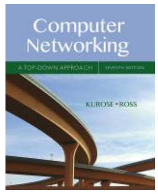
Chapter 2
应用层
Application Layer



Computer Networking: A Top-Down Approach
7th edition
Jim Kurose, Keith Ross
Pearson, 2016

应用层简介
The Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

应用层简介
The Application Layer: Overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-layer protocols
  - HTTP
  - SMTP, IMAP
  - DNS
- programming network applications
  - socket API

网络应用是计算机网路存在的理由
Some network apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix奈飞)
- P2P file sharing
- voice over IP (e.g., Skype)
- real-time video conferencing
- Internet search
- remote login
- …

Q: your favorites?

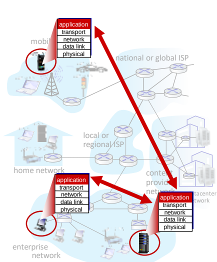
Creating a Network App

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



客户-服务器体系结构
Client-server architecture

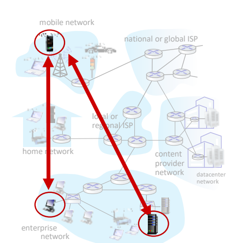
服务器 server:

- always-on host
- permanent IP address
- often in data centers, for scaling

客户clients:

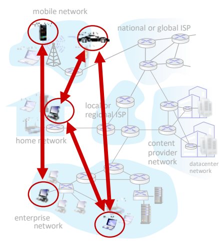
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP



对等方到对等方/P2P 体系结构
Peer-peer architecture

- *no* always-on server
- arbitrary end systems directly communicate
- Peers（对等方） request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing



进程通信
Processes Communicating

process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts communicate by exchanging messages

clients, servers

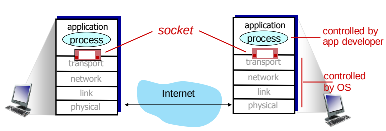
client process: process that initiates communication

server process: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes

进程与计算机网络之间的接口−套接字
Sockets

- process sends/receives messages to/from its socket
- 进程类比房子，进程的套接字类比于它的门
  - 发送进程将报文推出门（套接字）
  - 发送进程假定该门到另外一侧之间有传输基础设施，该设施将报文传送到目的进程的门口，报文通过目的进程的门（套接字）传递
  - two sockets involved: one on each side



进程寻址
Addressing Processes

- 标识 to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, *many* processes can be running on same host
- *标识包括IP地址和端口号*
  identifier includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

应用层协议定义什么？
An Application-layer Protocol Defines:

- 交换的报文类型 types of messages exchanged,
  - e.g., request, response
- 报文的语法 message syntax:
  - what fields in messages & how fields are described
- 报文的语义 message semantics
  - meaning of information in fields
- 规则 rules for when and how processes send & respond to messages

公共/开放协议
open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

专用协议
proprietary protocols:

- e.g., Skype

应用程序需要怎样的传输服务？
What transport service does an app need?

可靠数据传输
reliable data transportation

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

及时 timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

吞吐量 throughput

- 带宽敏感的应用需要运输协议确保可用吞吐量总是至少为r比特/秒
  apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- 弹性应用利用可供使用的吞吐量
  other apps ("elastic apps") make use of whatever throughput they get

安全 security

- encryption, data integrity, …

常见应用对传输服务的要求
Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	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, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

网络提供的传输服务
Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Q: why bother? *Why* is there a UDP?

流行的网络应用及其应用层协议和传输层协议
Internet transport protocols services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

TCP的安全加强
Securing TCP

TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Secure Socket Layer/ Transport Layer Security (SSL/TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

SSL/TLS implemented in application layer

- apps use TSL libraries, that use TCP in turn

SSL/TLS socket API

- cleartext passwords sent into socket traverse Internet encrypted
- see Chapter 8

The Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Web and HTTP

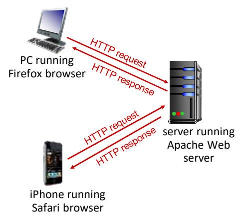
First, a quick review…

- web page consists of *objects（对象）*, each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file(HTML基本文件)* which includes *several referenced objects（引用对象）, each* addressable by a *URL,* e.g.,

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

HTTP Overview

HTTP: hypertext transfer protocol
- Web’s application layer protocol
- client/server model:
  - *client:* 浏览器（browser） that requests, receives, (using HTTP protocol) and “displays” Web objects
  - *server:* Web服务器（Web server） sends (using HTTP protocol) objects in response to requests



HTTP Overview (continued)

HTTP uses TCP:
- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is “stateless”
- server maintains *no* information about past client requests

aside

protocols that maintain “state” are complex!
- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled

两种类型的HTTP连接
HTTP Connections: two types

非持续连接
Non-persistent HTTP
1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

持续连接
Persistent HTTP
- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

Non-persistent HTTP: example

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing text, references to 10 jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80

1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80 “accepts” connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Non-persistent HTTP: example (cont.)

