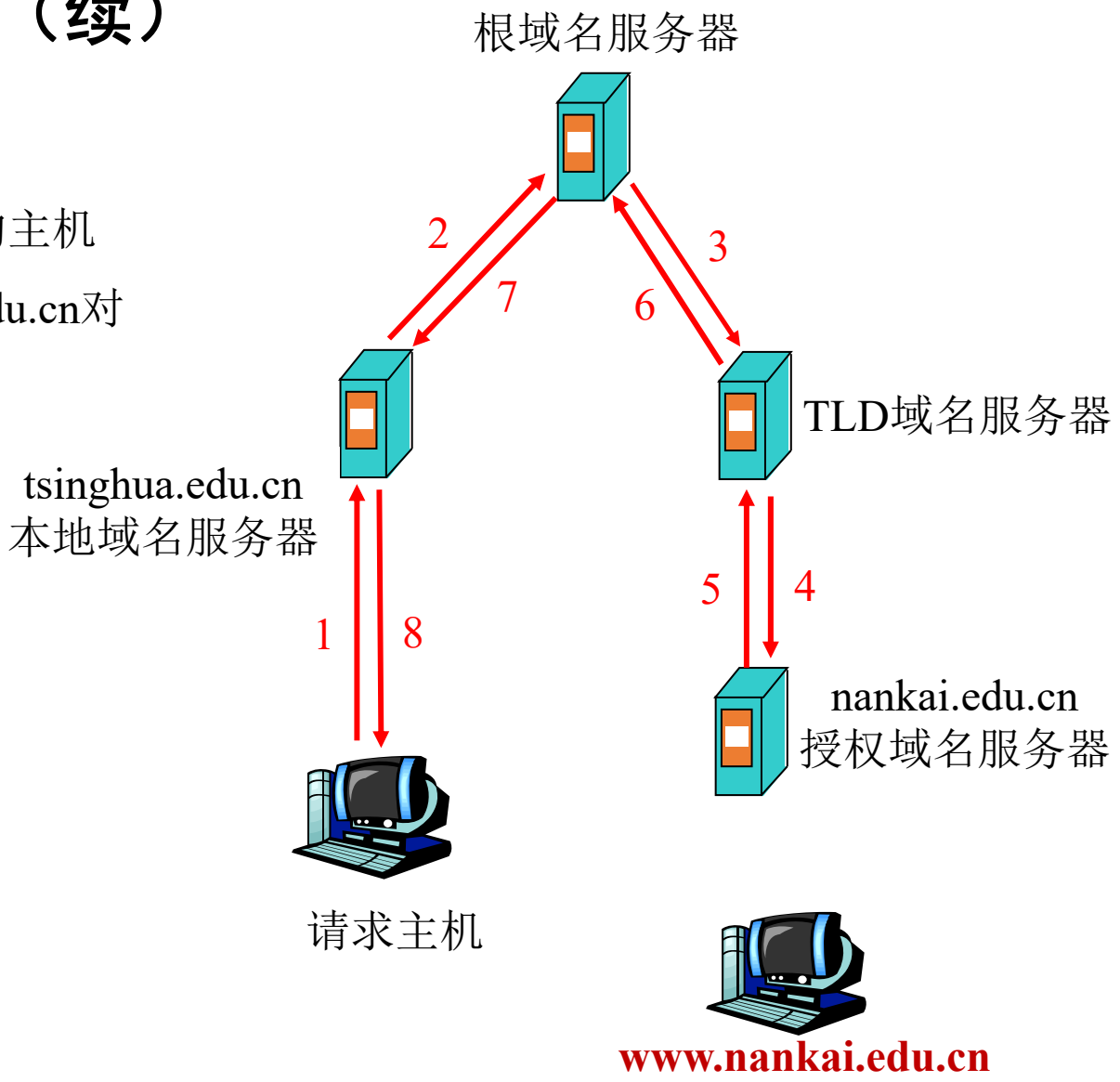
- 例如：
  - tsinghua.edu.cn域中的主机要解析www.nankai.edu.cn对应的IP地址
- 解析过程
  - 反复解析
  - 递归解析



## DNS域名解析示例（续）

- 例如：
  - tsinghua.edu.cn域中的主机要解析www.nankai.edu.cn对应的IP地址
- 解析过程
  - 反复解析
  - 递归解析

两者的优缺点？



## DNS域名服务器缓存

- 目的：降低非本地名字查询开销及查询延时
  - 通常地址与名字的绑定变化不频繁
- 服务器缓存名字与地址映射关系
  - 服务器学习到某个名字和地址的映射关系时，便进行缓存
  - 记录名字和地址的映射从何处获取
  - 基于授权服务器中的TTL值设置超时时间，缓存的映射关系经过一定时间会超时
  - TLD服务器通常会被本地域名服务器缓存，可以有效减少根域名服务器的访问频度

## DNS域名服务器缓存（续）

- 服务器使用缓存的映射关系响应客户端的请求
  - 标记为非授权（*nonauthoritative*）映射
  - 给出获取映射的服务器的域名和IP地址
- 客户机接收服务器响应
  - 映射有可能过时
  - 如果注重效率，客户端接受非授权响应
  - 如果注重准确性，客户机可以再联系授权服务器，验证映射是否仍有效

## 主机缓存

- 基本方法
  - 在启动时从本地域名服务器下载名字-地址映射数据库
  - 定期获取新的映射
  - 缓存最近用过的名字和地址映射
- 优点
  - 无需访问域名服务器，名字解析速度快
  - 本地服务器的故障不影响名字解析
  - 减低服务器的负载
- 缺点？

## DNS资源记录

- DNS使用区域数据库存储名字与地址映射关系，区域数据库由资源记录（Resource Records，RR）组成
- 资源记录结构
  - 名字（**name**）
  - TTL（Time To Live）：有效时间，通常为86400秒（24小时）
  - 类型（Type）：SOA、NS、A、AAAA、PTR、CNAME、MX
  - 类（Class）：例如，IN类
  - 值（**Value**）

## DNS资源记录（续）

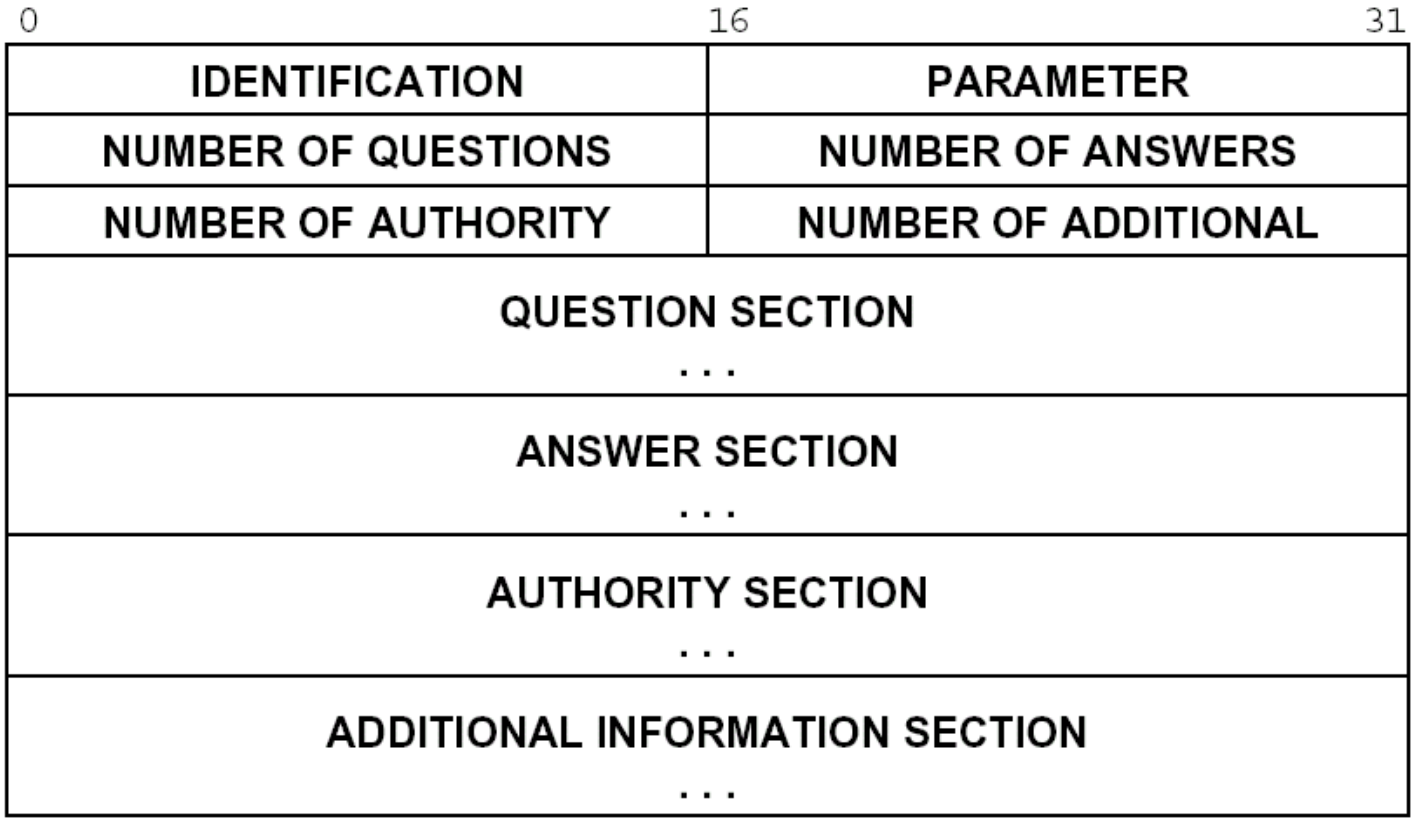
- 资源记录类型（常用）
  - SOA：区域数据库的开始，描述负责区域的域名服务器、版本信息，以及从属域名服务器备份时的一些参数等
  - NS：指定DNS服务器主机名（不使用IP地址）
  - A：将名称对应到IPv4的32位地址
  - AAAA：将名称对应到IPv6的128位地址
  - PTR：将IP 对应的名字
  - CNAME：别名，同一台主机可以有多个名字
  - MX：给出服务特定域的邮件服务器的主机名

## DNS资源记录：示例

| | | | | |
|---|---|---|---|---|
| cs.vu.nl. | 86400 | IN | SOA | star boss (serial, refresh, retry, expire, ttl) |
| cs.vu.nl. | 86400 | IN | TXT | “A University” |
| cs.vu.nl. | 86400 | IN | MX | 1 zephyer.cs.vu.nl. |
| cs.vu.nl. | 86400 | IN | MX | 2 top.cs.vu.nl. |
| flits.cs.vu.nl. | 86400 | IN | HINFO | Sun Unix |
| flits.cs.vu.nl. | 86400 | IN | A | 130.37.16.112 |
| flits.cs.vu.nl. | 86400 | IN | MX | 1 flits.cs.vu.nl. |
| flits.cs.vu.nl. | 86400 | IN | MX | 2 zephyer.cs.vu.nl. |
| flits.cs.vu.nl. | 86400 | IN | MX | 3 top.cs.vu.nl. |
| www.cs.vu.nl. | 86400 | IN | CNAME | top.cs.vu.nl. |
| ftp.cs.vu.nl. | 86400 | IN | CNAME | zephyer.cs.vu.nl. |
| zephyer | 86400 | IN | A | 130.37.56.201 |
| | | IN | HINFO | Sun Unix |

## DNS报文格式

- DNS包括*query*和*reply*两种报文



## DNS报文格式（续）

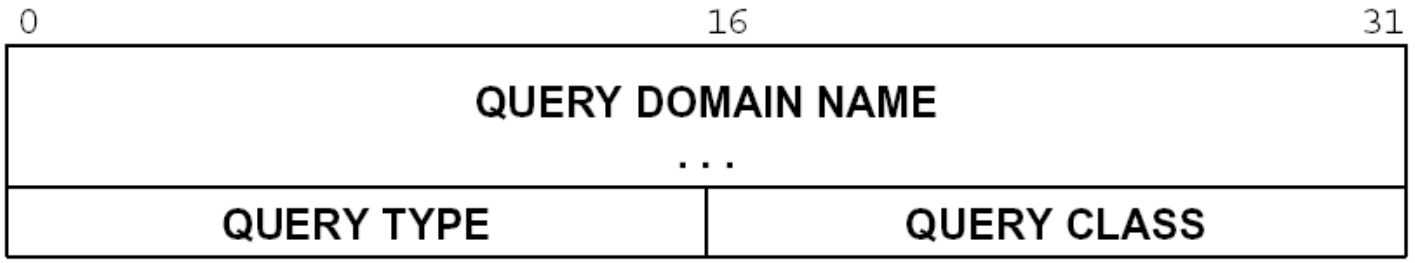
- 参数域的定义

| 参数域中的位 | 含义 |
| --- | --- |
| 0 | 报文类型：0-query 1-reply |
| 1-4 | 查询类型：<br>0-标准查询 1-反向查询 |
| 5 | 授权响应，则置1 |
| 6 | 报文被截断，则置1 |
| 7 | 期望递归，则置1 |
| 8 | 支持递归，则置1 |
| 9-11 | 保留 |
| 12-15 | 应答类型：<br>0-无错误<br>1-查询中格式错<br>2-服务器失效<br>3-名字不存在 |

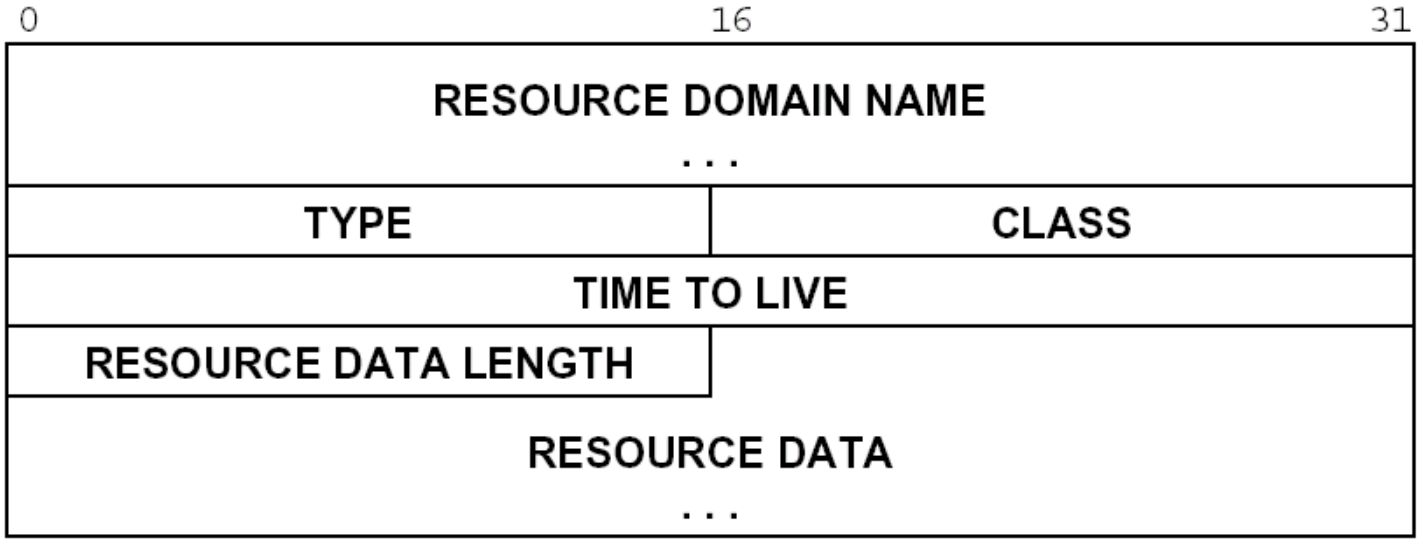
## DNS报文格式（续）

- Question: 携带查询的名字和其他参数
- Answer: 携带直接响应查询的资源记录
- Authority: 携带描述其他域名服务器的资源记录
  - 在应答中可以选择携带授权数据的SOA资源记录
- Additional:携带附加的资源记录

## DNS报文格式（续）




Question
格式




Answer
Authority
Additional
格式

## 域名格式压缩

- 报文中域名格式
  - 每个段第一个字节指定长度 (00xxxxxx=n)，后跟n个字节
  - 长度为0的段，表明域名结束
  - 例如：www.nankai.edu.cn，nankai.edu.cn
- 压缩格式
  - 域名的后缀部分经常重复，可以进行适当压缩
  - 指针：如果前两位为11，则段前两个字节的后14位为指针 (11xxxxxxxxxxxxxx)

## 域名格式压缩：示例

- 例如：需要查询名字F.ISI.ARPA, FOO.F.ISI.ARPA, ARPA, 和根，忽略其他域，这些名字可以表示为：

偏移量

| | | |
|---|---|---|
| 20 | 1 | F |
| 22 | 3 | I |
| 24 | S | I |
| 26 | 4 | A |
| 28 | R | P |
| 30 | A | 0 |

| | | |
|---|---|---|
| 40 | 3 | F |
| 42 | O | O |
| 44 | 11 | 20 |
| 64 | 11 | 26 |
| 92 | 0 | |

## 报文格式：示例

- 例如：一个邮件的客户端想向Mockapetris@ISI.EDU发送邮件，需要对域名ISI.EDU进行解析
  - Query: question section

    QNAME=ISI.EDU, QTYPE=MX, QCLASS=IN
  - Reply: answer section

    ISI.EDU. MX 10 VENERA.ISI.EDU.

