A note on the use of these ppt slides:
We’re making these slides freely available to all (faculty, students, readers). They’re in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:
❖ If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we’d like people to use our book!)
❖ If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2010
J.F Kurose and K.W. Ross, All Rights Reserved

Chapter 2
Application Layer

Computer Networking:
Jim Kurose, Keith Ross
Pearson/Addison-Wesley, April 2016.
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   • SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
Chapter 2: Application Layer

**Our goals:**

- Conceptual, implementation aspects of network application protocols
  - Transport-layer service models
  - Client-server paradigm
  - Peer-to-peer paradigm

- Learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS

- Programming network applications
  - Socket API
Some network apps

- e-mail
- web
- instant messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube)
- voice over IP
- real-time video conferencing
- cloud computing
- ...
- ...
- ...
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
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
Application architectures

- client-server
- peer-to-peer (P2P)
- hybrid of client-server and P2P
Client-server architecture

server:
- always-on host
- permanent IP address
- server farms for scaling

clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other
Pure P2P architecture

- no always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

highly scalable but difficult to manage
Hybrid of client-server and P2P

Skype
- voice-over-IP P2P application
- centralized server: finding address of remote party:
- client-client connection: direct (not through server)

Instant messaging
- chatting between two users is P2P
- centralized service: client presence detection/location
  - user registers its IP address with central server when it comes online
  - user contacts central server to find IP addresses of buddies
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*

**client process:** process that initiates communication

**server process:** process that waits to be contacted

- aside: applications with P2P architectures have client processes & server processes
Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process

- API: (1) choice of transport protocol; (2) ability to fix a few parameters (lots more on this later)
Sockets & Process Addressing

- Connection End point determined by *pair*
  - Host address: IP address is *Network Layer*
  - Port number: is *Transport Layer*

  example 1: 206.62.226.35, p21 + 198.69.10.2, p1500
  example 2: 206.62.226.35, p21 + 198.69.10.2, p1499

Ports (16 bit integer)

- Numbers (typical, vary by OS):
  - 0–1023 “reserved”, must be root
  - 1024 – 5000 “ephemeral”
  - Above 5000 for general use

  • Well-known reserved s
  - ftp 21/tcp
  - http 80/tcp
  - telnet 23/tcp
  - finger 79/tcp
  - smtp 25/tcp
Addressing processes

- 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?
Addressing processes

- 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

- 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…
App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

public-domain protocols:
- defined in IETF RFCs
- allows for interoperability
  - e.g., HTTP, SMTP

proprietary protocols:
- e.g., Skype

IETF: Internet Engineering Task Force (www.ietf.org)
What transport service does an app need?

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

**Timing/Latency**
- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

**Throughput**
- some apps (e.g., multimedia) require minimum amount of throughput to be “effective”
- other apps (“elastic apps”) make use of whatever throughput they get

**Security**
- encryption, data integrity, ...

## Transport service requirements of common apps

<table>
<thead>
<tr>
<th>Application</th>
<th>Data loss</th>
<th>Throughput</th>
<th>Time Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>file transfer</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>e-mail</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>Web documents</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>audio: 5kbps-1Mbps</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>video: 10kbps-5Mbps</td>
<td></td>
</tr>
<tr>
<td>stored audio/video</td>
<td>loss-tolerant</td>
<td>same as above</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>few kbps up</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td>instant messaging</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no</td>
</tr>
</tbody>
</table>

**elastic**: throughput between end-point adjusts to network conditions (TCP traffic)  
**vs. inelastic** (UDP - e.g. multimedia)
Internet transport protocols services

TCP service:

- *connection-oriented*: setup required between client and server processes
- *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 guarantees, security

UDP service:

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

Q: why bother? Why is there a UDP?
# Internet apps: application, transport protocols

<table>
<thead>
<tr>
<th>Application</th>
<th>Application layer protocol</th>
<th>Underlying transport protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-mail</td>
<td>SMTP [RFC 2821]</td>
<td>TCP</td>
</tr>
<tr>
<td>remote terminal access</td>
<td>Telnet [RFC 854]</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>HTTP [RFC 2616]</td>
<td>TCP</td>
</tr>
<tr>
<td>file transfer</td>
<td>FTP [RFC 959]</td>
<td>TCP</td>
</tr>
<tr>
<td>streaming multimedia</td>
<td>HTTP (e.g., YouTube), RTP [RFC 1889]</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>SIP, RTP, proprietary (e.g., Skype)</td>
<td>typically UDP</td>
</tr>
<tr>
<td>domain name service</td>
<td>DNS (RFC 1034, 1035)</td>
<td>typically UDP (local) but also TCP (zone)</td>
</tr>
</tbody>
</table>
Chapter 2: Application layer

2.1 Principles of network applications
   ▪ app architectures
   ▪ app requirements

2.2 Web and HTTP

2.3 FTP

2.4 Electronic Mail
   ▪ SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 Socket programming with TCP

2.8 Socket programming with UDP
Web and HTTP

First, a review...

