Telematics 1

Chapter 9
Internet Application Layer

- Principles of network applications
- Important application protocols
- Socket programming

Acknowledgement: Most of these slides have been prepared by J.F. Kurose and K.W. Ross with some additions compiled from other sources

Chapter 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 Applications

- ❑ E-mail
- ❑ Web
- ❑ Instant messaging
- ❑ Remote login
- ❑ P2P file sharing
- ❑ Multi-user network games
- ❑ Streaming stored video clips
- ❑ Internet telephone
- ❑ Real-time video conference
- ❑ Massive parallel computing

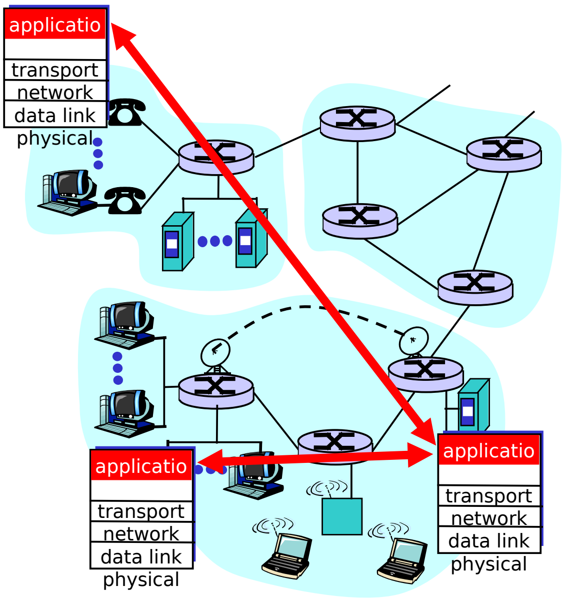
Creating a Network Application

Write programs that

- ❑ Run on different end systems and
- ❑ Communicate over a network.
- ❑ E.g., Web: Web server software communicates with browser software

No software written for devices in network core

- ❑ Network core devices do not function at app layer
- ❑ This design allows for rapid app development



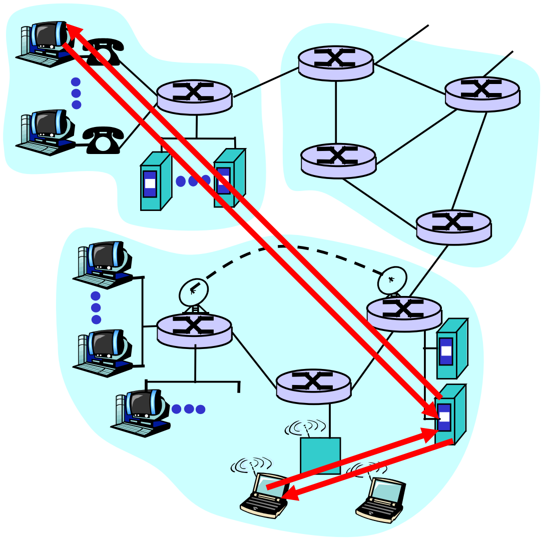
Chapter 1: Application Layer

- ❑ Principles of network applications
- ❑ Web and HTTP
- ❑ FTP
- ❑ Electronic Mail
  - ❑ SMTP, POP3, IMAP
- ❑ DNS
- ❑ P2P file sharing
- ❑ Socket programming with TCP
- ❑ Socket programming with UDP
- ❑ Building a Web server

Principles of Network Applications: Architectures

- ❑ Principle alternatives:
  - ❑ 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:

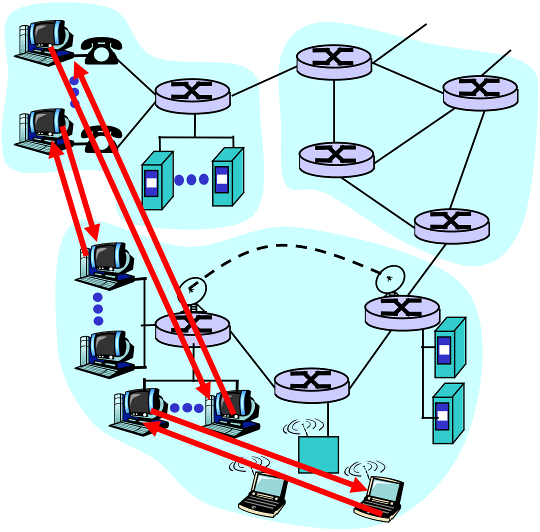
- ❑ communicate with server
- ❑ may be intermittently connected
- ❑ may have dynamic IP addresses
- ❑ do not communicate directly with each other

Pure P2P Architecture

- ❑ No always on server
- ❑ Arbitrary end systems directly communicate
- ❑ Peers are intermittently connected and change IP addresses
- ❑ Example: Gnutella

Highly scalable

But difficult to manage



Hybrid of Client-Server and P2P

(Original) Napster file sharing

- ❑ File transfer P2P
- ❑ File search centralized:
  - ■ Peers register content at central server
  - ■ Peers query same central server to locate content

Instant messaging

- ❑ Chatting between two users is P2P
- ❑ Presence detection/location centralized:
  - ■ 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

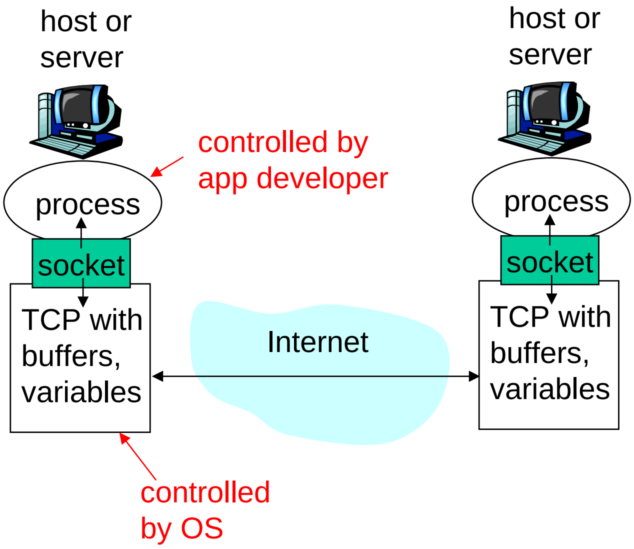
Client process: process that initiates communication

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


host or server
process
socket
TCP with buffers, variables
controlled by app developer
Internet
host or server
process
socket
TCP with buffers, variables
controlled by OS

- ❑ Application programming interface (API):
  (1) choice of transport protocol;
  (2) ability to fix a few parameters (lots more on this later)

Addressing Processes

- ❑ For a process to receive messages, it must have an identifier
- ❑ A host has a unique32-bit IP address
- ❑ Q: does the IP address of the host on which the process runs suffice for identifying the process?
- ❑ Answer: No, many processes can be running on same host

- ❑ Identifier includes both the IP address and port numbers associated with the process on the host.
- ❑ Example port numbers:
  - ❑ HTTP server: 80
  - ❑ Mail server: 25
- ❑ More on this later

Issues Defined by an Application-Layer Protocol

- Types of messages exchanged, e.g. request & response messages
- Syntax of message types: what fields in messages & how fields are delineated
- Semantics of the fields, ie, meaning of information in fields
- Rules for when and how processes send & respond to messages

Open vs. Proprietary protocols:

- Public-domain protocols:
  - open specification available to everyone
  - allows for interoperability
  - most protocols commonly used in the Internet are defined in RFCs
  - e.g. HTTP, FTP, SMTP
- Proprietary protocols:
  - defined by a vendor
  - specification often not publicly available
  - eg, KaZaA

What Transport Service does an Application 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”

Bandwidth

- Some apps (e.g., multimedia) require minimum amount of bandwidth to be “effective”
- Other apps (“elastic apps”) make use of whatever bandwidth they get

Transport Service Requirements of Common Applications

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
instant messaging	no loss	elastic	yes and no

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 bandwidth guarantees

UDP service:

- Unreliable data transfer between sending and receiving process
- Does not provide:
  - Connection setup,
  - Reliability,
  - Flow & congestion control,
  - Timing, or bandwidth guarantee

Q: Why bother?
Why is there a UDP?

Internet Applications & Transport Protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	proprietary (e.g. RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Dialpad)	typically UDP

Chapter 1: Application Layer

- Principles of network applications
- Web and HTTP
- FTP
- Electronic Mail
  - SMTP, POP3, IMAP
- DNS
- P2P file sharing
- Socket programming with TCP
- Socket programming with UDP
- Building a Web server

Web and HTTP

First some jargon

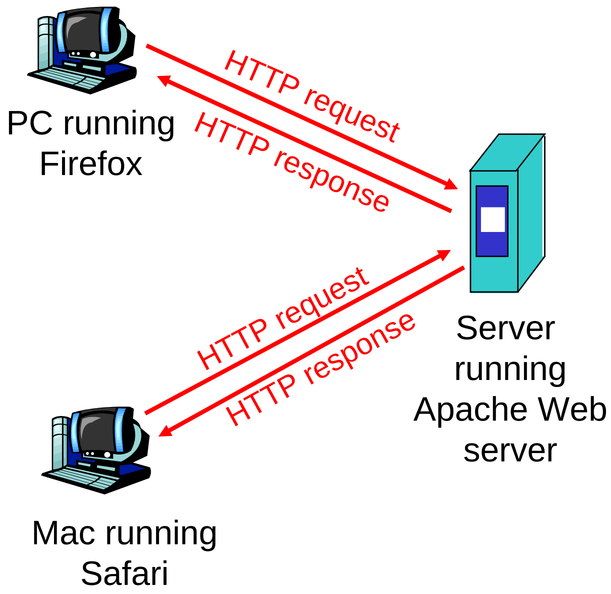
- Web page consists of objects
- Object can be HTML file, JPEG image, Java applet, audio file,…
- Web page consists of base HTML-file which includes several referenced objects
- Each object is addressable by a URL
- Example URL:

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

HTTP Overview

HTTP: Hypertext Transfer Protocol

- Web’s application layer protocol
- Client/Server model
  - *Client:* 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 2068



HTTP Overview (continued)

Uses TCP:

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

HTTP is “stateless”

- ❑ Server maintains no information about past client requests

aside

Protocols that maintain “state” are complex!