    MX 10 VAXA.ISI.EDU.
  - Reply: additional section

    VAXA.ISI.EDU. A 10.2.0.27

    A 128.9.0.33

    VENERA.ISI.EDU. A 10.1.0.52

    A 128.9.0.32

## 报文格式：示例

- 例如：一个查询 (QNAME=BRL.MIL, QTYPE=A)发送到C.ISI.EDU，则应答可能为：
  - answer section

    <empty>
  - authority section

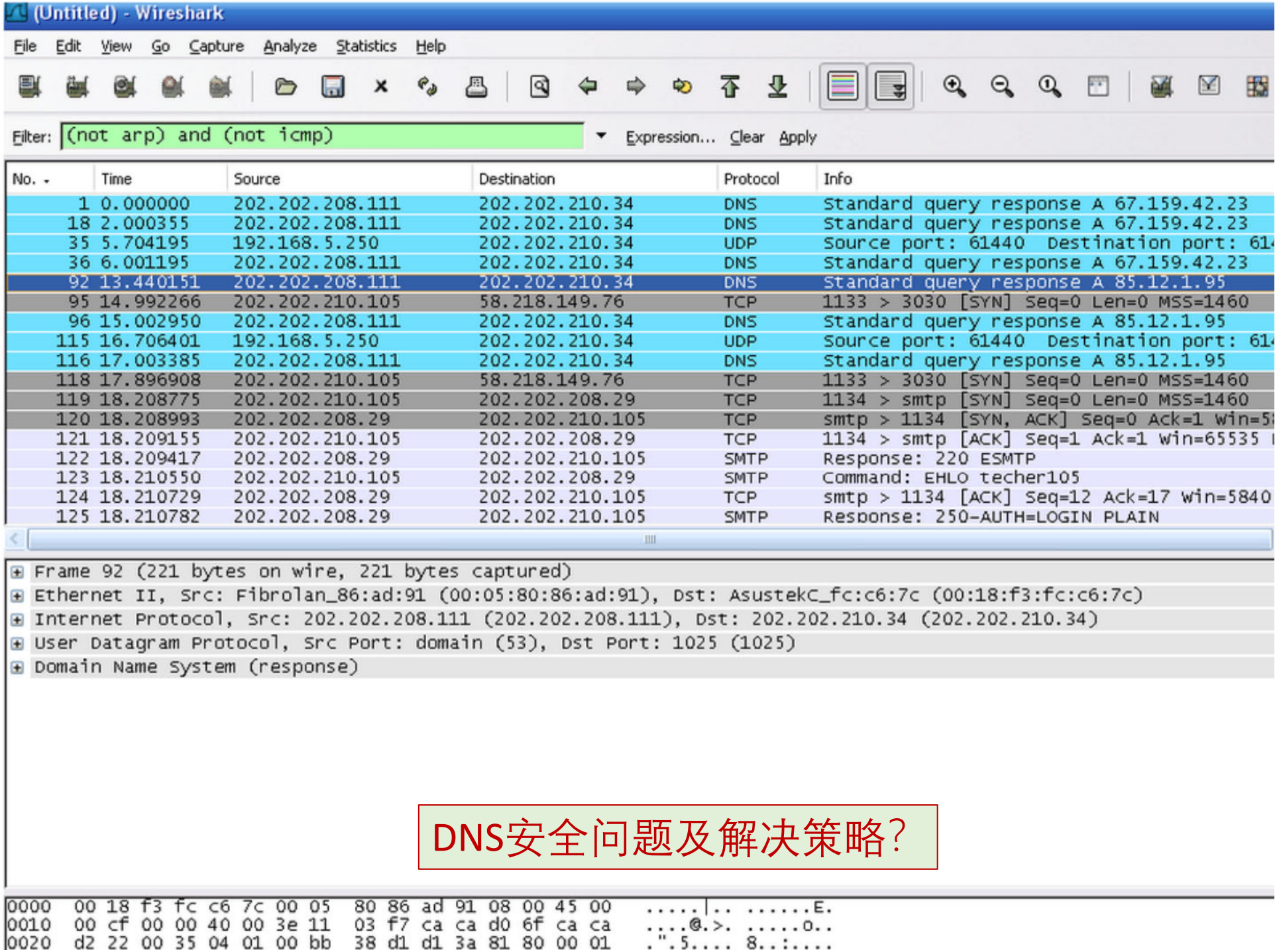
    MIL. 86400 IN NS SRI-NIC.ARPA.

    86400 IN NS A.ISI.EDU.
  - additional section

    A.ISI.EDU. A 26.3.0.103

    SRI-NIC.ARPA. A 26.0.0.73

    A 10.0.0.51




DNS安全问题及解决策略?

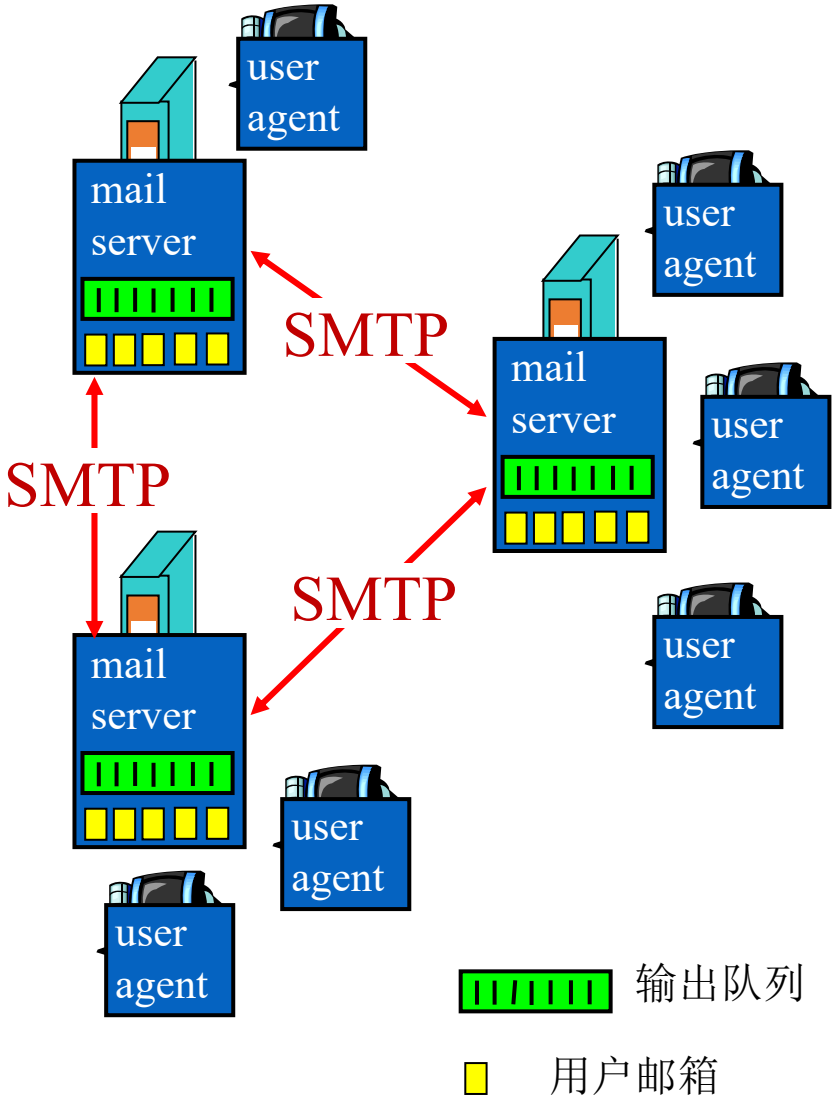
## 电子邮件系统

用户代理（接口）

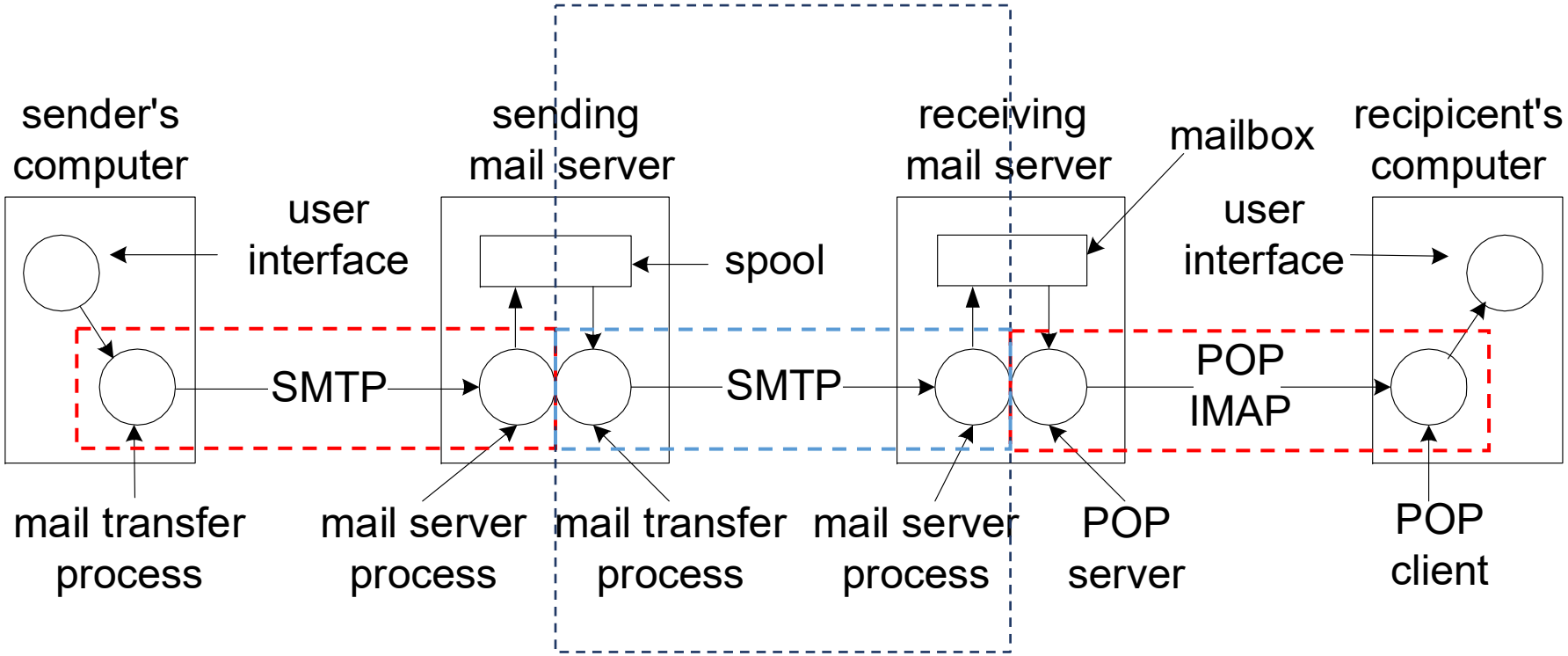
- 编辑和发送邮件
- 接收、读取和管理邮件
- 无统一标准

邮件服务器

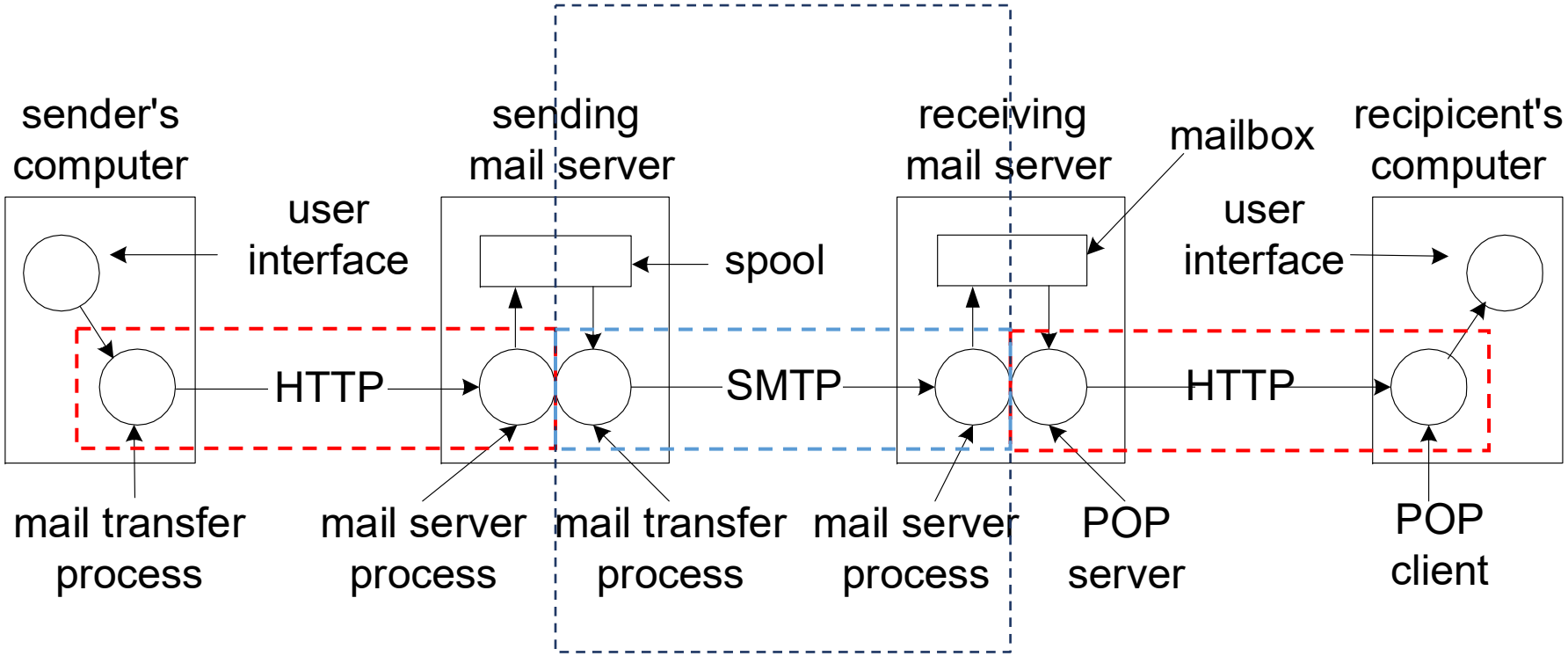
- **邮箱**：保存用户收到的消息
- **消息输出队列**：消息的发送队列
- **SMTP协议**：邮件服务器之间传递邮件使用的协议
  - smtp客户：发送邮件端
  - smtp服务器：接收邮件端