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing text, references to 10 jpeg images)

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

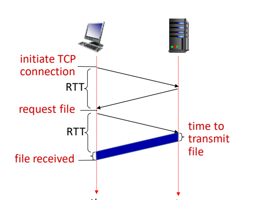
6. Steps 1-5 repeated for each of 10 jpeg objects

time

Non-persistent HTTP: response time

RTT (Round Trip Time 往返时间): time for a small packet to travel from client to server and back

HTTP response time (per object):
- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



Non-persistent HTTP response time = 2RTT+ file transmission time

Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

Persistent HTTP (HTTP1.1):

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects (cutting response time in half)

HTTP请求报文
HTTP Request Message

- two types of HTTP messages: *request, response*
- HTTP request message:
  - ASCII (human-readable format)

request line (GET, POST, HEAD commands)

header lines

carriage return character

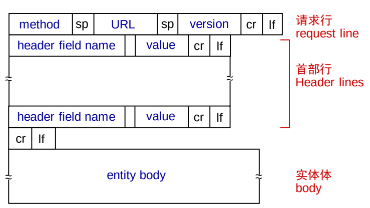
line-feed character

```
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
```

carriage return, line feed at start of line indicates end of header lines

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP request message: general format



Other HTTP Request Messages

POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

`www.somesite.com/animalsearch?monkeys&banana`

HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

HTTP Response Message

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP Response Status Codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK
- request succeeded, requested object later in this message

301 Moved Permanently
- requested object moved, new location specified later in this message (in Location: field)

400 Bad Request
- request msg not understood by server

404 Not Found
- requested document not found on this server

505 HTTP Version Not Supported

HTTP练习
Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet gaia.cs.umass.edu 80
```

- opens TCP connection to port 80 (default HTTP server port) at gaia.cs.umass. edu.
- anything typed in will be sent to port 80 at gaia.cs.umass.edu

2. type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu
```

- by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. look at response message sent by HTTP server!

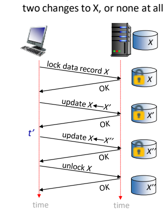
(or use Wireshark to look at captured HTTP request/response)

维护用户/服务器状态cookies
Maintaining user/server state: cookies

Recall: HTTP GET/response interaction is *stateless*

- HTTP协议是不保留状态的
  no notion of multi-step exchanges of HTTP messages to complete a Web "transaction"
  - no need for client/server to track "state" of multi-step exchange
  - all HTTP requests are independent of each other
  - no need for client/server to "recover" from a partially-completed-but-never-completely-completed transaction

a stateful protocol: client makes two changes to X, or none at all



Q: what happens if network connection or client crashes at *t'* ?

维护用户/服务器状态cookies
Maintaining user/server state: cookies

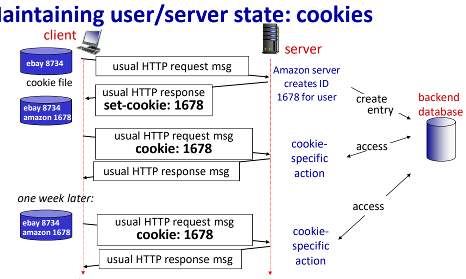
Web sites and client browser use *cookies* to maintain some state between transactions

cookies 技术有4个组件

1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

Example:
- Susan uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID (aka "cookie")
  - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

维护用户/服务器状态cookies
Maintaining user/server state: cookies



HTTP Cookies: comments

What cookies can be used for:
- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

Challenge: How to keep state:
- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: HTTP messages carry state

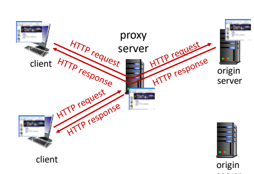
aside

cookies and privacy:
- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

Web caches (proxy servers)

Goal: satisfy client request without involving origin server

- user configures browser to point to a *Web cache*
- browser sends all HTTP requests to cache
  - *if* object in cache: cache returns object to client
  - *else* cache requests object from origin server, caches received object, then returns object to client



Web缓存（代理服务器）
Web caches (proxy servers)

- Web cache acts as both client and server
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

Why Web caching?

- reduce response time for client request
  - cache is closer to client
- reduce traffic on an institution's access link
- Internet is dense with caches
  - enables "poor" content providers to more effectively deliver content

Caching Example

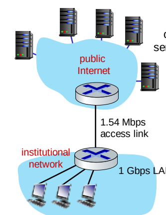
Scenario:

- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Average request rate from browsers to origin servers: 15/sec
  - average data rate to browsers: 1.50 Mbps

Performance:

- LAN utilization: .0015
- access link utilization = .97
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs

problem: large delays at high utilization!