- ❑ Past history (state) must be maintained
- ❑ If server/client crashes, their views of “state” may be inconsistent, must be reconciled

HTTP Connections

Nonpersistent HTTP

- ❑ At most one object is sent over a TCP connection.
- ❑ HTTP/1.0 uses nonpersistent HTTP

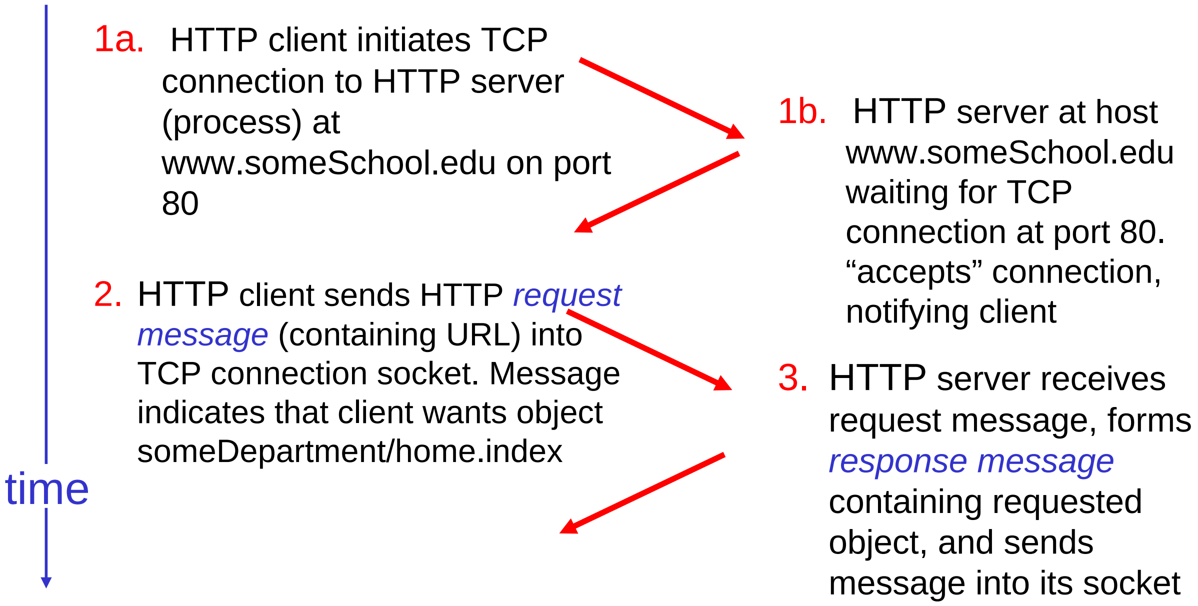
Persistent HTTP

- ❑ 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`**

(contains text, references to 10 jpeg images)



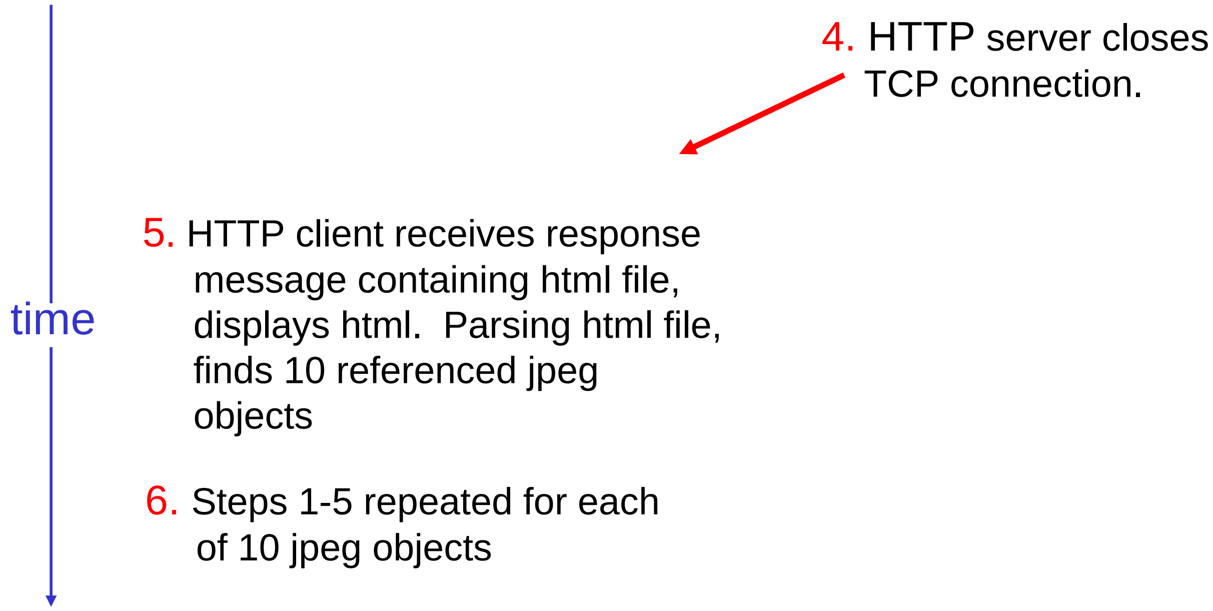
1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80

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

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

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

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

Response Time Modeling

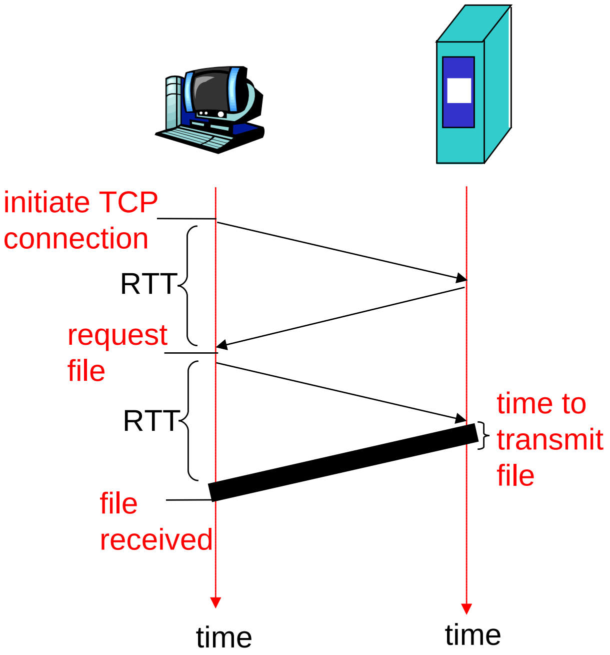
Definition of RTT:

- time to send 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 must work and allocate host resources for each TCP connection
- But 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 are sent over connection

Persistent without pipelining:

- Client issues new request only when previous response has been received
- One RTT for each referenced object

Persistent with pipelining:

- Default in HTTP/1.1
- Client sends requests as soon as it encounters a referenced object
- As little as one RTT for all the referenced objects

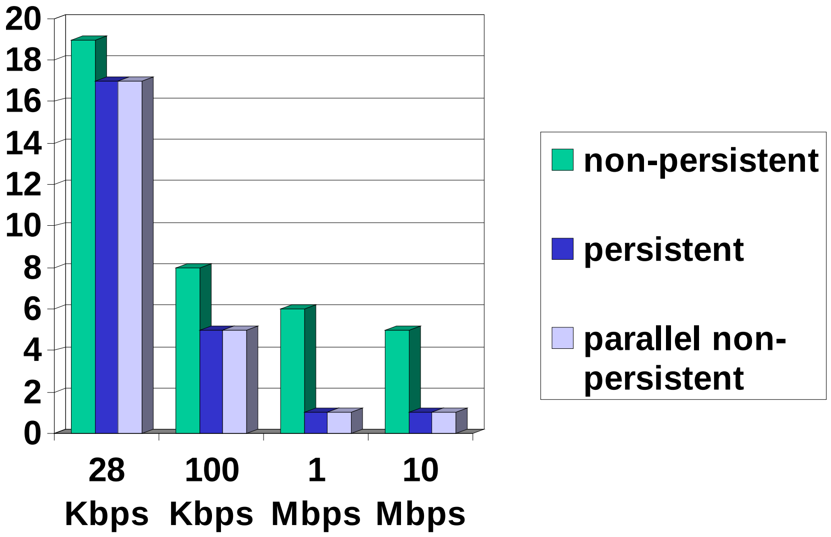
HTTP Response Time Modeling (incl. TCP Dynamics)

- Assume Web page consists of:
  - *1* base HTML page (of size *O* bits)
  - *M* images (each of size *O* bits)
- Non-persistent HTTP (one connection per object):
  - *M+1* TCP connections in series
  - *Response time = (M+1)O/R + (M+1)2RTT + sum of idle times*
- Persistent HTTP (one connection for all objects):
  - *2 RTT* to request and receive base HTML file
  - *1 RTT* to request and receive M images
  - *Response time = (M+1)O/R + 3RTT + sum of idle times*
- Non-persistent HTTP with X parallel connections
  - Suppose M/X integer.
  - 1 TCP connection for base file
  - M/X sets of parallel connections for images.
  - *Response time = (M+1)O/R + (M/X + 1)2RTT + sum of idle times*

HTTP Response Time Modeling (incl. TCP Dynamics)

RTT = 100 msec, O = 5 Kbytes, M=10 and X=5

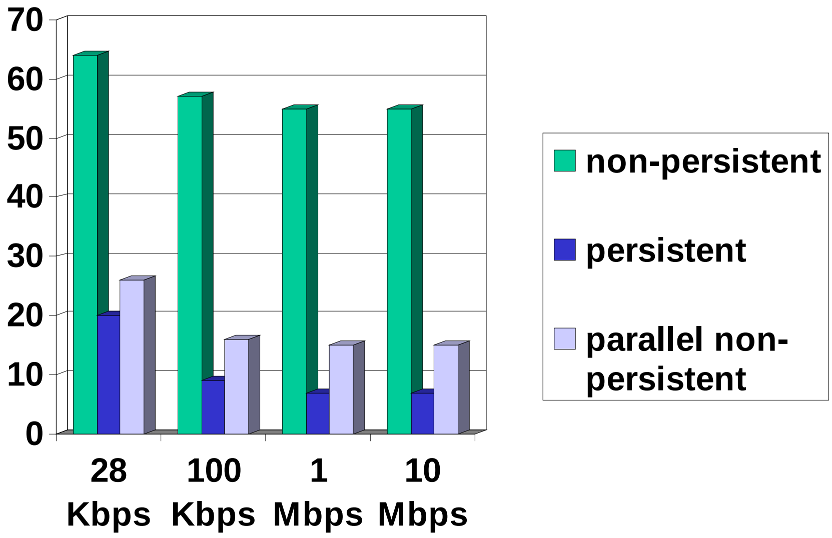
- For low bandwidth, connection & response time dominated by transmission time.
- Persistent connections only give minor improvement over parallel connections.
- Y-axis shows response time in seconds



HTTP Response Time Modeling (incl. TCP Dynamics)