## 客户/服务器模型（SMTP/POP/IMAP）



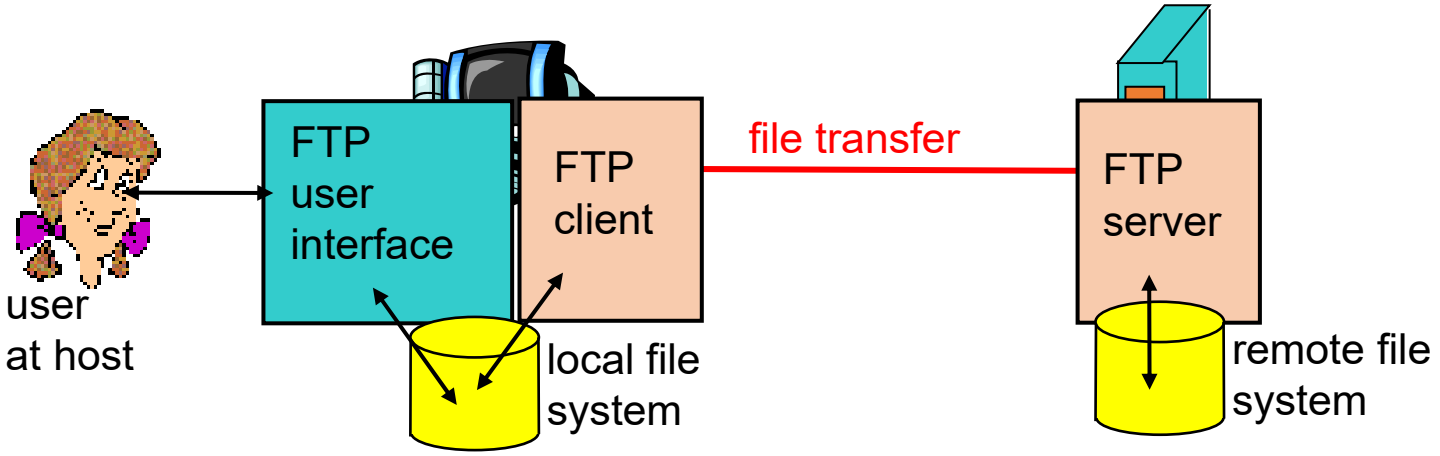
## 客户/服务器模型（SMTP/HTTP）



## SMTP: 简单邮件传输协议

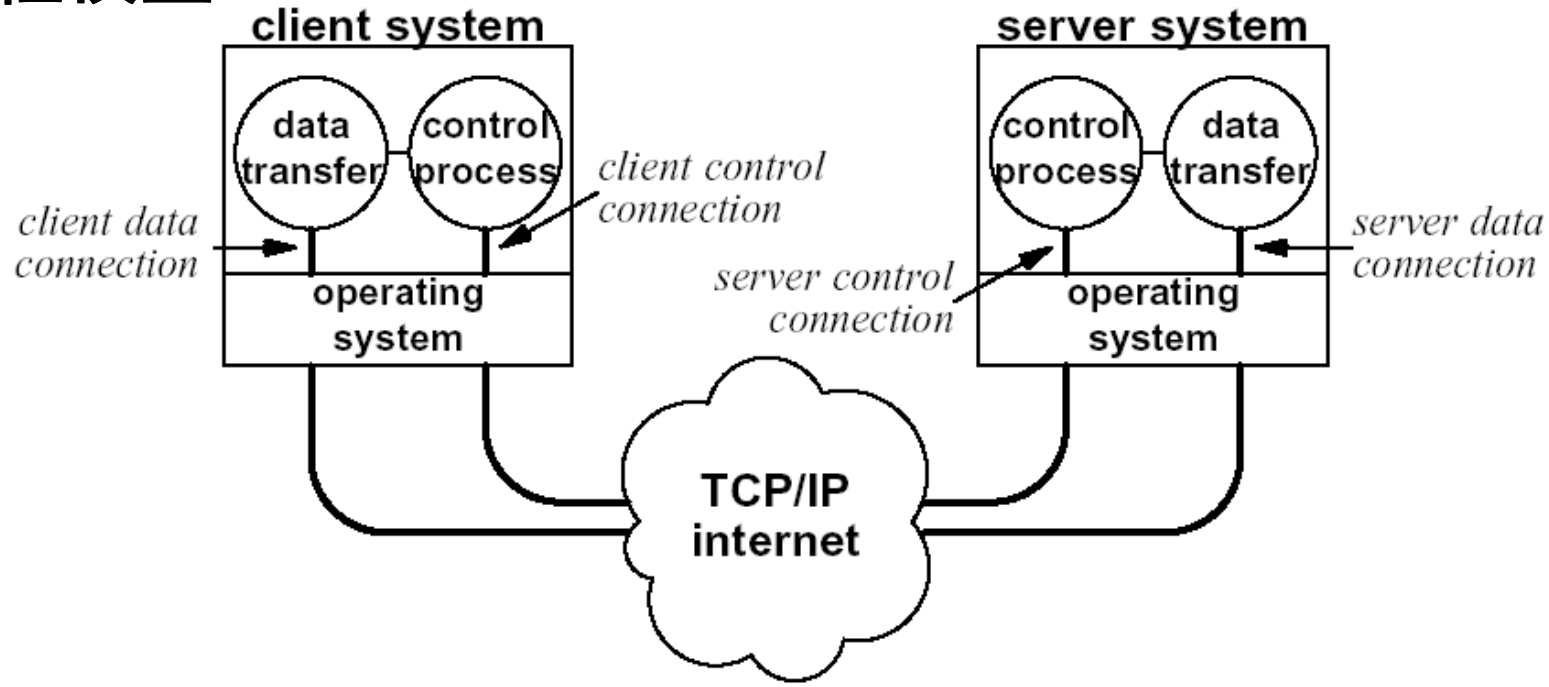
- 利用TCP可靠地从SMTP客户向SMTP服务器传递邮件
- SMTP的3个阶段：连接建立、邮件传送、连接关闭
- 命令/响应
  - 命令：ASCII字符串
  - 响应：状态码+短语
- 消息为7位ASCII
- RFC 821: SMTP

## FTP: 文件传输协议 (RFC 959)




- 基于TCP/IP的文件传输系统
- 客户/服务器模型
  - *client:* 初始化传输（无论上传还是下载）
  - *server:* 远端
- 客户使用TCP协议连接远端服务器

## FTP进程模型



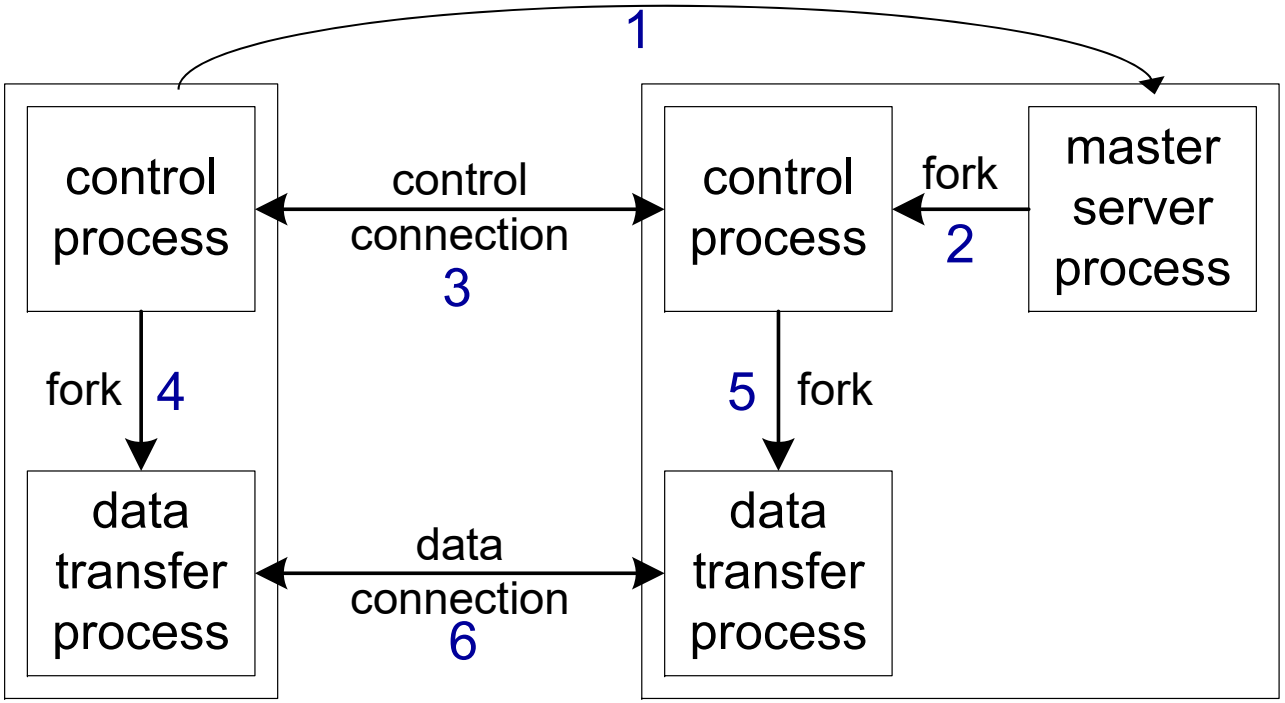

### 控制连接

- 带外控制
- TCP的21端口
- 客户和服务器之间维护的一个FTP会话
- 用于客户和服务器之间交换命令和响应
- 在整个会话期间保持活跃

### 数据传输连接

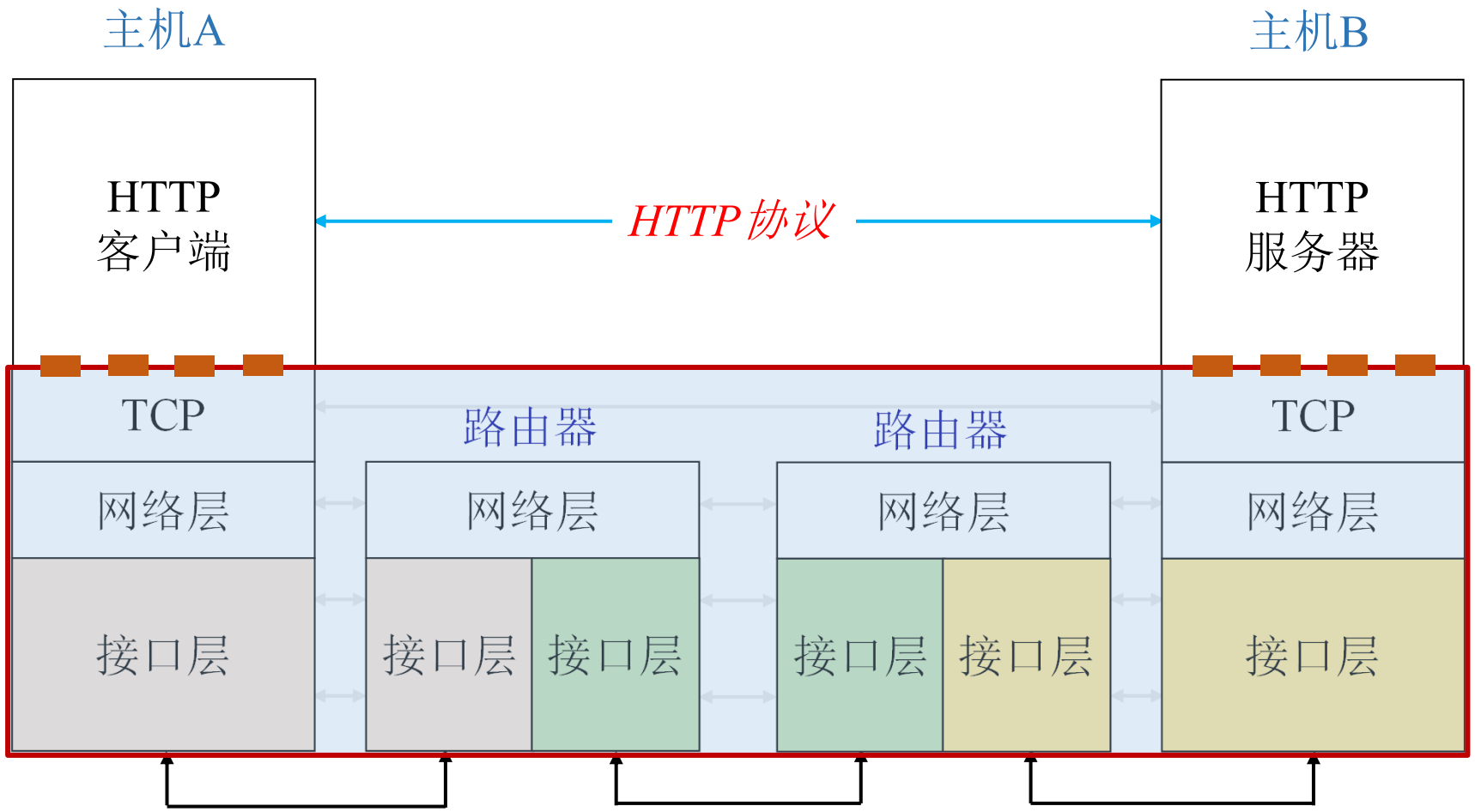
- TCP的20端口
- 用于传输数据
- 客户在一个会话上向服务器传输多个请求
- 每个文件请求都会建立一个数据连接
- 数据传输结束后，释放数据连接

## FTP进程模型（续）

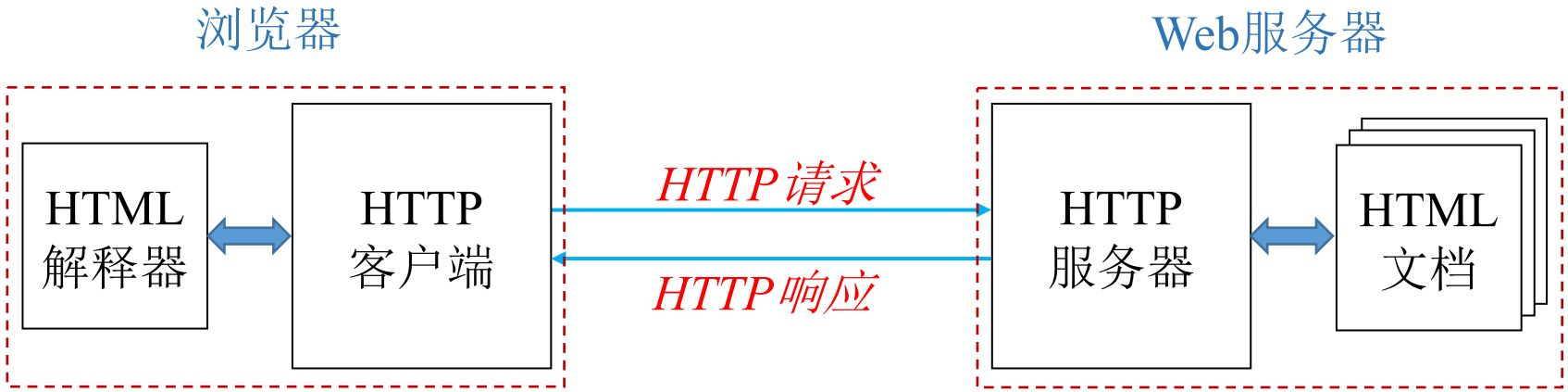



- Master server process: daemon at port 21
- Client control process: random port
- Server control process: port 21
- Server data transfer process: port 20
- Client data transfer process: random port

## Web客户/服务器模型


主机A
主机B
HTTP
客户端
HTTP协议
HTTP
服务器
TCP
路由器
路由器
TCP
网络层
网络层
网络层
网络层
接口层
接口层
接口层
接口层
接口层
接口层

## Web客户/服务器模型（续）



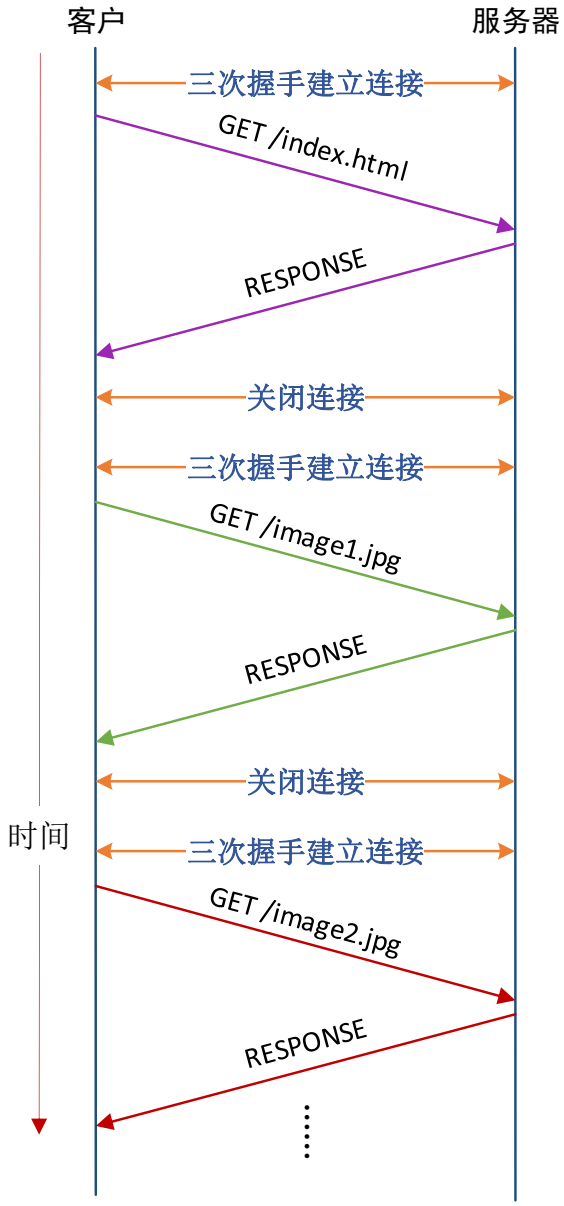

- 服务器
  - Web页面（HTML文档）：包含到多种对象的链接
  - 对象：可以是 HTML文档、 图像文件、视频文件、声音文件、脚本文件等
  - 对象用URL（统一资源定位符）编址：协议类型://主机名//路径和文件名
- 客户端
  - 发出请求、接收响应、解释HTML文档并显示；
  - 有些对象需要浏览器安装插件

## HTTP协议

- HTTP（ HyperText Transfer Protocol）
  - 传输层通常使用TCP协议，缺省使用TCP的80端口
  - HTTP为无状态协议，服务器端不保留之前请求的状态信息
    - ✓ 无状态协议：效率低、但简单
    - ✓ 有状态协议：维护状态相对复杂，需要维护历史信息，在客户端或服务器出现故障时，需要保持状态的一致性等
  - HTTP标准
    - ✓ HTTP/1.0: RFC 1945（1996年）
    - ✓ HTTP/1.1: RFC 2616（1999年）
    - ✓ HTTP/2: RFC 7540（2015年）、RFC 8740（2020年）
    - ✓ HTTP/3: RFC 9114 （2022年）