Caching Example: buy a faster access link

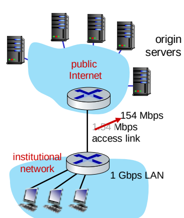
Scenario:

- access link rate: ~~1.54~~ 154 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

Performance:

- LAN utilization: .0015
- access link utilization = ~~.97~~ .0097
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + ~~minutes~~ msecs + usecs

Cost: faster access link (expensive!)



Caching Example: install a web cache

Scenario:

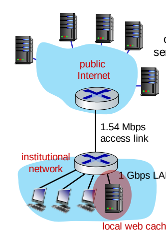
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

Performance:

- LAN utilization: .?
- access link utilization = ?
- average end-end delay = ?

How to compute link utilization, delay?

Cost: web cache (cheap!)

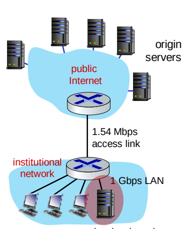


Caching Example: install a web cache

Calculating access link utilization, end-end delay with cache:

- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link
  = 0.6 * 1.50 Mbps = .9 Mbps
- utilization = 0.9/1.54 = .58
- average end-end delay
  = 0.6 * (delay from origin servers)
  + 0.4 * (delay when satisfied at cache)
  = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs

lower average end-end delay than with 154 Mbps link (and cheaper too!)

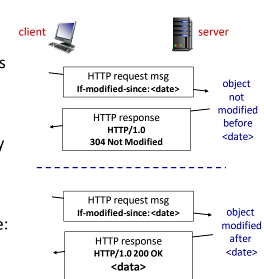


条件GET方法
Conditional GET

Goal: don't send object if cache has up-to-date cached version

- no object transmission delay
- lower link utilization

- *cache:* specify date of cached copy in HTTP request
  If-modified-since: <date>
- *server:* response contains no object if cached copy is up-to-date:
  HTTP/1.0 304 Not Modified



HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP1.1: introduced multiple, pipelined GETs over single TCP connection
- server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests
- smaller objects can be blocked (head-of-line (HOL) blocking，队首阻塞) behind large object, in server-to-client connection
- loss recovery (retransmitting lost TCP segments) stalls object transmission

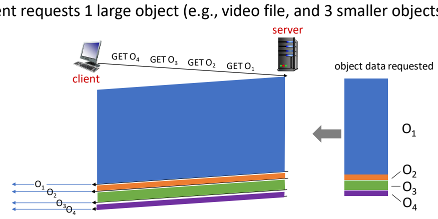
HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP/2: [RFC 7540, 2015] increased flexibility at server in sending objects to client:
- schedule transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking
- methods, status codes, most header fields unchanged from HTTP 1.1
- loss recovery still stalls object transmission
- HTTP/3: error, congestion control, security, more pipelining over UDP

HTTP/2: 缓解队首阻塞问题
HTTP/2: mitigating HOL blocking

Client requests 1 large object (e.g., video file, and 3 smaller objects)

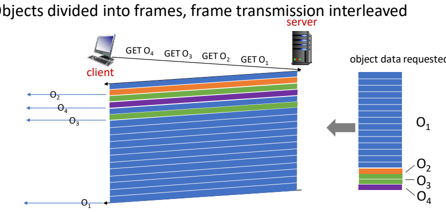


objects delivered in order requested: O_2, O_3, O_4 wait behind O_1

HTTP/2: 缓解队首阻塞问题
HTTP/2: mitigating HOL blocking

对象划分成帧，每个对象的帧交替传输

Objects divided into frames, frame transmission interleaved



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

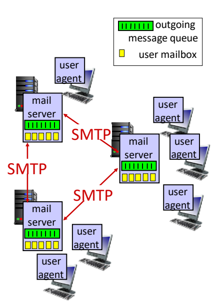
The Application Layer: Overview

- Principles of network applications
- Web and HTTP
- **E-mail, SMTP, IMAP**
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

E-mail

Three major components:
- user agents
- mail servers
- simple mail transfer protocol: SMTP

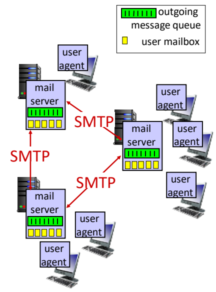
User Agent
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- outgoing, incoming messages stored on server



E-mail: mail servers

mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages
- *SMTP protocol* between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server