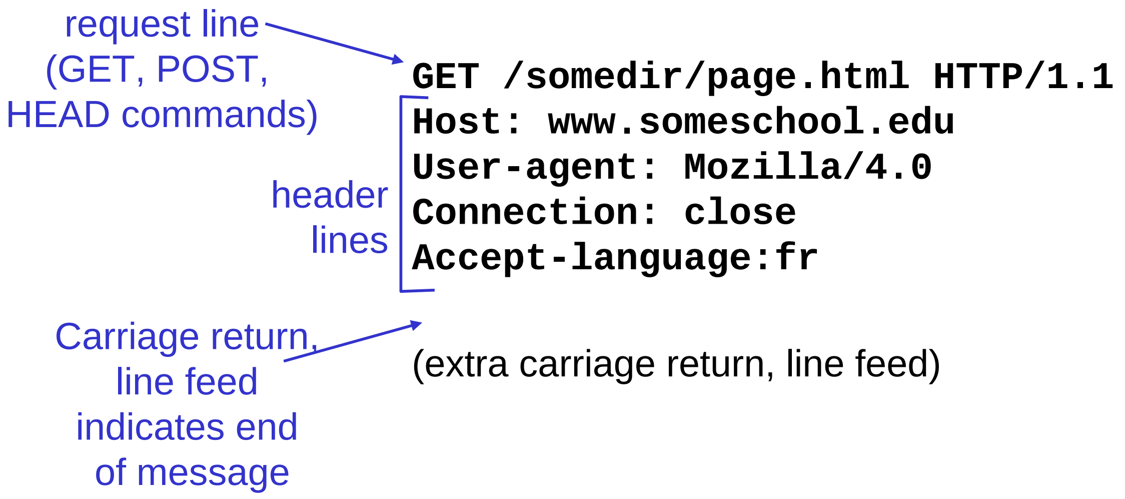
RTT =1 sec, O = 5 Kbytes, M=10 and X=5

- For larger RTT, response time dominated by TCP establishment & slow start delays.
- Persistent connections now give important improvement:
- Particularly in high delay•bandwidth networks



HTTP Request Message

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



request line (GET, POST, HEAD commands)

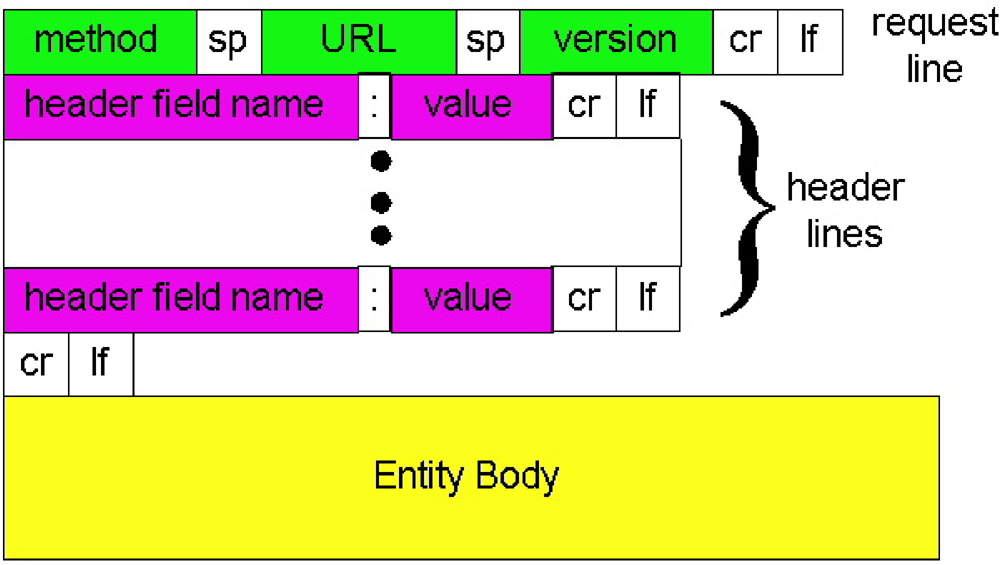
```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language:fr
```

header lines

Carriage return, line feed indicates end of message

(extra carriage return, line feed)

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

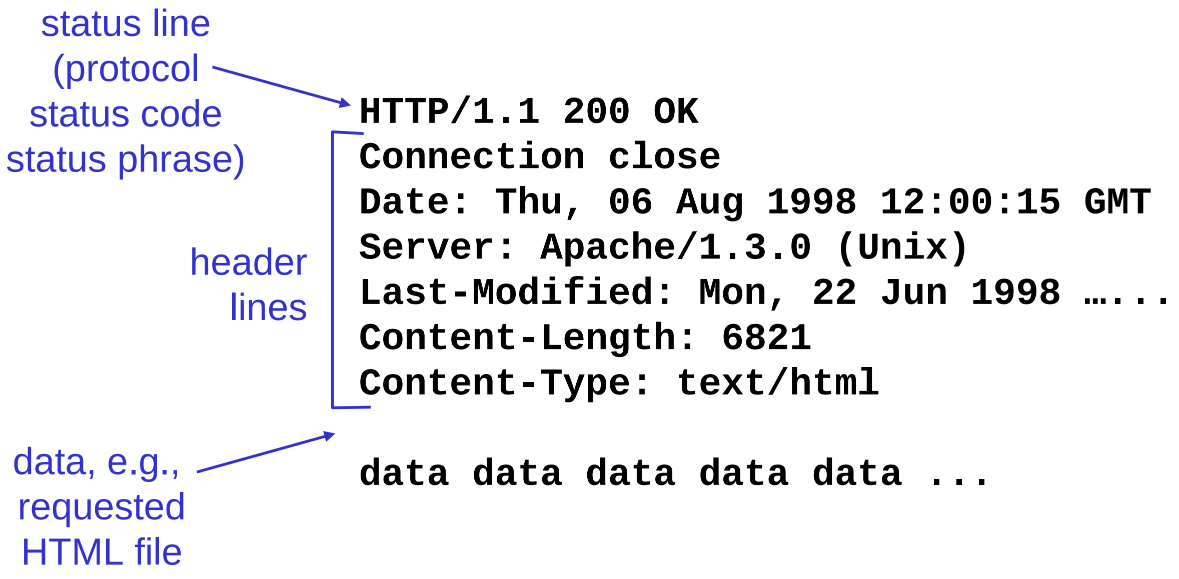
HTTP/1.0

- ❑ 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)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

HTTP Response Status Codes

In first line in server → client response message.

A few sample codes:

200 OK

- request succeeded, requested object later in this message

301 Moved Permanently

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

400 Bad Request

- request message not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Supported0

Trying Out HTTP (Client Side) for Yourself

1. Telnet to your favorite Web server:

```
telnet cis.poly.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at cis.poly.edu. Anything typed in sent to port 80 at cis.poly.edu

2. Type in a GET HTTP request:

```
GET /~ross/ HTTP/1.1
Host: cis.poly.edu
```

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

3. Look at response message sent by HTTP server!

User-Server State: Cookies

Many major Web sites use cookies

Four components:

1) Cookie header line in the HTTP response message

2) Cookie header line in HTTP request message

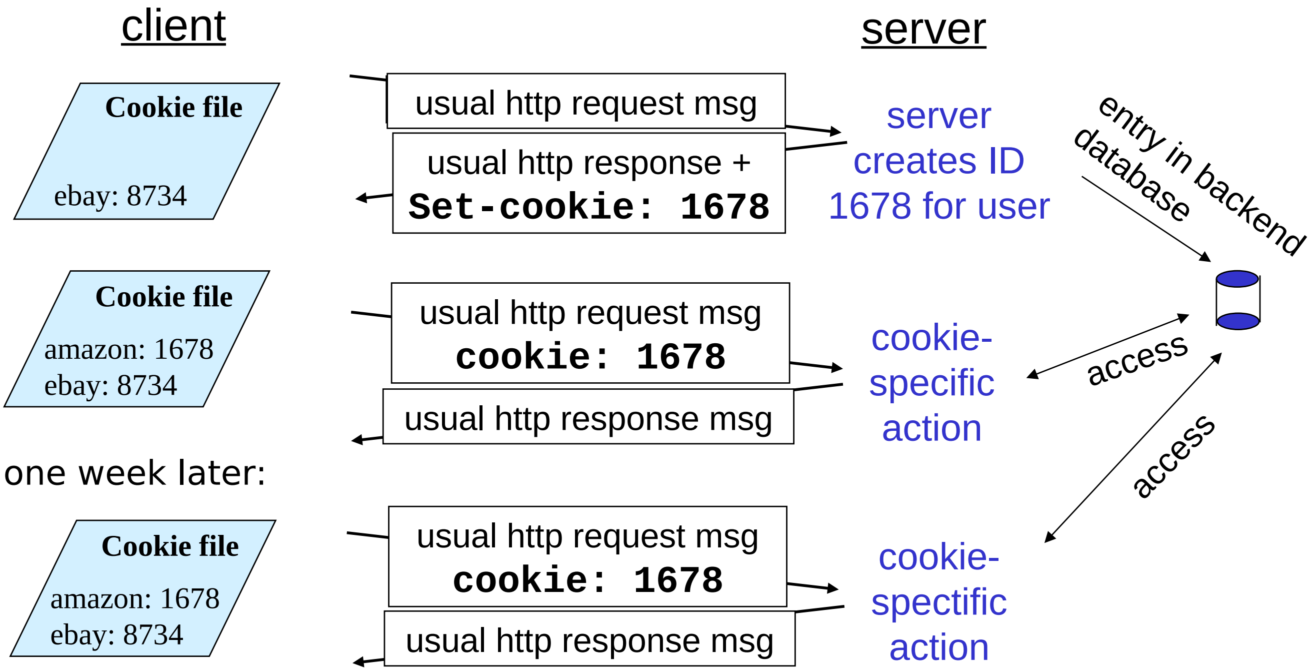
3) Cookie file kept on user's host and managed by user's browser

4) Back-end database at Web site

Example:

- Susan access Internet always from same PC
- She visits a specific e-commerce site for first time
- When initial HTTP requests arrives at site, site creates a unique ID and creates an entry in backend database for ID

Cookies: Keeping "State"



client

server

Cookie file

ebay: 8734

usual http request msg

usual http response +
`Set-cookie: 1678`

server creates ID 1678 for user

entry in backend database

Cookie file

amazon: 1678
ebay: 8734

usual http request msg
`cookie: 1678`

usual http response msg

cookie-specific action

access

access

one week later:

Cookie file

amazon: 1678
ebay: 8734

usual http request msg
`cookie: 1678`

usual http response msg

cookie-spectific action

Cookies (Continued)

What cookies can bring:

- Authorization
- Shopping carts
- Recommendations
- User session state (Web e-mail)

aside

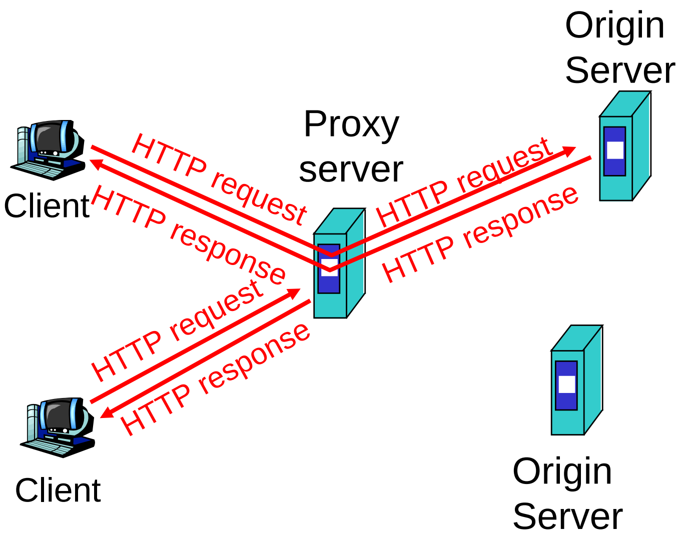
Cookies and privacy:

- Cookies permit sites to learn a lot about you
- You may supply name and e-mail to sites
- Search engines use redirection & cookies to learn yet more
- Advertising companies obtain info across sites

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
  - Otherwise cache requests object from origin server, then returns object to client



More About Web Caching

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

Why Web caching?

- ❑ Reduce response time for client request.
- ❑ Reduce traffic on an institution’s access link.
- ❑ Internet dense with caches enables “poor” content providers to effectively deliver content (but so does P2P file sharing)

Caching Example

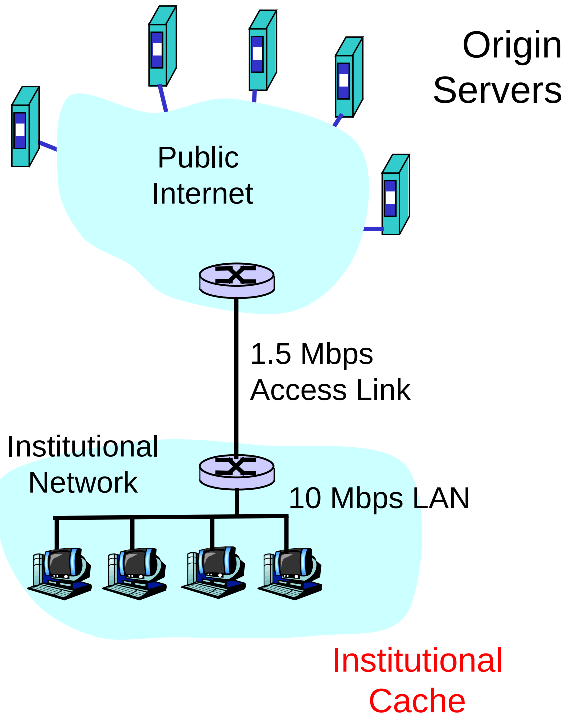
Assumptions

- ❑ average object size = 100,000 bits
- ❑ avg. request rate from institution’s browsers to origin servers = 15/sec
- ❑ delay from 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 (continued)

Possible solution

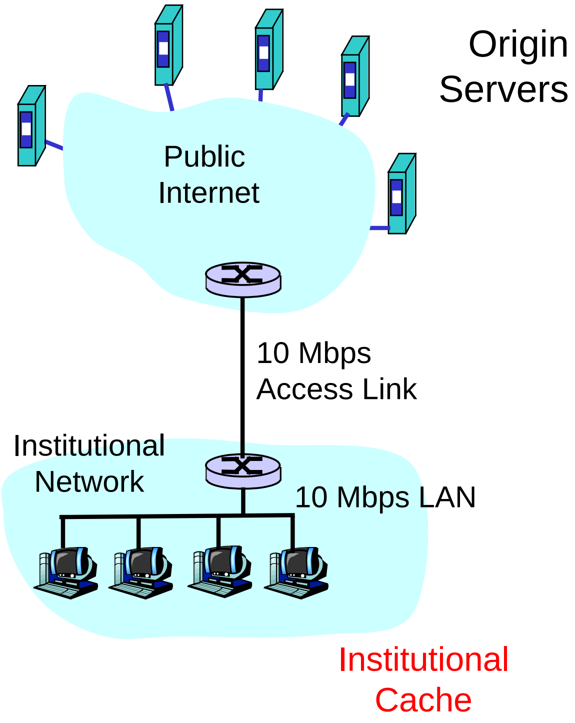
- ❑ Increase bandwidth of access link to, say, 10 Mbps

Consequences

- ❑ 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



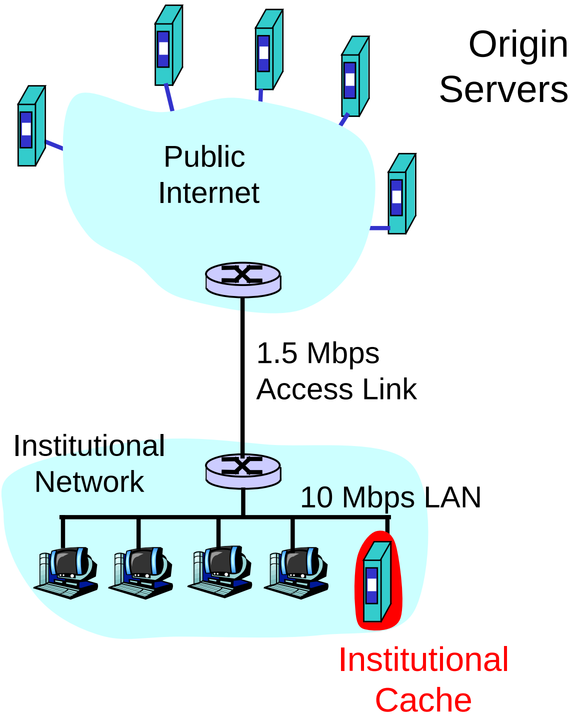
Caching Example (continued)

Install cache

- ❑ Suppose hit rate is .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 = .6*(2.01) secs + milliseconds < 1.4 secs

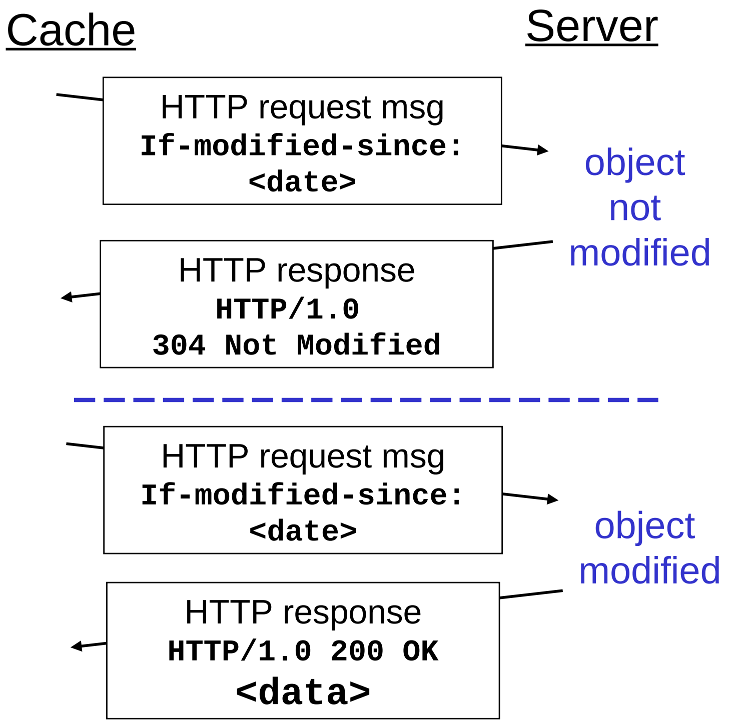


Conditional GET

- **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`



Cache

Server

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

object not modified

HTTP response
`HTTP/1.0 304 Not Modified`

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

object modified

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

Basic Web Server Tasks

Basic steps:

- Prepare for accepting requests
- Accept connection/request
- Read and process request
- Respond to request

(Ack: The following slides on web server tasks and architectures have been compiled from Hartmut Ritter's material [Rit04a])

Basic Web Server Tasks

- ❑ Prepare and accept requests:

```
s=socket();     // allocate listen socket

bind(s,80);     // bind socket to port 80

listen(s);         // indicate: ready to accept

while (1)  {
   newconn = accept(s);   // accept new requests
   /* when accept returns, we get a new socket which
     represents a new connection to a client */
   }
```

Basic Web Server Tasks

- ❑ Read and Process

```
read();          // read request
getsockname();   // get remote host name (to log)
setsockopt();    // set options, e.g. disable
                 // Nagle‘s algorithm
gettimeofday();  // get time of request

…                // Parse request, find file to send

stat();          // obtain file status and size
open();          // open requested file
read();          // read file into server
```

Basic Web Server Tasks

- ❑ Respond to Request