## HTTP/1.0示例

- 假设用户输入URL
  - http://www.nankai.edu.cn/computer/index.html
  - 页面包含10幅jpg图像




HTTP 1.0

## 非持久连接和持久连接

思考：持久连接中的TCP连接何时关闭？

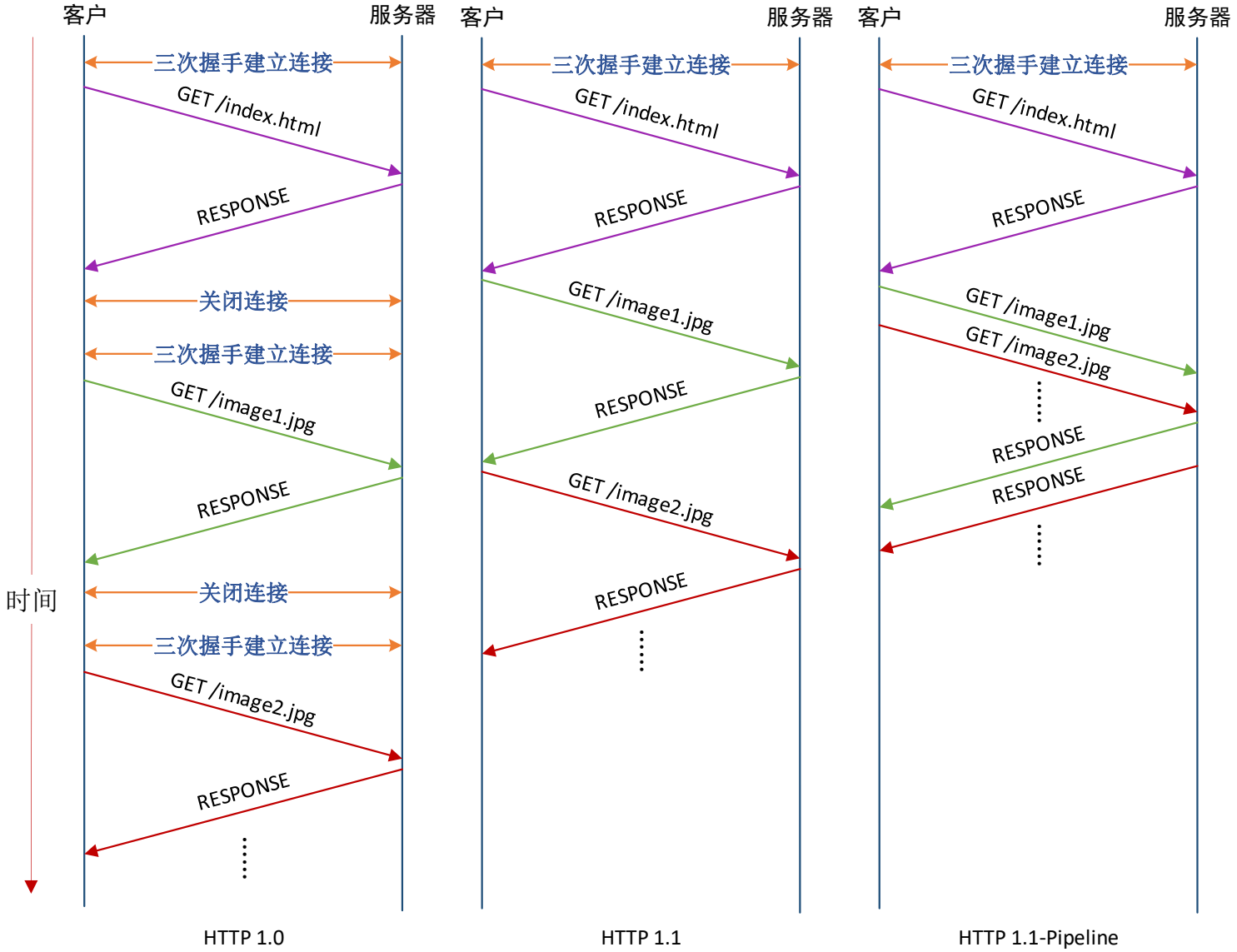
### 非持久连接

- HTTP/1.0缺省为非持久连接
  - 服务器接收请求、给出响应、关闭TCP连接
- 获取每个对象需要两阶段
  - 建立TCP连接
  - 对象请求和传输
- 每次连接需要经历TCP慢启动阶段

### 持久连接

- HTTP/1.1缺省为持久连接
  - 在相同的TCP连接上，服务器接收请求、给出响应；再接收请求、给出响应；响应后保持连接
- HTTP/1.1支持流水线机制
  - 需要按序响应
- 经历较少的慢启动过程，减少往返时间
  - 降低响应时间

# HTTP/1.0与HTTP/1.1比较



## 单选题 5分

设置

如果客户端通过浏览器从Web服务器获取一个包含4幅jpg图像页面，传输层使用TCP单连接，假设每次建立TCP连接需要一个RTT，那么使用HTTP1.0的非持久连接和HTTP1.1的持久连接（不使用流水线），获取完整的页面分别需要经历几个RTT

A 10个、6个

B 10个、5个

C 8个、6个

D 8个、5个

提交

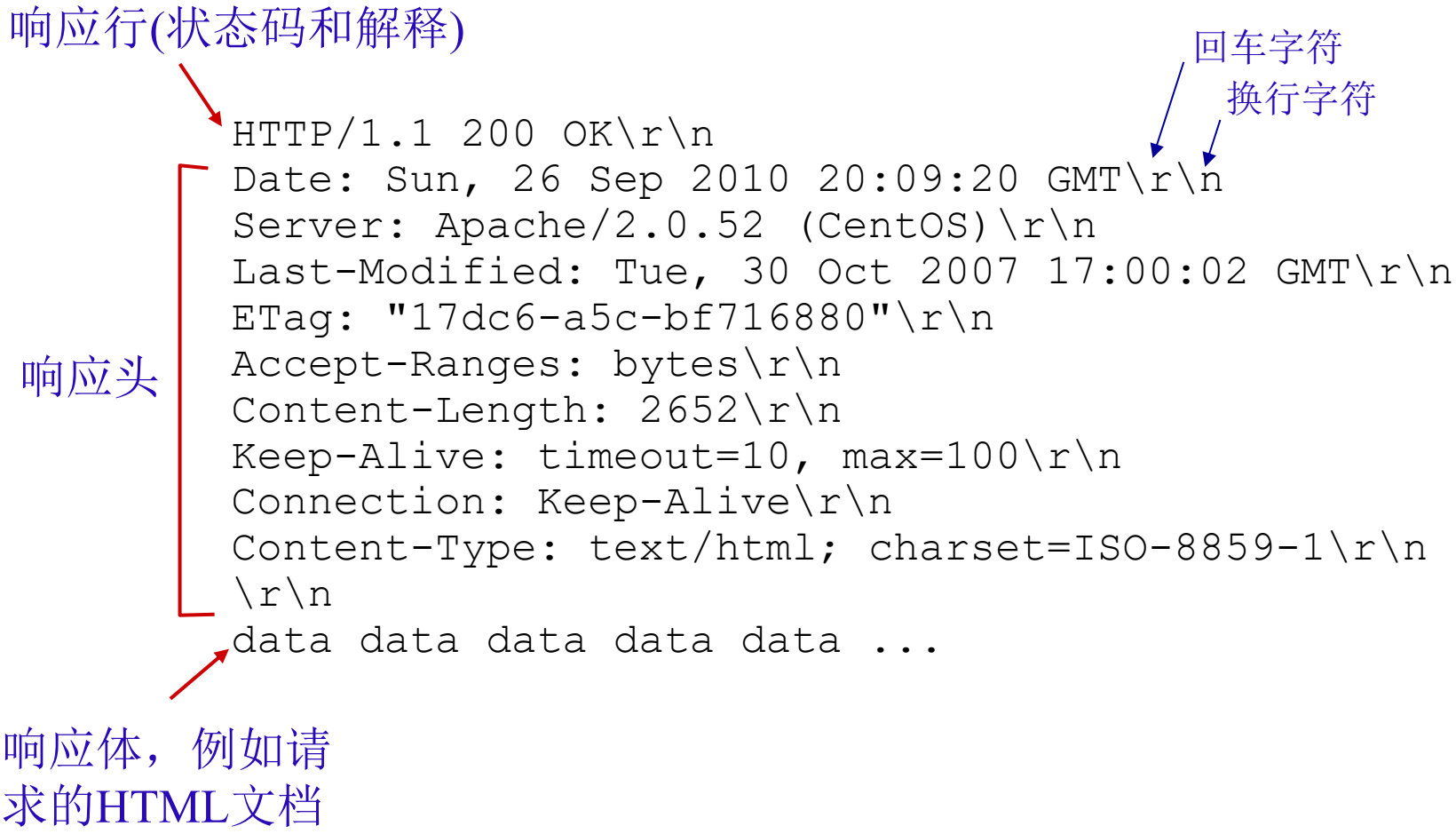
## HTTP报文类型

- HTTP两种报文：请求(request)、响应(response）
- HTTP请求报文：采用ASCII，数据部分采用MIME格式


HTTP/1.1请求方法：GET, POST, HEAD, PUT, DELETE

## HTTP报文类型

- HTTP响应报文：数据部分采用MIME格式



## HTTP报文类型

- 典型的状态码

**200 OK**

- 请求成功，被请求的对象包含在该响应的数据部分

**301 Moved Permanently**

- 请求的对象被移走，新的位置在响应中通过***Location:*** 给出

**400 Bad Request**

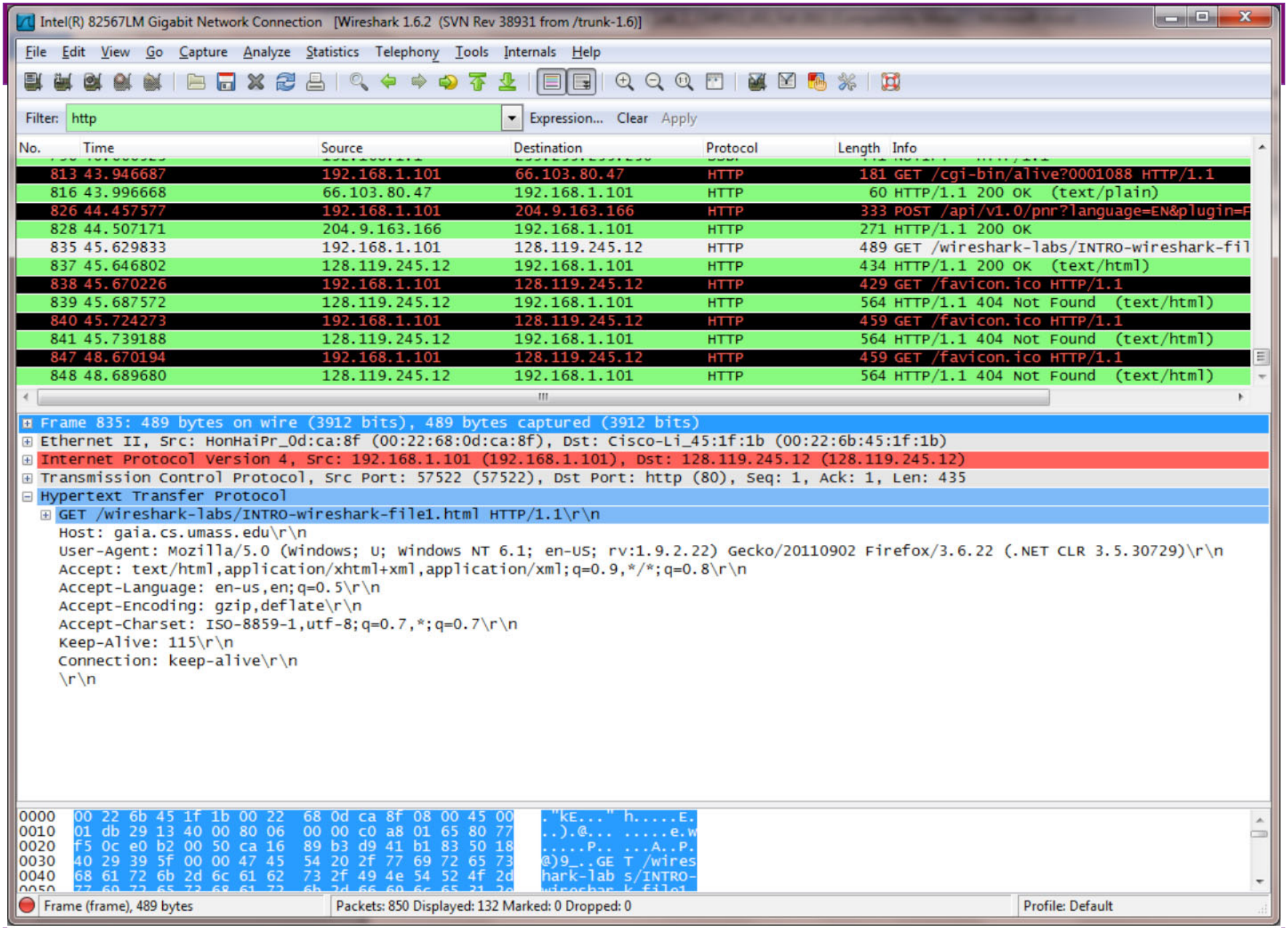
- 服务器不能解释请求报文

**404 Not Found**

- 服务器中找不到请求的文档

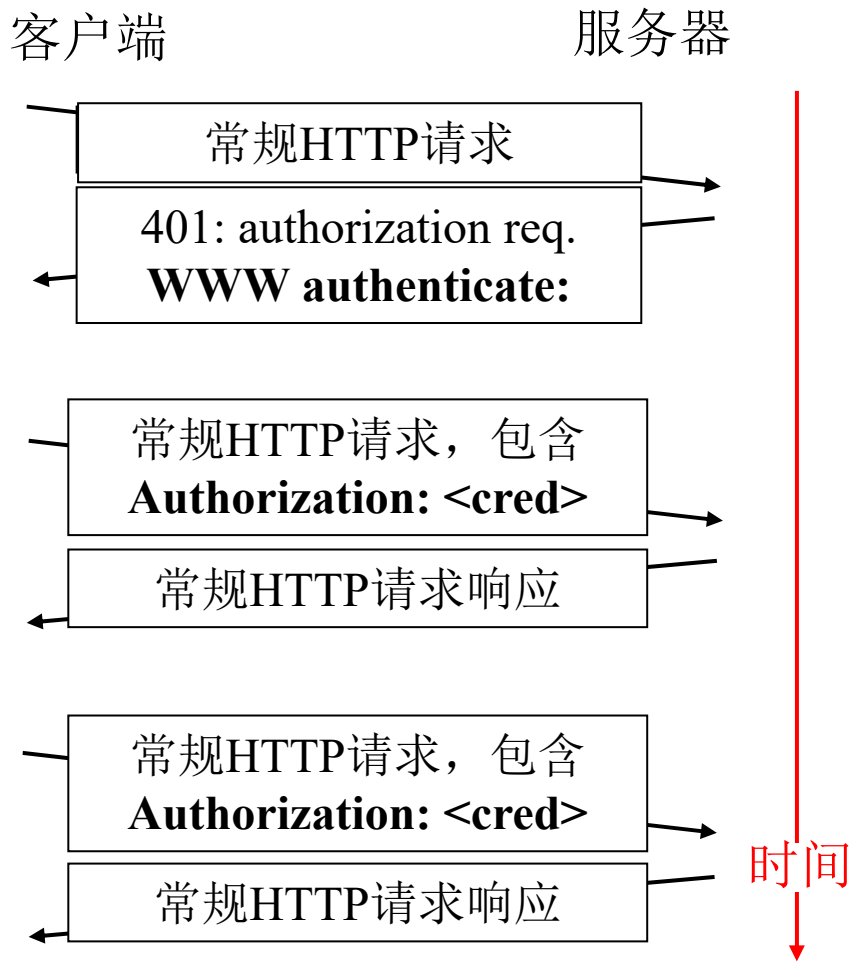
**505 HTTP Version Not Supported**

- 服务器不支持相应的HTTP版本


Intel(R) 82567LM Gigabit Network Connection [Wireshark 1.6.2 (SVN Rev 38931 from /trunk-1.6)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: http
Expression... Clear Apply
No. Time Source Destination Protocol Length Info
813 43.946687 192.168.1.101 66.103.80.47 HTTP 181 GET /cgi-bin/alive?0001088 HTTP/1.1
816 43.996668 66.103.80.47 192.168.1.101 HTTP 60 HTTP/1.1 200 OK (text/plain)
826 44.457577 192.168.1.101 204.9.163.166 HTTP 333 POST /api/v1.0/pnr?language=EN&plugin=F
828 44.507171 204.9.163.166 192.168.1.101 HTTP 271 HTTP/1.1 200 OK
835 45.629833 192.168.1.101 128.119.245.12 HTTP 489 GET /wireshark-labs/INTRO-wireshark-fil
837 45.646802 128.119.245.12 192.168.1.101 HTTP 434 HTTP/1.1 200 OK (text/html)
838 45.670226 192.168.1.101 128.119.245.12 HTTP 429 GET /favicon.ico HTTP/1.1
839 45.687572 128.119.245.12 192.168.1.101 HTTP 564 HTTP/1.1 404 Not Found (text/html)
840 45.724273 192.168.1.101 128.119.245.12 HTTP 459 GET /favicon.ico HTTP/1.1
841 45.739188 128.119.245.12 192.168.1.101 HTTP 564 HTTP/1.1 404 Not Found (text/html)
847 48.670194 192.168.1.101 128.119.245.12 HTTP 459 GET /favicon.ico HTTP/1.1
848 48.689680 128.119.245.12 192.168.1.101 HTTP 564 HTTP/1.1 404 Not Found (text/html)
Frame 835: 489 bytes on wire (3912 bits), 489 bytes captured (3912 bits)
Ethernet II, Src: HonHaiPr_0d:ca:8f (00:22:68:0d:ca:8f), Dst: Cisco-Li_45:1f:1b (00:22:6b:45:1f:1b)
Internet Protocol Version 4, Src: 192.168.1.101 (192.168.1.101), Dst: 128.119.245.12 (128.119.245.12)
Transmission Control Protocol, Src Port: 57522 (57522), Dst Port: http (80), Seq: 1, Ack: 1, Len: 435
Hypertext Transfer Protocol
GET /wireshark-labs/INTRO-wireshark-file1.html HTTP/1.1\r\n
Host: gaia.cs.umass.edu\r\n
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.22) Gecko/20110902 Firefox/3.6.22 (.NET CLR 3.5.30729)\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
0000 00 22 6b 45 1f 1b 00 22 68 0d ca 8f 08 00 45 00
0010 01 db 29 13 40 00 80 06 00 00 c0 a8 01 65 80 77
0020 f5 0c e0 b2 00 50 ca 16 89 b3 d9 41 b1 83 50 18
0030 40 29 39 5f 00 00 47 45 54 20 2f 77 69 72 65 73
0040 68 61 72 6b 2d 6c 61 62 73 2f 49 4e 54 52 4f 2d
Frame (frame), 489 bytes
Packets: 850 Displayed: 132 Marked: 0 Dropped: 0
Profile: Default