SMTP: RFC 5321

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
- direct transfer: sending server (acting like client) to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCI

Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail message "to" bob@someschool.edu

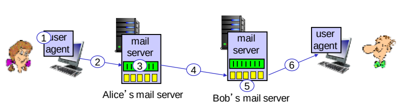
2) Alice's UA sends message to her mail server; message placed in message queue

3) client side of SMTP opens TCP connection with Bob's mail server

4) SMTP client sends Alice's message over the TCP connection

5) Bob's mail server places the message in Bob's mailbox

6) Bob invokes his user agent to read message



Sample SMTP interaction

```
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
```


Try SMTP interaction for yourself:

telnet <servername> 25

- see 220 reply from server
- enter HELO, MAIL FROM:, RCPT TO:, DATA, QUIT commands

above lets you send email without using e-mail client (reader)

Note: this will only work if <servername> allows telnet connections to port 25 (this is becoming increasingly rare because of security concerns)

SMTP是一种推协议

comparison with HTTP:

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

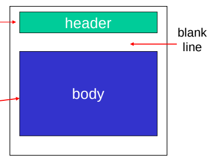
Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 531 (like HTTP)

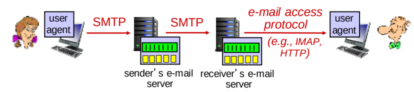
RFC 822 defines *syntax* for e-mail message itself (like HTML)

- header lines, e.g.,
  - To:
  - From:
  - Subject:

  these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
- Body: the "message" , ASCII characters only



邮件访问协议
Mail access protocols



- SMTP: delivery/storage of e-mail messages to receiver's server
- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
- HTTP: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages

The Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

域名系统
DNS: Domain Name System

people: many identifiers:
- SSN, name, passport #

Internet hosts, routers:
- IP address (32 bit) - used for addressing datagrams
- "name", e.g., cs.umass.edu - used by humans

Q: how to map between IP address and name, and vice versa ?

Domain Name System:
- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, *implemented as application-layer protocol*
  - complexity at network's "edge"

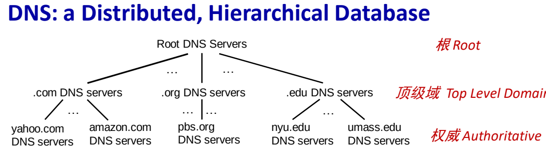
DNS: Services, Structure

DNS services
- 主机名到IP地址的映射 hostname to IP address translation
- 主机别名 host aliasing
  - 规范主机名 canonical
  - 多个很容易记忆的别名 alias names
- 邮件服务器别名 mail server aliasing
- 负载分配 load distribution
  - 冗余的服务器之间进行负载均衡 replicated Web servers: many IP addresses correspond to one name

Q: Why not centralize DNS?
- single point of failure
- traffic volume
- distant centralized database
- maintenance

A: doesn't scale!
- 单点故障
- 通信容量600B DNS queries per day
- 远距离集中式数据库
- 维护

DNS:分布式、层次数据库
DNS: a Distributed, Hierarchical Database



Client wants IP address for www.amazon.com:
- client queries root server to find .com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS: 根域名服务器
DNS: Root Name Servers

- 官方的，无法解析名字时应联系的地方official, contact-of-last-resort by name servers that can not resolve name
- *非常重要的网络功能incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name "servers" worldwide each "server" replicated many times (~200 servers in US)

顶级域名服务器、权威域名服务器
TLD, Authoritative Servers

Top-Level Domain (TLD) servers:
- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .cn, .uk, .fr, .ca, .jp
- Network Solutions: authoritative registry for .com, .net TLD
- Educause: .edu TLD

Authoritative DNS servers:
- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

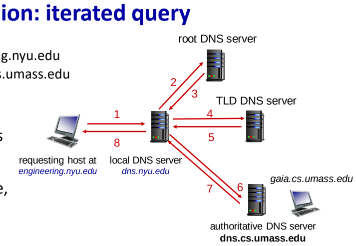
本地域名服务器
Local DNS Name Server

- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

DNS名字解析：迭代查询
DNS name resolution: iterated query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

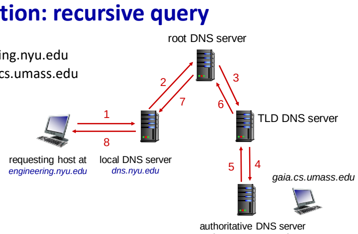
Iterated query:
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS名字解析：递归查询
DNS name resolution: recursive query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

Recursive query:
- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



Caching, Updating DNS Records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL, Time to Live)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best-effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire!
- update/notify mechanisms proposed IETF standard
  - RFC 2136

DNS Records

DNS: distributed database storing resource records (RR)

