2. Application Layer
2. Application layer: Outline

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 and TCP
2. Application layer: Goals

**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
- creating network applications
  - socket API
Some network apps

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

possible structure of applications:
- client-server
- peer-to-peer (P2P)
Client-server architecture

server:
- always-on host
- permanent IP address
- data centers for scaling

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

- no always-on server
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
Processes communicating

**process**: program running within a host

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

**clients, servers**

**client process**: process that initiates communication

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

- aside: even P2P applications have client processes & server processes
Sockets

- process sends/receives messages to/from its socket
- socket analogous to a dropbox at door
  - sending process shoves message into dropbox
  - sending process relies on transport to deliver message to dropbox at receiving 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 www.cs.umass.edu web server:
  - IP address: 128.119.240.84
  - 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

open protocols:
- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:
- e.g., Skype
What transport service does an app need?

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

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, ...

Application Layer 2-13
## Transport service requirements: 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, 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></td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>few kbps up</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>text messaging</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no, 100s msec</td>
</tr>
</tbody>
</table>
Common Internet transport services

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

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

_Q: Why bother? Why is there a UDP?_
<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>TCP or UDP</td>
</tr>
</tbody>
</table>

**Q:** Why might Skype use TCP?
Securing TCP

TCP & UDP
- no encryption
- cleartext passwds sent into socket traverse Internet in cleartext

SSL
- provides encrypted TCP connection
- data integrity
- end-point authentication

SSL is at app layer
- Apps use SSL libraries, which “talk” to TCP

SSL socket API
- cleartext passwds sent into socket encrypted before transmission
- See Chapter 7
Q1: TCP vs. UDP

Which of the following is true?

A. FTP uses UDP
B. HTTP uses UDP
C. UDP ensures in-order delivery but not reliability
D. HTTP uses TCP
Q2 Endpoint process identifier

- A network application process is identified uniquely by which of the following?
  A. IP address
  B. IP address, port
  C. IP address, port, MAC address
  D. domain name
Q3 Transport

Pick the true statement

A. TCP provides reliability and guarantees a minimum bandwidth.
B. TCP provides reliability while UDP provides bandwidth guarantees.
C. TCP provides reliability while UDP does not.
D. Neither TCP nor UDP provide reliability.
Q4 HTTP

- Persistent HTTP fetches multiple web objects over a single TCP connection while non-persistent HTTP uses a separate TCP connection for each object. True/false?

A. True
B. False
2. Application layer: Outline

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 and TCP
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, e.g.,

  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, (using HTTP protocol) and “displays” Web objects
  - **server**: Web server sends (using HTTP protocol) objects in response to requests

PC running Firefox browser

server running Apache Web server

iphone running Safari browser
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
  - cookies an exception

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
  - connection then closed
- downloading multiple objects required multiple connections

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

(contains text, references to 10 jpeg images)
Non-persistent 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

RTT (definition): time for a small packet to travel from client to server and back

HTTP 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
- non-persistent HTTP response time = 2RTT + file transmission time
**Persistent HTTP**

**non-persistent 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 /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
```

carriage return character
line-feed character
request line
(GET, POST,
HEAD commands)
header lines
carriage return,
line feed at start
of line indicates
end of header lines
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>value</td>
<td>cr</td>
<td>lf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>header field name</td>
<td>value</td>
<td>cr</td>
<td>lf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cr</td>
<td>lf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- status line (protocol status code status phrase)
- header lines
- data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\nDate: Sun, 26 Sep 2010 20:09:20 GMT\r\nServer: Apache/2.0.52 (CentOS)\r\nLast-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\nETag: "17dc6-a5c-bf716880"\r\nAccept-Ranges: bytes\r\nContent-Length: 2652\r\nKeep-Alive: timeout=10, max=100\r\nConnection: Keep-Alive\r\nContent-Type: text/html; charset=ISO-8859-1\r\n\r\ndata data data data data data data ...
```
HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:
  
  200 OK
  - request succeeded, requested object later in this 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
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!

   (or use Wireshark to look at captured HTTP request/response)
many Web sites use cookies

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

example:
- Susan always access Internet from her 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
- subsequent HTTP requests carry cookie
Cookies: keeping “state” (cont.)