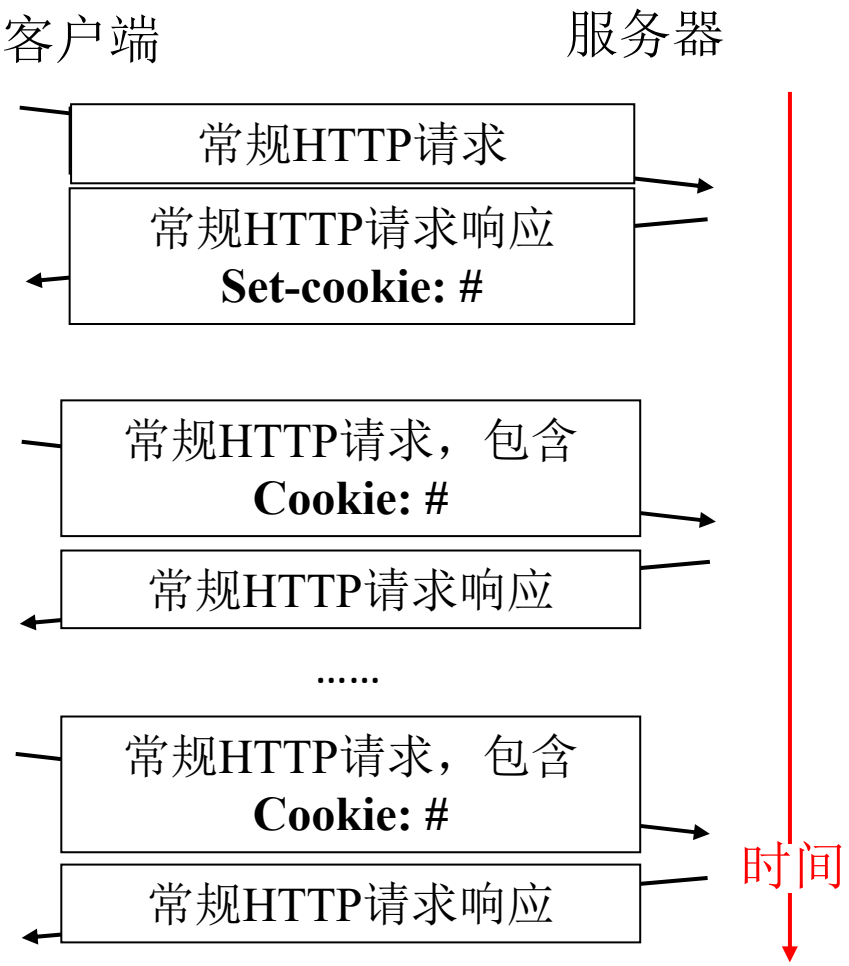
# 用户-服务器交互：认证

- 认证：控制对服务器内容的访问
  - 认证方法：通常使用“名字-口令”
  - 无状态：客户端需要在每个请求中携带认证信息
  - 每个请求头中包含***authorization:***
  - 如果请求头中无***authorization:***，则服务器拒绝访问，并在响应头中包含***WWW authenticate：***



# 用户-服务器状态：Cookies

- 服务器使用cookies保持状态
  - HTTP响应头中使用***set-cookie：***
    - 选择的cookie号具有唯一性
  - 后继的HTTP请求中使用***cookie：***
  - **Cookie文件**保存在用户的主机中，由用户主机中的浏览器管理
  - Web服务器建立后端数据库，记录用户信息
  - 例如：
    - **Set-Cookie**: SID=31d4d96e407aad42; Path=/; Domain=example.com
    - **Cookie**: SID=31d4d96e407aad42

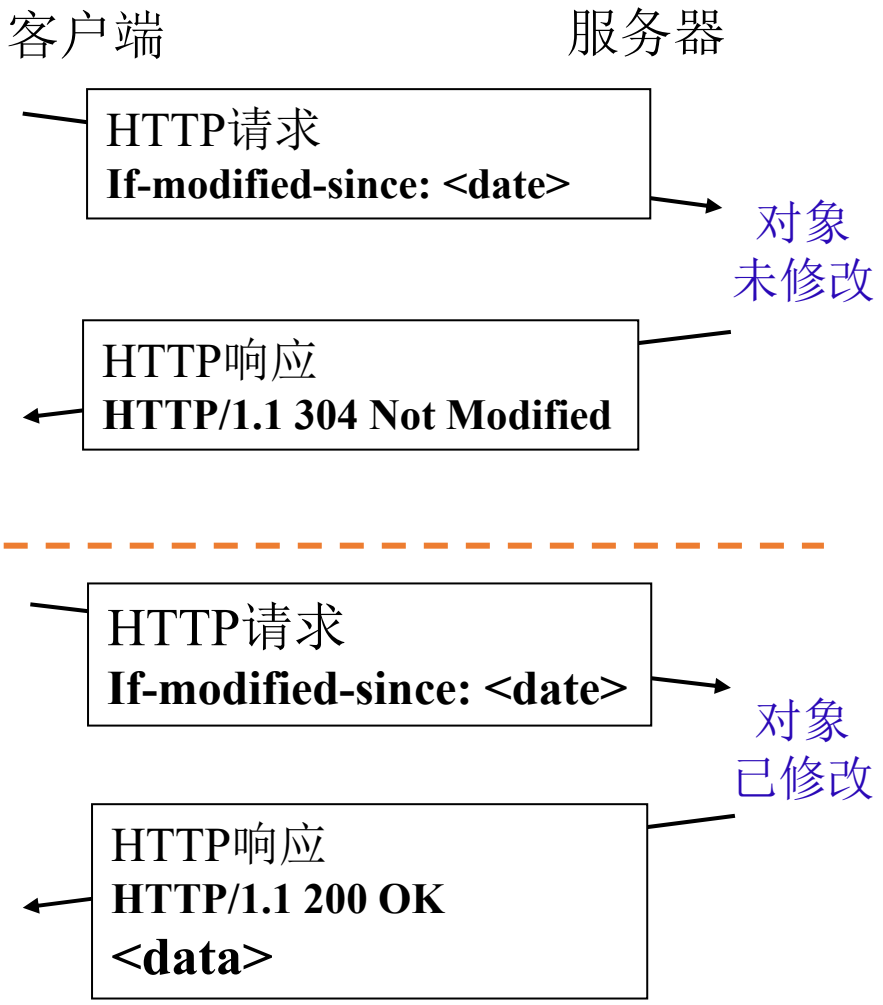


## Web缓存机制：客户端缓存

- **目标**：如果被请求的对象在客户端缓存有最近版本，则不需要发送该对象
- **客户端**：在发送的HTTP请求中指定缓存的时间，请求头包含

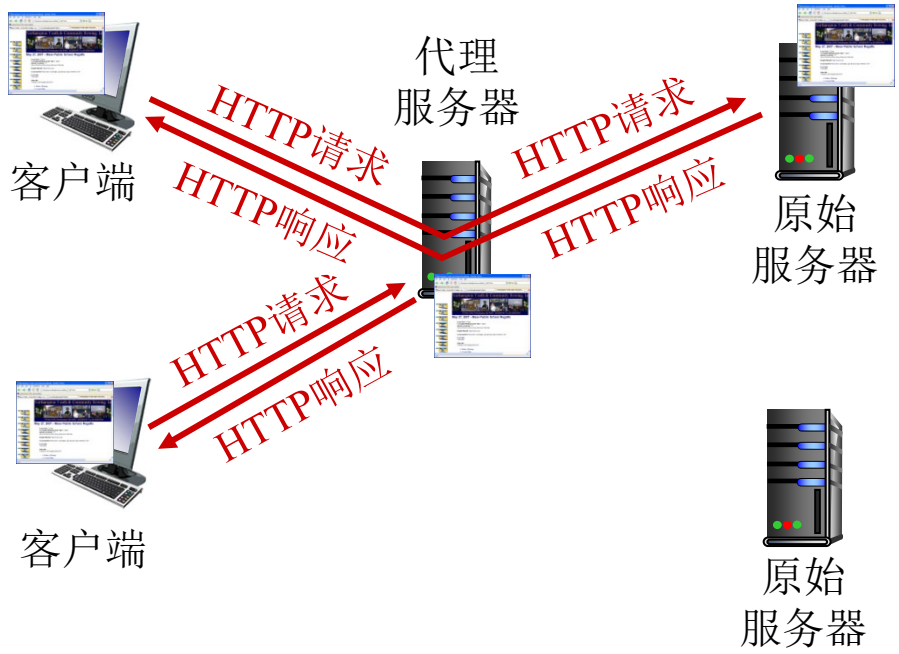
  **If-modified-since: <date>**
- **服务器**：如果缓存的对象是最新的，在响应时无需包含该对象，响应头包含

  **HTTP/1.1 304 Not Modified**



## Web缓存机制：代理服务器缓存

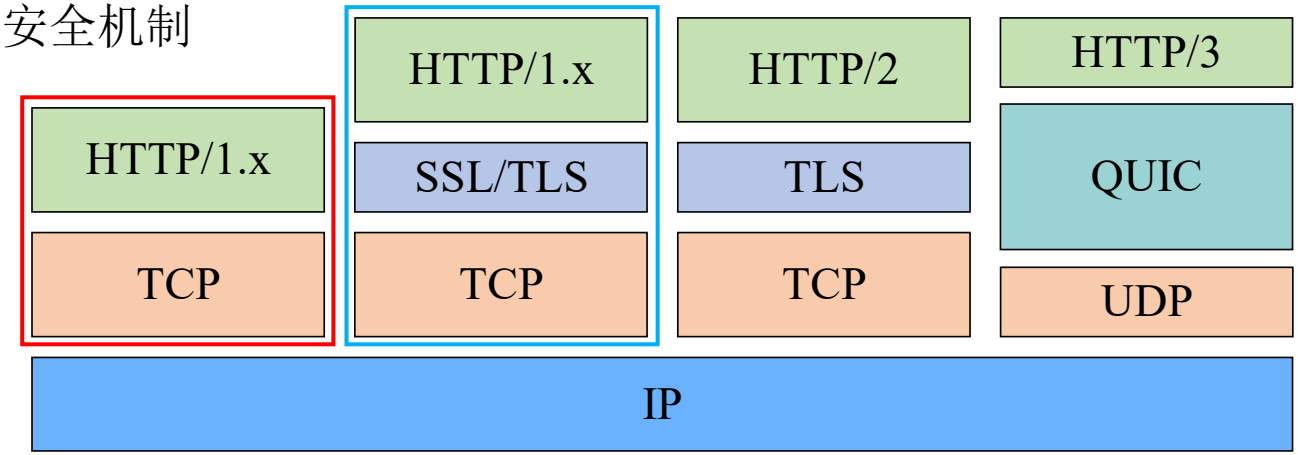
- 目标：由代理服务器进行缓存，尽量减少原始服务器参与
- 用户设置浏览器：通过代理服务器进行Web访问
- 浏览器将所有的HTTP请求发送到代理服务器
  - 如果缓存中有被请求的对象，则直接返回对象
  - 否则，代理服务器向原始服务器请求对象，再将对象返回给客户端



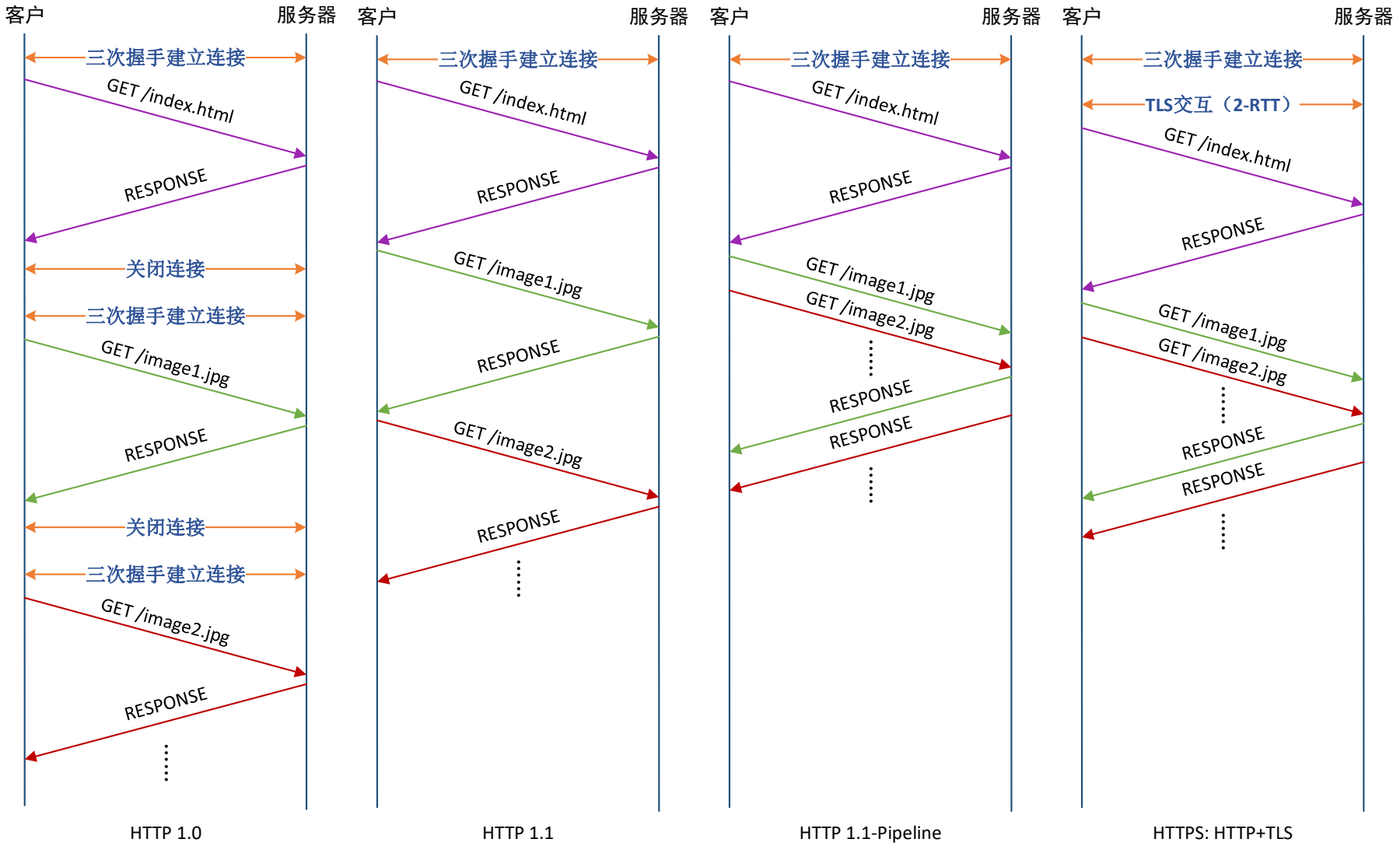

降低时延、减少网络流量

## HTTP发展现状

- HTTP/1.0（1996）
  - 无状态，非持久连接
- HTTP/1.1（1999）
  - 支持长连接和流水线机制
  - 缓存策略优化、部分资源请求及断点续传
- HTTPS：HTTP+TLS（2008）
  - 增加SSL/TLS（TLS 1.2）层，在TCP之上提供安全机制
- HTTP/2.0（2015、2020）
  - 目标：提高带宽利用率、降低延迟
  - 增加二进制格式、TCP多路复用、头压缩、服务端推送等功能
- HTTP/3.0（2022）
  - 运行在QUIC+UDP之上



