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

Computer Networking:
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

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 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 clips
- 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 TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server
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

**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
- Q: does IP address of host 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 RFCs
- allows for interoperability
  - e.g., HTTP, SMTP
Proprietary protocols:
- e.g., Skype
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

Timing
- 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>
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 (eg 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 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

a host-local, application-created, OS-controlled interface (a "door") into which application process can both send and receive messages to/from another application process
Socket-programming using TCP

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

**TCP service**: reliable transfer of bytes from one process to another
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 client-local 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 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 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

write reply to
connectionSocket

TCP connection setup

close clientSocket
Stream jargon

- A **stream** is a sequence of characters that flow into or out of a process.
- An **input stream** is attached to some input source for the process, e.g., keyboard or socket.
- An **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)
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
Example: Java client (TCP), cont.

```java
BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + '\n');
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();
```
Example: Java server (TCP)

```java
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;
        ServerSocket welcomeSocket = new ServerSocket(6789);
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
            String clientSentence = inFromClient.readLine();
            String capitalizedSentence = clientSentence.toUpperCase();
            System.out.println(capitalizedSentence);
        }
    }
}
```
Example: Java server (TCP), cont

Create output stream, attached to socket

Read in line from socket

Write out line to socket

DataOutputStream outToClient =
new DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes(capitalizedSentence);

End of while loop, loop back and wait for another client connection
Example: C echo client (TCP)

```
#include <sys/types.h> /* basic system data types */
#include <sys/socket.h> /* basic socket definitions */
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr;

    if (argc != 2) {
        printf("usage: tcpcli <IPaddress>
        
        exit(-1);
    }

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
```
Example: C echo client (TCP), cont.

Fill in socket structure with server information

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(6789);
servaddr.sin_addr.s_addr = inet_addr(argv[1]);
```

Establish connection with server

```
connect(sockfd,
    (struct sockaddr *) &servaddr,
    sizeof(servaddr));
```

Work with the established socket

```
str_cli(stdin, sockfd);          /* do it all */
exit(0);
```
Example: C echo client (TCP), cont.

```c
void str_cli(FILE *fp, int sockfd)
{
    char sendline[MAXLINE], recvline[MAXLINE];
    while (fgets(sendline, MAXLINE, fp) != NULL) {
        write(sockfd, sendline, strlen(sendline));
        if (readline(sockfd, recvline, MAXLINE) == 0)
            printf("str_cli: server terminated prematurely\n");
            exit(-1);
    }
    fputs(recvline, stdout);
}

Readline is also user defined.
```
Example: C echo server (TCP)

```c
#include <sys/types.h>   /* basic system data types */
#include <sys/socket.h>  /* basic socket definitions */
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>

#define MAXLINE 1024
#define LISTENQ 16    /* max size of queue */

int main(int argc, char **argv)
{
    int   listenfd, connfd;
    pid_t   childpid;
    int   clilen;
    struct sockaddr_in   cliaddr, servaddr;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
```

Helpful includes

Define socket address structure

Create TCP socket
Example: C echo server (TCP), cont.

- Fill in structure to accept conns from any local interface
  
  ```c
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(6789);
  ```

- Assign structure to the socket
  ```c
  bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
  ```

- Convert socket to a listening socket
  ```c
  listen(listenfd, LISTENQ);
  ```

- Wait until new conn. is established
  ```c
  for ( ; ; ) {
      clilen = sizeof(cliaddr);
      connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
  }
  ```

- Work with the established conn.
  ```c
  str_echo(connfd); /* process the request */
  close(connfd);    /* close connected socket */
  }
  } /* end main */
Example: C echo server (TCP), cont.

```c
void str_echo(int sockfd)
{
    ssize_t n;
    char line[MAXLINE];

    for (; ; ) {
        if ( (n = readline(sockfd, line, MAXLINE)) == 0)
            return; /* connection closed by other end */
        write(sockfd, line, n);
    }
} /* end str_echo */
```

Get line from client
Write line back to client

Readline is also user defined.
Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 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 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:

  http://www.someschool.edu/someDept/pic.gif

  - method
  - 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 1.0: RFC 1945
- HTTP 1.1: RFC 2616

![Diagram of PC running Explorer, Server running Apache Web server, and Mac running Firefox]
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.
- HTTP/1.0 uses nonpersistent HTTP

Persistent HTTP
- Multiple objects can be sent over single TCP connection between client and server.
- HTTP/1.1 uses persistent connections in default mode
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

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

3. HTTP server receives request message, forms response message containing requested object, and sends message into its socket
Nonpersistent HTTP (cont.)