- web page consists of objects
- object can be HTML file, JPEG image, Java applet, audio file,...
- web page consists of base HTML-file which includes several referenced objects
- each object is addressable by a URL
- example URL:

  www.someschool.edu/someDept/pic.gif

  \[\text{host name} \quad \text{path name}\]
Simple html file (hi.html)

<table>
<thead>
<tr>
<th>Make a file called &quot;hi.html&quot; containing this</th>
<th>Use your a Web browser to open the file &quot;hi.html,&quot; and it will look something like this</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;!DOCTYPE html PUBLIC &quot;-//IETF//DTD HTML 2.0//EN&quot;&gt;</code>&lt;HTML&gt;&lt;HEAD&gt;&lt;TITLE&gt;A Small Hello&lt;/TITLE&gt;&lt;/HEAD&gt;&lt;BODY&gt;&lt;H1&gt;Hi&lt;/H1&gt;&lt;P&gt;This is very minimal &quot;hello world&quot; HTML document.&lt;/P&gt;&lt;/BODY&gt;&lt;/HTML&gt;`</td>
<td><strong>Hi</strong>&lt;br&gt;This is a very minimal &quot;hello world&quot; HTML document.</td>
</tr>
</tbody>
</table>

Only the elements that you place in the BODY element (that is, between `<BODY>` and `</BODY>` ) ever get displayed in a Web browser's window.

In this example, only the contents of the H1 element (between `<H1>` and `</H1>` ) and the P element (between `<P>` and `</P>` ) are displayed.

The phrase between `<TITLE>` and `</TITLE>` will be displayed in the browser's title bar.
HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- Client/server model
  - **client**: browser that requests, receives, "displays" Web objects
  - **server**: Web server sends objects in response to requests
HTTP overview (continued)

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
HTTP connections

**non-persistent HTTP**
- at most one object sent over TCP connection.

**persistent HTTP**
- multiple objects can be sent over single TCP connection between client, server.
Nonpersistent HTTP

Suppose user enters URL:
www.someSchool.edu/someDepartment/home.index

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
Nonpersistent HTTP (cont.)

4. HTTP server closes TCP connection.


6. Steps 1-5 repeated for each of 10 jpeg objects.
Non-Persistent HTTP: Response time

definition of RTT: time for a small packet to travel from client to server and back.

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

total = 2RTT + transmit time
**Persistent HTTP**

**non-persistent HTTP issues:**
- requires 2 RTTs per object
- OS overhead for *each* TCP connection

**persistent HTTP**
- GET message contains a *keep-alive field* to ask the server to leave TCP 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
2: Application Layer

Non-persistent HTTP

Browser receives main html file, parses it and finds references for 2 embedded objects

Open TCP connection

Download object1

Open TCP connection

Download object2

Close TCP connection

persistent HTTP

Browser receives main html file, parses it and finds references for 2 embedded objects

Open TCP connection

Download object1

Close TCP connection

Download object2

Close TCP connection
Persistent HTTP with Pipelining

**Persistent HTTP:**
- After main HTML file received, server keeps TCP connection open
- Browser sends one HTTP GET request for first embedded object and wait for a response
- After object is received, browser sends a second HTTP GET request for the second object
- Browser repeats the process until all objects are received
- Time wasted in waiting between consecutive Request/Response transactions

**Persistent HTTP with pipelining**
- After main HTML file received, server keeps TCP connection open
- Browser sends multiple HTTP GET requests in parallel to the server, one for each embedded object
- All GET requests use the SAME TCP connection
- Server send objects back-to-back to the browser
- Avoid waiting time between consecutive Request/Response transactions ➔ faster page load time
HTTP with Parallel connections

persistent HTTP issues:
- Only works if all embedded objects stored on the same server
- If some embedded objects are stored on a different server, browser must open separate TCP connection to each server

HTTP with Parallel connection
- Most modern browsers open parallel TCP connections to speed up page download
- After main html file downloaded and parsed, the browser immediately opens multiple TCP connections, one for each embedded object
- Each TCP connection is used to GET only one object and then closed
- This works regardless of whether objects stored on the same server or not
persistent HTTP with pipelining

HTTP with parallel connections
Tweaking browser settings (Firefox example)
## Sample Firefox settings

- Relevant HTTP settings can be found in the page: `about:config`
- Search for: "network.http."