RR format: `(name, value, type, ttl)`

type=A
- `name` is hostname
- `value` is IP address

type=NS
- `name` is domain (e.g., foo.com)
- `value` is hostname of authoritative name server for this domain

type=CNAME
- `name` is alias name for some "canonical" (the real) name
- www.ibm.com is really servereast.backup2.ibm.com
- `value` is canonical name

type=MX
- `value` is name of mailserver associated with `name`

DNS Protocol and Messages

DNS *query* and *reply* messages, both have same *format:*

message header:
- 标识符 identification: 16 bit # for query, reply to query uses same #
- 标志 flags:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative

2 bytes	2 bytes
identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

DNS Protocol and Messages

DNS *query* and *reply* messages, both have same *format:*

2 bytes	2 bytes
identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

name, type fields for a query — questions (variable # of questions)

RRs in response to query — answers (variable # of RRs)

records for authoritative servers — authority (variable # of RRs)

additional " helpful" info that may be used — additional info (variable # of RRs)

Inserting Records into DNS

Example: new startup "Network Utopia"
- register name networkuptopia.com at *DNS registrar* (DNS注册登记机构 e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts NS, A RRs into .com TLD server:
    `(networkutopia.com, dns1.networkutopia.com, NS)`
    `(dns1.networkutopia.com, 212.212.212.1, A)`
- create authoritative server locally with IP address `212.212.212.1`
  - type A record for www.networkuptopia.com
  - type MX record for networkutopia.com

DNS Security

DDoS attacks
- bombard root servers with traffic
  - not successful to date
  - traffic filtering
  - local DNS servers cache IPs of TLD servers, allowing root server bypass
- bombard TLD servers
  - potentially more dangerous

Redirect attacks
- man-in-middle
  - intercept DNS queries
- DNS poisoning
  - send bogus relies to DNS server, which caches

Exploit DNS for DDoS
- send queries with spoofed source address: target IP
- requires amplification

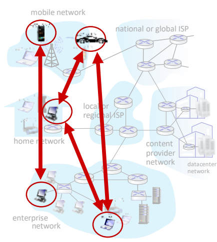
DNSSEC
[RFC 4033]

The Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

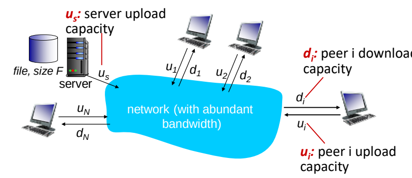
Peer-to-peer (P2P) Architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- examples: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)



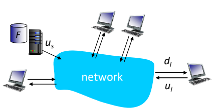
File distribution: client-server vs P2P

Q: how much time to distribute file (size *F*) from one server to *N* peers?
- peer upload/download capacity is limited resource



File distribution time: client-server

- *server transmission:* must sequentially send (upload) *N* file copies:
  - time to send one copy: F/u_s
  - time to send *N* copies: NF/u_s
- *client:* each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}

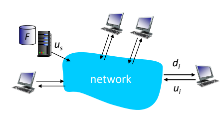


time to distribute F to N clients using client-server approach $D_{c\text{-}s} \geq max\{NF/u_s, F/d_{min}\}$

increases linearly in N

File distribution time: P2P

- *server transmission:* must sequentially send (upload) *N* file copies:
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *clients: 系统必须向N个对等方每个上载F比特，因此总交付iNF比特。系统整体的上载能力等于服务器的上载速率加上每个单独的对等方的上载速率* max upload rate (limiting max download rate) is $u_s + \Sigma u_i$

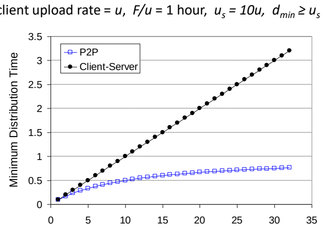


time to distribute F to N clients using P2P approach $D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$

increases linearly in *N* …
… but so does this, as each peer brings service capacity

Client-server vs. P2P: example

client upload rate = *u*, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$

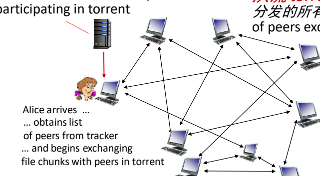


P2P file distribution: BitTorrent

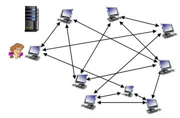
- file divided into 256Kb chunks(文件块)
- peers in torrent send/receive file chunks

追踪器 tracker: tracks peers participating in torrent

洪流 torrent: 参与一个特定文件分发的所有对等发的集合 group of peers exchanging chunks of a file



P2P file distribution: BitTorrent



- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

Requesting chunks:

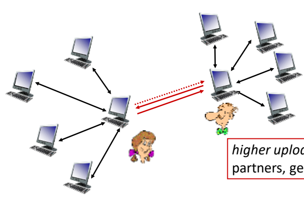
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, 最稀缺优先技术 rarest first

Sending chunks: 一报还一报 tit-for-tat

- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked（阻塞） by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - “optimistically unchoke” this peer
  - newly chosen peer may join top 4

BitTorrent: tit-for-tat

(1) Alice “optimistically unchokes” Bob
(2) Alice becomes one of Bob’s top-four providers; Bob reciprocates（报答）
(3) Bob becomes one of Alice’s top-four providers



higher upload rate: find better trading partners, get file faster !