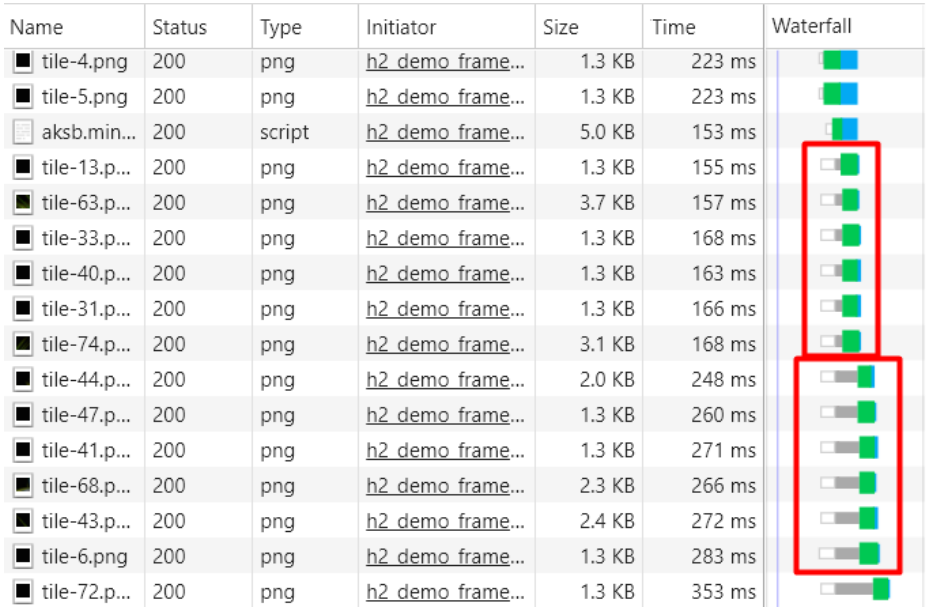
# 2.6 Web服务与HTTP协议




例：页面index.html包含10幅图像

# HTTP 1.1存在的问题

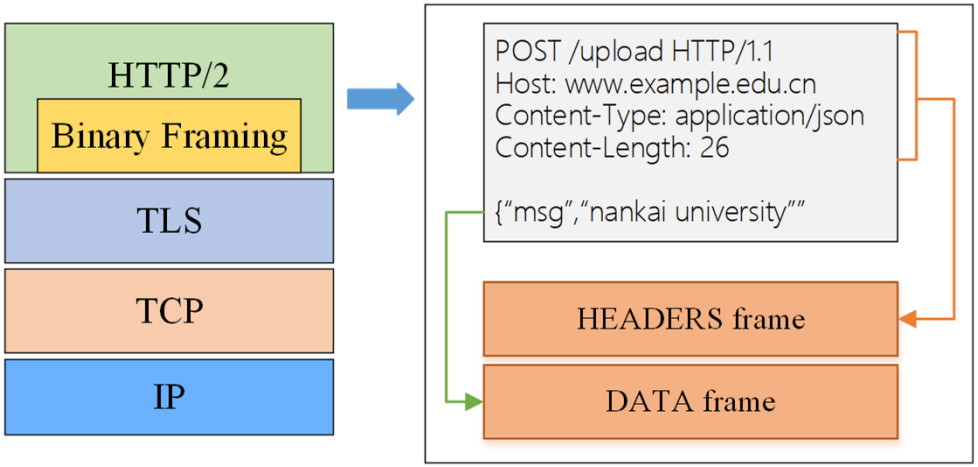
- HTTP 1.1的问题
  - 队头阻塞问题
    - 基于文本协议的问答有序模式，先请求的必须先响应
  - 传输效率问题
    - 文本格式、冗长重复的头部等
- HTTP 1.1队头阻塞的解决策略
  - 浏览器建立多个TCP连接
    - 一般最多可以建立6个TCP连接
    - 通过不同TCP连接传送的请求没有响应顺序的要求
    - 耗费较多的计算和存储资源

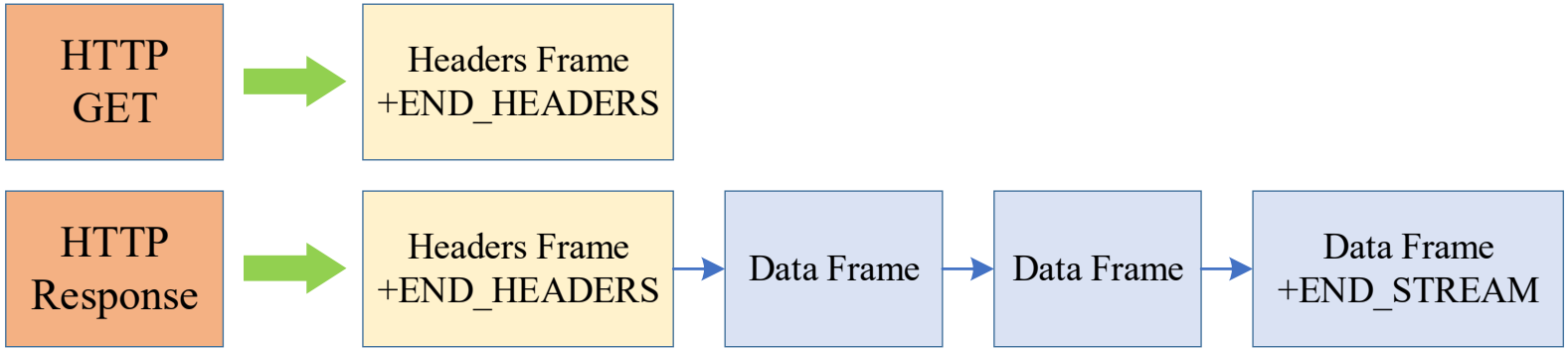


| Name | Status | Type | Initiator | Size | Time | Waterfall |
|---|---|---|---|---|---|---|
| tile-4.png | 200 | png | h2_demo_frame... | 1.3 KB | 223 ms | |
| tile-5.png | 200 | png | h2_demo_frame... | 1.3 KB | 223 ms | |
| aksb.min... | 200 | script | h2_demo_frame... | 5.0 KB | 153 ms | |
| tile-13.p... | 200 | png | h2_demo_frame... | 1.3 KB | 155 ms | |
| tile-63.p... | 200 | png | h2_demo_frame... | 3.7 KB | 157 ms | |
| tile-33.p... | 200 | png | h2_demo_frame... | 1.3 KB | 168 ms | |
| tile-40.p... | 200 | png | h2_demo_frame... | 1.3 KB | 163 ms | |
| tile-31.p... | 200 | png | h2_demo_frame... | 1.3 KB | 166 ms | |
| tile-74.p... | 200 | png | h2_demo_frame... | 3.1 KB | 168 ms | |
| tile-44.p... | 200 | png | h2_demo_frame... | 2.0 KB | 248 ms | |
| tile-47.p... | 200 | png | h2_demo_frame... | 1.3 KB | 260 ms | |
| tile-41.p... | 200 | png | h2_demo_frame... | 1.3 KB | 271 ms | |
| tile-68.p... | 200 | png | h2_demo_frame... | 2.3 KB | 266 ms | |
| tile-43.p... | 200 | png | h2_demo_frame... | 2.4 KB | 272 ms | |
| tile-6.png | 200 | png | h2_demo_frame... | 1.3 KB | 283 ms | |
| tile-72.p... | 200 | png | h2_demo_frame... | 1.3 KB | 353 ms | |

# HTTP 2.0协议基本原理

- 二进制分帧传输
  - 不改变HTTP原有的语义
  - 将HTTP请求和响应分割成帧，采用二进制编码
  - 帧为最小传输单位
- 最常用的 HTTP 请求 / 响应的帧形式



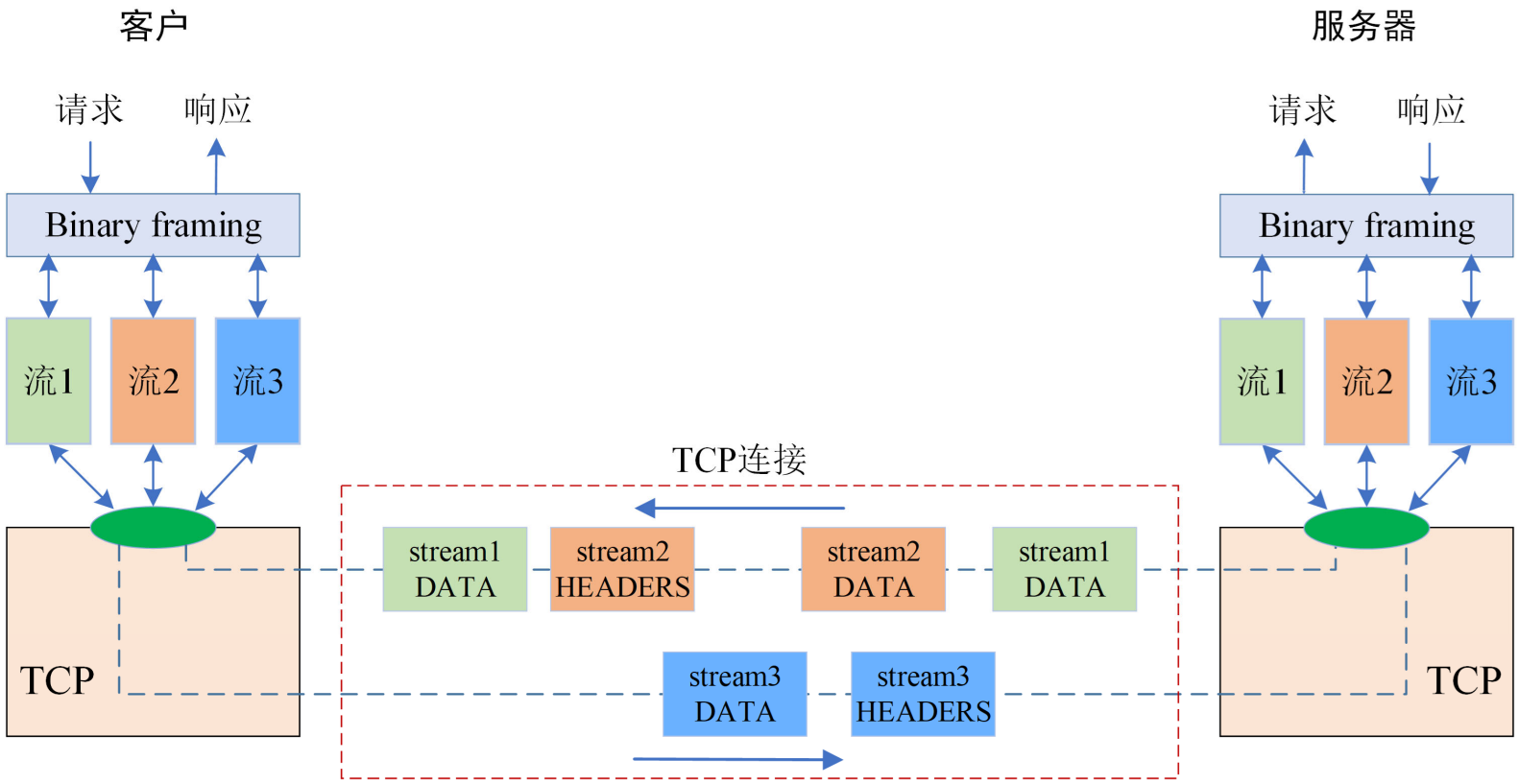



## HTTP 2.0协议基本原理（续）

- TCP连接复用：提高连接利用率，解决HTTP的队头阻塞问题
  - 消息(Message)：HTTP一次请求或响应，包含一个或多个帧
  - 流(Stream)：简单看成一次请求和应答，包含多个帧
  - 每个TCP连接中可以承载多个流，不同流的帧可以交替穿插传输
  - 流的创建与标识
    - Stream ID：标识一个流。客户端创建的流，ID为奇数；服务器创建的流，ID为偶数；0x00和0x01用于特定场景；Stream ID 不能重复使用，如果一条连接上ID分配完，会新建一条连接。接收端通过Stream ID进行消息的组装。
    - 流创建：发送和接收到HEADERS帧（包含新Stream ID）时创建
    - 流优先级：可以依据重要性为流设置不同的优先级（1~256），在HEADERS帧中承载

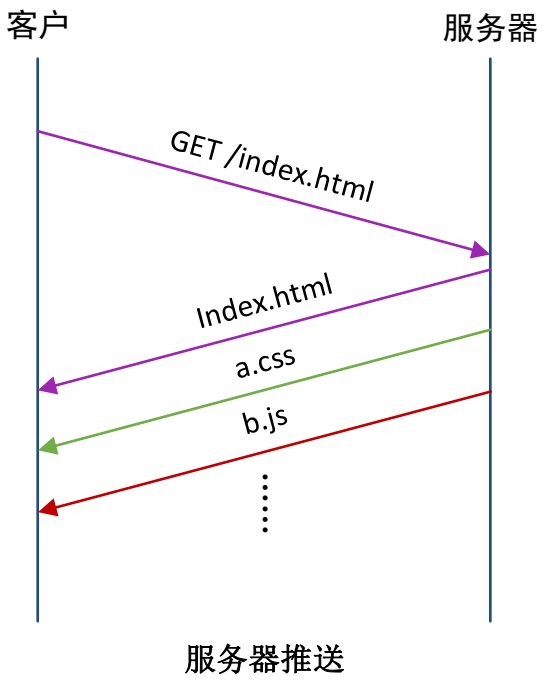
## HTTP 2.0协议基本原理（续）

- TCP连接复用（示例）



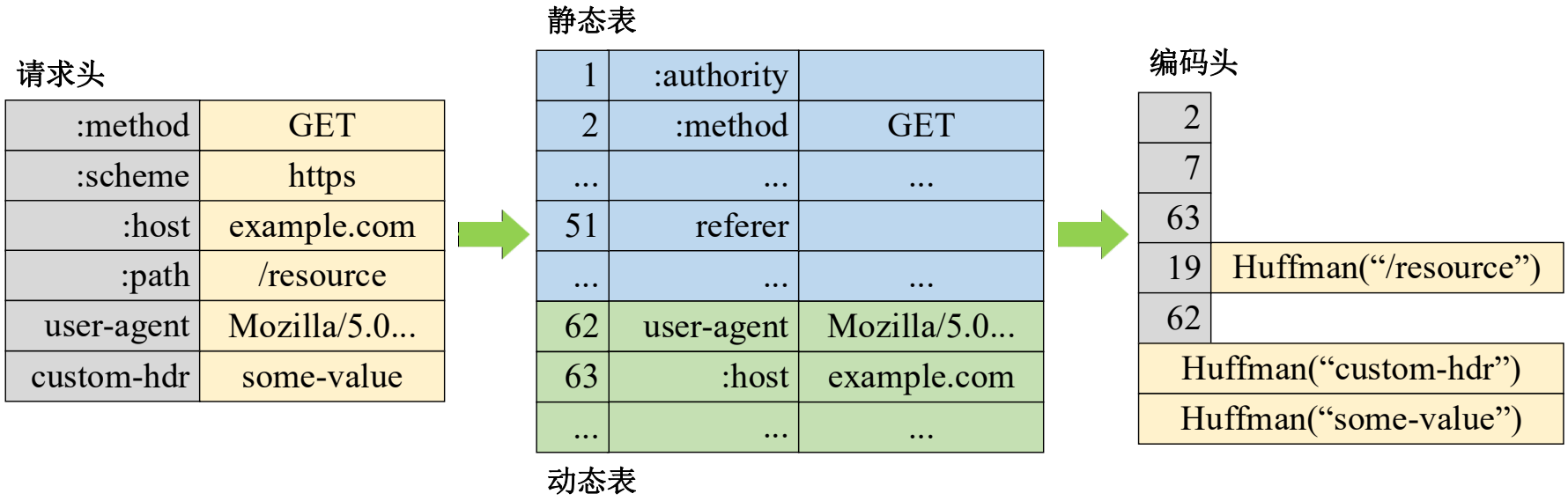
# HTTP 2.0协议基本原理（续）

- 服务器推送：提高响应速度
  - 服务器在请求之前先推送响应信息到客户端，推送的响应信息可以在客户端被缓存
- HTTP头压缩（HPACK）
  - 请求头由大量的键值组成，多个请求的键值重复程度很高
  - 静态表：定义通用HTTP头域，常用键值无需重复传送，直接引用内部字典的整数索引
  - 动态表： 两边交互发现新的头域，添加到动态表
  - 自定义键值：采用Huffman编码