<table>
<thead>
<tr>
<th>parameter</th>
<th>non-persistent</th>
<th>persistent</th>
<th>Persistent with pipelining</th>
<th>Parallel connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>network.http.keep-alive</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>network.http.max-connections-per-server</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>network.http.max-persistent-connections-per-server</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>network.http.pipelining</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>network.http.pipelining.aggressive</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
**HTTP request message**

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

```
GET /index.html HTTP/1.1\r\nHost: www-net.cs.umass.edu\r\nUser-Agent: Firefox/3.6.10\r\nAccept: text/html,application/xhtml+xml\r
Accept-Language: en-us,en;q=0.5\r\nAccept-Encoding: gzip,deflate\r\nAccept-Charset: ISO-8859-1,utf-8;q=0.7\r\nKeep-Alive: 115\r\nConnection: keep-alive\r\n\r\n```

Carriage return character

Line-feed character
HTTP request message: general format

<table>
<thead>
<tr>
<th>method</th>
<th>sp</th>
<th>URL</th>
<th>sp</th>
<th>version</th>
<th>cr</th>
<th>lf</th>
</tr>
</thead>
<tbody>
<tr>
<td>header field name</td>
<td>:</td>
<td>value</td>
<td>cr</td>
<td>lf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>header field name</td>
<td>:</td>
<td>value</td>
<td>cr</td>
<td>lf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cr</td>
<td>lf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entity Body
**Uploading form input**

**POST method:** (query keywords inside “Entity body” of request message)
- web page often includes form input
- input is uploaded to server in entity body

**URL method:**
- uses GET method
- input is uploaded in URL field of request line:
  
  www.somesite.com/animalsearch?monkeys&banana

- Entity body is empty
Method types

HTTP/1.0
- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1
- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field
HTTP response message

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

...
HTTP response status codes

- status code appears in 1st line in server->client response message.
- some sample codes:
  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
User-server state: cookies

many Web sites use cookies

four components:

1) cookie header line of HTTP response message
2) cookie header line in 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 always access Internet from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID
Cookies: keeping “state” (cont.)

Client:
- eBay 8734
- Cookie file

Server:
- Amazon server creates ID 1678 for user

One week later:
- eBay 8734
- Amazon 1678

Diagram:
- Usual HTTP request msg
- Set-cookie: 1678
- Usual HTTP response msg
- Cookie: 1678
- Usual HTTP response msg
- Backend database

Access:
- Cookie-specific action
- Cookie-specific action
- Create entry
Cookies (continued)

what cookies can bring:
- authorization
- shopping carts
- recommendations
- user session state
  (Web e-mail)

how to keep “state”:
- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

cookies and privacy:
- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

aside
**Web caches (proxy server)**

**Goal:** satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client
More about Web caching

- cache acts as both client and server
- typically cache is installed by ISP (university, company, residential ISP)

why Web caching?
- reduce response time for client request
- reduce traffic on an institution’s access link.
- Internet dense with caches: enables “poor” content providers to effectively deliver content (but so does P2P file sharing)
Caching example

assumptions

- average object size = 100,000 bits
- avg. request rate from institution’s browsers to origin servers = 15/sec
- delay from internet gateway router to any origin server and back = 2 sec

consequences

- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + milliseconds

Note: For defn. of Utilization, see Ch. 1 slide 49
Caching example (cont)

possible solution

- increase bandwidth of access link to, say, 10 Mbps

consequence

- utilization on LAN = 15%
- utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + msecs + msecs
- often a costly upgrade
**Caching example (cont)**

**possible solution:**
- install cache

**consequence**
- suppose hit rate is 0.4
  - 40% requests will be satisfied almost immediately
  - 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)
- total avg delay = Internet delay + access delay + LAN delay = \(0.6 \times 2.01\) secs + \(0.4\) milliseconds < 1.4 secs
Conditional GET (original server & cache synchronization issue)

- **Goal:** don't send object if cache has up-to-date cached version
- 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
- server: response contains object modified after <date>:
  - HTTP/1.0 200 OK
  - <data>

Application 2-51
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic mail
   ▪ SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
FTP: the file transfer protocol