```
write(); // send HTTP header to client
write(); // send file to client

close(); // close file
close(); // shutdown connection

write(); // log request
```

Web Server Architectures

Four basic models:

- ❑ Process model
- ❑ Thread model
- ❑ In-kernel model
- ❑ Event-driven model

1. Process Model

- A process is assigned to perform all steps required to process a request
- When processing done,
  the process is ready to accept a new connection

- Typically multiple processes needed (20-200)
- One process blocks (e.g. read()),
  OS chooses next process to run
- Concurrency limited by max number of processes

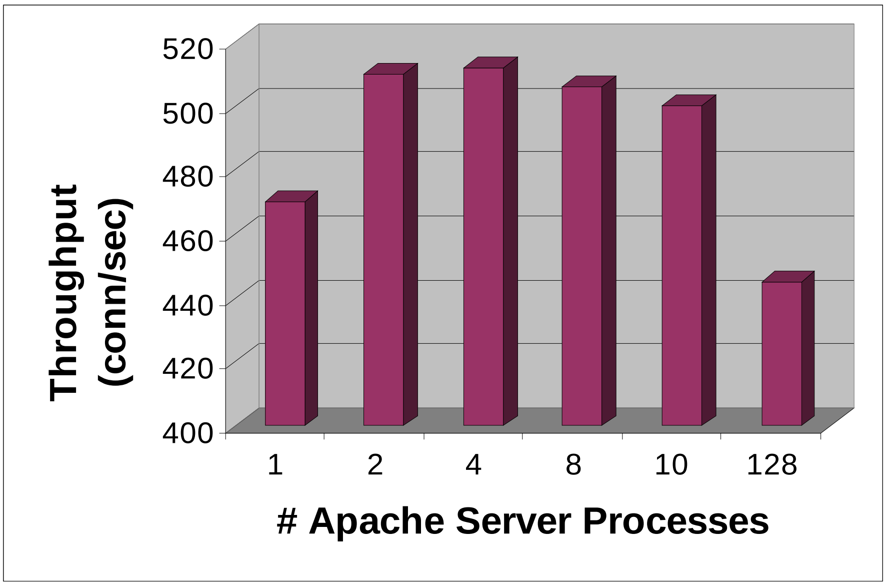
- Example: Apache on UNIX
  (most widely used web server, >60%)

1. Process Model

- Advantages:
  - Synchronization when handling different requests inherent in process model
  - Protection between processes
    (one process crashes, others unaffected)

- Disadvantage:
  - Slow (fork is expensive, context switching overhead)
  - Difficult to perform optimizations that rely on global information
    (e.g. cache URLs)

1. Process Model



Note: server is very slow machine

2. Thread Model

Use threads instead of processes

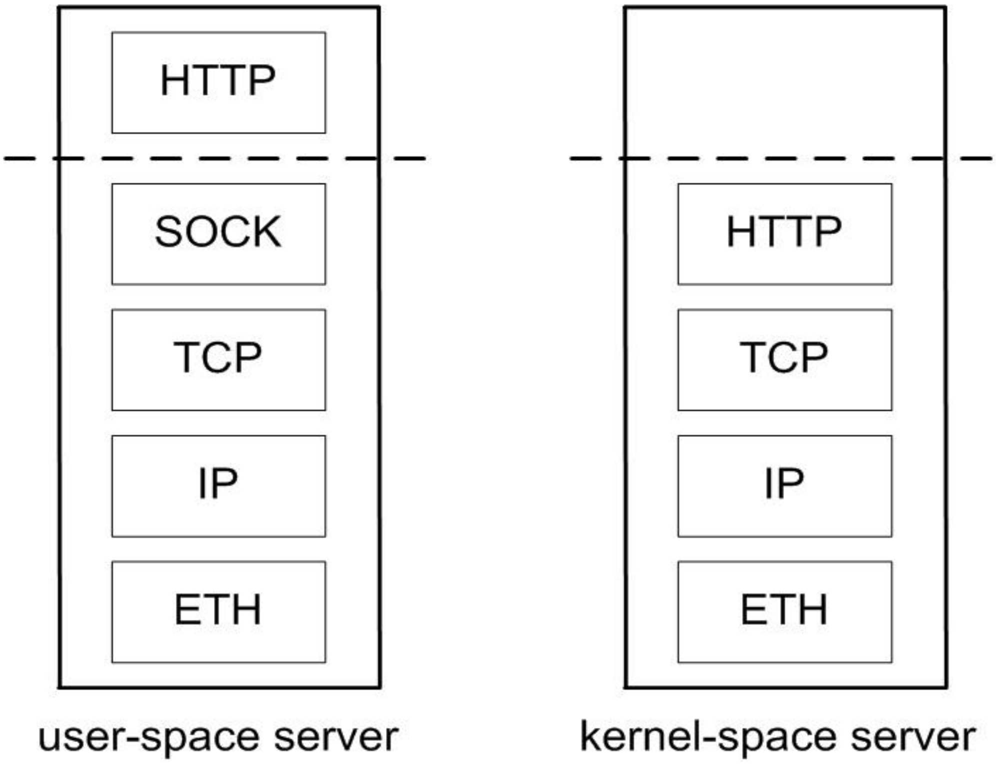
- Motivation:
  - Thread creation and destruction cheaper
  - Sharing data between threads easier than between processes, but synchronization required for shared data
- Problem:
  - OS support required (otherwise one blocked thread blocks whole address space)
- Examples: JAWS, IIS, Apache (Windows)

2. Thread Model

- Advantages:
  - Faster than processes
  - Sharing enabled by default

- Disadvantages:
  - Requires OS support
  - Can exhaust per-process limits (e.g. max. number of open file descriptors)
  - Limited control over scheduling decisions

3. In-kernel Model



3. In-kernel Model

- ❑ One option: whole server in the kernel
- ❑ Most often: only static files served from kernel, other requests go to regular user-space server (khttpd, AFPA)
- ❑ Dedicated kernel thread for HTTP requests

3. In-kernel Model

- ❑ Advantages:
  - ❑ Avoids copies to/from user space
  - ❑ Very fast, if tightly integrated with kernel (khttpd is not)
- ❑ Disadvantages:
  - ❑ Bugs can crash whole machine
  - ❑ Harder to debug and extend
  - ❑ Inherently OS-specific

3. In-kernel Model

- Examples:
  - khttpd:
    in Linux kernel, threaded, web server moved into kernel, uses sockets
  - TUX (Red Hat):
    in Linux kernel, threaded, requires new API for dynamic content
  - Advanced Fast Path Architecture (AFPA) (for Linux, W2k, AIX):
    Minimizes context switching and scheduling overhead,
    using software interrupts to perform tasks such as parsing requests and sending responses

4. Event-driven Model

Use a single event-driven server process to perform concurrent processing of multiple requests:

```
 while (1) {
    /*accept all new connection requests*/
    /*call select() on active file descriptors*/
    for each fd:
          if (fd ready for reading) call read();
          if (fd ready for writing) call write();
}
```

- Examples: Zeus, Flash, Boa, Mathopd, ScatterWeb EWS

4. Event-driven Model

- Single HTTPServer-module

- List of HTTP-connections

