



网络应用层

刘志敏

liuzm@pku.edu.cn



提纲

- 网络应用层
- Web 和 HTTP
- 电子邮件：SMTP, POP3, IMAP
- DNS
- P2P 应用
- socket 编程
- 流媒体传输及其协议
- QoS及其技术
- 网络安全概述

网络应用

大量网络应用，促进互联网的发展
编写程序：

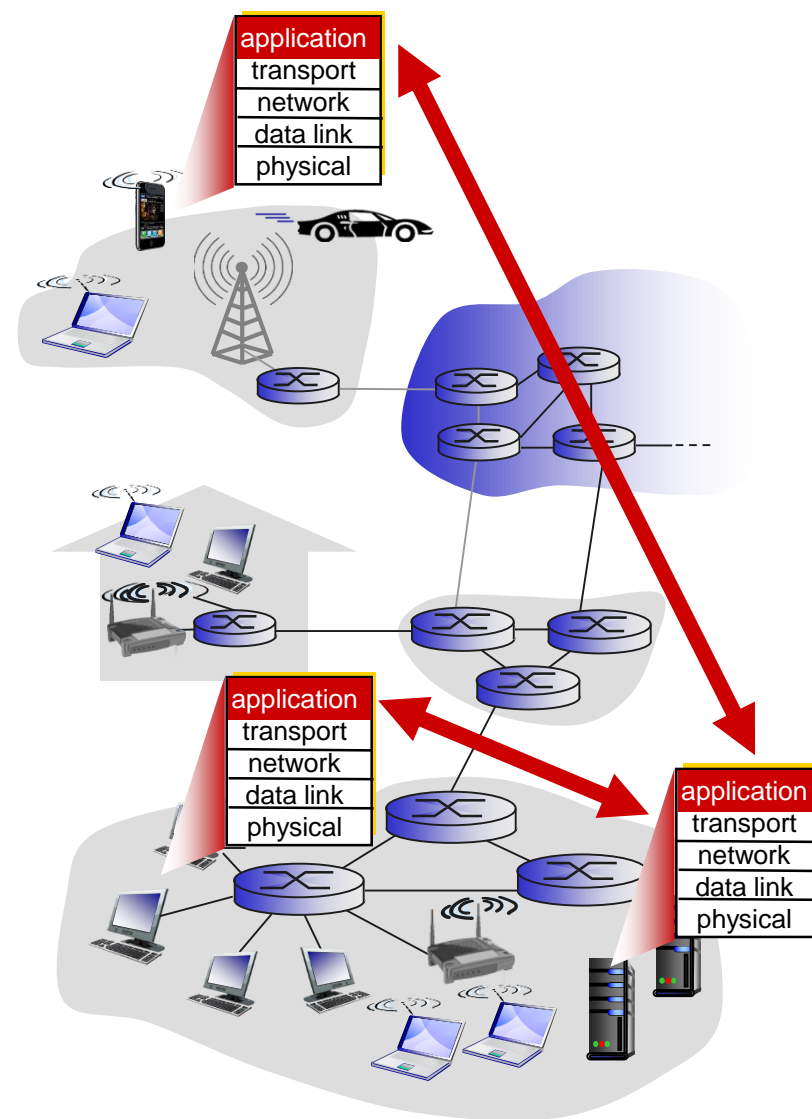
- 运行在不同主机上
- 利用网络实现数据交换
 - 例如：web 服务器与浏览器之间的通信

无需编写网络核心设备上的程序

- 在网络核心设备不运行用户应用程序
- 在主机上运行程序，便于开发及推广应用

应用程序体系结构：

- client-server (C/S)
- peer-to-peer (P2P)



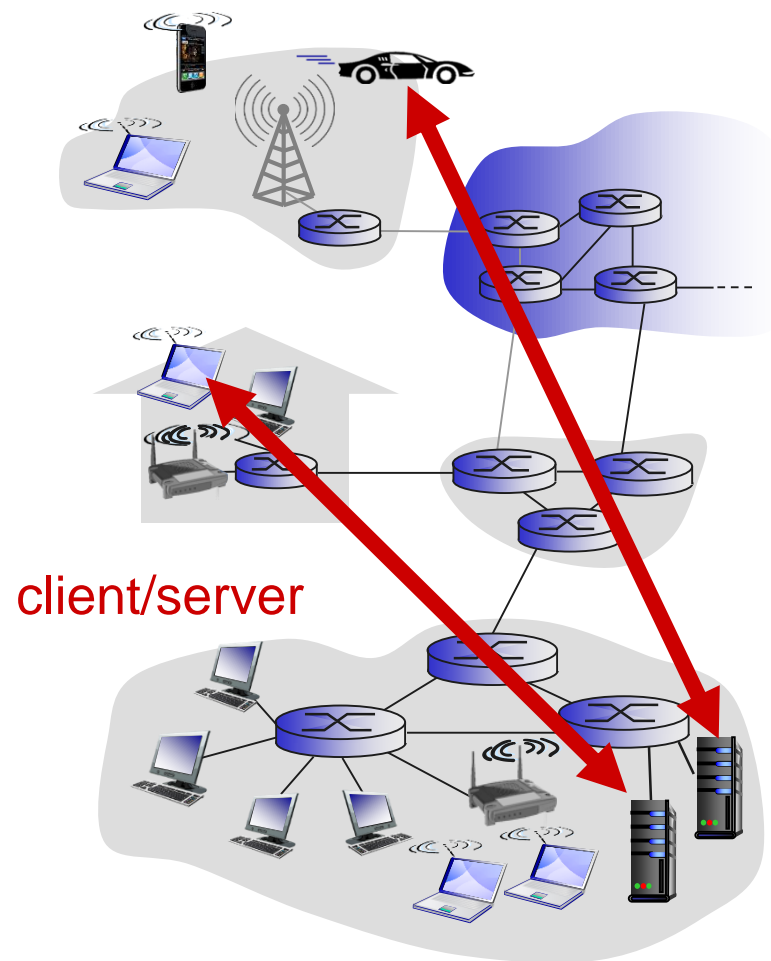
客户机服务器结构

服务器:

- 在主机上运行的程序，等待客户机请求
- 固定的IP地址
- 数据中心：服务器集群提供服务，如搜索引擎如谷歌百度、电子商务如阿里等

客户机:

- 与服务器通信
- 通信可以是间断的，可以是动态的IP地址
- 客户机之间不直接通信，通过服务器间接通信

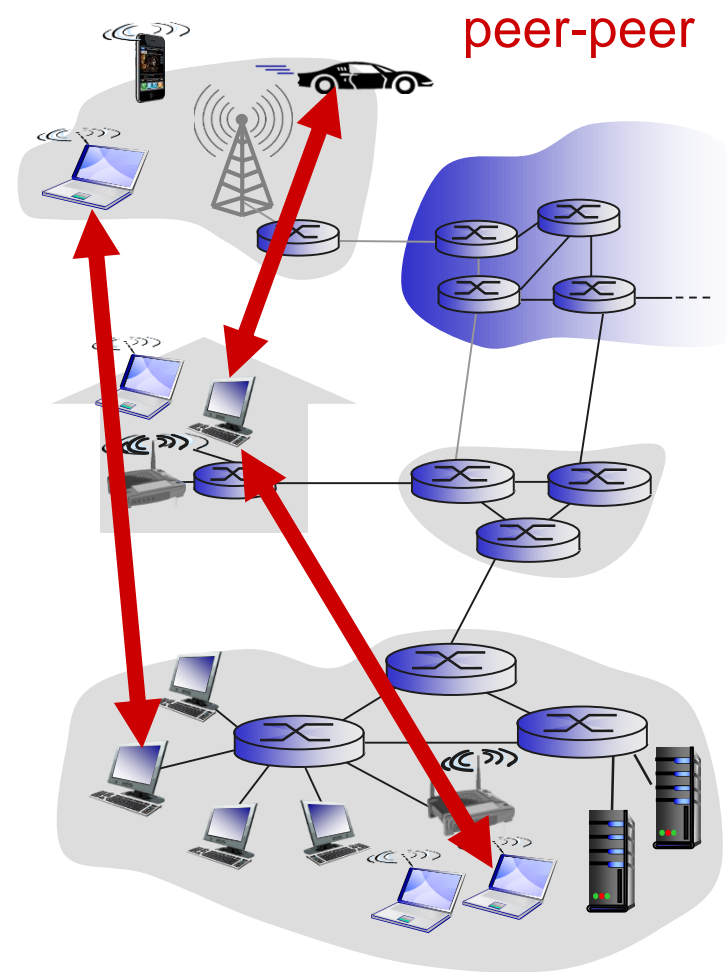


P2P 体系结构

- 典型的应用包括文件共享，如迅雷
- 没有永远的服务器
- 主机之间直接通信
- 主机向其他主机请求服务，同时也向其他主机提供服务
- 主机之间是间断连接的，IP地址可变

自扩展性

- 例如在文件共享中，对每个对等方增加了网络负载，同时也增加了服务能力
- 管理复杂

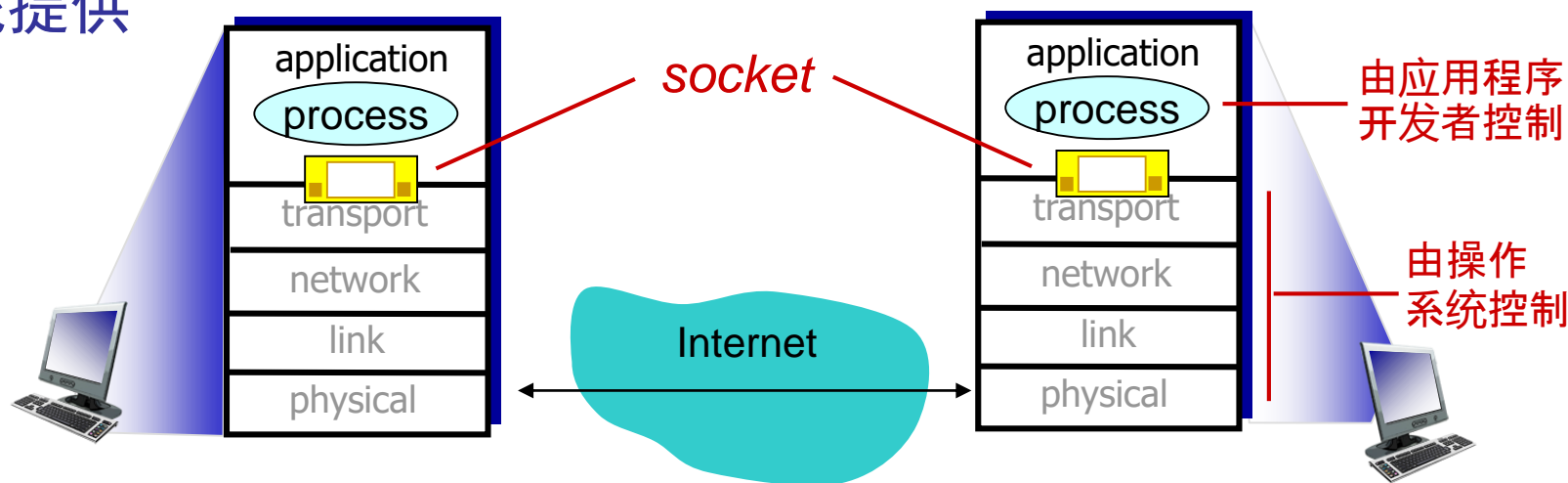


进程通信

进程：在主机上运行的应用程序

- 同一主机上的两个进程之间采用进程间通信IPC（**inter-process communication**），由操作系统提供
- 不同主机上的进程之间交换**消息**实现通信
- **client process**: 通信的发起方；**server process**: 等待建立连接方
- 在P2P中进程既是客户机又是服务器