One week later:

- eBay Server
  - eBay 8734
  - Cookie file
  - Usual HTTP request msg
  - Usual HTTP response msg
  - Cookie: 1678
  - Usual HTTP response msg

- Amazon Backend Database
  - Amazon 1678
  - Create entry
  - Set-cookie: 1678
  - Usual HTTP request msg
  - Usual HTTP response msg
  - Cookie: 1678
  - Usual HTTP response msg
  - Cookie-specific action
  - Access

Application Layer 2-39
Cookies (continued)

**cookies uses:**
- 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

**“stateful” protocols:**
- protocol endpoints maintain state at sender/receiver over multiple transactions
  - cookies in 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
  - if 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
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

**why Web caching?**

1. reduce response time for client request
2. reduce traffic on an institution’s access link
3. reduce server load (as does P2P file sharing)
Caching example:

**assumptions:**
- avg object size: $S=100K$ bits
- avg request rate from browsers to origin servers: $A=15$/sec
- avg data rate to browsers: $R=1.50$ Mbps
- access link rate: $C=1.54$ Mbps
- RTT from institutional router to any origin server: $T=200$ ms

**consequences:**
- LAN utilization: $0.15\%$
- access link utilization $\approx 99\%$
- total delay = Internet delay + access delay + LAN delay
  $= 200$ ms + $\approx$ minutes + $\mu$secs
Caching example: fatter access link

assumptions:
- avg object size: $S=100K$ bits
- avg request rate from browsers to origin servers: $A=15$/sec
- avg data rate to browsers: $R=1.50$ Mbps
- access link rate: $C=1.54$ Mbps
- RTT from institutional router to any origin server: $T=200$ ms

consequences:
- LAN utilization: 0.15% (as before)
- access link utilization = 99% or 9.9%
- total delay = Internet delay + access delay + LAN delay
  = 200 ms + ~minutes + usecs
  ≈ ms

Cost: increased access link speed (not cheap!)
Caching example: install local cache

assumptions:
- avg object size: \( S = 100K \) bits
- avg request rate from browsers to origin servers: \( A = 15/\text{sec} \)
- avg data rate to browsers: \( R = 1.50 \) Mbps
- access link rate: \( C = 1.54 \) Mbps
- RTT from institutional router to any origin server: \( T = 200 \) ms

consequences:
- LAN utilization: 0.15% (as before)
- access link utilization = ?
- total delay = ?

How to compute link utilization, delay?

Cost: web cache (cheap!)
assumptions:
- avg object size: $S = 100K$ bits
- avg request rate from browsers to origin servers: $A = 15$/sec
- avg data rate to browsers: $R = 1.50$ Mbps
- access link rate: $C = 1.54$ Mbps
- RTT from institutional router to any origin server: $T = 200$ ms
Caching example: install local cache

Calculating access link utilization, delay with cache:

- Suppose cache hit rate is 0.4
  - p = 40% requests satisfied at cache, 60% (=1-p) satisfied at origin

- Access link utilization:
  - 60% of request data rate

- Data rate to browsers over access link:
  - \( pR = 0.6 \times 1.50 \text{ Mbps} = 0.9 \text{ Mbps} \)
  - Utilization \( u = \frac{pR}{C} = \frac{0.9}{1.54} = 0.58 \)
  - Transmission delay \( d = \frac{S}{C} = 0.067s \)
  - Queuing delay \( q = \frac{S}{C}/(1-u) = 0.16s \)

- Total delay
  - \((1-p)\text{miss\_delay} + p\text{hit\_delay}\)
  - \((1-p)*(\text{delay from origin servers}) + p*(\text{delay when satisfied at cache})\)
  - \((1-p)*(T+d+q) + p*(?)\)

- Benefit: Lower latency without costly upgrade!
- \( \approx 250\text{ms} \)
**Conditional GET**

- **Goal:** don’t send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- **cache:** specify date of cached copy in HTTP request
  - `If-modified-since: <date>`
- **server:** response contains no object if cached copy is up-to-date:
  - `HTTP/1.0 304 Not Modified`

```
HTTP 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>
```

Application Layer 2-48
Q1: HTTP conn. persistence

Which of the following is true about persistent HTTP compared to non-persistent HTTP

A. Persistent HTTP improves throughput using more connections.
B. Persistent HTTP improves download time by reducing the number of connection setup round trips.
C. Persistent HTTP improves throughput by sending fewer HTTP requests.
D. Persistent HTTP improves download time by sending fewer HTTP requests.
Q2: HTTP conn. persistence

Among the following, in which case would you get the greatest improvement in performance with persistent HTTP compared to non-persistent?