The Application Layer: Overview

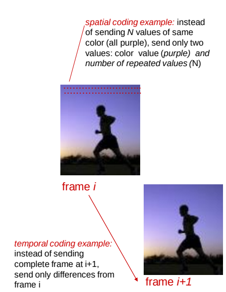
- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- challenge: 规模 scale - how to reach ~1B users?
  - single mega-video server won’t work (why?)
- challenge: 异构性 heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution: distributed, application-level infrastructure*

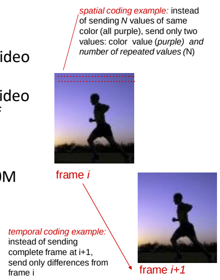
Multimedia: Video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy（冗余） *within* and *between* images to decrease # bits used to encode image
  - spatial (within image)
  - temporal (from one image to next)



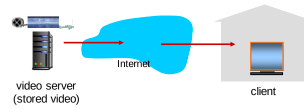
Multimedia: Video

- CBR: (固定码率 constant bit rate): video encoding rate fixed
- VBR: (可变码率 variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- examples:
  - MPEG 1 (CD-ROM) 1.5 Mbps, 700M
  - MPEG2 (DVD) 3-6 Mbps，4.3G
  - MPEG4 (often used in Internet, 64Kbps – 12 Mbps)



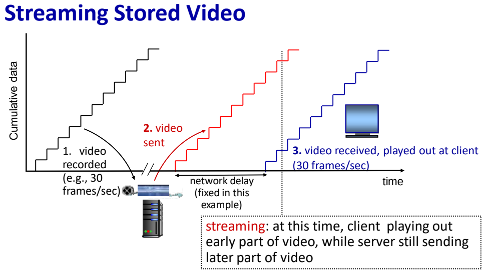
流媒体存储视频
Streaming Stored Video

simple scenario:



Main challenge:
- server-to-client bandwidth will *vary* over time. with changing network congestion levels (in house, in access network, in network core, at video server)
- packet loss and delay due to congestion will delay playout, or result in poor video quality

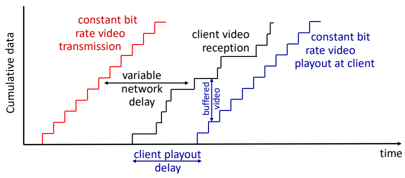
流媒体存储视频
Streaming Stored Video



流媒体存储视频
Streaming Stored Video: Challenges

- continuous playout constraint: once client playout begins, playback must match original timing
  - … but network delays are variable (jitter), so will need client-side buffer to match playout requirements
- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted

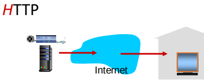
流媒体存储视频：播放缓冲
Streaming Stored Video: playout buffering



- *客户端缓冲和播放延迟：补偿网络附加延迟、延迟抖动 client-side buffering and playout delay:* compensate for network-added delay, delay jitter

HTTP上的动态自适应流媒体协议 DASH
Streaming multimedia: DASH

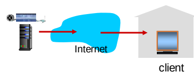
- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
- *server:*
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - *告示文件 manifest file:* provides URLs for different chunks



- *client:*
  - periodically measures server-to-client bandwidth
  - consulting manifest, requests one chunk at a time
    - chooses maximum coding rate sustainable given current bandwidth
    - can choose different coding rates at different points in time (depending on available bandwidth at time)

HTTP上的动态自适应流媒体协议
Streaming multimedia: DASH

- *"intelligence"* at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is "close" to client or has high available bandwidth)



Streaming video = encoding + DASH + playout buffering

内容分发网络
Content Distribution Networks (CDNs)

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- option 1: single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of video sent over outgoing link

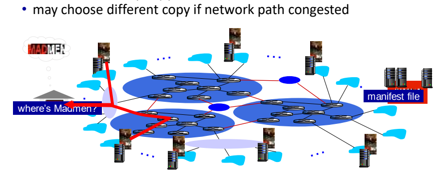
....quite simply: this solution *doesn't scale*