进程与网络间接口——Socket：负责消息收发的进程，是应用层的编程接口，由操作系统提供

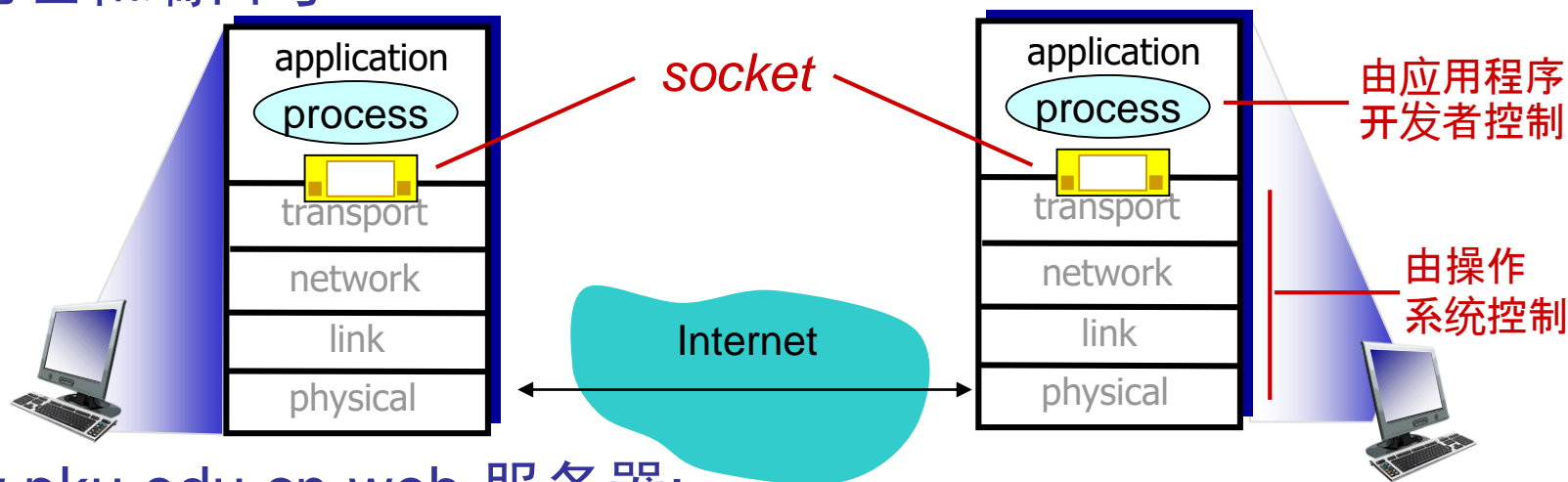


进程寻址

- 进程要接收消息，必须有标识
- 主机设备有32位的IP地址
- 用IP地址可以标识进程吗？否！因为同一主机上有很多个进程
- 标识进程用主机的IP地址和端口号

■ 例如：

- HTTP server: 80
- mail server: 25



■ 发送HTTP消息给www.pku.edu.cn web 服务器:

- IP address: 162.105.131.160
- port number: 80



应用层定义的协议

- 交换消息的类别
 - 例如, request, response
- 消息的语法
 - 消息中的字段以及字段如何划分
- 消息的语义
 - 字段的含义
- 进程何时以及如何发送和响应消息的规则

开放协议:

- RFC 定义
- 允许互操作
- 例如, HTTP, SMTP

专用协议

- 例如, Skype

应用程序所需的数据传输服务

可靠传输

- 某些应用程序(如文件传输、web服务)需要可靠数据传输
- 其他应用程序(如音频)可允许部分丢失

实时性

- 某些应用程序(如网络电话、交互式游戏)要求低延迟

吞吐量

- 某些应用程序(如多媒体)需要保证最低数据率, 为带宽敏感的业务
- 其他应用程序可以使用任何数据率

安全性

- 保密、数据可靠性

application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100' s msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100' s msec
text messaging	no loss	elastic	yes and no



互联网提供的传输服务

TCP 提供的服务

- 发送进程与接收进程之间的可靠传输
- 流量控制：发送方以适于接收方的速率能力
- 拥塞控制：网络过载时限制发送速率
- 不提供：定时，最小数据率保证，安全性
- 面向连接：客户端和服务端进程之间需要建立连接

UDP提供的服务

- 发送进程与接收进程之间的不可靠数据传输
- 不提供：可靠性、流量控制、拥塞控制、定时、数据率保证、安全性以及连接建立

为何有两种协议？为何存在UDP？



互联网：应用及其应用层协议

application		application layer protocol	underlying transport protocol
remote terminal access	e-mail	SMTP [RFC 2821]	TCP
	Web	Telnet [RFC 854]	TCP
	file transfer	HTTP [RFC 2616]	TCP
	streaming multimedia	FTP [RFC 959]	TCP
	Internet telephony	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
		SIP, RTP, proprietary (e.g., Skype)	TCP or UDP



TCP安全

TCP & UDP

- 没有加密
- 明文直接发送给socket 经互联网明文传输

SSL

- 提供加密的TCP连接
- 数据完整性
- 对通信对端的认证

SSL 是在应用层

- 应用层使用SSL与TCP会话

SSL socket API

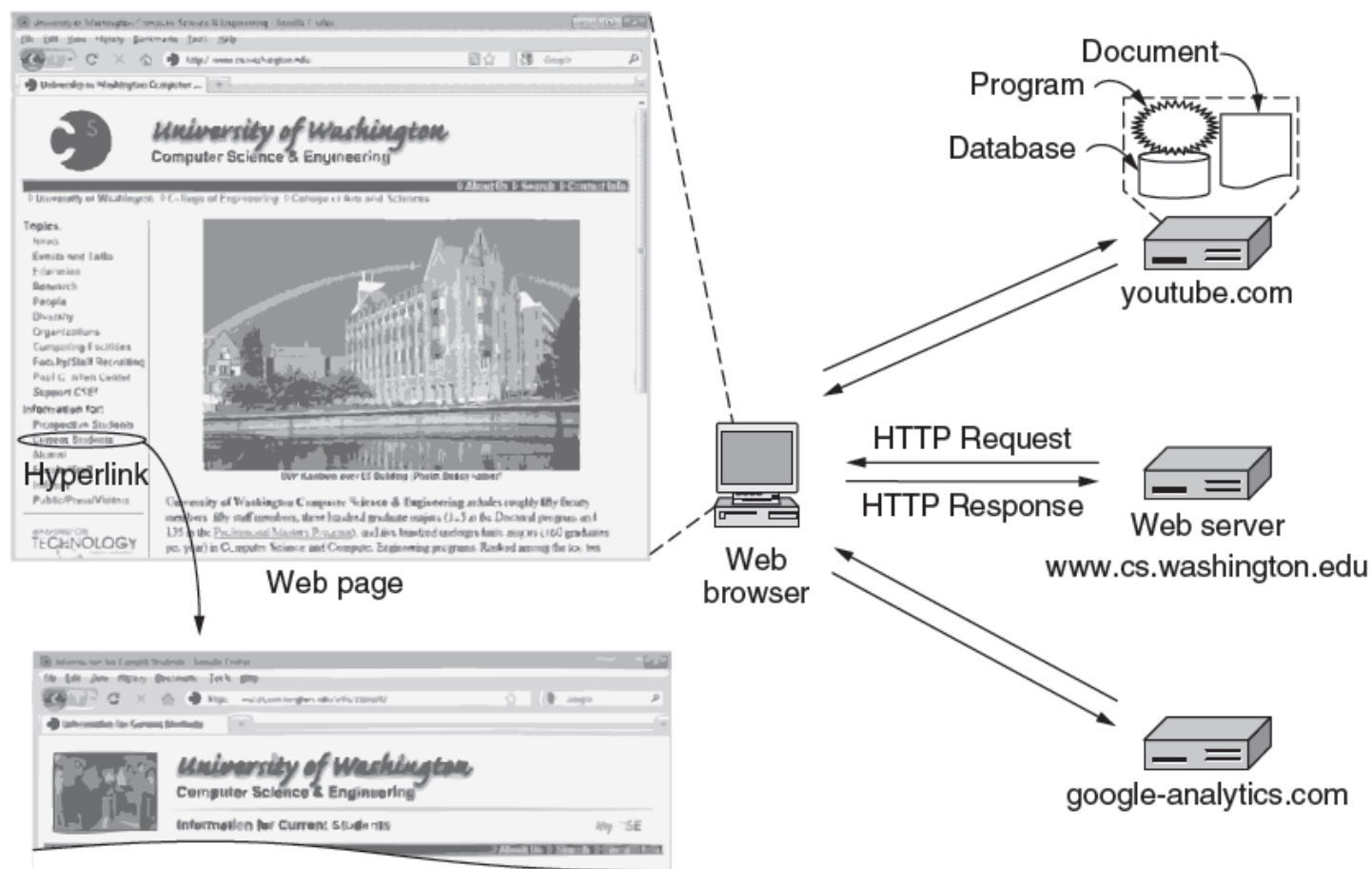
- 明文直接发送给socket 经互联网加密传输



提纲

- 网络应用层
- Web 和 HTTP
- 电子邮件：SMTP, POP3, IMAP
- DNS
- P2P 应用
- 视频流和内容分发网络CDN
- socket 编程

Web 和 HTTP



在Web服务中，为了使一个页面指向另一个，需要发送命名并定位页面，要解决的问题：

1. 该页面的名字？

URL

2. 该页面在哪里？

DNS→IP

3. 如何访问该页面？

HTTP (用TCP承载)

- Web页面用HTML描述，内含超文本及超链接，用URL（统一资源定位符）定义
- HTTP协议：浏览器与服务器之间的数据传输协议



Web 和 HTTP

- 用户选择页面中的一个链接时，客户端（浏览器）的动作：

1. 确定URL
2. 请求DNS解析IP地址，得到DNS响应
3. 建立TCP连接
4. 发送HTTP请求页面
5. 得到服务器的HTTP响应
6. 提取其他的URL
7. 显示页面/index.html
8. 若短时间没有向同一服务器发出请求，则释放TCP连接

- 服务器端的动作

1. 接受客户端的TCP连接
2. 获取页面路径，即文件名
3. 从磁盘上读取文件
4. 将文件内容发送给客户
5. 释放TCP连接



Web 和 HTTP

- **web 页面** 包括对象
- 对象可以是HTML文件、JPEG 图像、Java小程序、音频文件等
- web 页面由HTML文件组成，可以包括多个对象
- 每个对象的地址为一个 **URL**

`www.someschool.edu/someDept/pic.gif`

host name

path name

HTTP 概述

HTTP: hypertext transfer protocol

- Web应用层协议
- 客户机/服务器模式
 - 客户机：浏览器，采用HTTP发送请求，显示Web页面
 - 服务器：采用HTTP对收到的请求进行响应

使用TCP

- 客户机向服务器80端口发起TCP连接
- 服务器响应客户机的TCP连接请求
- 在浏览器与服务器之间交换HTTP消息
- 关闭TCP连接

HTTP 是“无状态的”

- 服务器不维护客户机的请求信息
 - 维护客户机状态太复杂了，因客户机数量多，宕机后还要恢复





HTTP：持续连接与非持续连接

HTTP非持续连接

- 经过TCP连接发送数据后，立即关闭连接
- 下载多个对象时需要多个TCP连接

HTTP持续连接

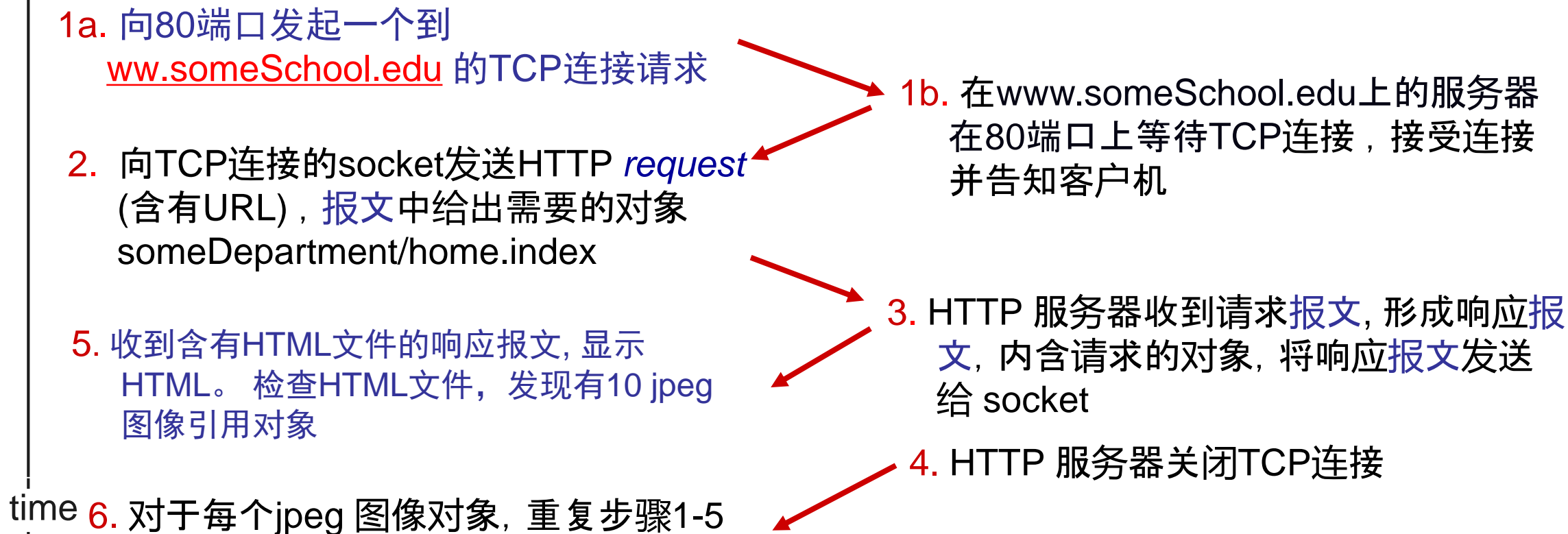
- 可以在一条客户机-服务器的TCP连接上传输多个对象
- 服务器发送响应后保持连接
- 同一对客户机服务器的后续HTTP消息经过该连接传输
- 客户机发送请求后还可以继续发送新的请求