A. Low capacity (bits/sec) network paths
B. High capacity network paths
C. Long-distance network paths
D. High capacity, short-distance network paths
E. High capacity, long-distance network paths
Q3: Web caching

If the cache captured a fraction $p=0.3$ of requests, what is the average delay contributed by transmission delays alone (i.e., no queuing) for each object? Ignore LAN transmission delays.

A. $S/C$
B. $pS/C$
C. $(1-p)S/C$
D. $(p + AS/C)(S/C)$
E. $(1-p)(AS/C)(S/C)$

**assumptions:**
- Avg object size: $S=100K$ bits
- Avg request rate from browsers to origin servers: $A=15/\text{sec}$
- Access link rate: $C=1.54$ Mbps
- RTT from institutional router to any origin server: $T=200$ ms
Q4 HTTP download time

Consider a web page with a base file of size $S_0$ bits and N inline objects each of size $S$ bits being downloaded by a client over a link of capacity $C$ bits/sec and RTT $T$. How much time is saved by using persistent HTTP compared to non-persistent assuming requests for all inline objects are sent in a pipelined manner?

A. $T$
B. $T(2N-1)$
C. $NT + S/C$
D. $T + NS/C$
E. $T(N-1)$
Q5 HTTP download time

- Consider a web page with a base file of size $S_0$ bits and $N$ inline objects each of size $S$ bits being downloaded by a client over a link of capacity $C$ bits/sec and RTT $T$. How much time is saved by using persistent HTTP compared to non-persistent assuming requests for all inline objects are sent in a sequential manner, i.e., a request for the next object is sent after the previous object has been completely received?

A. $T$
B. $NT$
C. $(2N-1)T$
D. $2NT$
Q6 HTTP download time

Consider a web page with a base file of size $S_0$ bits and $N$ inline objects each of size $S$ bits being downloaded by a client over a link of capacity $C$ bits/sec and RTT $T$ bits/sec and RTT $T$. How much time will persistent HTTP (with pipelined requests) take if it used two parallel connections? Assume both connections are set up in parallel at the start, they share the available capacity equally, and inline objects are equally split across them.

A. $(2T + S_0) + T + NS/C$
B. $(2T + S_0) + NT + NS/C$
C. $(2T + S_0) + NS/C$
2. Application layer: Outline

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 and TCP
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, using TCP
- client authorized over control connection
- client browses remote directory, sends commands over control connection
- when server receives file transfer command, server opens 2\textsuperscript{nd} TCP data 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
2. Application layer: Outline

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 and TCP
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, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server

Diagram:
- User agents (email clients)
- Mail servers
- SMTP protocol
- User mailbox
- Outgoing message queue
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
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP, FTP)
  - 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 “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 vs HTTP

SMTP
- persistent connections
- 7-bit ASCII request/response + status codes
- CRLF CRLF for end of message
- Push
- Multiple objects sent in multipart message

HTTP
- persistent or non-persistent
- ASCII request/response + status codes
- CRLF or CRLF CRLF for end of message
- Pull
- Single object encapsulated in its own response message
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 MAIL FROM, RCPT TO: 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, download
- **IMAP**: Internet Mail Access Protocol [RFC 1730]: more features, including 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 POP3 “download and delete” mode
  - Bob cannot re-read e-mail if he changes client
- POP3 “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
2. Application layer: Outline

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 and TCP
DNS: domain name system

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

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

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

**Domain Name System:**
- **distributed database** implemented in hierarchy of many name servers
- **application-layer protocol**: hosts, name servers communicate to resolve names → addresses
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network’s “edge”
DNS: services, structure

**DNS services**
- Resolution
  - hostname → IP address
- Aliasing
  - canonical, alias names
  - mail server aliasing
- Load balancing with replicated web servers:
  - many IP addresses correspond to one name

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

A: *doesn’t scale!*
DNS: a distributed, hierarchical database

client wants IP for www.amazon.com; 1st approx:
- client queries root server to find .com TLD DNS server
- client queries .com TLD DNS server for amazon.com auth server
- client queries amazon.com DNS auth server to get IP address for www.amazon.com
DNS: root name servers