内容分发网络
Content Distribution Networks (CDNs)

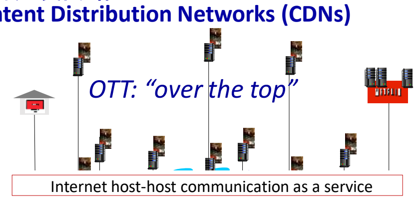
- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- option 2: store/serve multiple copies of videos at multiple geographically distributed sites *(CDN)*
  - *深入 enter deep:* push CDN servers deep into many access networks
    - close to users
    - Akamai: 240,000 servers deployed in more than 120 countries (2015)
  - *邀请做客 bring home:* smaller number (10's) of larger clusters in POPs near (but not within) access networks
    - used by Limelight

内容分发网络
Content Distribution Networks (CDNs)

- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested



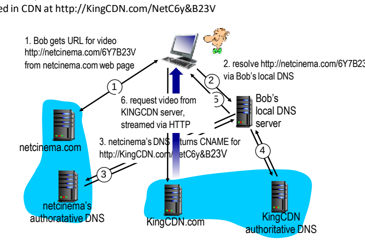
内容分发网络
Content Distribution Networks (CDNs)



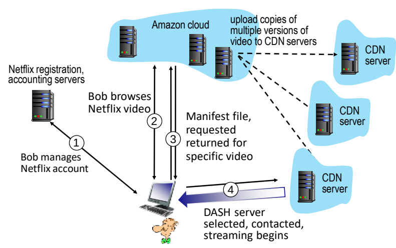
OTT challenges: coping with a congested Internet
- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

CDN content access: a closer look

Bob (client) requests video http://netcinema.com/6Y7B23V
- video stored in CDN at http://KingCDN.com/NetC6y&B23V



Case study: Netflix



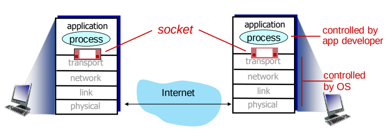
The Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:
- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:
1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

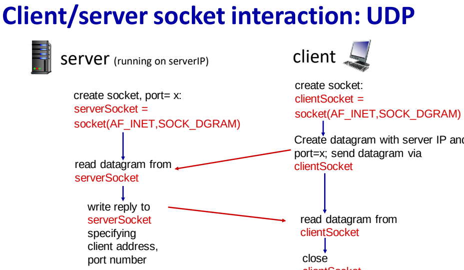
Socket programming with UDP

UDP: no "connection" between client & server
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:
- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



Example app: UDP client

Python UDPClient

include Python's socket library
```
from socket import *
serverName = 'hostname'
serverPort = 12000
```
create UDP socket for server
```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```
get user keyboard input
```
message = raw_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()
```


Example app: UDP server

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 programming with TCP

Client must contact server
- server process must first be running
- server must have created socket (door) that welcomes client's contact

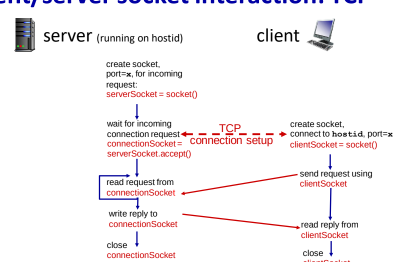
Client contacts server by:
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

Application viewpoint

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
```
create TCP socket for server, remote port 12000
```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
```
No need to attach server name, port
```
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```


Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
```
create TCP welcoming socket
```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
```
server begins listening for incoming TCP requests
```
serverSocket.listen(1)
print 'The server is ready to receive'
```
loop forever
```
while True:
```
server waits on accept() for incoming requests, new socket created on return
```
    connectionSocket, addr = serverSocket.accept()
```
read bytes from socket (but not address as in UDP)
```
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                          encode())
```
close connection to this client (but *not* welcoming socket)
```
    connectionSocket.close()
```

Chapter 2: Summary

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming: TCP, UDP sockets

Chapter 2: Summary

Most importantly: learned about *protocols*!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data:* info(payload) being communicated

important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- "complexity at network edge"

作　业

- 用户A与B的邮箱分别为a@XXX.com与B@YYY.com，请从应用角度简述A发送邮件给B的过程以及在这个过程中涉及的应用层协议如何协作完成该任务。

作　业

- SMS、iMessage和WhatsApp都是智能手机即时通信系统。在因特网上进行一些研究后，为这些系统分别写一段它们所使用协议的文字。然后撰文解释它们的差异所在。