非持续连接的HTTP

设用户输入URL: www.someSchool.edu/someDepartment/home.index
(contains text, references to 10 jpeg images)

HTTP 客户机

HTTP 服务器

- 
- ```
sequenceDiagram
 participant Client as HTTP 客户机
 participant Server as HTTP 服务器
 Note over Client: 1a. 向80端口发起一个到
www.someSchool.edu 的TCP连接请求
 Note over Server: 1b. 在www.someSchool.edu上的服务器
在80端口上等待TCP连接，接受连接
并告知客户机
 Note over Client: 2. 向TCP连接的socket发送HTTP request
(含有URL)，报文中给出需要的对象
someDepartment/home.index
 Note over Server: 3. HTTP 服务器收到请求报文，形成响应报文，
内含请求的对象，将响应报文发送给 socket
 Note over Server: 4. HTTP 服务器关闭TCP连接
 Note over Client: 5. 收到含有HTML文件的响应报文，显示
HTML。检查HTML文件，发现有10 jpeg
图像引用对象
 Note over Client: 6. 对于每个jpeg 图像对象，重复步骤1-5
```
- 1a. 向80端口发起一个到 www.someSchool.edu 的TCP连接请求
- 1b. 在www.someSchool.edu上的服务器在80端口上等待TCP连接，接受连接并告知客户机
2. 向TCP连接的socket发送HTTP *request* (含有URL)，*报文*中给出需要的对象 someDepartment/home.index
3. HTTP 服务器收到请求*报文*，形成响应*报文*，内含请求的对象，将响应*报文*发送给 socket
4. HTTP 服务器关闭TCP连接
5. 收到含有HTML文件的响应报文，显示HTML。检查HTML文件，发现有10 jpeg 图像引用对象
6. 对于每个jpeg 图像对象，重复步骤1-5
- time ↓

# 非持续连接的HTTP：响应时间

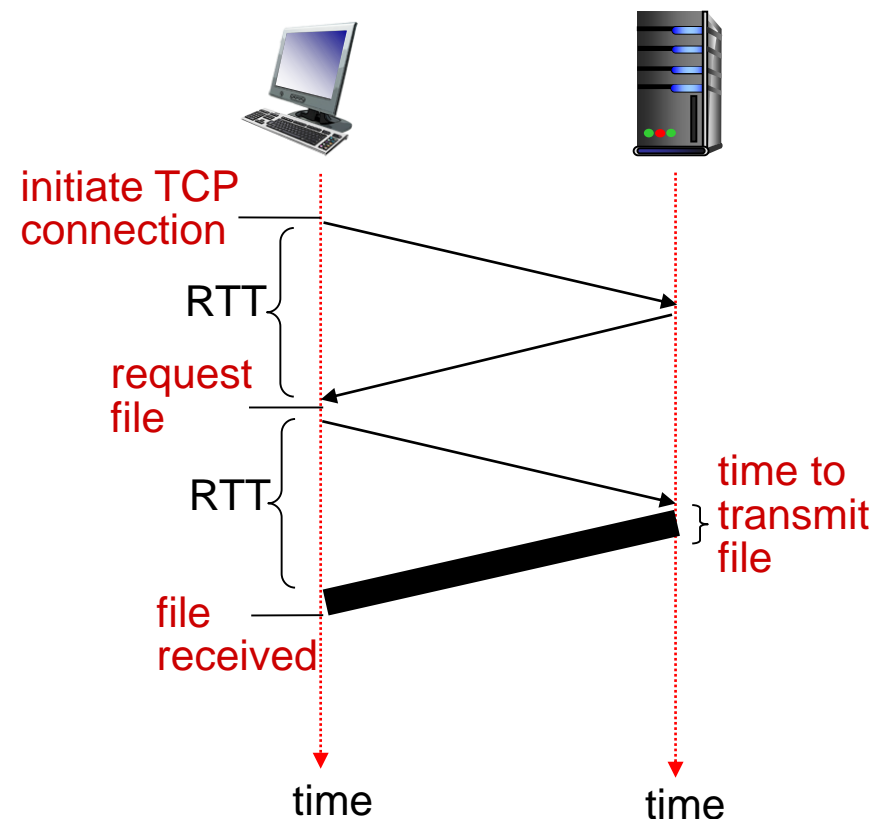
定义RTT：分组从客户机到服务器再返回客户机的时间

HTTP 响应时间：

- 1个RTT 建立TCP连接
- 1个RTT HTTP请求并返回HTTP响应
- 文件传输时间
- 响应时间 =  $2\text{RTT} + \text{文件传输时间}$

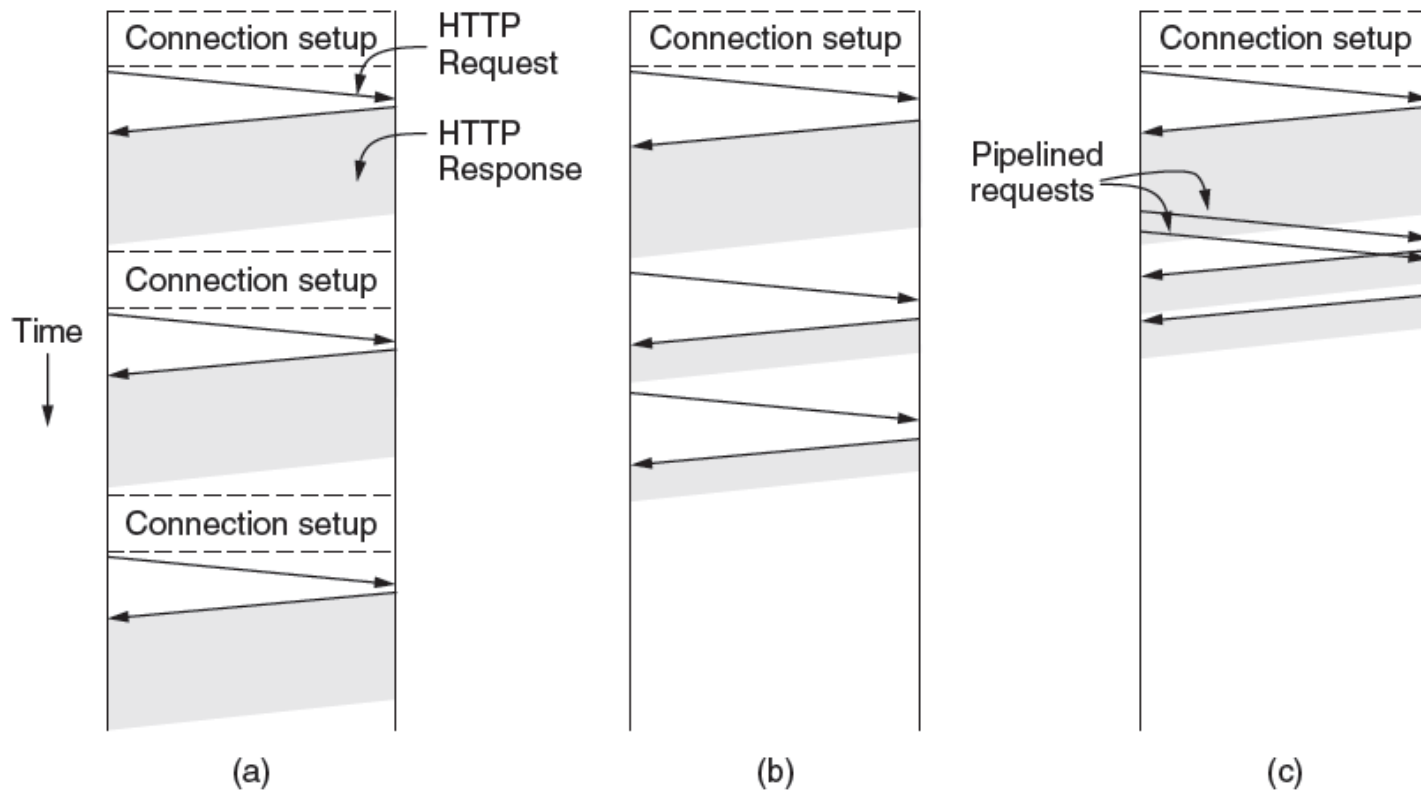
非持续连接HTTP的问题：

- 下载每个对象需要2倍RTT
- OS对每个TCP连接的开销
- 浏览器经常同时开启多个TCP连接获取HTTP对象



# HTTP: 非持续连接与持续连接

- (a) 为每个请求都建立1个连接，多个请求需要多个连接：HTTP1.0
- (b) 在一个持续连接上发送一系列的请求：HTTP1.1 连接重用，减少连接建立时间，TCP 传输更快，没有TCP慢启动过程；一个HTTP响应之后再发出下一个请求
- (c) 一个持续连接和流水线式请求：HTTP1.1 连接重用，可发送多个请求



# HTTP报文格式

- 两类HTTP报文: *request, response*
- HTTP 请求报文:
  - ASCII (human-readable format)

请求行

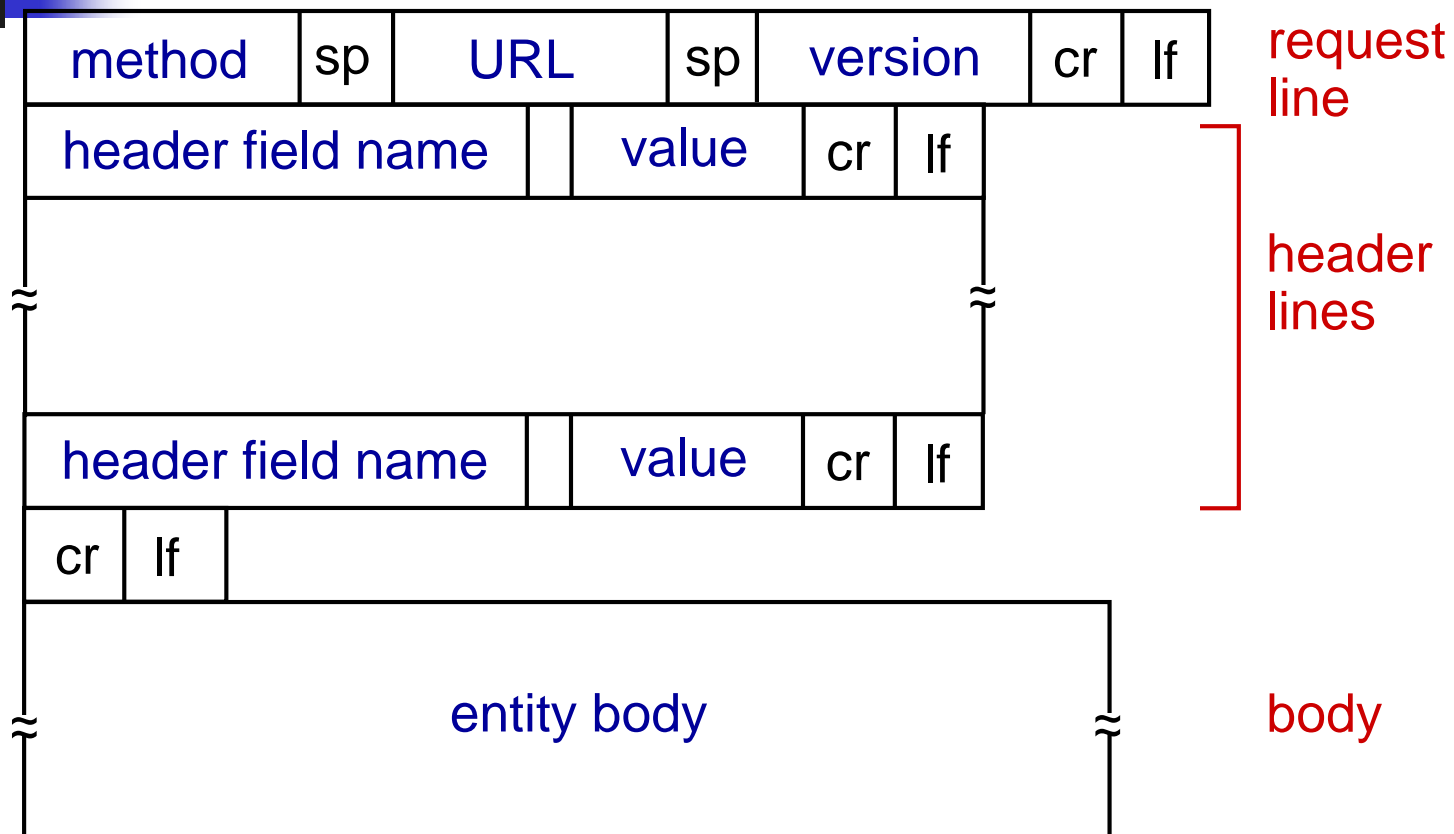
(GET, POST,  
HEAD命令)