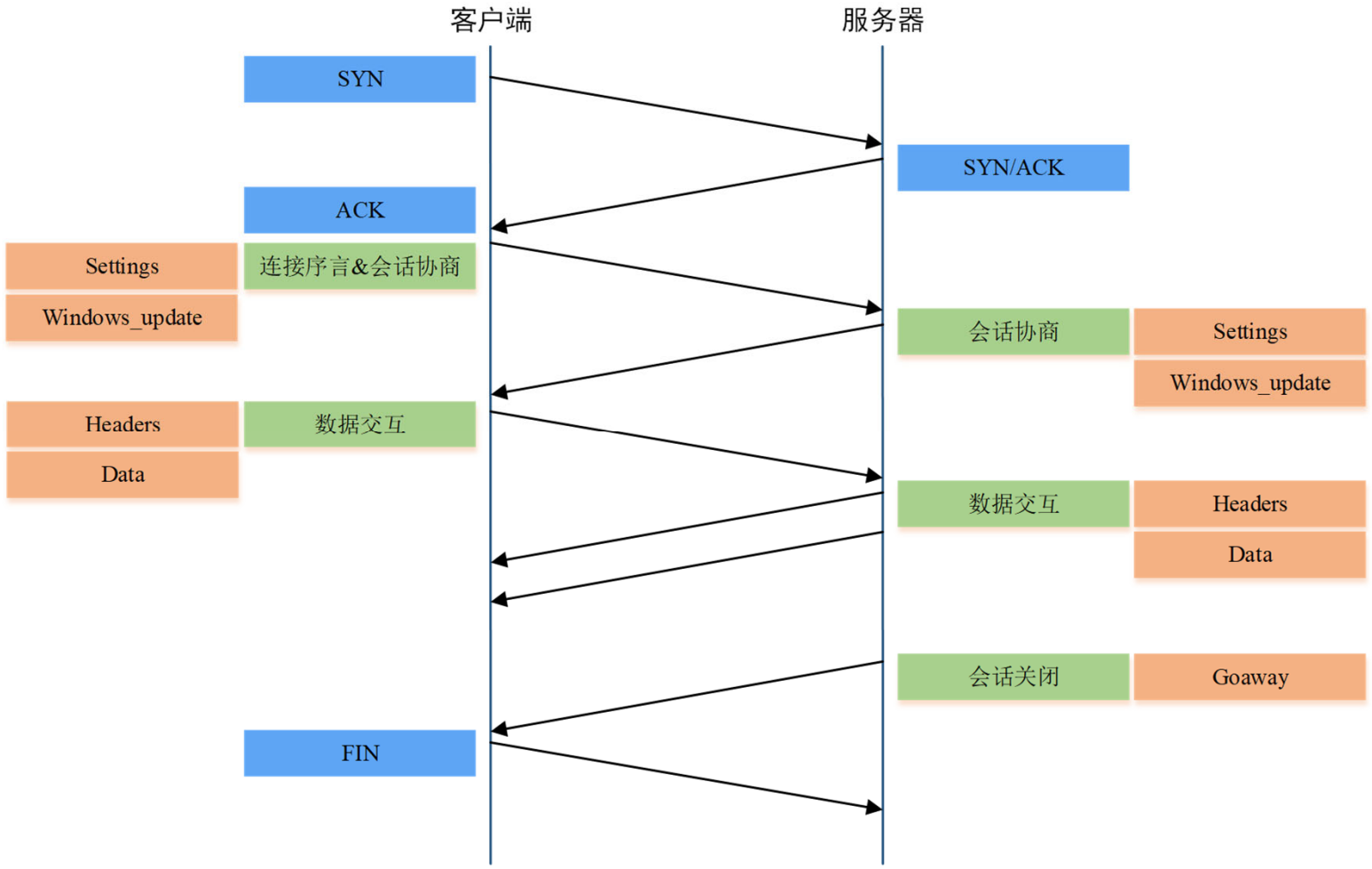
# HTTP 2.0协议基本原理（续）

- HTTP头压缩示例



# HTTP 2.0协议基本原理（续）

- HTTP连接流程



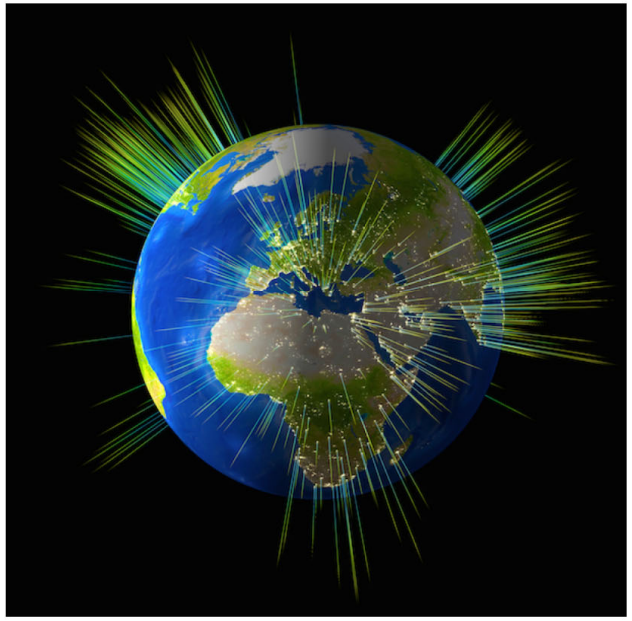
# HTTP 2.0协议基本原理（续）

Akamai

**HTTP/2 is the future of the Web, and it is here!**

**Your browser supports HTTP/2!**

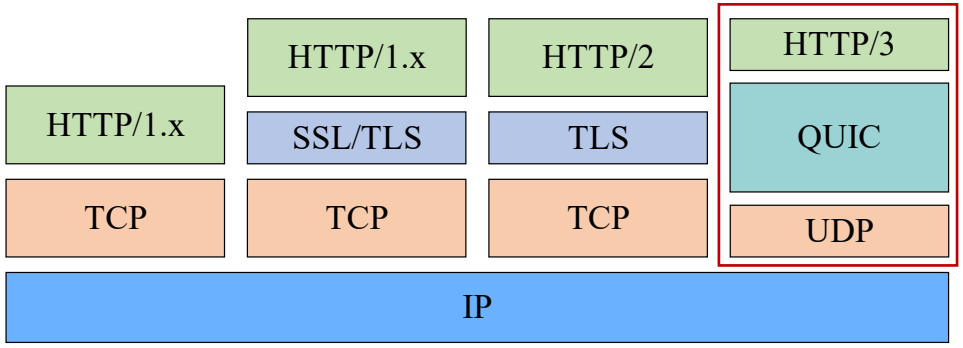
This is a demo of HTTP/2's impact on your download of many small tiles making up the Akamai Spinning Globe.

| HTTP/1.1 | HTTP/2 |
| --- | --- |
| Latency: 138ms<br>Load time: 9.68s | Latency: 131ms<br>Load time: 1.63s |
|  |  |

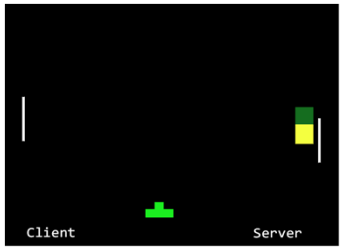
https://http2.akamai.com/demo

# HTTP 2.0协议基本原理（续）

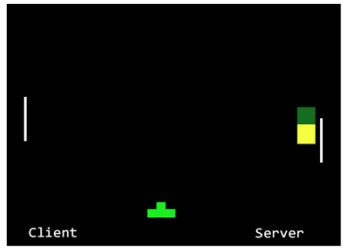
- HTTP 2.0协议解决的问题
  - 通过引入流机制，解决了HTTP队头阻塞问题，提高了传输效率
  - 通过二进制编码、头压缩机制提高了网络带宽利用率
  - 通过服务器推送，加快了页面响应速度
- HTTP 2.0协议没有解决的问题
  - TCP+TLS的多次交互，造成启动延迟问题
  - 移动主机和多宿主机的连接迁移问题
  - TCP队头阻塞问题




动画：同时发送两个流，黄颜色的流上有丢包



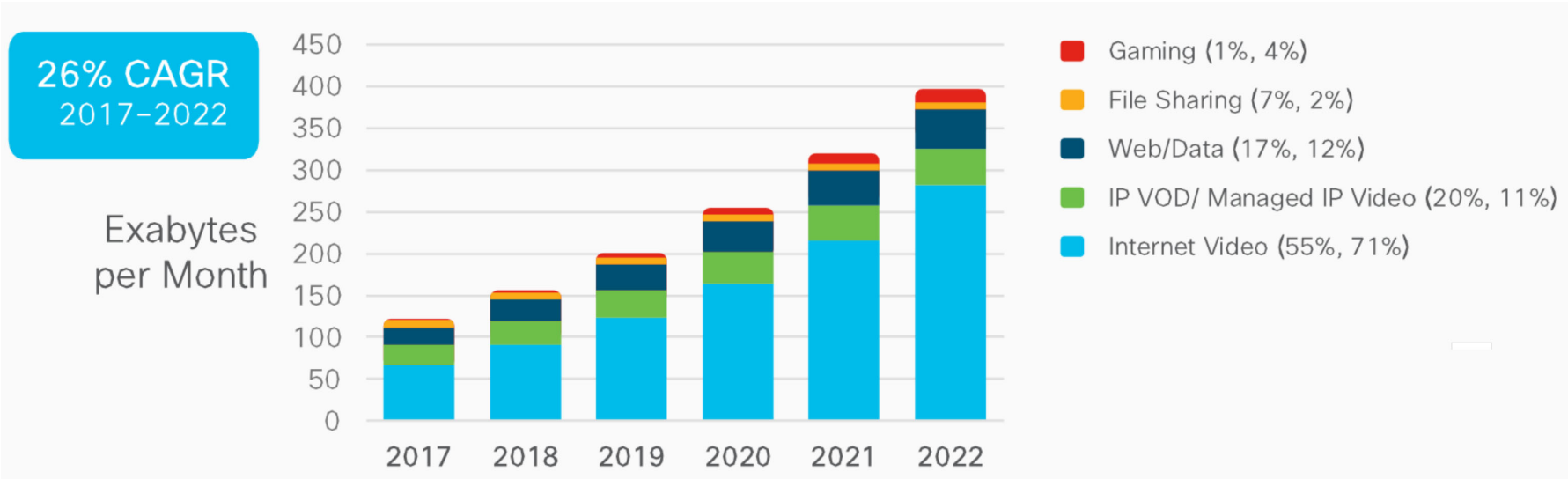

TCP丢包会阻塞
所有后续的流




QUIC丢包只会
影响相关的流

# 互联网IP流量发展趋势

全球IP流量（按应用分类）




- IP Video流量到2022年将占82%

参考：Cisco Visual Networking Index: Forecast and Trends, 2017–2022 (white paper), Cisco, Feb. 2019

**关注文章Waiting Times In Quality of Experience For Web Based Services关于QoE的观点**

## 内容分发网络概述

CDN (Content Distribution Network)

- 基本思想源于MIT对Web服务瞬间拥塞问题的解决（1998）
  - 一种Web缓存系统，靠近网络边缘（用户）提供内容服务
  - 目前提供更丰富的服务，包括静态内容、流媒体、用户上传视频等
- 主要优点
  - 降低响应时延，避免网络拥塞
  - 避免原始服务器过载及防止DDoS攻击
  - 分布式架构，具有良好的可扩展性
  - 对用户透明，无需用户感知

## 内容分发网络概述（续）

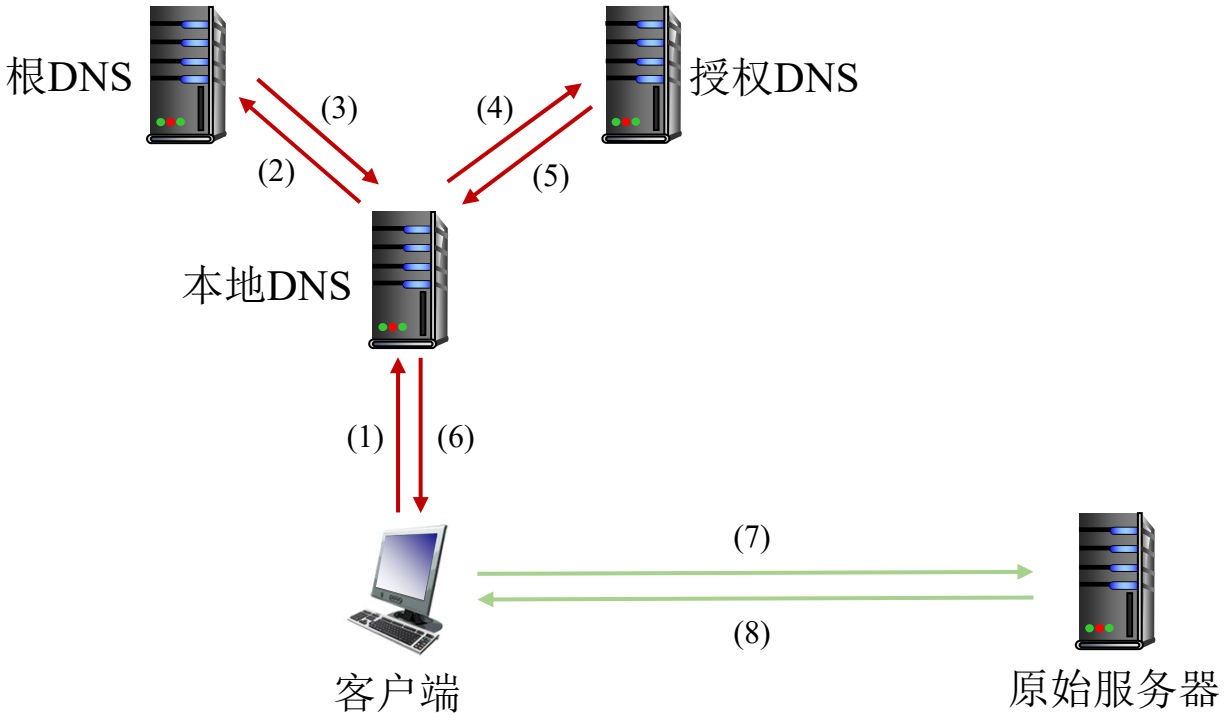
- **关键问题**
  - CDN服务器如何布局
  - 在哪里缓存、缓存哪些内容
  - 如何进行重定向，将请求调度到较近或负载较轻的CDN服务器
- **CDN服务提供**
  - CDN服务提供商：Akamai、蓝讯、世纪互联、网宿等
  - 互联网内容提供商（ICP）：腾讯、百度等
  - 互联网服务提供商（ISP）：移动、联通、电信等

✓ 腾讯云CDN：2100+节点覆盖国内移动、联通、电信及十几家中小型运营商，以及全球50+国家地区，全网带宽120Tbps+

✓ Akamai: 240,000部署在 > 120 个国家内 (2015)