- transfer file to/from remote host
- client/server model
  - client: side that initiates transfer (either to/from remote)
  - server: remote host
- ftp: RFC 959
- ftp server: port 21
FTP: separate control, data connections

- FTP client contacts FTP server at port 21, TCP is transport protocol
- Client authorized over control connection
- Client browses remote directory by sending commands over control connection.
- When server receives file transfer command, server opens 2nd TCP connection (for file) to client
- After transferring one file, server closes data connection.
- Server opens another TCP data connection to transfer another file (one new connection per file).
- Control connection: "out of band"
- FTP server maintains "state": current directory, earlier authentication
**FTP commands, responses**

**sample commands:**
- sent as ASCII text over control channel
- `USER username`
- `PASS password`
- `LIST` return list of file in current directory
- `RETR filename` retrieves (gets) file
- `STOR filename` stores (puts) file onto remote host

**sample return codes**
- status code and phrase (as in HTTP)
- 331 Username OK, password required
- 125 data connection already open; transfer starting
- 425 Can’t open data connection
- 452 Error writing file
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   - SMTP, POP3, IMAP
2.5 DNS

2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
Electronic 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, elm, Mozilla Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server
Electronic 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
Electronic Mail: SMTP [RFC 2821]

- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII
Scenario: Alice sends message to Bob

1) Alice uses UA to compose message and “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 email client (reader)
SMTP: final words

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

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

SMTP: protocol for exchanging email msgs
RFC 822: standard for text message format:

- header lines, e.g.,
  - To:
  - From:
  - Subject:
    different from SMTP commands!

- body
  - the “message”, ASCII characters only
Mail access protocols

- **SMTP**: delivery/storage to receiver's server
- Mail access protocol: retrieval from server (PULL)
  - **POP**: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - **IMAP**: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
  - **HTTP**: gmail, Hotmail, Yahoo! Mail, etc. (web-based email)
**POP3 protocol**

**authorization phase**
- **client commands:**
  - `user`: declare username
  - `pass`: password
- **server responses**
  - `+OK`
  - `-ERR`

**transaction phase, client:**
- `list`: list message numbers
- `retr`: retrieve message by number
- `dele`: delete
- `quit`

---

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

---

Application 2-66
**POP3 (more) and IMAP**

**more about POP3**
- previous example uses “download and delete” mode.
- Bob cannot re-read e-mail if he changes client
- “download-and-keep”: copies of messages on different clients
- POP3 is stateless across sessions

**IMAP**
- keeps all messages in one place: at server
- allows user to organize messages in folders
- keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name
Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P applications
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
DNS: Domain Name System

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

**Internet hosts, routers:**
- IPv4 address (32 bit) - used for addressing datagrams
- “name”, e.g., www.yahoo.com - used by humans

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

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

DNS services
- hostname to IP address translation
- host aliasing
  - Canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: set of IP addresses for one canonical name

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

doesn't scale!
Distributed, Hierarchical Database

- **Root DNS Servers**
  - com DNS servers
    - yahoo.com DNS servers
  - org DNS servers
    - amazon.com DNS servers
    - pbs.org DNS servers
  - edu DNS servers
    - poly.edu DNS servers
    - umass.edu DNS servers

**client wants IP for www.amazon.com; 1st approx:**
- client queries a root server to find com DNS server
- client queries com DNS server to get amazon.com DNS server (top level domain name server)
- client queries amazon.com DNS server to get IP address for **www.amazon.com** (org. level `authoritative` name server)
DNS: Root name servers

- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server

13 root name servers worldwide:

- a Verisign, Dulles, VA
- b USC-ISI Marina del Rey, CA
- c Cogent, Herndon, VA (also LA)
- d U Maryland College Park, MD
- e NASA Mt View, CA
- f Internet Software C. Palo Alto, CA (and 36 other locations)
- g US DoD Vienna, VA
- h ARL Aberdeen, MD
- i Autonomica, Stockholm (plus 28 other locations)
- j Verisign, (21 locations)
- k RIPE London (also 16 other locations)
- l ICANN Los Angeles, CA
- m WIDE Tokyo (also Seoul, Paris, SF)