标题行

行首的回车  
换行表示标  
题行的结束

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\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\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

# HTTP报文格式：请求报文



## 常见方法

| Method  | Description               |
|---------|---------------------------|
| GET     | Read a Web page           |
| HEAD    | Read a Web page's header  |
| POST    | Append to a Web page      |
| PUT     | Store a Web page          |
| DELETE  | Remove the Web page       |
| TRACE   | Echo the incoming request |
| CONNECT | Connect through a proxy   |
| OPTIONS | Query options for a page  |



# HTTP报文格式: header field name

| Header            | Type     | Contents                                             |
|-------------------|----------|------------------------------------------------------|
| User-Agent        | Request  | Information about the browser and its platform       |
| Accept            | Request  | The type of pages the client can handle              |
| Accept-Charset    | Request  | The character sets that are acceptable to the client |
| Accept-Encoding   | Request  | The page encodings the client can handle             |
| Accept-Language   | Request  | The natural languages the client can handle          |
| If-Modified-Since | Request  | Time and date to check freshness                     |
| If-None-Match     | Request  | Previously sent tags to check freshness              |
| Host              | Request  | The server's DNS name                                |
| Authorization     | Request  | A list of the client's credentials                   |
| Referer           | Request  | The previous URL from which the request came         |
| Cookie            | Request  | Previously set cookie sent back to the server        |
| Set-Cookie        | Response | Cookie for the client to store                       |
| Server            | Response | Information about the server                         |





# HTTP报文格式: header field name

|                  |          |                                                      |
|------------------|----------|------------------------------------------------------|
| Content-Encoding | Response | How the content is encoded (e.g., <i>gzip</i> )      |
| Content-Language | Response | The natural language used in the page                |
| Content-Length   | Response | The page's length in bytes                           |
| Content-Type     | Response | The page's MIME type                                 |
| Content-Range    | Response | Identifies a portion of the page's content           |
| Last-Modified    | Response | Time and date the page was last changed              |
| Expires          | Response | Time and date when the page stops being valid        |
| Location         | Response | Tells the client where to send its request           |
| Accept-Ranges    | Response | Indicates the server will accept byte range requests |
| Date             | Both     | Date and time the message was sent                   |
| Range            | Both     | Identifies a portion of a page                       |
| Cache-Control    | Both     | Directives for how to treat caches                   |
| ETag             | Both     | Tag for the contents of the page                     |
| Upgrade          | Both     | The protocol the sender wants to switch to           |

# HTTP报文格式： 响应报文

## 状态码

| Code | Meaning      | Examples                                           |
|------|--------------|----------------------------------------------------|
| 1xx  | Information  | 100 = server agrees to handle client's request     |
| 2xx  | Success      | 200 = request succeeded; 204 = no content present  |
| 3xx  | Redirection  | 301 = page moved; 304 = cached page still valid    |
| 4xx  | Client error | 403 = forbidden page; 404 = page not found         |
| 5xx  | Server error | 500 = internal server error; 503 = try again later |

状态行  
(协议及状态码)

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
```

标题行

data, e.g.,  
requested  
HTML file