- contacted when no info about top-level or auth server
- root name server can:
  - return top-level or auth name server address
  - or contact auth server and return final resolved address

13 root name "servers" worldwide

- a. Verisign, Los Angeles CA (5 other sites)
- b. USC-ISI Marina del Rey, CA
- c. Cogent, Herndon, VA (5 other sites)
- d. U Maryland College Park, MD
- e. NASA Mt View, CA (5 other sites)
- f. Internet Software C. Palo Alto, CA (and 48 other sites)
- g. US DoD Columbus, OH (5 other sites)
- h. ARL Aberdeen, MD
- i. Netnod, Stockholm (37 other sites)
- j. Verisign, Dulles VA (69 other sites)
- k. RIPE London (17 other sites)
- l. ICANN Los Angeles, CA (41 other sites)
- m. WIDE Tokyo (5 other sites)
TLD, authoritative servers

top-level domain (TLD) servers:

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

authoritative DNS servers:

- organization’s own DNS server(s), providing authoritative hostname to IP mappings for organization’s named hosts
- can be maintained by organization or service provider
Local DNS name server

- does not strictly belong to hierarchy
- deployed by ISP (residential, company, university)
  - also called “default name server”
- acts as proxy between host and DNS hierarchy
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
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 at upper levels of hierarchy?
DNS: caching, updating records

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

**DNS**: distributed 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
- **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**

- *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></th>
<th>identification</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td># questions</td>
<td># answer RRs</td>
<td></td>
</tr>
<tr>
<td># authority RRs</td>
<td># additional RRs</td>
<td></td>
</tr>
<tr>
<td>questions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>answers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>authority</td>
<td></td>
</tr>
<tr>
<td></td>
<td>additional info</td>
<td></td>
</tr>
</tbody>
</table>
### DNS protocol, messages

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td># questions</td>
<td># answer RRs</td>
</tr>
<tr>
<td># authority RRs</td>
<td># additional RRs</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**

*2 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)
- create authoritative server type A record for www.networkuptopia.com; type MX record for networkutopia.com
Attacking DNS

DDoS attacks

- Bombard root servers with traffic
  - Not successful to date
  - Traffic Filtering
  - Local DNS servers cache IPs of TLD servers, bypassing root
- Bombard TLD servers
  - Potentially more dangerous

Redirect attacks

- Man-in-middle
  - Intercept queries
- DNS poisoning
  - Send bogus replies to DNS server that caches

Exploit DNS for DDoS

- Send queries with spoofed source address: target IP
- Requires amplification
Q1: HTTP vs. FTP

Which of the following is not true?

A. HTTP and FTP are client-server protocols
B. HTTP separates control and data across two connections while FTP does not
C. FTP separates control and data across two connections while HTTP does not
D. Both HTTP and FTP use multiple connections to complete typical user operations
E. Both HTTP and FTP allow clients to upload (send) as well as download (receive) data
Q2: HTTP vs SMTP

- Which of the following is not true?
  A. HTTP is pull-based, SMTP is push-based
  B. HTTP uses a separate header for each object, SMTP uses a multipart message format
  C. SMTP uses persistent connections
  D. HTTP uses client-server communication but SMTP does not
Q3: Mail agent protocols

- Which of the following is not a difference between POP3 and IMAP?
  A. Session state maintenance
  B. Folders
  C. Use of TCP
Q4: DNS

Which one of the following pairs are respectively maintained by the client-side ISP and the domain name owner?

A. Local, Authoritative
B. Root, Top-level domain
C. Root, Local
D. Top-level domain, authoritative
E. Authoritative, Top-level domain
2. Application layer: Outline

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 and TCP
P2P architecture

- no always-on server
- arbitrary host-host communication
- intermittent connectivity with changing IP addresses

examples:
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)
File distribution: client-server vs P2P

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

- $u_s$: server upload capacity
- $u_i$: peer $i$ upload capacity
- $d_i$: peer $i$ download capacity
- $d_N$: total download capacity of all peers
- $u_N$: total upload capacity of all peers

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

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

\[
D_{cs} \geq \max\{NF/u_s , F/d_{min}\}
\]

- Increases linearly in \( N \)
File distribution time: P2P

- **server transmission**: must upload at least one copy
  - time to send one copy: $F/u_s$

- **client**: each client must download file copy
  - min client download time: $F/d_{\text{min}}$