  ```
  list[index];
  ```

-
  ```
  myHttpConnection[index].status =
  {response, begin, end, dynamic}
  ```

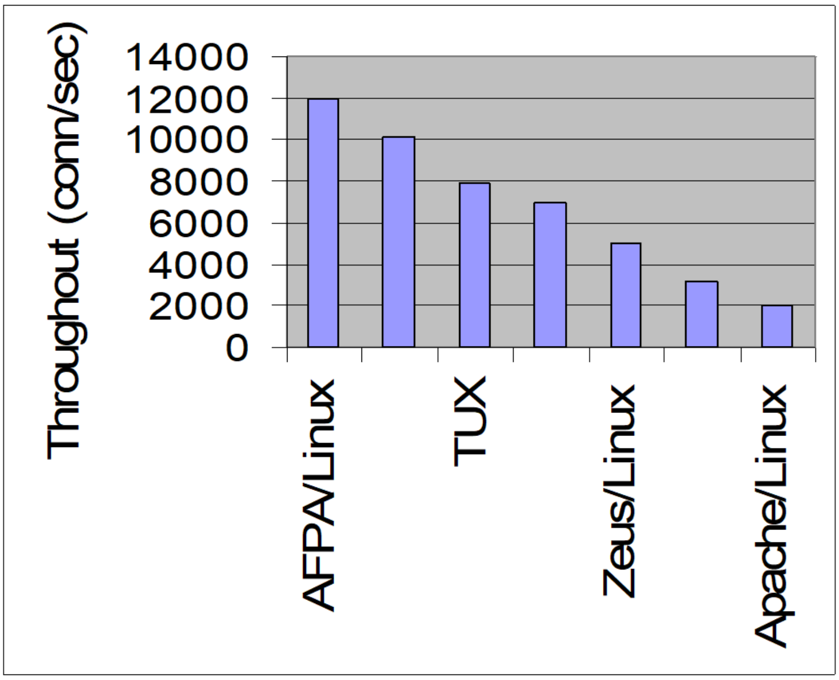
-
  ```
  myHttpConnection[index].PosInPage = readpointer;
  ```

4. Event-driven Model

- Advantages:
  - Very fast, no context switches
  - Sharing inherent (only one process), no locks needed
  - Complete control over scheduling decisions
  - No complex OS support needed

- Disadvantages:
  - Per-process resource limits
  - Not every OS has full asynchronous I/O, so can still block on read. Flash uses helper processes to avoid this.

Performance Comparison



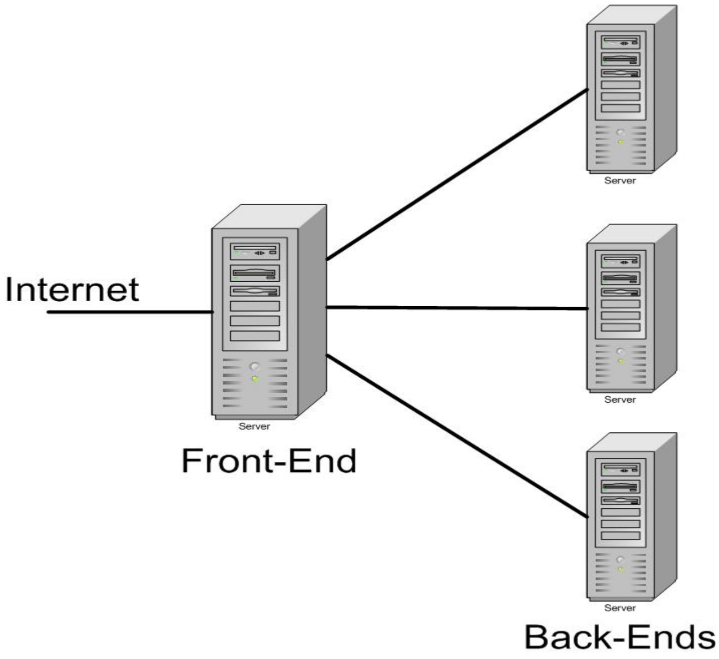
1 KB file, fast server

Web Server Clusters

- ❑ Two ways of increasing capacity:
  - ❑ Single larger machine
  - ❑ Cluster of cheap standard machines, e.g. PCs.
- ❑ Latter approach currently dominating:
  - ❑ Scalability
  - ❑ High availability
  - ❑ Cost

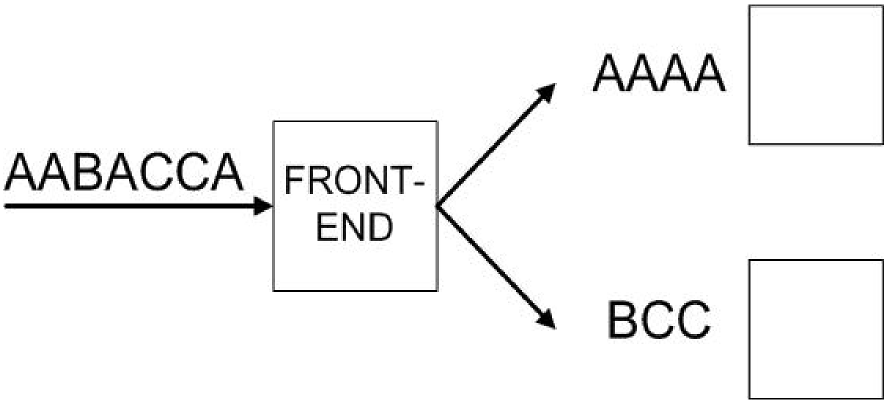
Web Server Clusters

- Typical architecture:



Web Server Clusters

- Important design issue: request distribution
- Traditional: round robin
- More efficient: content-based



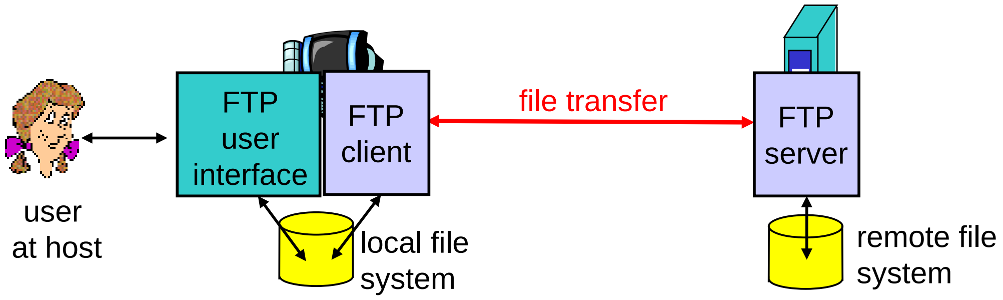
Acknowledgements/Literature on Web Server Architecture

- [Rit04] H. Ritter. *Embedded Internet - Chapter 3.3 Web Server Architecture.* Course slides, WS04/05, Freie Universität Berlin, 2004.
  - Hartmut says thank you to: Eric Nahum for providing his web server tutorial, Thiemo Voigt for compiling most of these slides
- References:
  - B. Krishnamurthy, J. Mogul and D. Kristol. Key Differences between HTTP/1.0 and HTTP/1.1. Wolrd Wide Web Conf., May 1999
  - V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. USENIX Technical Conference, June 1999.
  - V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. ASPLOS, Oct. 1998
  - P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey. High Performance Memory-Based Web Servers: Kernel and User-Space Performance. USENIX Technical Conference, June 2001.
  - E. Nahum, T. Barzilai and D. Kandlur. Performance Issues in WWW Servers. Transactions on Networking, Vol. 10, No. 1, February 2002.

Chapter 1: Application Layer

- Principles of network applications
- Web and HTTP
- FTP
- Electronic Mail
  - SMTP, POP3, IMAP
- DNS
- P2P file sharing
- Socket programming with TCP
- Socket programming with UDP
- Building a Web server

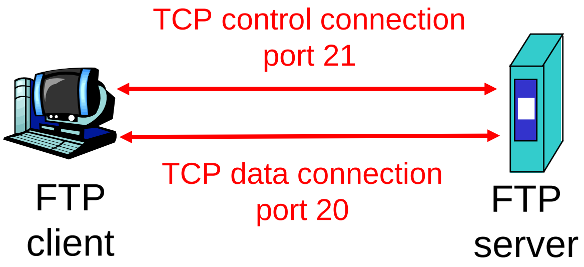
FTP: The File Transfer Protocol



- Transfer file to/from remote host
- Client/Server model
  - *Client:* side that initiates transfer (either to/from remote)
  - *Server:* remote host
- FTP specified in RFC 959
- FTP server port: 21

FTP: Separate Control & Data Connections

- FTP client contacts FTP server at port 21, specifying TCP as transport protocol
- Client obtains authorization over control connection
- Client browses remote directory by sending commands over control connection.
- When server receives a command for a file transfer, the server opens a TCP data connection to client
- After transferring one file, server closes connection.



- Server opens a second 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 1: Application Layer

- Principles of network applications
- Web and HTTP
- FTP
- Electronic Mail
  - SMTP, POP3, IMAP
- DNS
- P2P file sharing
- Socket programming with TCP
- Socket programming with UDP
- Building a Web server

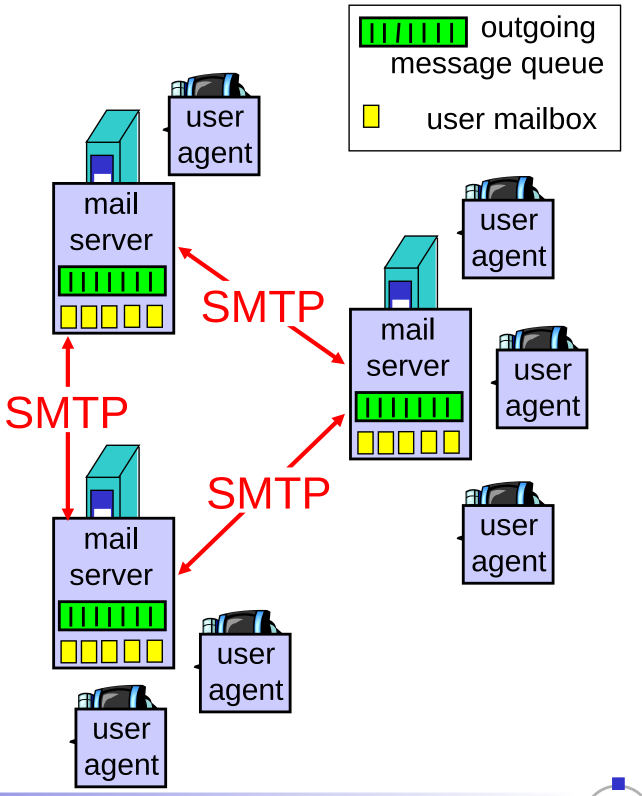
Electronic Mail

Three major components:

- ❑ User agents
- ❑ Mail servers
- ❑ Simple mail transfer protocol: SMTP

User Agent

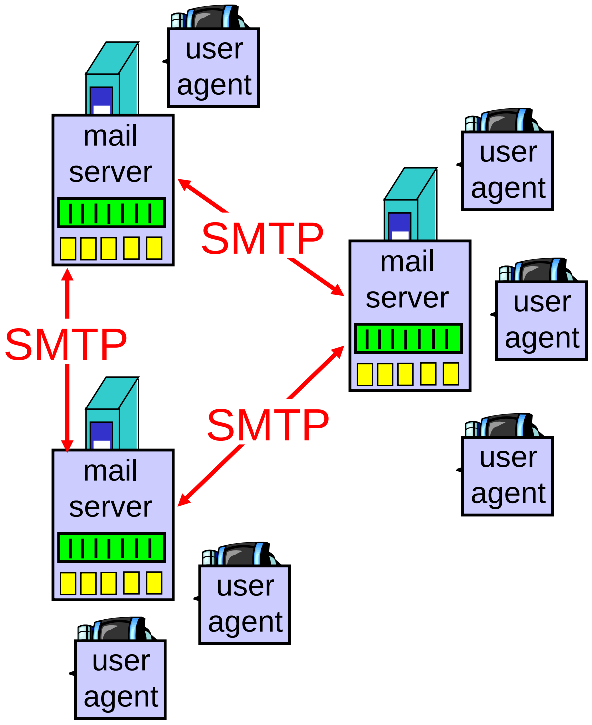
- ❑ A.k.a. “mail reader”
- ❑ Composing, editing, reading mail messages
- ❑ E.g., Outlook, Mozilla Firefox, mail client on mobile phone etc.
- ❑ Outgoing, incoming messages stored on server



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

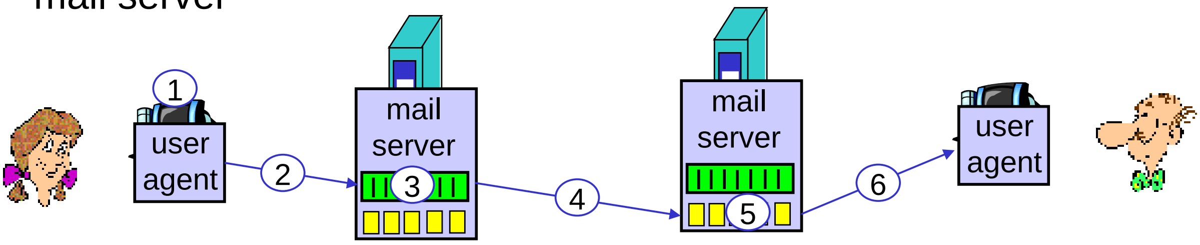


Electronic Mail: SMTP [RFC 2821]

- 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

- Type: `telnet servername 25`
- See 220 reply from server
- Enter commands:
  - `HELO`
  - `MAIL FROM`
  - `RCPT TO`
  - `DATA`
  - `QUIT`
- This lets you send email without using email client (reader)

SMTP: Final Words

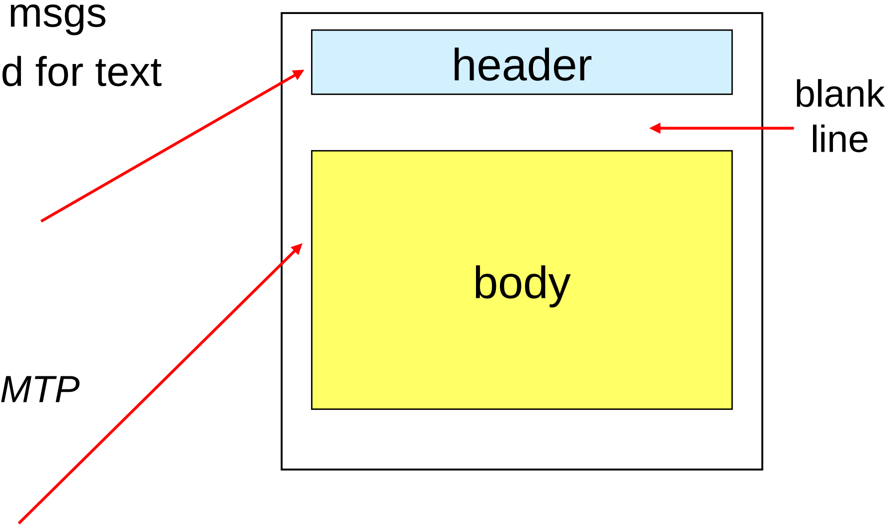
- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message
  - Btw. what would happen if you ever typed a single "." in one line in your email?
  - How could this be avoided?

Comparison with HTTP:

- Both have ASCII command/response interaction, status codes
- HTTP:
  - **Pull:** initiator asks responder for what it wants
  - each object encapsulated in its own response msg
- SMTP:
  - **Push:** initiator sends what it wants to communicate to responder
  - Multiple objects sent in multipart msg

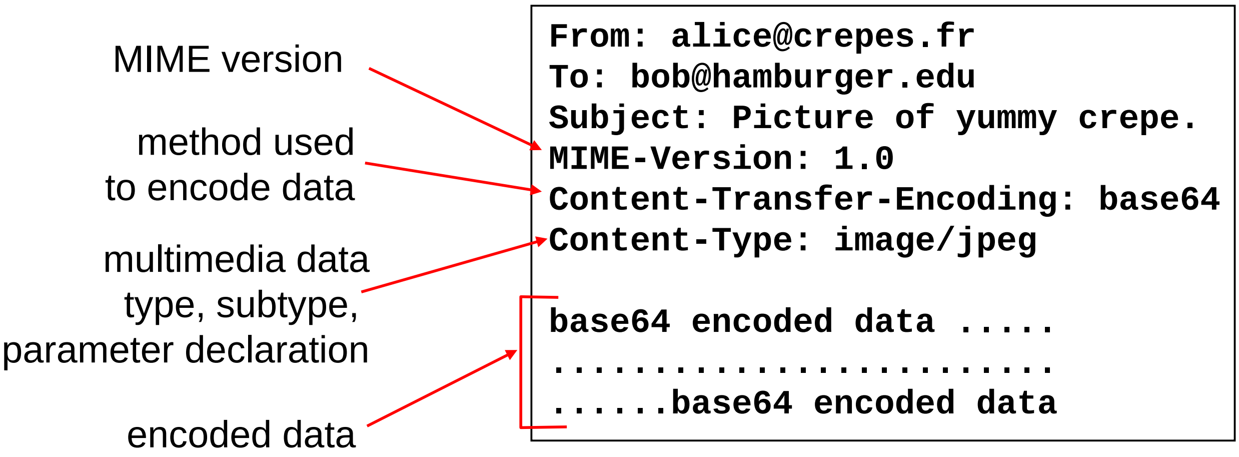
Mail Message Format

- 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



Message Format: Multimedia Extensions

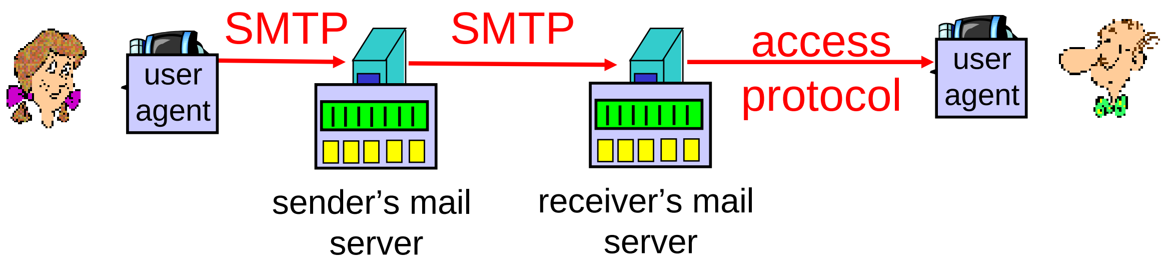
- MIME: Multimedia Mail Extension, RFC 2045, 2056
- Additional lines in msg header declare MIME content type


MIME version
method used to encode data
multimedia data type, subtype, parameter declaration
encoded data

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.........................
......base64 encoded data
```