```
data data data data data ...
```



# HTTP报文格式： 响应报文中的状态码

- 在服务器发给客户机的响应消息的第一行中给出状态码
- 常见的状态码

## 200 OK

- request succeeded, requested object later in this msg

## 301 Moved Permanently

- requested object moved, new location specified later in this msg (Location:)

## 400 Bad Request

- request msg not understood by server

## 404 Not Found

- requested document not found on this server

## 505 HTTP Version Not Supported



# 用户与服务器的交互：cookies

HTTP是面向非连接的服务，请求、响应，如何实现持续连接HTTP？ cookies

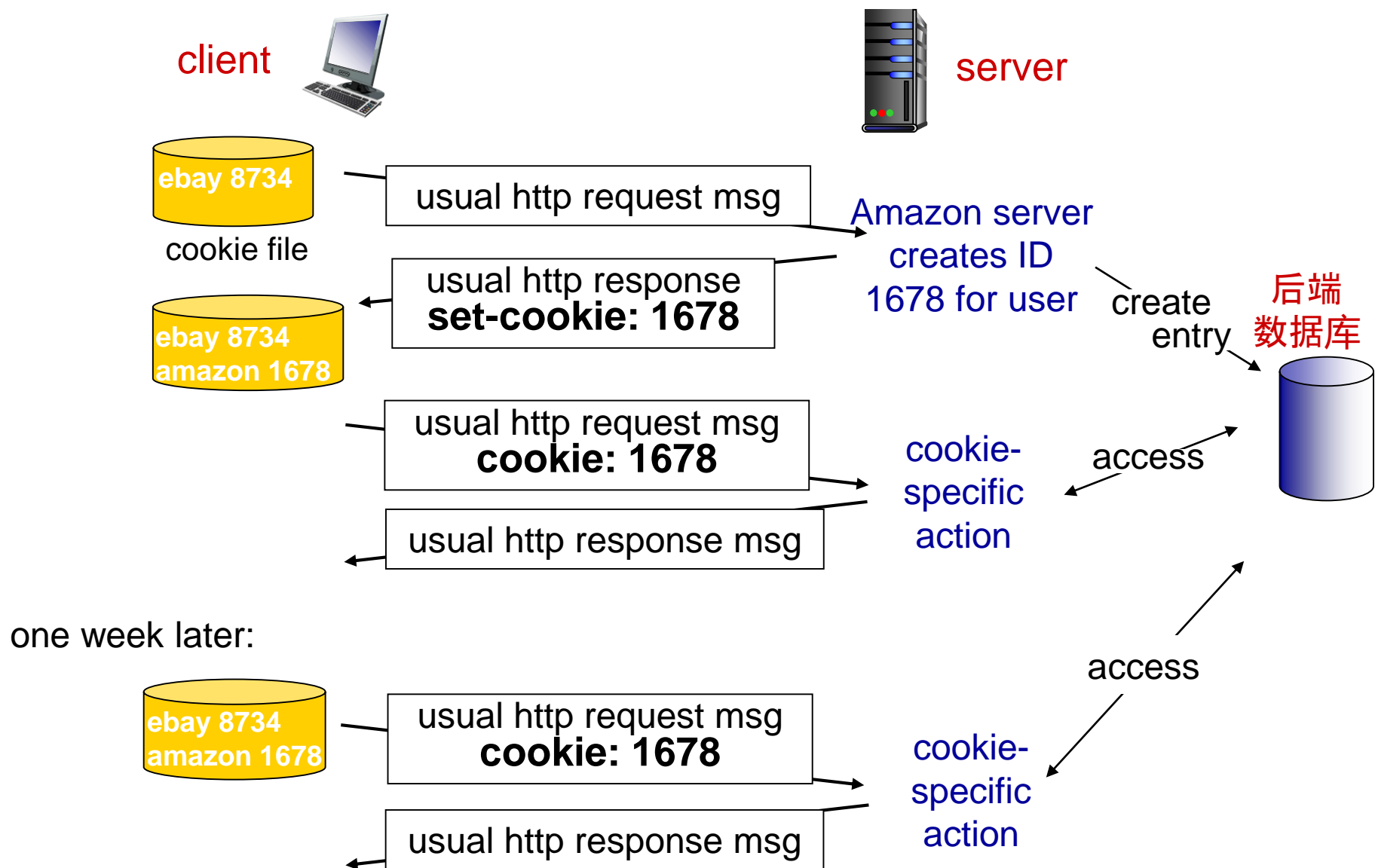
与Cookie技术相关的四个组件

- 1) HTTP响应报文中的cookie首部
- 2) 在下一个HTTP请求报文中的cookie首部
- 3) 保存在用户主机上的cookie文件，由用户浏览器读取
- 4) 保存在Web端的后端数据库中

例如：

- Susan 总是从PC端访问互联网
- 首次访问指定的电子商务网站e-commerce site
- 当初始的HTTP请求到达Web服务器后，Web服务器产生：
  - 唯一的ID识别码
  - 该ID对应于后端数据库的表项

# 用户与服务器的交互: cookies





# 用户与服务器的交互: cookies

## cookies 可用于

- 用户认证
- 购物计费
- 产品推荐
- 记录用户会话状态 (Web e-mail)

## 如何保持“状态”：

- HTTP的C/S端点：维护发送方/接收方的状态
- Cookie：http消息承载状态

此外

### Cookie和隐私：

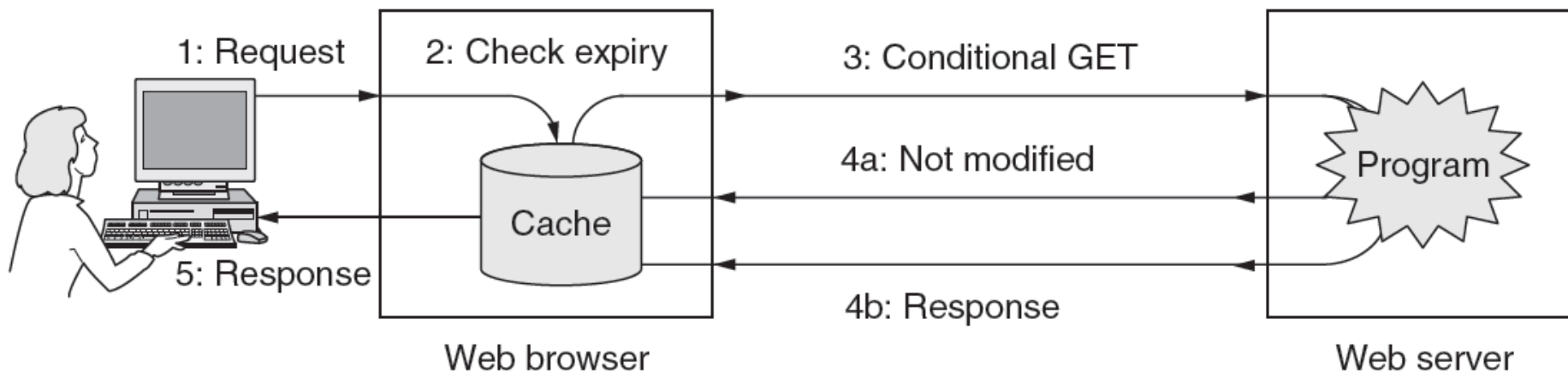
- Cookie允许网站了解用户的很多信息
- 用户可以向站点提供名称和电子邮件

# Web 页面缓存

Web缓存：保留已获取的网页供日后处理，避免重复传输，减少网络流量和延迟。

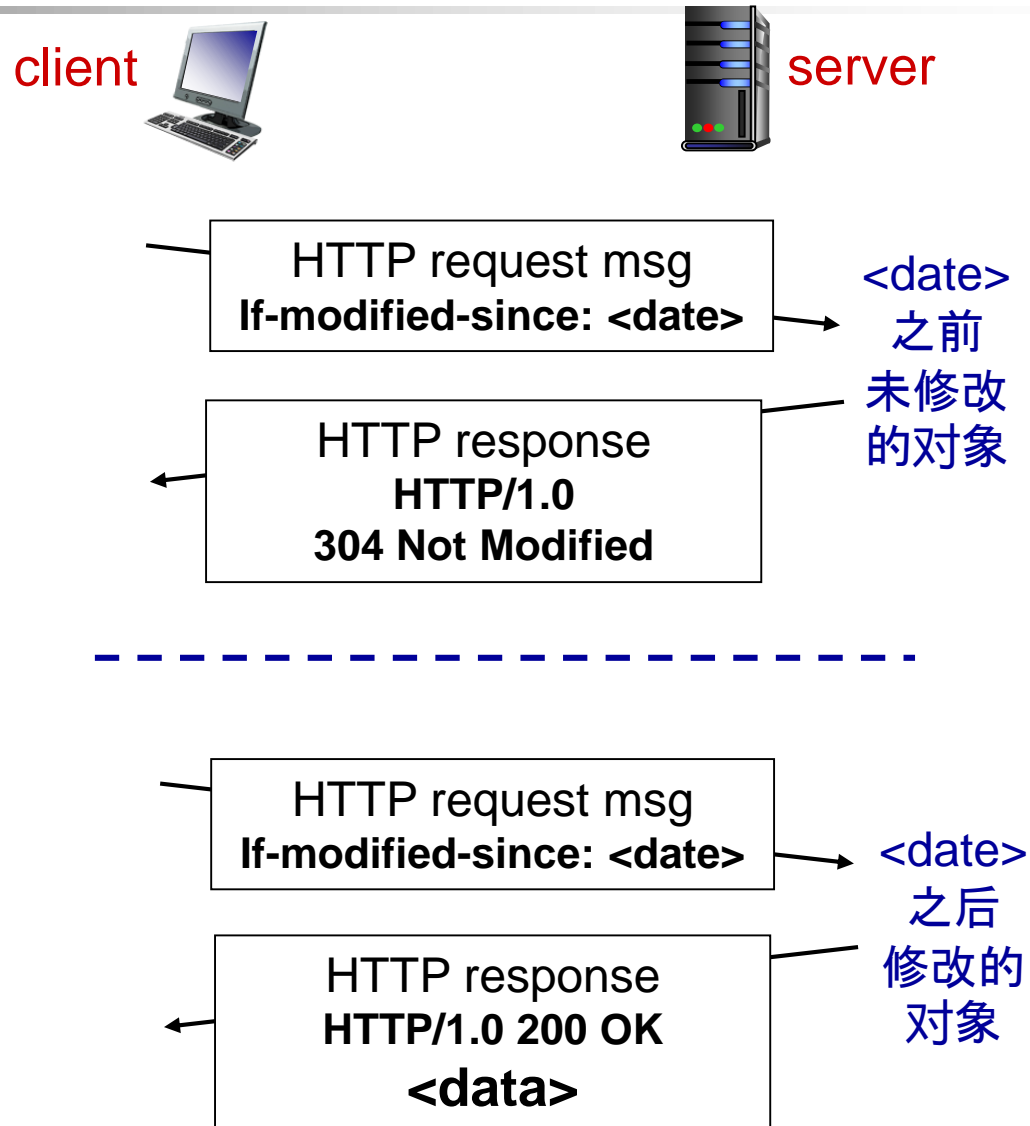
问题：如何确定旧的页面与新的内容相同？

- 1) 利用GET消息头的`expiry`判断是否过期，浏览器根据此前页面的更新间隔预测
- 2) 询问服务器缓存的副本是否过期



# 条件GET方法

- **目的：**如果缓存中有最新的缓存内容，则不发送对象
  - 没有对象传输延迟
  - 降低链路传输数据量
- **缓存：**在HTTP请求中指定缓存副本的日期  
`If-modified-since: <date>`
- **服务器：**如果缓存的副本是最新的，则在响应中不包含对象：  
`HTTP/1.0 304 Not Modified`

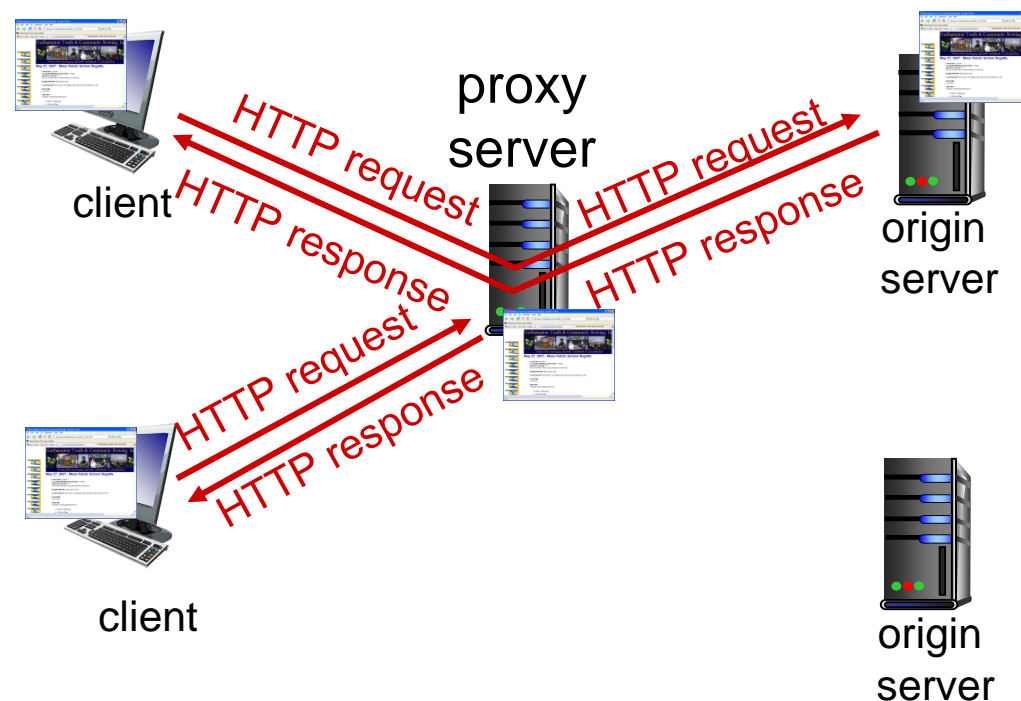




# Web 缓存 (代理服务器)

代理服务器: 代替源服务器满足用户请求

- 用户配置浏览器: Web请求到 Web缓存
- 浏览器发送HTTP请求到Web缓存
- 若对象在缓存中, 则缓存返回对象
- 否则, 缓存向源服务器请求对象, 并返回对象给客户机





# Web 缓存 (代理服务器)

---

- 缓存功能类似客户机也类似服务器
  - 对于初始请求，是服务器
  - 对于初始服务器，是客户机
- 缓存由ISP安装

## 为何采用缓存？

- 降低客户机请求的响应时间
- 降低访问链路的数据量
- 大量的互联网缓存：使内容ISP能够有效地提供内容

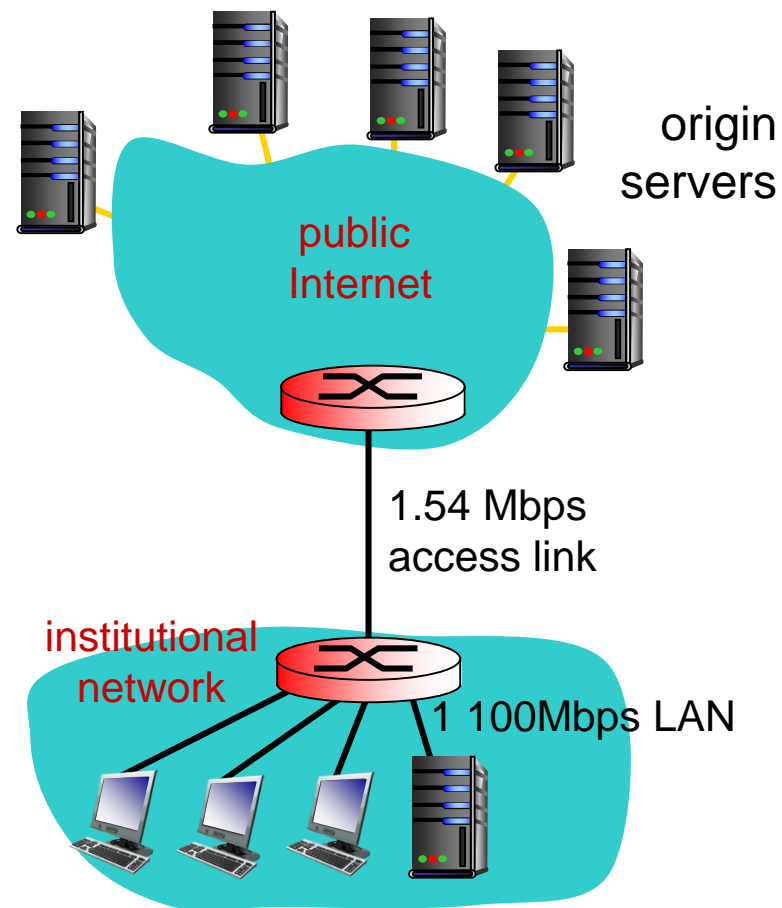
# Web缓存示例

## 假设

- 网页上的对象平均大小: 100Kb
- 客户机向服务器的平均请求频率: 15/sec
- 浏览器的平均数据率: 1.50 Mbps
- RTT: 2 sec
- 接入网链路速率: 1.54 Mbps

## 结果

- LAN 利用率: 15%
- 接入链路的利用率 = **97%** *problem!*
  - $1.50 / 1.54 = 0.97$
- 总时延 = 互联网时延 + 接入网时延 + LAN时延  
= 2 sec + minutes + u secs



# Web缓存示例: 增大接入网带宽

## 假设

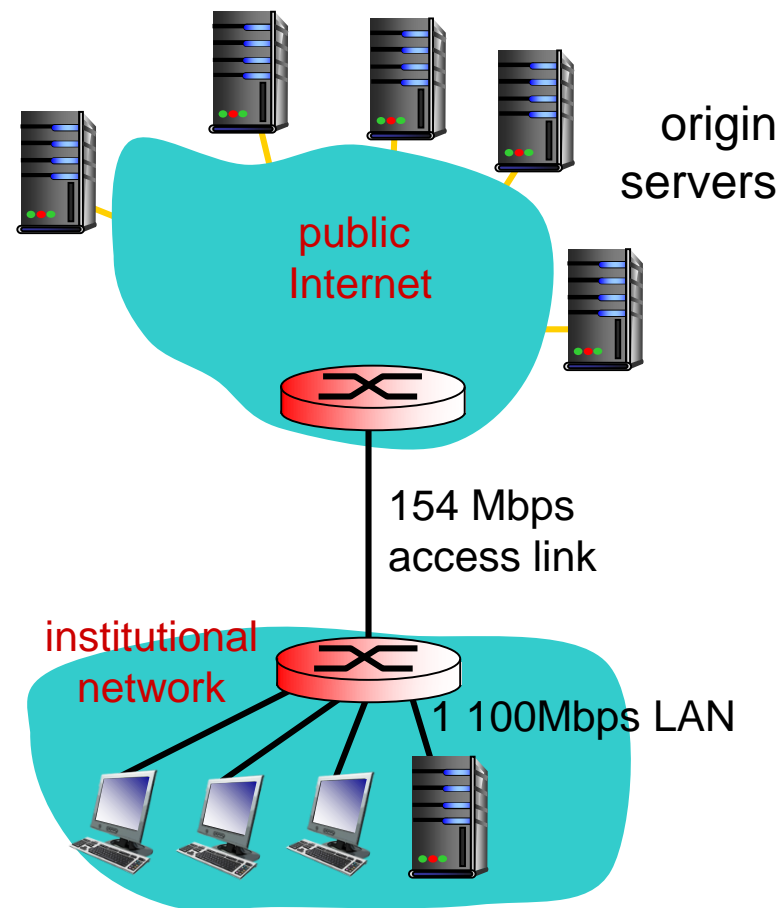
- 网页上的对象平均大小: 100Kb
- 客户机向服务器的平均请求频率: 15/sec
- 浏览器的平均数据率: 1.50 Mbps
- RTT: 2 sec
- 接入网链路速率: 1.54 Mbps

154 Mbps

## 结果

- LAN 利用率: 15%
- 接入链路的利用率 = 97% → 0.97%
- 总时延 = 互联网时延 + 接入网时延 + LAN时延  
= 2 sec + minutes + u secs  
→ m secs

增大接入网带宽, 费用增大



# Web缓存示例: 安装本地缓存

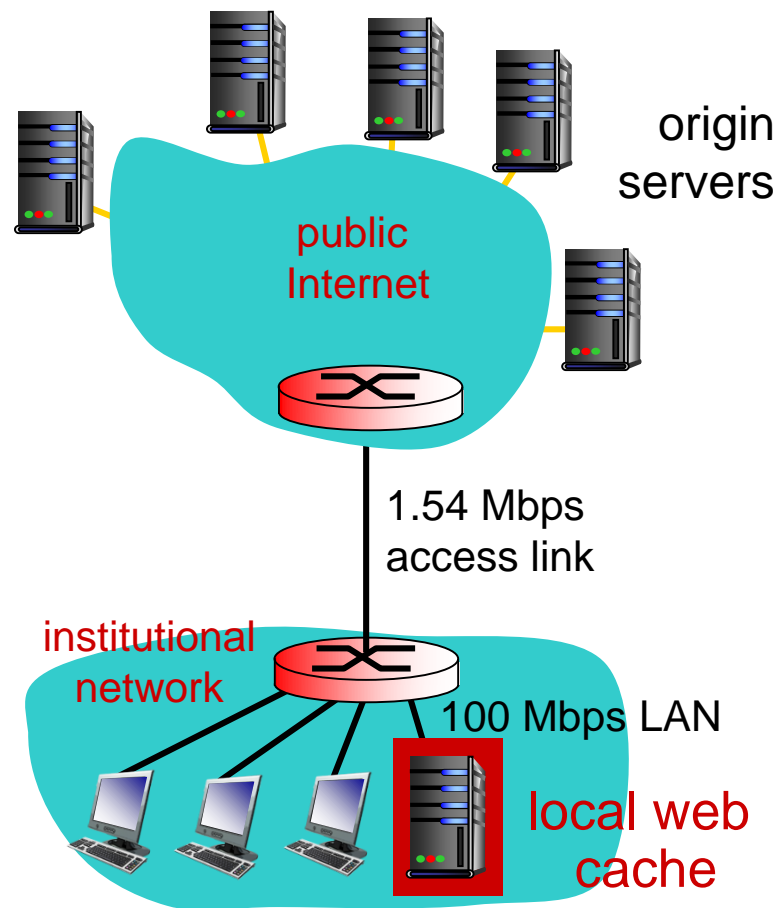
## 假设

- 网页上的对象平均大小: 100Kb
- 客户机向服务器的平均请求频率: 15/sec
- 浏览器的平均数据率: 1.50 Mbps
- RTT: 2 sec
- 接入网链路速率: 1.54 Mbps

## 结果

- LAN 利用率: 15%
- 接入链路的利用率 = 97%
- 总时延 = 互联网时延 + 接入网时延 + LAN时延  
= 2 sec + minutes + u secs
- 如何计算链路利用率和延迟?

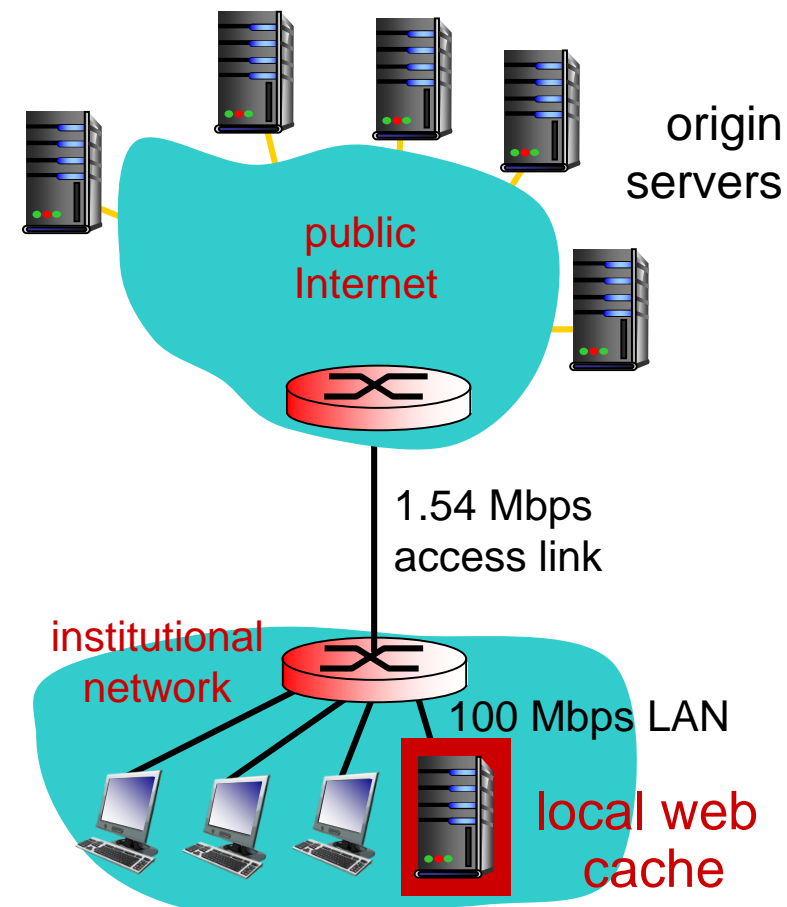
性价比: web 缓存 (廉价!)



# Web缓存示例: 安装本地缓存

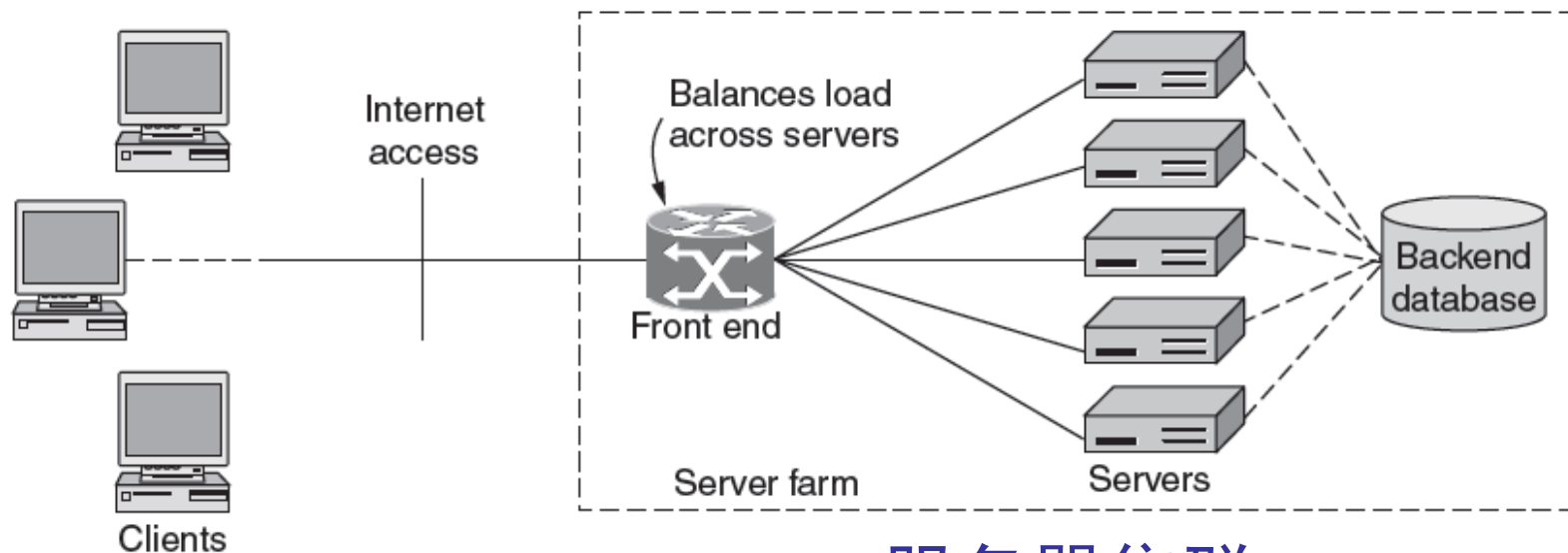
## 计算有缓存时接入网链路利用率和延迟:

- 假设缓存命中率为0.4
  - 40% 的请求在缓存中, 60% 请求在原服务器上
- 接入网链路利用率
  - 60% 的请求经过接入网
- 经过接入网的数据率为
  - $= 1.5 * 0.6 \text{ Mbps} = 0.9 \text{ Mbps}$
  - 利用率为  $= 0.9 / 1.54 = 0.58$
- 平均延迟