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
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
```

Carriage return, line feed only indicates end of message
HTTP request message: general format
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

- **status line**
  - protocol
  - status code
  - status phrase
  - `HTTP/1.1 200 OK\r\n      Connection close\r\n      Date: Thu, 06 Aug 1998 12:00:15 GMT\r\n      Server: Apache/1.3.0 (Unix)\r\n      Last-Modified: Mon, 22 Jun 1998...
      Content-Length: 6821\r\n      Content-Type: text/html\r\n      \r\n      data data data data data ...

- **header lines**

- **data, e.g., requested HTML file**
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!
Let’s look at HTTP in action

- telnet example
- Ethereal/Wireshark example (in lab)
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.)

One week later:

usual http request msg

cookie: 1678

usual http response msg

cookie: 1678

usual http response msg

Amazon server creates ID 1678 for user

create entry

backend database

cookie-specific action

access

access
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)
Where is Caching Performed?

Assume: there is at least one cache between user and content on server.
- Cache in client browser
- Workgroup or institutional proxy cache
- Local ISP proxy cache
- Remote ISP/CDN proxy cache (for server)

Read *A Web Caching Primer* for more details.
Caching example

Assumptions

- average object size = 100,000 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, 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
- 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 msec)
- total avg delay = Internet delay + access delay + LAN delay = 0.6*(2.01) secs + 0.4*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
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
- 2.9 Building a Web server
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.
- 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., Eudora, Outlook, elm, Mozilla Thunderbird
- 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
- 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
  - **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.
**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
```
POP3 (more) and IMAP

More about POP3

- Previous example uses “download and delete” mode.
- Bob cannot re-read email if he changes client.
- “Download-and-keep”: copies of messages on different clients.
- POP3 is stateless across sessions.

IMAP

- Keep all messages in one place: the server.
- Allows user to organize messages in folders.
- IMAP 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
- 2.9 Building a Web server
DNS: Domain Name System

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

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

Q: map between IP addresses and name?

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

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
- client queries amazon.com DNS server to get IP address for www.amazon.com
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 defined root name servers worldwide
TLD and Authoritative Servers

- **Top-level domain (TLD) servers:**
  - responsible for com, org, net, edu, etc, and all top-level country domains 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?
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 under design by IETF
  - RFC 2136
DNS records

DNS: distributed db 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
  - *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 RRs</td>
</tr>
<tr>
<td>number of authority RRs</td>
<td>number of additional RRs</td>
</tr>
<tr>
<td>questions (variable number of questions)</td>
<td></td>
</tr>
<tr>
<td>answers (variable number of resource records)</td>
<td></td>
</tr>
<tr>
<td>authority (variable number of resource records)</td>
<td></td>
</tr>
<tr>
<td>additional information (variable number of resource records)</td>
<td></td>
</tr>
</tbody>
</table>
**DNS protocol, messages**

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

- create authoritative server Type A record for www.networkuptopia.com; Type MX record for networkutopia.com
- How do people get IP address of your Web site?
DNS Tools

nslookup:
- can find IP given name
- can find name given IP
- can show other RR

whois:
- can show information about domain and owner
- can show information about “owner” of an IP address or network
- online whois servers
  - http://www.arin.net/whois/index.html
  - And others...

dig and host:
- Show more detail
- Often available on Suns, Linux
- Online:
  - And others…
Content distribution networks (CDNs)

Content replication

- challenging to stream large files (e.g., video) from single origin server in real time
- solution: replicate content at hundreds of servers throughout Internet
  - content downloaded to CDN servers ahead of time
  - placing content "close" to user avoids impairments (loss, delay) of sending content over long paths
  - CDN server typically in edge/access network
Content distribution networks (CDNs)

Content replication

- CDN (e.g., Akamai) customer is the content provider (e.g., CNN)
- CDN replicates customers' content in CDN servers.
- when provider updates content, CDN updates servers
**CDN example**

1. **origin server** (www.foo.com)
   - distributes HTML
   - replaces:
     http://www.foo.com/sports.ruth.gif
     with

2. **CDN company** (cdn.com)
   - distributes gif files
   - uses its authoritative DNS server to route redirect requests
More about CDNs

**routing requests**

- CDN creates a “map”, indicating distances from leaf ISPs and CDN nodes
- when query arrives at authoritative DNS server:
  - server determines ISP from which query originates
  - uses “map” to determine best CDN server
- CDN nodes create application-layer overlay network
Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 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
**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, size $F$