作　业

- 考虑当浏览器发送一个HTTP GET报文时，通过Wireshark俘获到下列ASCII字符串（即这是一个HTTP GET报文的实际内容）。字符*<cr><lf>*是回车和换行符（即下面文本中的斜体字符串*<cr>*表示了单个回车符，该回车符包含在HTTP首部中的相应位置）。回答下列问题，指出你在下面HTTP GET报文中找到答案的地方。

```
GET /cs453/index.html HTTP/1.1<cr><lf>Host: gai
a.cs.umass.edu<cr><lf>User-Agent: Mozilla/5.0 (
Windows;U; Windows NT 5.1; en-US; rv:1.7.2) Gec
ko/20040804 Netscape/7.2 (ax) <cr><lf>Accept:ex
t/xml, application/xml, application/xhtml+xml, text
/html;q=0.9, text/plain;q=0.8,image/png,*/*;q=0.5
<cr><lf>Accept-Language: en-us,en;q=0.5<cr><lf>Accept-
Encoding: zip,deflate<cr><lf>Accept-Charset: ISO
-8859-1,utf-8;q=0.7,*;q=0.7<cr><lf>Keep-Alive: 300<cr>
<lf>Connection:keep-alive<cr><lf><cr><lf>
```

a. 由浏览器请求的文档的URL是什么？
b. 该浏览器运行的是HTTP的何种版本？
c. 该浏览器请求的是一条非持续连接还是一条持续连接？
d. 该浏览器所运行的主机的IP地址是什么？
e. 发起该报文的浏览器的类型是什么？在一个HTTP请求报文中，为什么需要浏览器类型？

作　业

- 下面文本中显示的是来自服务器的回答，以响应上述问题中HTTP GET报文。回答下列问题，指出你在下面报文中找到答案的地方。

```
HTTP/1.1 200 OK<cr><lf>Date: Tue, 07 Mar 2008
12:39:45GMT<cr><lf>Server: Apache/2.0.52 (Fedora)
<cr><lf>Last-Modified: Sat, 10 Dec2005 18:27:46
GMT<cr><lf>ETag: "526c3-f22-a88a4c80"<cr><lf>Accept-
Ranges: bytes<cr><lf>Content-Length: 3874<cr><lf>
Keep-Alive: timeout=max=100<cr><lf>Connection:
Keep-Alive<cr><lf>Content-Type: text/html; charset=
ISO-8859-1<cr><lf><cr><lf><!doctype html public "-
//w3c//dtd html 4.0transitional//en"><lf><html><lf>
<head><lf> <meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1"><lf> <meta
name="GENERATOR" content="Mozilla/4.79 [en] (Windows NT
5.0; U) Netscape]"><lf> <title>CMPSCI 453 / 591 /
NTU-ST550ASpring 2005 homepage</title><lf></head><lf>
<much more document text following here (not shown)>
```

a. 服务器能否成功地找到那个文档？该文档提供回答是什么时间？
b. 该文档最后修改是什么时间？
c. 文档中被返回的字节有多少？
d. 文档被返回的前5个字节是什么？该服务器同意一条持续连接吗？

作　业

- 假定你在浏览器中点击一条超链接获得Web页面。相关联的URL的IP地址没有缓存在本地主机上,因此必须使用DNS lookup以获得该IP地址。如果主机从DNS得到IP地址之前已经访问了*n*个DNS服务器；相继产生的RTT依次为RTT_1、…、RTT_n。进一步假定与链路相关的Web页面只包含一个对象，即由少量的HTML文本组成。令RTT_0表示本地主机和包含对象的服务器之间的RTT值。假定该对象传输时间为零，则从该客户点击该超链接到它接收到该对象需要多长时间？

作　业

- 假定你能够访问所在系的本地DNS服务器中的缓存。你能够提出一种方法来粗略地确定在你所在系的用户中最为流行的Web服务器（你所在系以外）吗？解释原因。

作　业

- 考虑使用一种客户-服务器体系结构向*N*个对等方分发一个*F*比特的文件。假定一种某服务器能够同时向多个对等方传输的流体模型，只要组合速率不超过u_s，则以不同的速率向每个对等方传输。
  a. 假定$u_s/N \le d_{min}$。定义一个具有NF/u_s，分发时间的分发方案。
  b. 假定$u_s/N \ge d_{min}$。定义一个具有F/d_{min}分发时间的分发方案。
  c. 得出最小分发时间通常是由$\max\{NF/u_s, F/d_{min}\}$所决定的结论。

作　业

- 假定Bob加入BitTorrent,但他不希望向任何其他对等方上载任何数据（因此称为搭便车）。
  a. Bob声称他能够收到由该社区共享的某文件的完整副本。Bob所言是可能的吗？为什么？
  b. Bob进一步声称他还能够更为有效地进行他的“搭便车”，方法是利用所在系的计算机实验室中的多台计算机（具有不同的IP地址）。他怎样才能做到这些呢？