  - $= 0.6 * (\text{来自原服务器的延迟}) + 0.4 * (\text{缓存中的延迟})$
  - $= 0.6 (2.01) + 0.4 (\sim m \text{ secs}) = \sim 1.2 \text{ secs}$
  - 低于154 Mbps 链路 (且**成本低!**)



# 大型Web服务

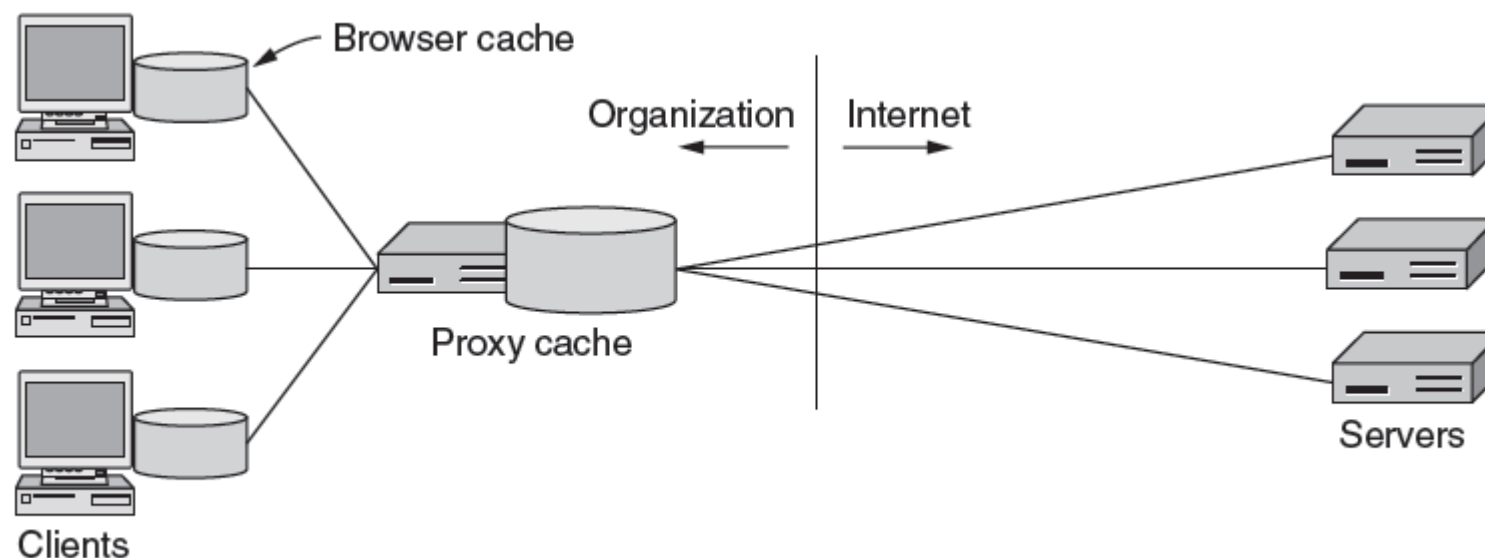
- 提供良好性能，需要加快服务器或客户机的处理速度，主要技术：
- 1) 客户机采用缓存技术：页面缓存，降低流量
- 2) 服务器采用服务器集群，并页面缓存（降低服务器的计算开销）
  - 多台Web服务器与公共后端数据库连接，任意一台服务器所获取的数据是一致的，可以避免单台服务器成为通信瓶颈
- 3) 前端将入境请求分发到服务器缓存中：由路由器中窥探IP、TCP及HTTP头部数据（如Web Cookie），确保将同一TCP连接转发给同一台服务器



服务器集群

# 大型Web服务

- 4) 减少对服务器的访问频度：在Web代理上共享缓存（页面的二级缓存），例如为某一组织设置一个代理，减少访问服务器的流量以及响应时间
- 5) 使用DNS重定向，实现各服务器之间的负载均衡



Web浏览器与服务器之间的代理缓存





# 提纲

---

- 网络应用层
- Web 和 HTTP
- 电子邮件：SMTP, POP3, IMAP
- DNS
- P2P 应用
- 视频流和内容分发网络CDN
- socket 编程

# 电子邮件

三大组件： 用户代理、邮件服务器、  
简单邮件传输协议SMTP

## 用户代理

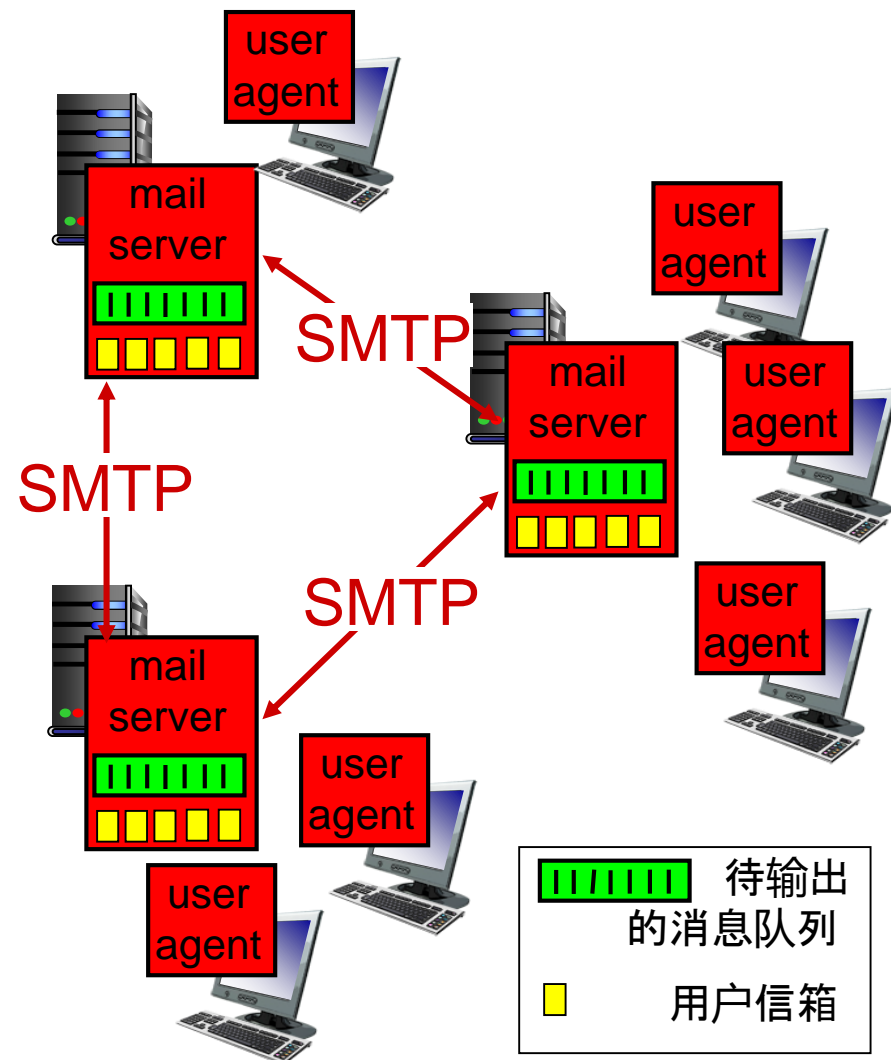
- 功能：邮件编辑、读邮件，如Outlook, iPhone mail client
- 与邮件服务器交换存储的消息

## 邮件服务器

- 邮箱：保存用户消息
- 消息队列：邮件队列

SMTP：邮件服务器之间发送邮件

- 客户机：发送邮件的服务器
- 服务器：接收邮件的服务器





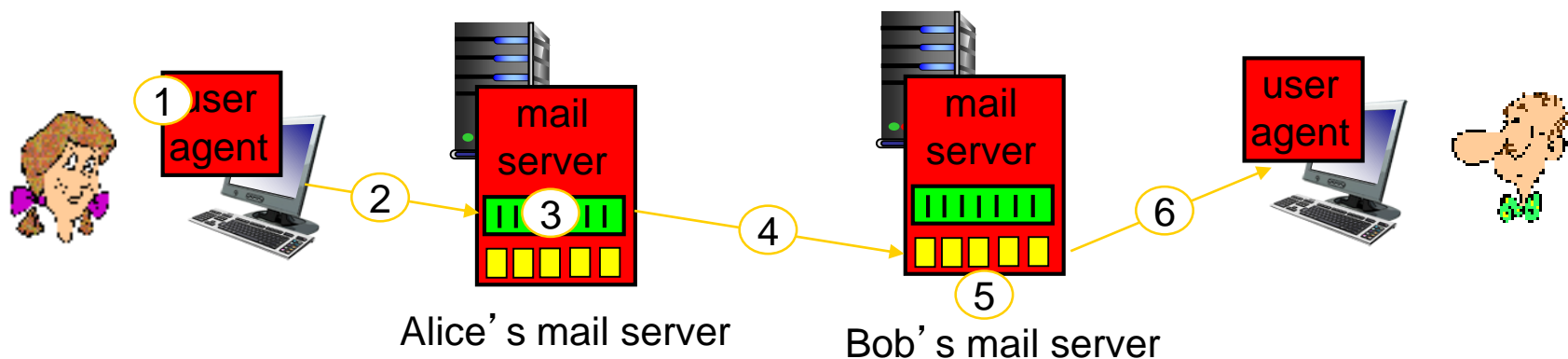
# 电子邮件: SMTP [RFC 2821]

---

- 使用TCP25端口将电子邮件从客户端可靠地传输到服务器
- 直接传输: 从发送邮件服务器到接收邮件服务器
- 转移的三个阶段:
  - 握手 (问候)
  - 信息传递
  - 关闭
- 命令/响应交互 (类似于HTTP)
  - 命令: ASCII文本
  - 回复: 状态码和短语
- 消息必须为7位ASCII

# 场景: Alice 给 Bob 发送消息

- 1) Alice使用UA编写消息 “to bob@some school.edu
- 2) Alice的UA将消息发送到她的邮件服务器；消息存储消息队列中
- 3) SMTP的客户端打开其与Bob邮件服务器的TCP连接
- 4) SMTP客户端通过TCP连接发送Alice的消息
- 5) Bob的邮件服务器将邮件放在Bob的邮箱中
- 6) Bob调用他的UA来读取消息





# SMTP交互示例

---

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C:
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```



# SMTP 与 HTTP 对比

---

- SMTP使用持久连接
- SMTP要求邮件（标头和正文）为7位ASCII格式
- SMTP服务器使用CRLF，CRLF确定消息结尾
- HTTP：获取
- SMTP：推送
- 两者都有ASCII命令/响应交互、状态码
- HTTP：每个对象都封装在其响应消息中
- SMTP：在多个邮件中发送多个对象



# 请完善邮件客户程序

- SMTPClient.py
- 功能：向接收方发送电子邮件。
- 创建与邮件服务器的TCP连接，采用SMTP与邮件服务器交互；经该邮件服务器向某接收方发送电子邮件，最后关闭TCP连接。
- 程序输出见右侧。
- 请完善该程序。

S:220 pku.edu.cn Anti-spam GT for Coremail System (mispb-1ea67e80-64e4-49d5-bd9f-3beae24b9f2-pku.edu.cn[20200728])

S1:250 OK

S2:250 Mail OK

S3:550 relaying denied

250 reply not received from server.

S4:503 bad sequence of commands

354 reply not received from server.

S5:502 Error: command not implemented

250 reply not received from server.

S6:502 Error: command not implemented

221 reply not received from server.

# 邮件报文格式

SMTP：交换电子邮件的协议

RFC 822：文本消息格式标准

标题行，例如：

收件人：

发件人：

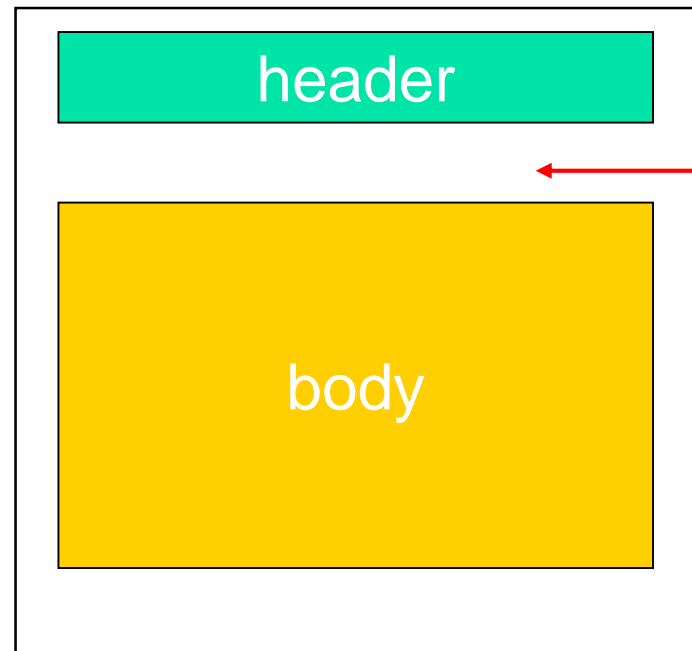
主题

与SMTP的 MAIL from, RCPT TO:命令不同！

正文：“信息” 仅限ASCII字符

标题行

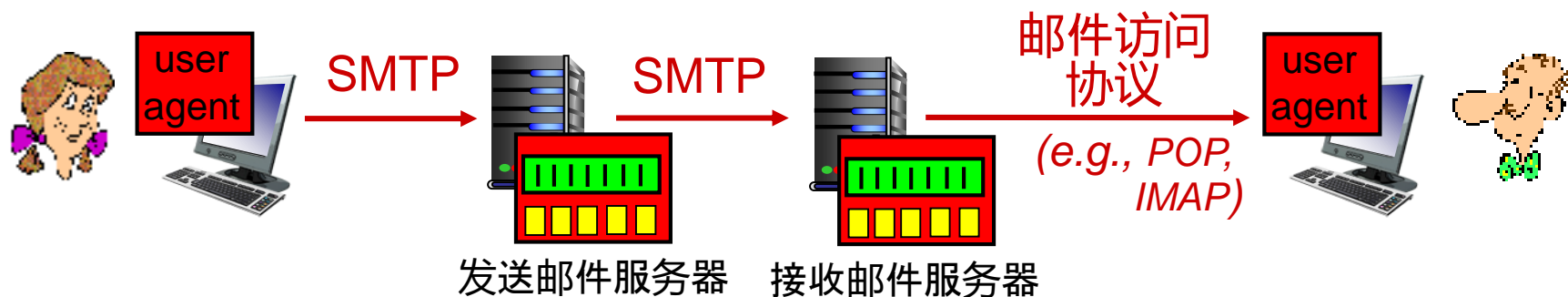
正文



空行



# 邮件访问协议



- **SMTP**: 传送/储存邮件到收方服务器
- **邮件访问协议**: 从服务器检索邮件
  - **POP**: Post Office Protocol [RFC 1939]: 认证, 下载
  - **IMAP**: Internet Mail Access Protocol [RFC 1730]: 更多功能, 包括操作服务器上存储的消息
  - **HTTP**: gmail, Hotmail, Yahoo! Mail, etc.

# POP3 协议

## 认证阶段

- 客户机命令:
  - **user**: 声明用户名
  - **pass**: 口令
- 服务器响应
  - **+OK**
  - **-ERR**

## 传输阶段, 客户机:

- **list**: 列出消息编号
- **retr**: 按号码检索邮件
- **dele**: 删除
- **Quit**

示例为使用POP3“下载并删除”模式  
POP3在会话之间是无状态的

认证  
阶段

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on
```