- n Verisign, (21 locations)
TLD and Authoritative Servers

Top-level domain (TLD) servers:

- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for com TLD
- Educause for edu TLD

Authoritative DNS servers:

- organization’s DNS servers, providing authoritative hostname to IP mappings for organization’s servers (e.g., Web, mail).
- can be maintained by organization or service provider
Local 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
  - acts as proxy, forwards query into hierarchy
**DNS name resolution example**

- host at cis.poly.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 name resolution example**

**recursive query:**
- Puts burden of name resolution on contacted name server
- Heavy load?

---

**Diagram:**
- **Requesting host:** cis.poly.edu
- **Local DNS server:** dns.poly.edu
- **Authoritative DNS server:** dns.cs.umass.edu
- **TLD DNS server:**
- **Root DNS server:**

---

Application 2-76
DNS: caching and updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time
  - TLD servers typically cached in local name servers
    - Thus root name servers not often visited
- update/notify mechanisms proposed IETF standard
  - RFC 2136
DNS records

**DNS:** distributed db storing resource records (RR)

**RR format:** \[(name, value, type, ttl)\]  

**ttl:** time to live

name (i/p), value (o/p) field depends on type

- **Type=A** (from authoritative name server)
  - 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, messages

### DNS protocol: query and reply messages, both with same message format

**msg header**

- **identification**: 16 bit # for query, reply to query uses same #
- **flags**:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative

<table>
<thead>
<tr>
<th>identification</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of questions</td>
<td>number of answer RRss</td>
</tr>
<tr>
<td>number of authority RRss</td>
<td>number of additional RRss</td>
</tr>
</tbody>
</table>

- questions (variable number of questions)
- answers (variable number of resource records)
- authority (variable number of resource records)
- additional information (variable number of resource records)
**DNS protocol, messages**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identification</td>
<td></td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>number of questions</td>
<td></td>
</tr>
<tr>
<td>number of answer RRs</td>
<td></td>
</tr>
<tr>
<td>number of authority RRs</td>
<td></td>
</tr>
<tr>
<td>number of additional RRs</td>
<td></td>
</tr>
<tr>
<td>questions</td>
<td>(variable number of questions)</td>
</tr>
<tr>
<td>answers</td>
<td>(variable number of resource records)</td>
</tr>
<tr>
<td>authority</td>
<td>(variable number of resource records)</td>
</tr>
<tr>
<td>additional info</td>
<td>(variable number of resource records)</td>
</tr>
</tbody>
</table>

Name, type fields for a query

RRs in response to query

Records for authoritative servers

Additional "helpful" info that may be used

12 bytes
Inserting records into DNS

- example: new startup “Network Utopia”
- register name networkuptopia.com at DNS registrar (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts two RRs into com TLD server:
    
    (networkutopia.com, dns1.networkutopia.com, NS)
    (dns1.networkutopia.com, 212.212.212.1, A)

- Other people are now able to get IP address of your Web site?
- create Type MX record for networkutopia.com
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   - SMTP, POP3, IMAP
2.5 DNS

2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
Pure P2P architecture

- *no always-on server*
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

Three topics:
- file distribution
- searching for information
- case Study: Skype
**File Distribution: Server-Client vs P2P**

**Question**: How much time to distribute file from one server to $N$ peers?

- $u_s$: server upload bandwidth
- $u_i$: peer $i$ upload bandwidth
- $d_i$: peer $i$ download bandwidth
File distribution time: server-client

- server sequentially sends $N$ copies (one to each client):
  - $NF/u_s$ time
- client $i$ takes $F/d_i$ time to download

---

Time to distribute $F$ to $N$ clients using client-server approach

$$d_{cs} = \max \{ NF/u_s, \frac{F}{\min(d_i)} \}$$

increases linearly in $N$ (for large $N$)
File distribution time: P2P

- server must send one copy: $F/u_s$ time
- client $i$ takes $F/d_i$ time to download
- NF bits must be downloaded (aggregate)
  - fastest possible upload rate: $u_s + \sum u_i$

$$d_{p2p} \geq \max \{ F/u_s, F/min(d_i), NF/(u_s + \sum u_i) \}$$
Server-client vs. P2P: example

Client upload rate = u, $F/u = 1$ hour, $u_s = 10u$, $d_{\text{min}} \geq u_s$
File distribution: BitTorrent

P2P file distribution

**tracker**: tracks peers participating in torrent

**torrent**: group of peers exchanging chunks of a file

peer

obtain list of peers

trading chunks
BitTorrent (1)

- file divided into 256KB *chunks*.
- peer joining torrent:
  - has no chunks, but will accumulate them over time
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers.
- peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain
BitTorrent (2)