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: 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 and IMAP

More about POP3:

- Previous example uses “download and delete” mode.
- Bob cannot re-read e-mail if he changes client
- “Download-and-keep”: enables copies of messages on different clients (requires to organize messages into folders on each client)
- 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 1: Application Layer

- ❑ Principles of network applications
- ❑ Web and HTTP
- ❑ FTP
- ❑ Electronic Mail
  - ❑ SMTP, POP3, IMAP
- ❑ DNS
- ❑ P2P file sharing
- ❑ Socket programming with TCP
- ❑ Socket programming with UDP
- ❑ 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* for hosts, 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

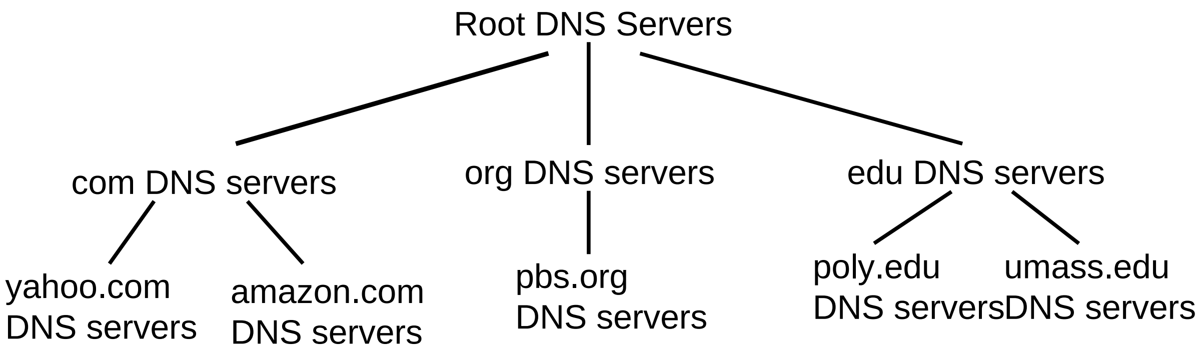
- ❑ Hostname to IP address translation
- ❑ Host aliasing
  - ❑ Canonical and alias names
- ❑ Mail server aliasing
- ❑ Load distribution
  - ❑ Replicated Web servers: set of IP addresses for one canonical name

Why not centralize DNS?

- ❑ Single point of failure
- ❑ Traffic volume
- ❑ Distant centralized database
- ❑ One central authority for worldwide name resolution undesirable (“who owns the Internet?”)
- ❑ Maintenance

⇒ does not *scale!*

Distributed, Hierarchical Database



Root DNS Servers

com DNS servers

org DNS servers

edu DNS servers

yahoo.com DNS servers

amazon.com DNS servers

pbs.org DNS servers

poly.edu DNS servers

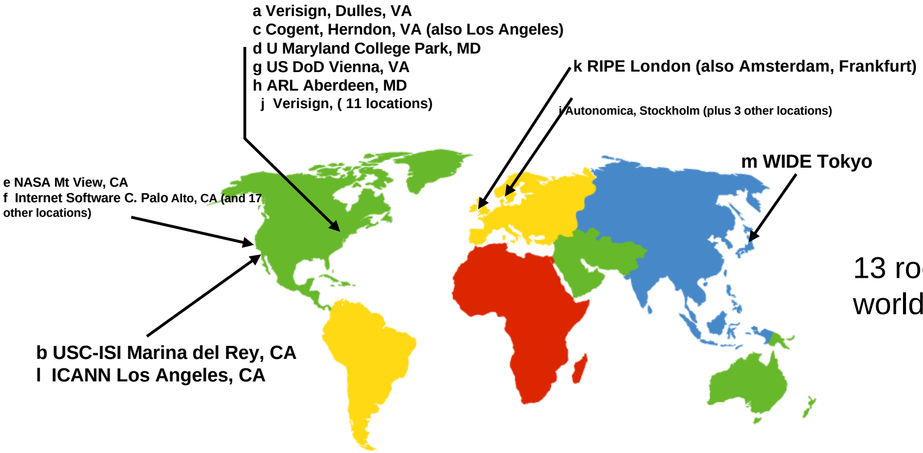
umass.edu DNS servers

Client wants IP for www.amazon.com; 1st approx:

- ❑ Client queries a root server to find com DNS server
- ❑ Client queries com DNS server to get amazon.com DNS server
- ❑ 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



a Verisign, Dulles, VA
c Cogent, Herndon, VA (also Los Angeles)
d U Maryland College Park, MD
g US DoD Vienna, VA
h ARL Aberdeen, MD
j Verisign, (11 locations)

k RIPE London (also Amsterdam, Frankfurt)

i Autonomica, Stockholm (plus 3 other locations)

m WIDE Tokyo

e NASA Mt View, CA
f Internet Software C. Palo Alto, CA (and 17 other locations)

b USC-ISI Marina del Rey, CA
l ICANN Los Angeles, CA

13 root name servers worldwide

TLD, Authoritative and Local DNS Servers

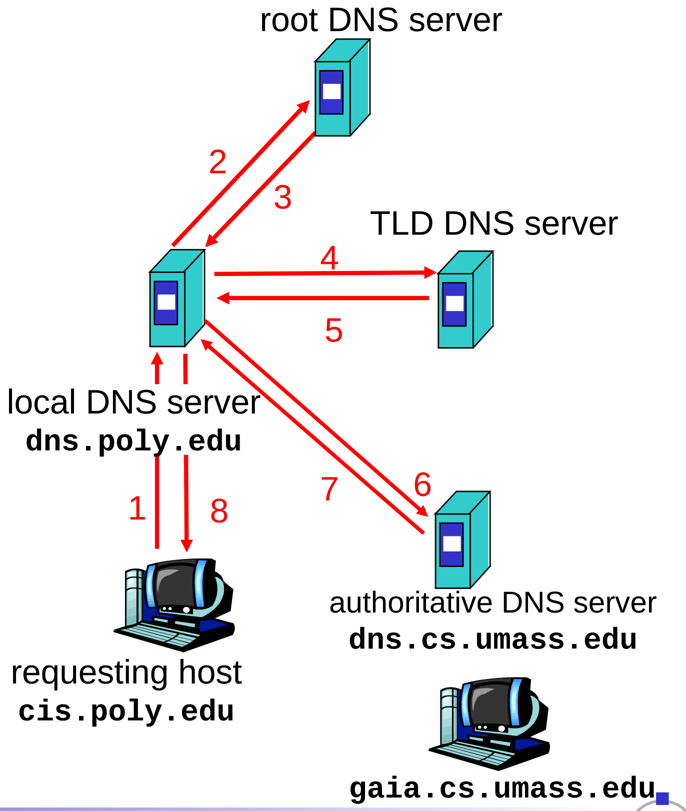
- ❑ Top-level domain (TLD) servers:
  - ❑ responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp
  - ❑ Network solutions maintains servers for **`com`** TLD
  - ❑ Educause for **edu** TLD
- ❑ Authoritative DNS servers:
  - ❑ organization’s DNS servers, providing authoritative hostname to IP mappings for organization’s servers (e.g., Web and mail).
  - ❑ Can be maintained by organization or service provider
- ❑ Local DNS servers:
  - ❑ Does not strictly belong to hierarchy
  - ❑ Each ISP (residential ISP, company, university) has one
    - ■ Also called “default name server”
  - ❑ When a host makes a DNS query, query is sent to its local DNS server
    - ■ Acts as a proxy, forwards query into hierarchy

Iterative Queries: 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”

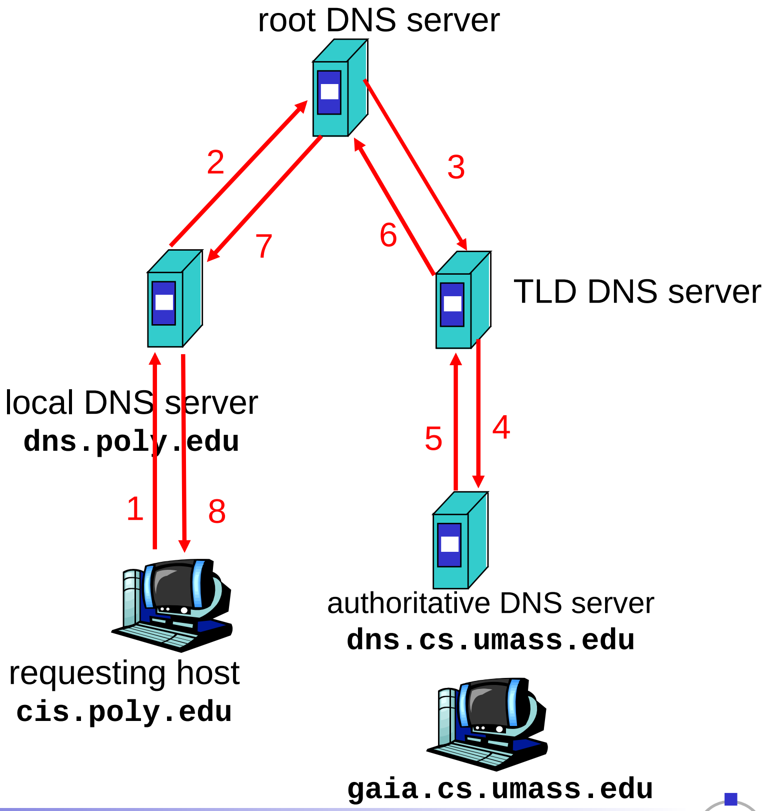


Recursive Queries: Example

Recursive query:

- Puts burden of name resolution on contacted name server
- Heavy load?