传输  
阶段

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

# 用Foxmail发送邮件的过程

- 用Wireshark捕获数据，使用SMTP显示过滤

Info

```
S: 220 pku.edu.cn Anti-spam GT for Coremail System (mispb-1ea67e80-64e4-49d5-bd9f-3beae24b9f2-pku.edu.cn[20200728])
C: EHLO DESKTOP-3AIM6J9
S: 250-mail | PIPELINING | AUTH LOGIN PLAIN | AUTH=LOGIN PLAIN | coremail 1Uxr2xKj7kG0xkI17xGrU7I0s8FY2U3Uj8Cz28x1UUU
C: AUTH LOGIN
S: 334 dXNlcm5hbWU6
C: User: bG11em1AcGt1LmVkdS5jbG==
S: 334 UGFzc3dvcmQ6
C: Pass: bHptOTMxMTA0
S: 235 Authentication successful
C: MAIL FROM: <liuzm@pku.edu.cn>
S: 250 Mail OK
C: RCPT TO: <liuzm@pku.edu.cn>
S: 250 Mail OK
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: DATA fragment, 436 bytes
C: DATA fragment, 1460 bytes
C: DATA fragment, 278 bytes
from: "liuzm@pku.edu.cn" <liuzm@pku.edu.cn>, subject: test, (text/plain) (text/html)
S: 250 Mail OK queued as 54FpogB3_2Y4iJxgrYQ+Aw--.60172S2
C: QUIT
S: 221 Bye
```



# 提纲

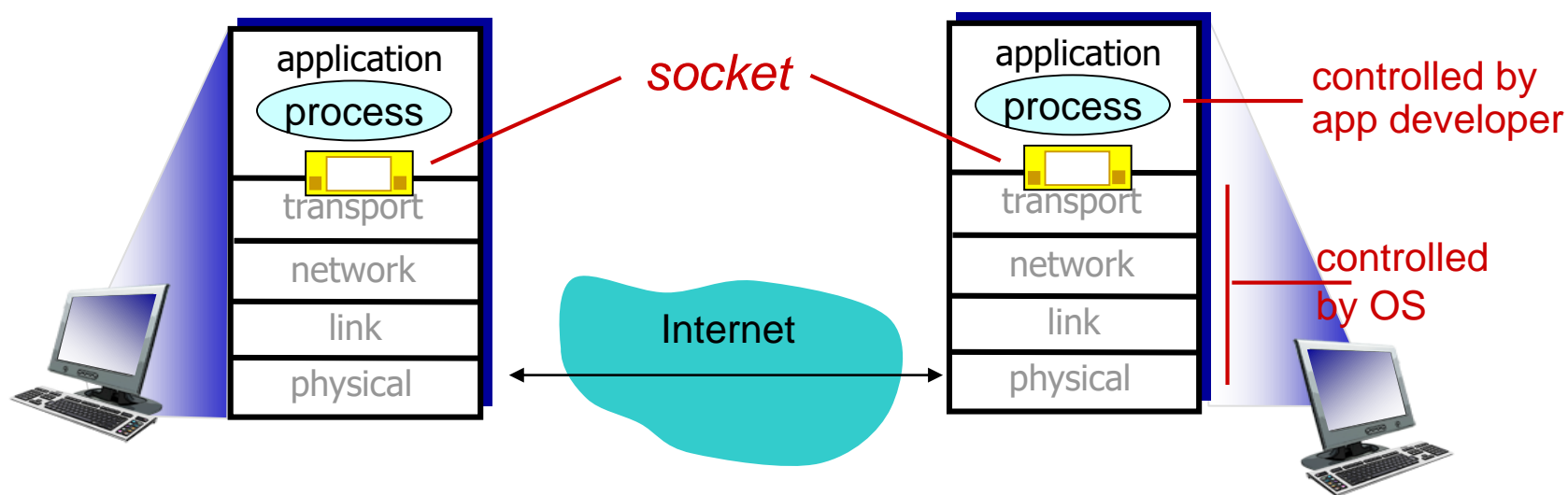
---

- 网络应用层
- Web 和 HTTP
- 电子邮件：SMTP, POP3, IMAP
- DNS
- P2P 应用
- 视频流和内容分发网络CDN
- socket 编程

