Chapter 2
Application Layer

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 1996-2009
J.F Kurose and K.W. Ross, All Rights Reserved
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 UDP
- 2.8 Socket programming with TCP

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 clips
- social networks
- voice over IP
- real-time video conferencing
- grid 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 UDP
- 2.8 Socket programming with TCP

Application architectures

- Client-server
  - Including data centers / cloud computing
- 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

Google Data Centers

- Estimated cost of data center: $600M
- Google spent $2.4B in 2007 on new data centers
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

- Note: 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)*
Addressing processes

- to receive messages, process must have identifier
- host device has unique 32-bit IP address
- Exercise: use `ipconfig` from command prompt to get your IP address (Windows)

Q: does IP address of host on which process runs suffice for identifying the process?

- A: No, many processes can be running on same

- Identifier includes both IP address and port numbers associated with process on host.

- Example port numbers:
  - HTTP server: 80
  - Mail server: 25

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 RFCs
- allows for interoperability
- e.g., HTTP, SMTP, BitTorrent

Proprietary protocols:

- e.g., Skype, ppstream
What transport service does an app need?

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

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

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

Security
- Encryption, data integrity, ...

<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 video:10kbps-5Mbps</td>
<td>yes, 100's msec</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>
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>
</tbody>
</table>
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 UDP
- 2.8 Socket programming with TCP

Web and HTTP

First some jargon
- 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

  host name  path name
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

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**

**Nonpersistent HTTP**
- At most one object is sent over a TCP connection.

**Persistent HTTP**
- Multiple objects can be sent over single TCP connection between client and 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

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

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

Nonpersistent HTTP issues:
- requires 2 RTTs per object
- OS overhead for each TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

Persistent HTTP
- 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

HTTP request message

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

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
```
(extra carriage return, line feed)
HTTP request message: general format

```
method  sp  URL  sp  version   cr   lf
header field name : value  cr   lf
header field name : value  cr   lf
...                                                

Entity Body
```

Uploading form input

Post method:
- 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
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
Connection close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 ...
Content-Length: 6821
Content-Type: text/html

data data data data data ...

**HTTP response status codes**

In first line in server->client response message.
A few 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 (Location:)

400 Bad Request
- request message not understood by server

404 Not Found
- requested document not found on this server

505 HTTP Version Not Supported

---

**Trying out HTTP (client side) for yourself**

1. Telnet to your favorite Web server:
   
   ```
   telnet cis.poly.edu 80
   
   Opens TCP connection to port 80
   (default HTTP server port) at cis.poly.edu.
   Anything typed in sent to port 80 at cis.poly.edu
   ```

2. Type in a GET HTTP request:
   
   ```
   GET /~ross/ HTTP/1.1
   Host: cis.poly.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!
User-server state: cookies

Many major 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 always 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.)

<table>
<thead>
<tr>
<th>client</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebay 8734</td>
<td>Amazon server</td>
</tr>
<tr>
<td>cookie file</td>
<td>creates ID</td>
</tr>
<tr>
<td>eBay 8734</td>
<td>1678 for user</td>
</tr>
<tr>
<td>cookie 1678</td>
<td>create entry</td>
</tr>
<tr>
<td>cookie: 1678</td>
<td>access</td>
</tr>
<tr>
<td>cookie file</td>
<td>backend database</td>
</tr>
<tr>
<td>eBay 8734</td>
<td>access</td>
</tr>
<tr>
<td>cookie: 1678</td>
<td>cookie-specific action</td>
</tr>
<tr>
<td>one week later:</td>
<td>cookie-specific action</td>
</tr>
<tr>
<td>eBay 8734</td>
<td>cookie: 1678</td>
</tr>
<tr>
<td>Amazon server</td>
<td>usual http response msg</td>
</tr>
<tr>
<td>cookie: 1678</td>
<td>usual http response msg</td>
</tr>
<tr>
<td>cookie file</td>
<td>usual http response msg</td>
</tr>
<tr>
<td>eBay 8734</td>
<td>usual http response msg</td>
</tr>
<tr>
<td>cookie: 1678</td>
<td>usual http response msg</td>
</tr>
<tr>
<td>Amazon server</td>
<td>usual http response msg</td>
</tr>
</tbody>
</table>
**Cookies (continued)**

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

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

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

---

**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 = 1,000,000 bits
- avg. request rate from institution's browsers to origin servers = 15/sec
- delay from institutional router to any origin server and back to router = 2 sec

Consequences
- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay = 2 sec + minutes + milliseconds
Caching example (cont)

possible solution
- increase bandwidth of access link to, say, 100 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
- suppose hit rate is 0.4

consequence
- 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^-6 m sec)
- total avg delay = Internet delay + access delay + LAN delay
  = .6*(2.01) secs + .4*milliseconds * 1.4 secs
**Conditional GET**

- **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

```
HTTP request msg
If-modified-since: <date>

HTTP response
HTTP/1.0 304 Not Modified
```

```
HTTP request msg
If-modified-since: <date>

HTTP response
HTTP/1.0 200 OK
<data>
```