Pulling Chunks
- at any given time, different peers have different subsets of file chunks
- periodically, a peer (Alice) asks each neighbor for list of chunks that they have.
- Alice sends requests for her missing chunks
  - rarest first

Sending Chunks: tit-for-tat
- Alice sends chunks to four neighbors currently sending her chunks at the highest rate
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - newly chosen peer may join top 4
  - “optimistically unchoke”
BitTorrent: Tit-for-tat

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

With higher upload rate, can find better trading partners & get file faster!
P2P Case study: Skype

- inherently P2P: pairs of users communicate.
- proprietary application-layer protocol (inferred via reverse engineering)
- hierarchical overlay with SNs
- Index maps usernames to IP addresses, ports; distributed over SNs
Peers as relays

- problem when both Alice and Bob are behind "NATs".
  - NAT prevents an outside peer from initiating a call to insider peer

- solution:
  - using Alice’s and Bob’s SNs, *relay* is chosen
  - each peer initiates session with relay.
  - peers can now communicate through NATs via relay
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   • SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
Socket programming

**Goal:** learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented
Socket-programming using TCP

**Socket**: a door between application process and end-to-end-transport protocol (UCP or TCP)

**TCP service**: reliable transfer of *bytes* from one process to another
Socket programming \textit{with TCP}

\textbf{Client must contact server}
\begin{itemize}
  \item server process must first be running
  \item server must have created socket (door) that welcomes client’s contact
\end{itemize}

\textbf{Client contacts server by:}
\begin{itemize}
  \item creating client-local TCP socket
  \item specifying IP address, port number of server process
  \item when \textit{client creates socket:} client TCP establishes connection to server TCP
\end{itemize}

\begin{itemize}
  \item when contacted by client, server TCP creates new socket for server process to communicate with client
    \begin{itemize}
      \item allows server to talk with multiple clients
      \item source port numbers used to distinguish clients (more in Chap 3)
    \end{itemize}
\end{itemize}

\textbf{application viewpoint}

\textit{TCP provides reliable, in-order transfer of bytes (“pipe”) between client and server}
Client/server socket interaction: TCP

Server (running on hostid)

create socket, port=x, for incoming request:
welcomeSocket = ServerSocket()

wait for incoming connection request
connectionSocket = welcomeSocket.accept()

read request from connectionSocket
write reply to connectionSocket
close connectionSocket

Client

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

send request using clientSocket
read reply from clientSocket
close clientSocket

TCP connection setup
Stream jargon

- **stream** is a sequence of characters that flow into or out of a process.
- **input stream** is attached to some input source for the process, e.g., keyboard or socket.
- **output stream** is attached to an output source, e.g., monitor or socket.
Socket programming with TCP

Example client-server app:

1) client reads line from standard input (inFromUser stream), sends to server via socket (outToServer stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (inFromServer stream)
# Chapter 2: Application layer

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Principles of network applications</td>
</tr>
<tr>
<td>2.2</td>
<td>Web and HTTP</td>
</tr>
<tr>
<td>2.3</td>
<td>FTP</td>
</tr>
<tr>
<td>2.4</td>
<td>Electronic Mail</td>
</tr>
<tr>
<td></td>
<td>- SMTP, POP3, IMAP</td>
</tr>
<tr>
<td>2.5</td>
<td>DNS</td>
</tr>
<tr>
<td>2.6</td>
<td>P2P applications</td>
</tr>
<tr>
<td>2.7</td>
<td>Socket programming with TCP</td>
</tr>
<tr>
<td>2.8</td>
<td>Socket programming with UDP</td>
</tr>
</tbody>
</table>
Socket programming with UDP

UDP: no “connection” between client and server
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

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

application viewpoint:

UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server
Client/server socket interaction: UDP

Server (running on hostid)

create socket, port= x.
serverSocket = DatagramSocket()

read datagram from serverSocket

write reply to serverSocket specifying client address, port number

Client

create socket, clientSocket = DatagramSocket()

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

read datagram from clientSocket

close clientSocket
Chapter 2: Summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
  - hybrid
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, Skype
- socket programming
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 being communicated

**Important themes:**

- control vs. data msgs
  - in-band, out-of-band

- centralized vs. decentralized

- stateless vs. stateful

- reliable vs. unreliable msg transfer

- “complexity at network edge”