- **clients**: as aggregate must download $NF$ bits
  - max upload rate (limiting max download rate) is $u_s + \Sigma u_i$

\[
D_{P2P} \geq \max\{F/u_s,F/d_{\text{min}},NF/(u_s + \Sigma u_i)\}
\]

---

Application Layer 2-94
Client-server vs. P2P: example

client upload rate = $u$, $F/u = 1$ hour, $u_s = 10u$, $d_{\text{min}} \geq u_s$
P2P: BitTorrent and precursors
P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

**tracker**: tracks peers participating in torrent

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

Alice arrives …
… obtains list of peers from tracker
… and begins exchanging file chunks with peers in torrent
P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)

- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- **churn**: peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent
BitTorrent: requesting, sending file chunks

**requesting chunks:**
- at any given time, different peers have different chunks
- periodically, Alice asks each peer for their list of chunks
- Alice seeks missing chunks from peers, rarest first

**sending chunks: tit-for-tat**
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, start sending
  - “optimistically unchoke” this peer
  - newly chosen peer may join top 4
BitTorrent: tit-for-tat

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

higher upload rate: find better trading partners, get file faster!
Distributed Hash Table (DHT)
Distributed Hash Table (DHT)

- DHT: a distributed P2P database
- database has \((key, value)\) pairs; examples:
  - key: ss number; value: human name
  - key: movie title; value: peer IP address
- Distribute the \((key, value)\) pairs over the (millions of peers)
- a peer queries DHT with key
  - DHT returns values that match the key
- peers can also insert \((key, value)\) pairs
Q: how to assign keys to peers?

- central issue:
  - assigning (key, value) pairs to peers.

- basic idea:
  - convert each key to an integer
  - assign integer to each peer
  - put (key, value) pair in the peer that is closest to the key
DHT Identifiers

- assign n-bit integer identifier to each peer in range \([0,2^n-1]\) for some \(n\).
- require each key to be an integer in same range
- to get integer key, hash original key, e.g., key = hash("Led Zeppelin IV")
Assign keys to peers

- rule: assign key to the peer that has the closest ID.
- convention: closest is the immediate successor of the key if no peer exists
- e.g., $n=4$; peers: 1,3,4,5,8,10,12,14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1
Simplistic circular DHT

- “Overlay” network where each peer *only* aware of immediate successor and predecessor.
Simplistic circular DHT

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

Define closest as closest successor

Who’s responsible for key 1111?
Circular DHT with shortcuts

- 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 responsible for key 1111?
Peer churn

handling peer churn:
- each peer knows address of its two successors
- each peer periodically pings its two successors to check aliveness
- if immediate successor leaves, choose next successor as new immediate successor

example: peer 5 abruptly leaves
- peer 4 detects peer 5 departure; 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?
Q1: What protocol?

- When your mail client contacts a mail server like “mail.cs.umass.edu”, what does it use to infer the address of this server?
  
  A. IMAP  
  B. SMTP  
  C. POP3  
  D. DNS  
  E. HTTP
Q2: What protocol?

- What transport protocol does DNS use for requests and responses?
  A. TCP
  B. UDP
  C. HTTP
Q3: P2P

- BitTorrent is typically used as a hybrid P2P + client-server system.
  
  A. True
  B. False
Q4: P2P

- BitTorrent uses tit-for-tat in each round to
  A. Determine which chunks to download
  B. Determine from which peers to download chunks
  C. Determine to which peers to upload chunks
  D. Determine which peers to report to the tracker as uncooperative
  E. Determine whether or how long it should stay after completing download
Q5: Ideal P2P

With a server of upload capacity $C$ and $K$ clients with uniform upload capacity $U$ and uniform download capacity $D$, how much time does it take for an ideal P2P system to transmit a file of size $S$ to all $K$ clients?