Network (with abundant bandwidth)
File distribution time: server-client

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

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

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

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: $u_s + \sum u_i$

$$d_{P2P} = \max \left\{ \frac{F}{u_s}, \frac{F}{\min(d_i)}, \frac{NF}{u_s + \sum u_i} \right\}$$
Server-client vs. P2P: example

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

- P2P file distribution

**tracker**: tracks peers participating in torrent

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

- Obtain list of peers
- Trading chunks
- Peer
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!
Distributed Hash Table (DHT)

- DHT = distributed P2P database
- Database has (key, value) pairs:
  - key: ss number; value: human name
  - key: content type; value: IP address
- Peers query DB with key
  - DB returns values that match the key
- Peers can also insert (key, value) peers
DHT Identifiers

- Assign integer identifier to each peer in range $[0, 2^n-1]$.
  - Each identifier can be represented by $n$ bits.

- Require each key to be an integer in the same range.

- To get integer keys, hash original key.
  - eg, $key = h(“Led Zeppelin IV”)$
  - This is why they call it a distributed “hash” table
How to assign keys to peers?

- Central issue:
  - Assigning (key, value) pairs to peers.

- Rule: assign key to the peer that has the closest ID.

- Convention in lecture: closest is the immediate successor of the key.

- Ex: n=4; peers: 1, 3, 4, 5, 8, 10, 12, 14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1
Circular DHT (1)

- Each peer *only* aware of immediate successor and predecessor.
- "Overlay network"
Circle DHT (2)

O(N) messages on avg to resolve query, when there are N peers

Who's resp for key 1110?

Define closest as closest successor
Each peer keeps track of IP addresses of predecessor, successor, short cuts.

Reduced from 6 to 2 messages.

Possible to design shortcuts so $O(\log N)$ neighbors, $O(\log N)$ messages in query.

Who's resp for key 1110?
Peer Churn

- To handle peer churn, require each peer to know the IP address of its two successors.
- Each peer periodically pings its two successors to see if they are still alive.

- Peer 5 abruptly leaves
- Peer 4 detects; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8’s immediate successor its second successor.
- What if peer 13 wants to join?
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; 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 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
Example: Java client (UDP)

Output: sends packet (recall that TCP sent "byte stream")

Input: receives packet (recall that TCP received "byte stream")

client UDP socket
Example: Java client (UDP)

```java
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("hostname");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
    }
}
```
Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

```java
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram to server

```java
clientSocket.send(sendPacket);
```

Read datagram from server

```java
DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
String modifiedSentence = new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
```
Example: Java server (UDP)

```java
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        while(true) {
            DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
        }
    }
}
```
Example: Java server (UDP), cont

String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress, port);

serverSocket.send(sendPacket);

End of while loop, loop back and wait for another datagram
Example: C echo client (UDP)

```c
[int main(int argc, char **argv) {
  int sockfd;
  struct sockaddr_in servaddr;

  if (argc != 2) {
    printf("usage: udpcli <IPaddress>\n");
    exit(-1);
  }
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(6789);
  servaddr.sin_addr.s_addr = inet_addr(argv[1]);
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  dg_cli(stdin, sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
  exit(0);
}
```

Set up socket parameters

Create UDP socket

Work with the established socket
Example: C echo client (UDP) cont.

```c
void
dg_cli(FILE *fp, int sockfd, const struct sockaddr *pservaddr, int servlen)
{
    int n;
    char sendline[MAXLINE], recvline[MAXLINE + 1];

    while (fgets(sendline, MAXLINE, fp) != NULL) {
        sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
        n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
        recvline[n] = 0;        /* null terminate */
        fputs(recvline, stdout);
    }
}
```
Example: C echo server (UDP)

```c
#include <sys/types.h>   /* basic system data types */
#include <sys/socket.h>  /* basic socket definitions */
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>
#define MAXLINE 1024

int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(6789);

    bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
    dg_echo(sockfd, (struct sockaddr *) &cliaddr, sizeof(cliaddr));
}
```
Example: C echo server (UDP) cont.

```c
void
dg_echo(int sockfd, struct sockaddr *pcliaddr, int clilen)
{
    int n;
    int len;
    char mesg[MAXLINE];

    for (; ; ) {
        len = clilen;
        n = recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
        sendto(sockfd, mesg, n, 0, pcliaddr, len);
    }
}
```
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”