# Socket 编程

目标：学习如何构建使用套接字进行通信的客户机/服务器应用程序

套接字：应用程序进程和端传输协议之间的接口





# Socket 编程

---

传输层提供的服务：两类套接字

- **UDP**：不可靠数据报
- **TCP**：可靠的面向连接的字节流传输
- **应用示例**
  1. 客户机从键盘读取一行字符(数据)，并将数据发送到服务器
  2. 服务器接收数据并将字符转换为大写
  3. 服务器将修改后的数据发送到客户端
  4. 客户端接收修改的数据并在其屏幕上显示



# Socket 编程 采用UDP

**UDP：**客户端和服务端之间没有“连接”

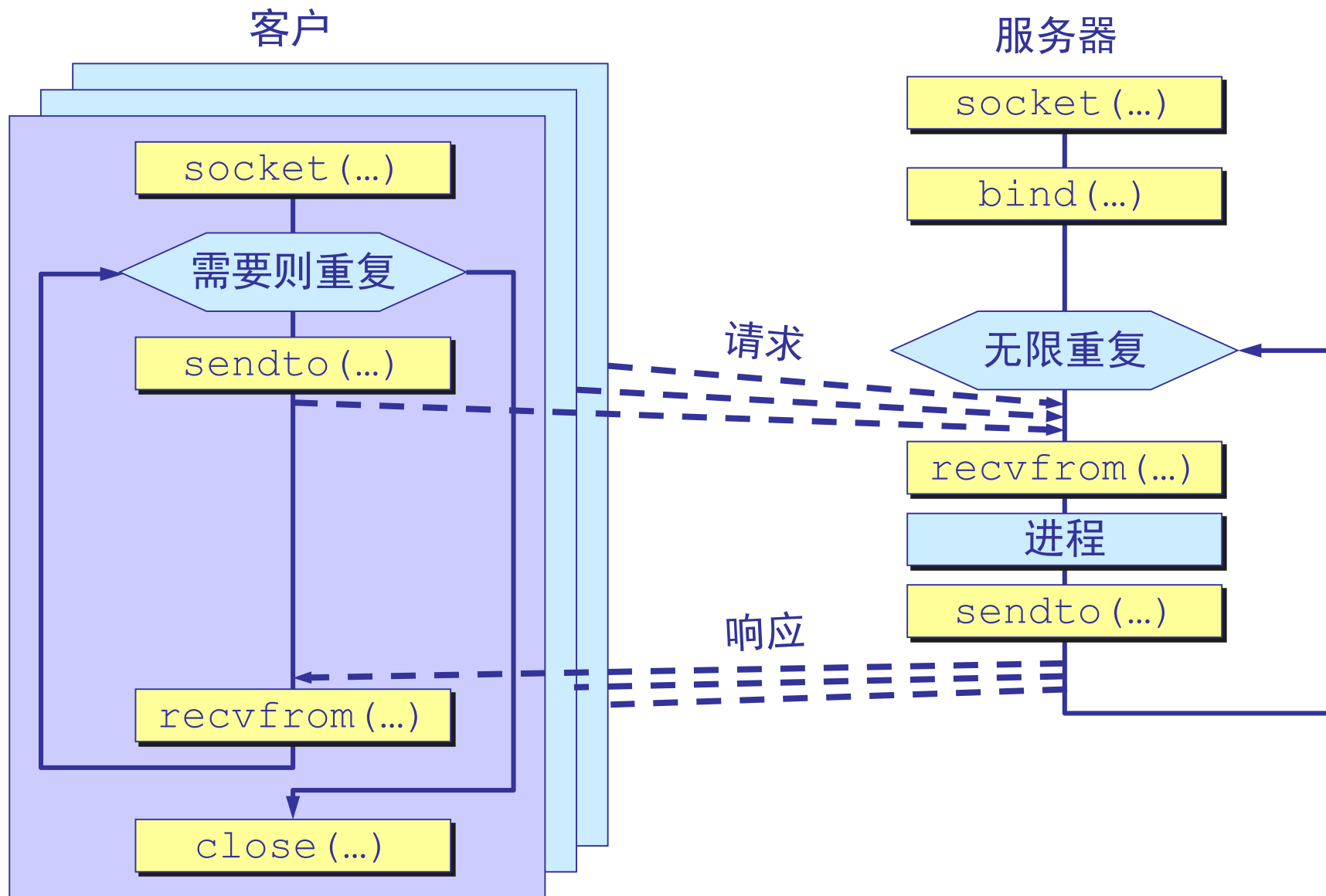
- 发送数据前不握手
- 发送方显式地将IP目标地址和端口#附加到每个数据报文上
- 接收方从收到的数据报文中提取发送方IP地址和端口#

**UDP：**传输的数据可能丢失或无序

应用角度

- UDP在客户端和服务端之间提供不可靠的数据传输

# 用套接字实现进程间通信：无连接服务





# 客户端/服务器套接字交互：UDP

## server (running on *serverIP*)

create socket, port= x:

*serverSocket* =  
socket(AF\_INET,SOCK\_DGRAM)

↓  
read datagram from  
*serverSocket*

↓  
write reply to  
*serverSocket*  
specifying  
client address,  
port number

## client

create socket:

*clientSocket* =  
socket(AF\_INET,SOCK\_DGRAM)

↓  
Create datagram with server IP and  
port=x; send datagram via  
*clientSocket*

↓  
read datagram from  
*clientSocket*

↓  
close  
*clientSocket*





# 示例应用程序：UDP客户端

## *Python UDP Client*

include Python's socket  
library

→ `from socket import *`

`serverName = 'localhost'`

`serverPort = 12000`

create UDP socket for  
server

→ `clientSocket = socket(AF_INET, SOCK_DGRAM)`

get user keyboard  
input

→ `message = input('Input lowercase sentence:')`

Attach server name, port to  
message; send into socket

→ `clientSocket.sendto(message.encode(),(serverName, serverPort))`

read reply characters from  
socket into string

→ `modifiedMessage, serverAddress = clientSocket.recvfrom(2048)`

print out received string  
and close socket

→ `print(modifiedMessage.decode())`

`clientSocket.close()`



# 示例应用程序：UDP服务器

## *Python UDPServer*

```
from socket import *
serverPort = 12000

create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 → serverSocket.bind(('', serverPort))
print ('The server is ready to receive')

loop forever → while True:
 Read from UDP socket into message, getting client's address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)
 modifiedMessage = message.decode().upper()
 send upper case string back to this client → serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```



# Socket 编程 采用TCP

## 客户端必须与服务器建立连接

- 必须首先运行服务器进程
- 服务器必须已经创建了套接字等待客户与之联系

## 客户端联系服务器的方式

- 创建TCP套接字，指定服务进程的IP地址和端口号
- 当客户端创建套接字时：客户端TCP建立到服务器TCP的连接

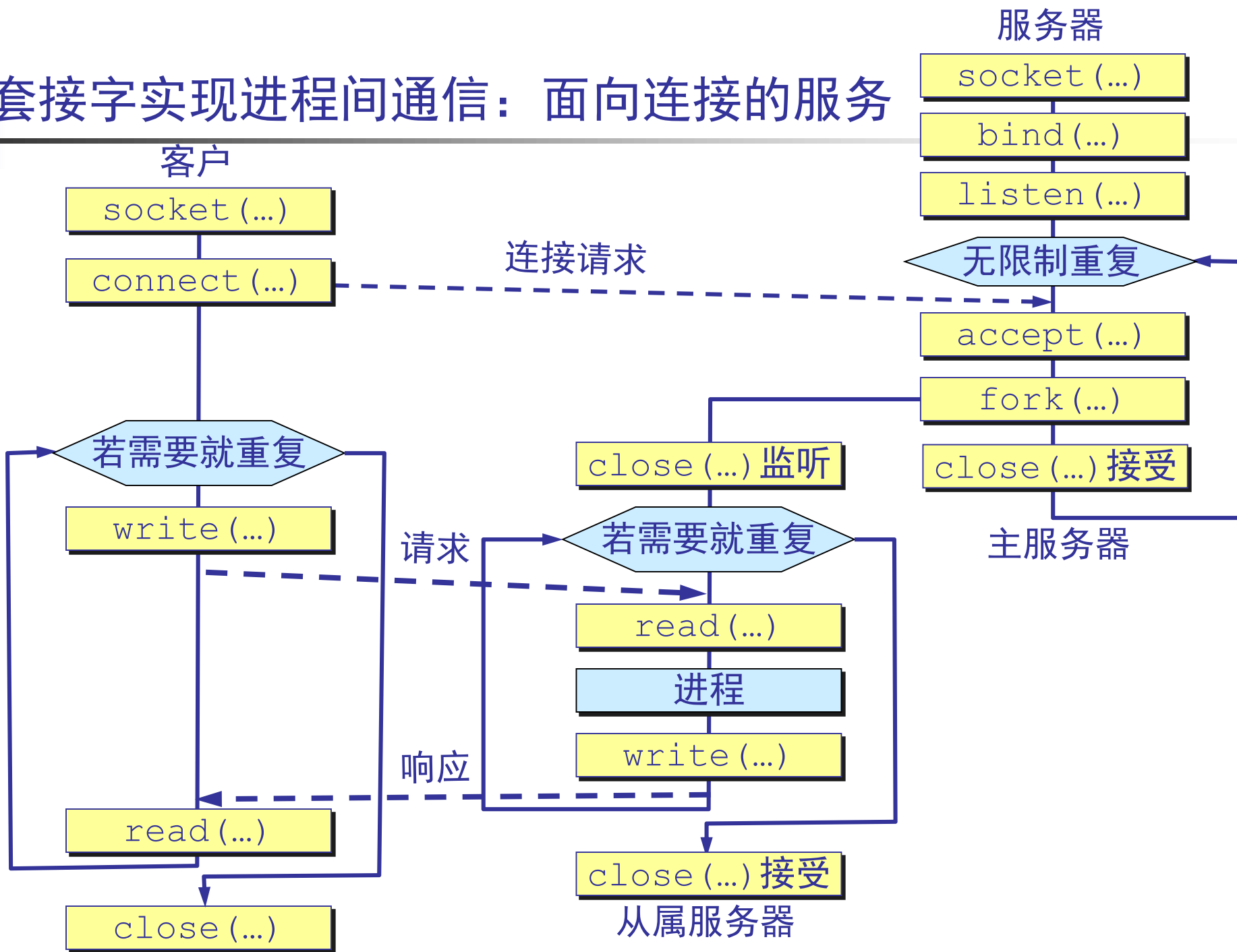
- 当客户端联系时，服务器TCP为服务器进程创建新的套接字，以便与特定的客户端通信

- 允许服务器与多个客户端对话
- 用于区分客户端的源端口号

## 应用角度

TCP在客户端和服务端之间提供可靠的字节流传输（“管道”）

# 用套接字实现进程间通信：面向连接的服务



# 客户端/服务器套接字交互： TCP

server (running on `hostid`)

client

create socket,  
port=`x`, for incoming  
request:  
`serverSocket = socket()`

wait for incoming  
connection request  
`connectionSocket =`  
`serverSocket.accept()`

read request from  
`connectionSocket`

write reply to  
`connectionSocket`

close  
`connectionSocket`

create socket,  
connect to `hostid`, port=`x`  
`clientSocket = socket()`

send request using  
`clientSocket`

read reply from  
`clientSocket`

close  
`clientSocket`

TCP  
connection setup

# 示例应用程序：TCP客户端

## *Python TCPClient*

create TCP socket for  
server, remote port 12000

```
from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

No need to attach server  
name, port

# 示例应用程序：TCP服务器

## *Python TCPServer*

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print('The server is ready to receive')

while True:
 connectionSocket, addr = serverSocket.accept()
 sentence = connectionSocket.recv(1024).decode()
 capitalizedSentence = sentence.upper()
 connectionSocket.send(capitalizedSentence.encode())
 connectionSocket.close()
```

create TCP welcoming socket →

server begins listening for incoming TCP requests →

loop forever →

server waits on accept() for incoming requests, new socket created on return →

read bytes from socket →

close connection to this client (but *not* welcoming socket) →





# 小结

---

- 应用层体系结构: C/S、P2P
- 应用层业务需求
  - 可靠性, 带宽, 延迟
- 互联网传输服务模式
  - 面向连接, 可靠: TCP
  - 不可靠, 数据报: UDP
- 特殊协议
  - HTTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent
- 视频流媒体, CDN
- socket 编程: TCP, UDP sockets
- 典型的请求/回复消息交换:
  - 客户请求信息或服务
  - 服务器用数据、状态码进行响应
- 消息格式:
  - 标题: 提供数据信息的字段
  - 数据: 正在通信的信息 (有效载荷)