A. $\max\left(\frac{S}{D}, \frac{S}{C}, \frac{KS}{C+KD}\right)$
B. $\frac{KS}{C}$
C. $\min\left(\frac{S}{C}, \frac{S}{U}, \frac{S}{D}\right)$
D. $\max\left(\frac{S}{C}, \frac{S}{D}, \frac{S}{C/K+U}\right)$
E. $\frac{KS}{C+KD+KU}$
Q6: DHT

- Which of the following is not true?
  A. DHTs distribute portions of a hash table across peers.
  B. The key corresponding to an object (e.g., movie) depends on the current number of peers.
  C. Which peer is responsible for an object depends on the current number of peers.
Q7: DHT

- In a circular DHT with N peers and M objects where each peer maintains a pointer only to its immediate neighbors, the arrival or departure of a single peer
  A. Causes a constant number of peers to update a constant amount of routing information
  B. Causes $O(N)$ peers to update a constant amount of routing information
  C. Causes $O(N)$ peers to update $O(M)$ routing information
  D. Causes a constant number of peers to update $O(M)$ routing information
2. Application layer: Outline

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 and TCP
**Socket programming**

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

**socket:** dropbox between application process and end-end-transport protocol
Socket programming

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

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

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

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

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

**Server (running on serverIP)**

- create socket, port= x:
  - serverSocket = DatagramSocket(x)
- 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 specifying client address, port number
- close clientSocket
import java.io.*;
import java.net.*;

class UDPCClient {
    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();

        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)

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

\[
\text{DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);}
\]

send datagram to server

\[
\text{clientSocket.send(sendPacket);}
\]

DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);

read datagram from server

\[
\text{clientSocket.receive(receivePacket);}
\]

\[
\text{String modifiedSentence = new String(receivePacket.getData());}
\]

\[
\text{System.out.println("FROM SERVER:" + modifiedSentence);}
\]

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);

            // Process received packet here...
        }
    }
}
```
Example: Java server (UDP)

```java
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);
```

- get IP addr, port #, of sender
- create datagram to send to client
- write out datagram to socket
- end of while loop, loop back and wait for another datagram
Example app: UDP client

Python UDPC client

- include Python's socket library
- create UDP socket for server
- get user keyboard input
- Attach server name, port to message; send into socket
- read reply characters from socket into string
- print out received string and close socket

```python
import socket

serverName = 'hostname'
serverPort = 12000

clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

message = raw_input('Input lowercase sentence: ')

clientSocket.sendto(message, (serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

print modifiedMessage

clientSocket.close()
```
Example app: UDP server

Python UDPServer

```python
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(("", serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

- create UDP socket
- bind socket to local port number 12000
- loop forever
- Read from UDP socket into message, getting client’s address (client IP and port)
- send upper case string back to this client
Socket programming with TCP

client must contact server
- server must be first running
- server must have created socket (dropbox) that welcomes client’s contact

client connects to server by:
- creating TCP socket, specifying IP address, port number of server process
- client socket is now bound to that specific server

server accepts connect by:
- creating new connection-specific socket
- allows server to talk with multiple clients

application viewpoint:
TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server
Client/server socket interaction: TCP

**server** *(running on hostid)*

- create socket, port=x, for incoming request:
  - `serverSocket = ServerSocket()`
- wait for incoming connection request
  - `connectionSocket = serverSocket.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`
Example: Java client (TCP)

```java
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));

        String sentence;
        String modifiedSentence;

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

        DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
    }
}
```
Example: Java client (TCP)

```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();
```

create input stream attached to socket

send line to server

read line from server

close socket (clean up behind yourself!)
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();

            capitalizedSentence = clientSentence.toUpperCase();

            System.out.println(capitalizedSentence);
        }
    }
}
Example: Java server (TCP)

create output stream, attached to socket → DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

read in line from socket → clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '
';

write out line to socket → outToClient.writeBytes(capitalizedSentence);

end of while loop, loop back and wait for another client connection
**Example app: TCP client**

**Python TCPClient**

```python
import socket

serverName = 'servername'
serverPort = 12000

clientSocket = socket.socket(socket.AF_INET,
                             socket.SOCK_STREAM)

clientSocket.connect((serverName, serverPort))

sentence = raw_input('Input lowercase sentence: ')

clientSocket.send(sentence)

modifiedSentence = clientSocket.recv(1024)

print 'From Server:', modifiedSentence

clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port
Example app: TCP server

**Python TCPServer**

```python
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

- create TCP welcoming socket
- server begins listening for incoming TCP requests
- loop forever
- server waits on accept() for incoming requests, new socket created on return
- read bytes from socket (but not address as in UDP)
- close connection to this client (but *not* welcoming socket)
2. Application layer: Summary

Our study of network apps now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, DHT
- socket programming: TCP, UDP sockets
2. Application layer: 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”