## CDN基本原理

- 传统Web服务访问



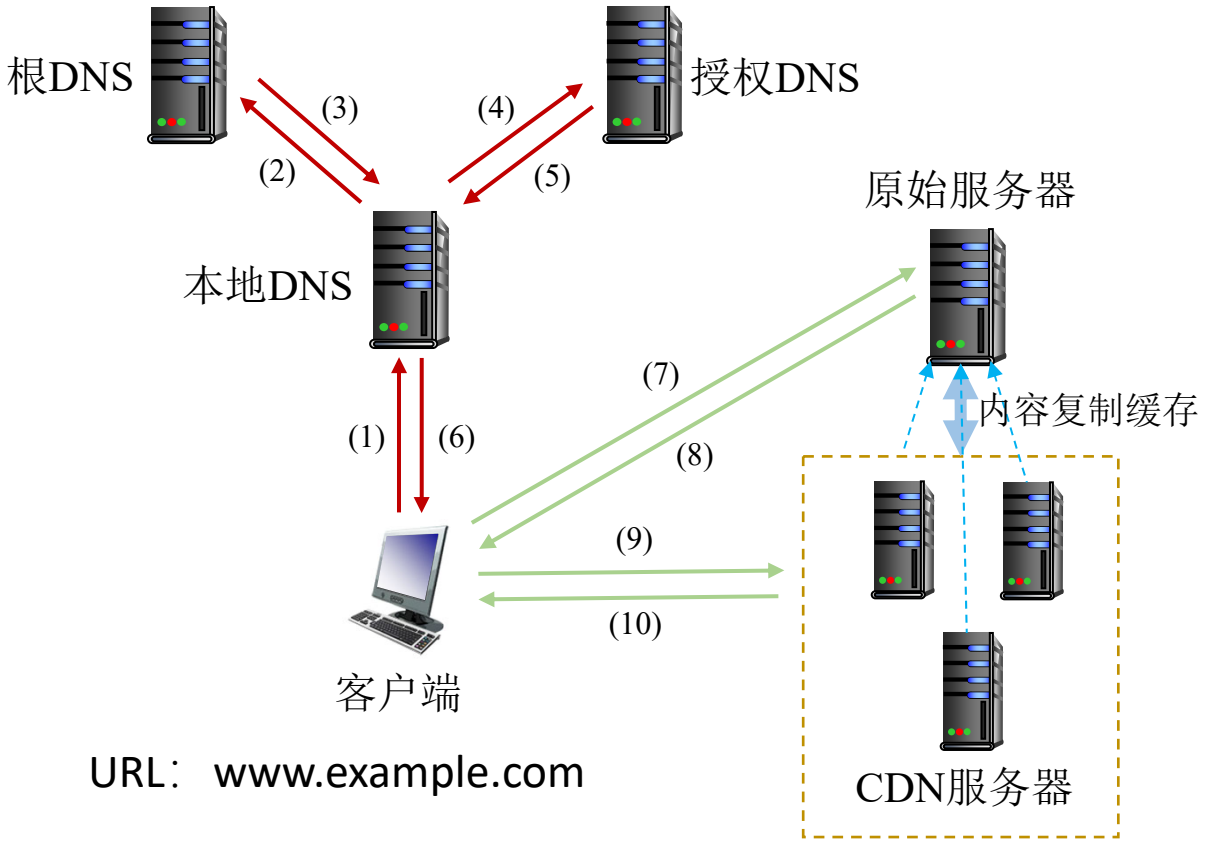

URL： www.example.com

## CDN基本原理

- CDN的实现机制（1）

HTTP重定向

- ✓ 原始服务器决策CDN服务器
- ✓ HTTP响应：状态码30X，***Location:***指明新的位置

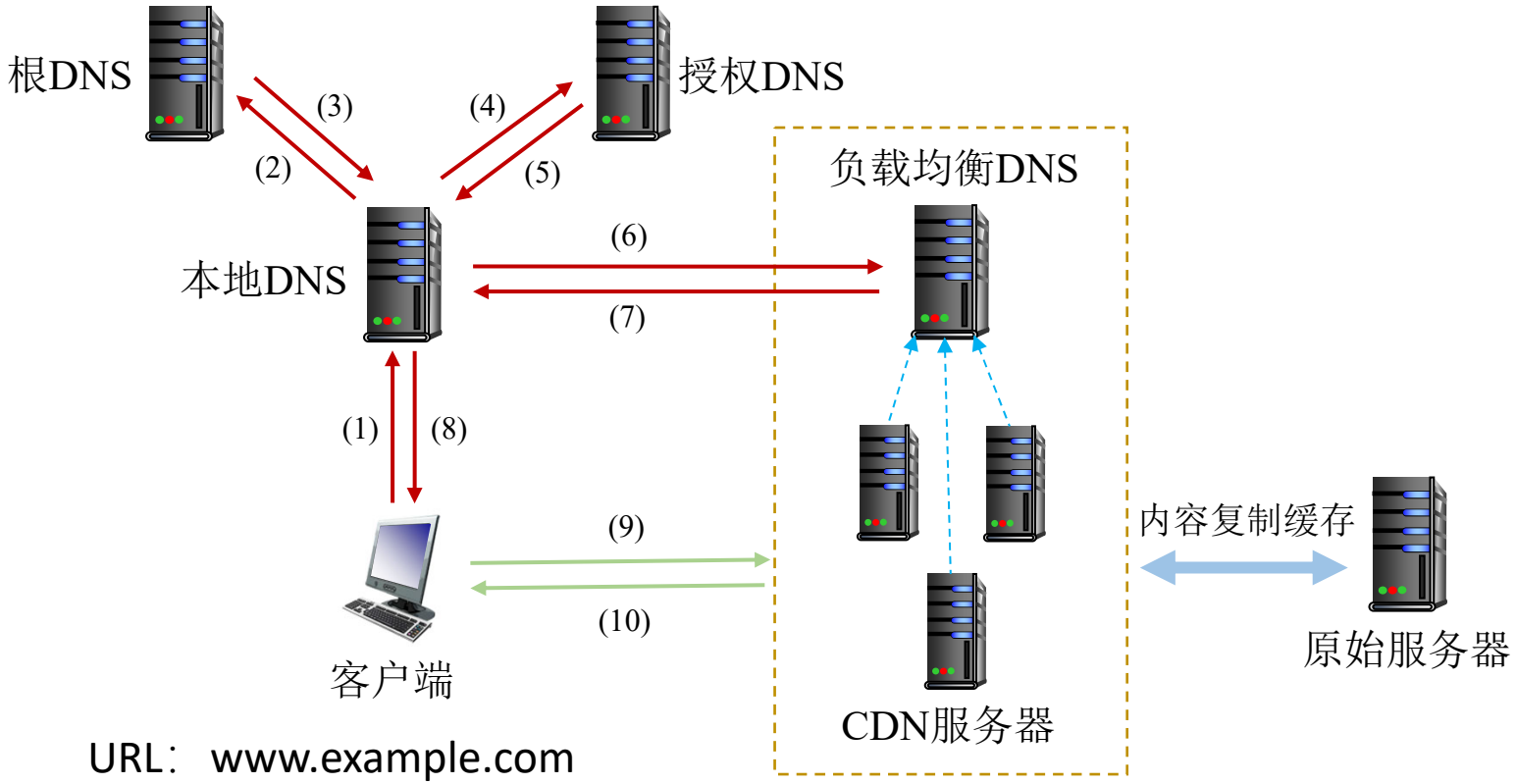


## CDN基本原理

- CDN的实现机制（2）

DNS辅助

- ✓ 负载均衡DNS负责决策CDN服务器选择
- ✓ 负载均衡DNS需要收集CDN服务器的位置和负载情况
- ✓ 如果找不到被请求的对象，需要从原始服务器获取




扩展：CDN服务器的层次化组织→移动边缘缓存→移动边缘计算

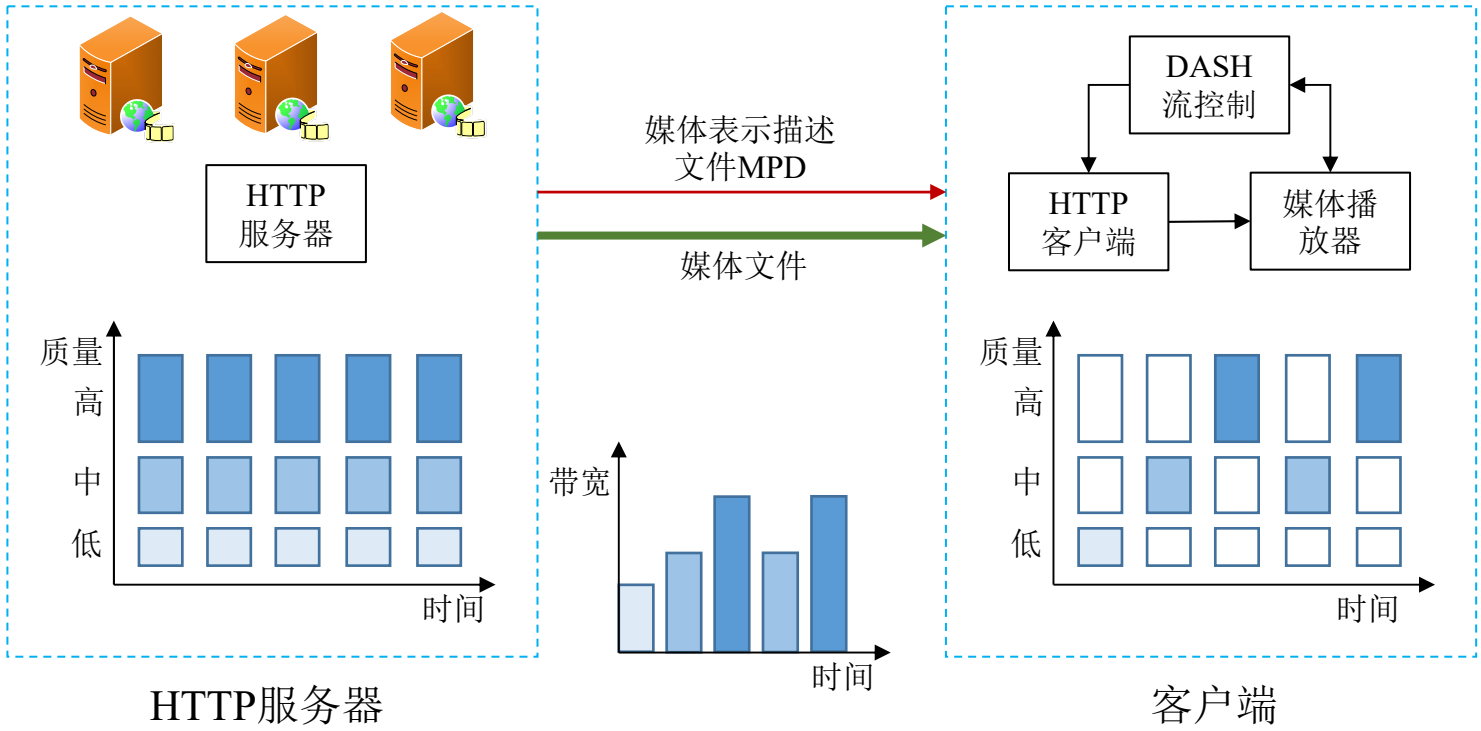
## DASH概述

- DASH (Dynamic Adaptive Streaming over HTTP)
  - MPEG组织制定的标准（标准号ISO/IEC 23009-1）
    - 也称MPEG-DASH
  - 基于HTTP的流媒体传输协议
    - 基于HTTP渐进式下载（类流媒体）
  - 类似协议：
    - HTTP Live Streaming（HLS），苹果公司
    - HTTP Dynamic Streaming（HDS），Adobe公司
    - Microsoft Smooth Streaming （MSS），微软公司

## DASH概述

- 基本思想：
  - 完整视频被拆分为固定时长 (2s-10s)、不同码率的视频片段(segment)
  - 视频片段与媒体表示描述 (Media Presentation Description, MPD) 文件一同存放于DASH服务器
  - 客户端根据自身设备性能、当前网络条件、客户端缓冲大小等自适应选择一种视频码率进行下载

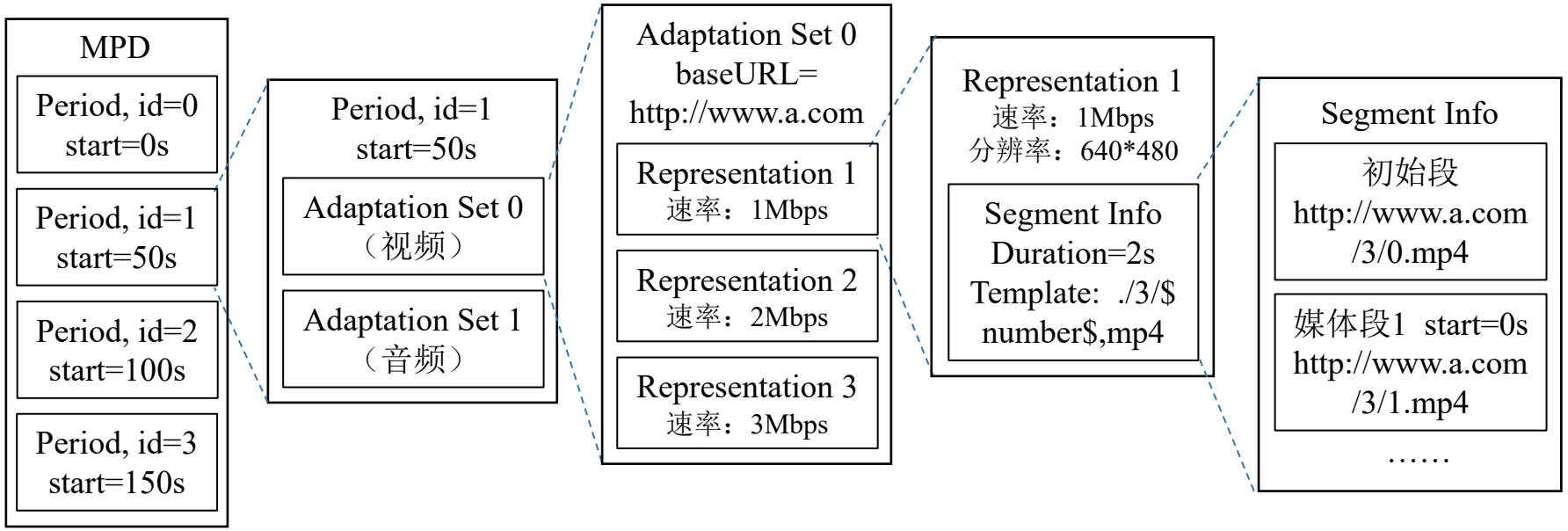
## DASH基本原理




- 例如：HTTP服务器中保存有高中低三种质量的视频片段，DASH客户端评估网络状况，通常在保证视频流畅的前提下，获取最高质量的视频片段

## DASH基本原理（续）

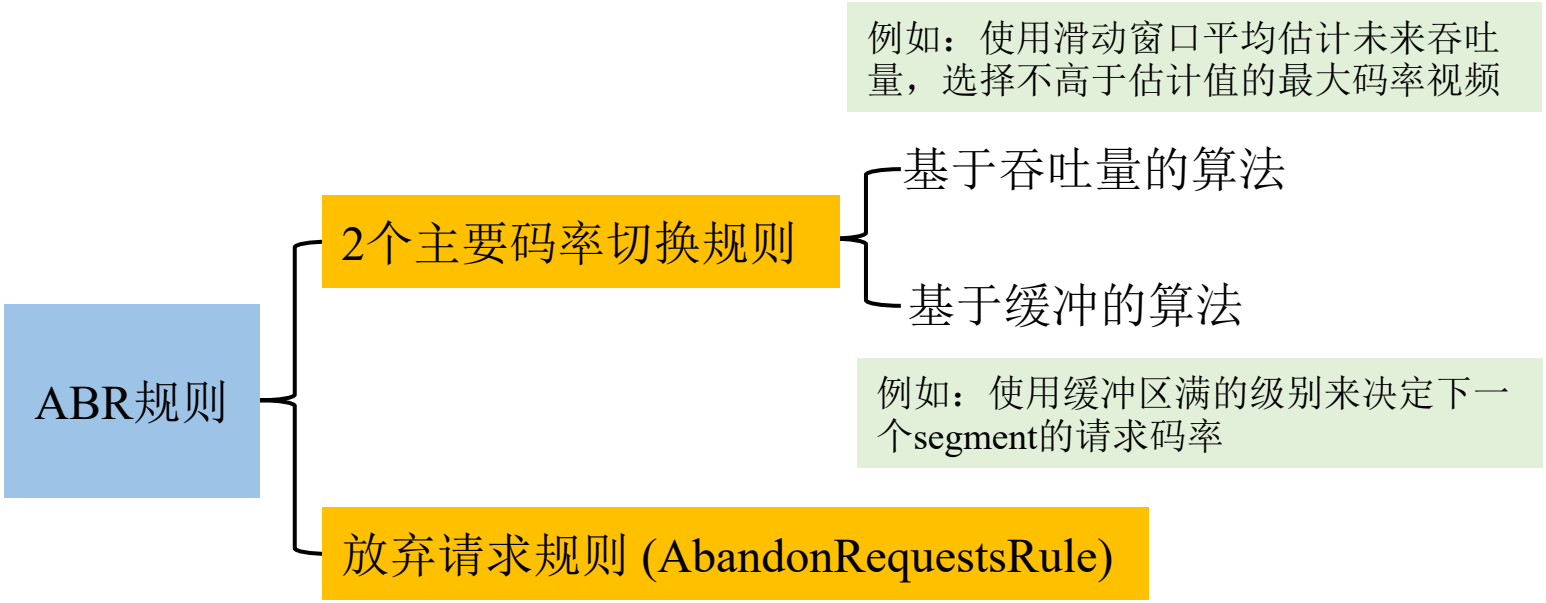
- MPD (Media Presentation Description) 文件
  - 一种 XML 文件，描述了DASH流媒体中视频/音频文件信息



## DASH基本原理

典型的DASH开源播放器dash.js

- 自适应码率（Adaptive bitrate，ABR）规则




在下载视频块的过程中，如果算法检测到下载该视频块的过程过有卡顿，则终止当前请求，转而重新下载相应的低码率视频块。

扩展：考虑多DASH服务器同时传输

# 总 结

- 应用协议与进程通信模型
- 传输层服务对应用的支持
- Socket编程
- 域名系统DNS
- 电子邮件服务与协议
- 文件传输服务与协议
- Web服务与HTTP协议
- 内容分发网络CDN
- 动态自适应流媒体协议DASH