⇒ Not done by root or TLD name servers



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
  - RFC 2136 and RFC 3007 (updated version)
  - E.g. used by DHCP servers to update DNS entries in servers
  - Alternatively, there is also DDNS (Dynamic DNS) over HTTPS for updating DNS entries of hosts that regularly get new IP addresses assigned (e.g. DSL routers often support interacting with so called-DynDNS providers)

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 IP address of authoritative name server for this domain
- Type=MX
  - **`value`** is name of mailserver associated with **`name`**
- Type=CNAME
  - **`name`** is alias name for some “canonical” (the real) name

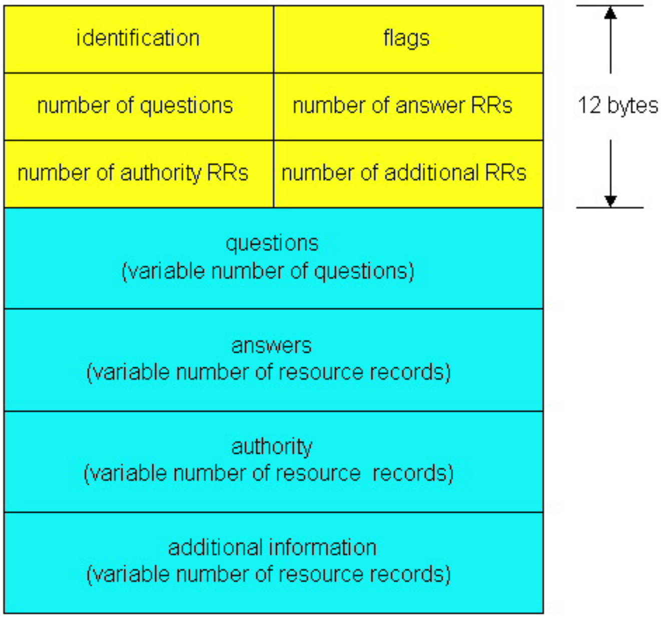
    `www.ibm.com` is really **`servereast.backup2.ibm.com`**
  - **`value`** is canonical name

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



identification | flags
number of questions | number of answer RRs
number of authority RRs | number of additional RRs
questions (variable number of questions)
answers (variable number of resource records)
authority (variable number of resource records)
additional information (variable number of resource records)

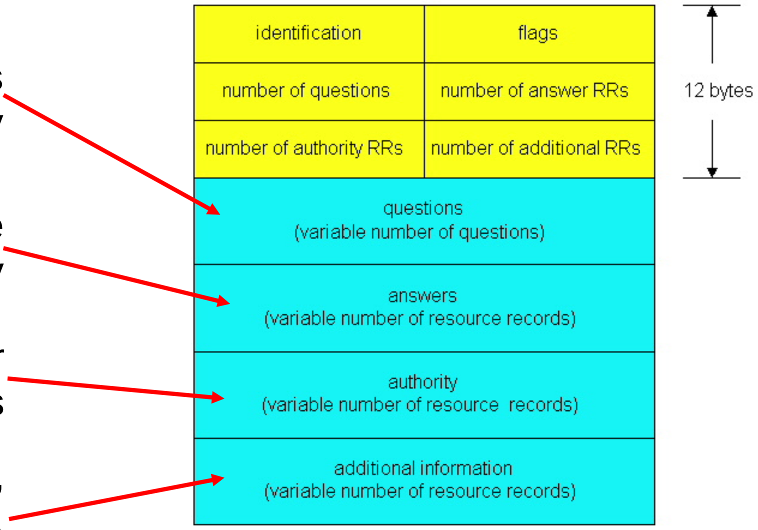
DNS Protocol, Messages

Name, type fields for a query

RRs in reponse to query

records for authoritative servers

additional “helpful” info that may be used



identification | flags
number of questions | number of answer RRs
number of authority RRs | number of additional RRs
questions (variable number of questions)
answers (variable number of resource records)
authority (variable number of resource records)
additional information (variable number of resource records)

Inserting Records Into DNS

- ❑ Example: just created startup “Network Utopia”
- ❑ Register name networkutopia.com at a registrar (e.g., Network Solutions)
  - ❑ Need to provide registrar with names and IP addresses of your authoritative name server (primary and secondary)
  - ❑ Registrar inserts two RRs into the com TLD server:

```
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
```

- ❑ Put in authoritative server Type A record for www.networkutopia.com and Type MX record for networkutopia.com
- ❑ How do people get the IP address of your Web site?

Chapter 1: Application Layer

- ❑ Principles of network applications
- ❑ Web and HTTP
- ❑ FTP
- ❑ Electronic Mail
  - ❑ SMTP, POP3, IMAP
- ❑ DNS
- ❑ P2P file sharing
- ❑ Socket programming with TCP
- ❑ Socket programming with UDP
- ❑ Building a Web server

P2P File Sharing

Example

- ❑ Alice runs P2P client application on her notebook computer
- ❑ Intermittently connects to Internet; gets new IP address for each connection
- ❑ Asks for “Hey Jude”
- ❑ Application displays other peers that have copy of Hey Jude
- ❑ Alice chooses one of the peers, Bob.
- ❑ File is copied from Bob’s PC to Alice’s notebook: HTTP
- ❑ While Alice downloads, other users uploading from Alice.
- ❑ Alice’s peer is both a Web client and a transient Web server.
- ❑ All peers are servers = highly scalable!

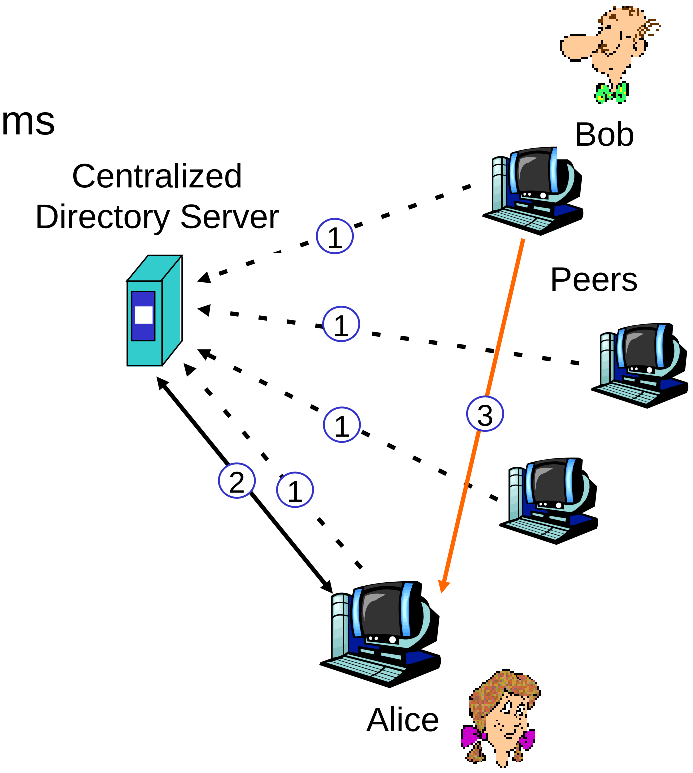
P2P: Centralized Directory

Original “Napster” design

1) When peer connects, it informs central server:
   - ❑ IP address
   - ❑ Content

2) Alice queries for “Hey Jude”

3) Alice requests file from Bob



P2P: Problems With Centralized Directory

- ❑ Single point of failure
- ❑ Performance bottleneck
- ❑ Copyright infringement

File transfer is decentralized, but locating content is highly centralized

Query Flooding: Gnutella

General Properties:

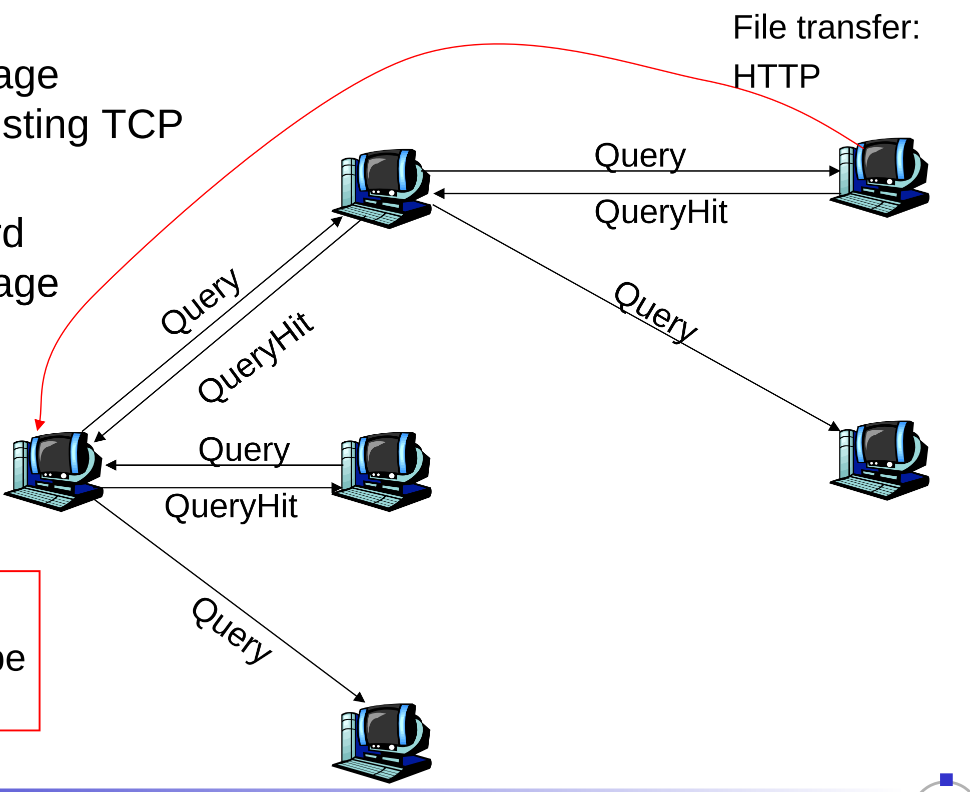
- ❑ Fully distributed
  - ❑ no central server
- ❑ Public domain protocol
- ❑ Many Gnutella clients implementing protocol

Overlay network: graph

- ❑ Edge between peer X and Y if there’s a TCP connection
- ❑ All active peers and edges is overlay net
- ❑ Edge is not a physical link
- ❑ Given peer will typically be connected with < 10 overlay neighbors

Gnutella: Protocol

- ❑ Query message sent over existing TCP connections
- ❑ Peers forward Query message
- ❑ QueryHit sent over reverse path



Scalability:
limited scope flooding

Gnutella: Peer Joining

- ❑ Joining peer X must find some other peer in Gnutella network: use list of candidate peers
- ❑ X sequentially attempts to make TCP with peers on list until connection setup with Y
- ❑ X sends Ping message to Y; Y forwards Ping message.
- ❑ All peers receiving Ping message respond with Pong message
- ❑ X receives many Pong messages. It can then setup additional TCP connections

Peer leaving?

- ❑ Your task: Search for information using your favorite search engine...

Chapter 1: Application Layer

- Principles of network applications
- Web and HTTP
- FTP
- Electronic Mail
  - SMTP, POP3, IMAP
- DNS
- P2P file sharing
- Socket programming with TCP
- Socket programming with UDP
- Building a Web server

Socket Programming

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

Socket API

- Introduced in BSD4.1 UNIX, 1981
- Sockets are explicitly created, used, released by applications
- Client/Server paradigm
- Two types of transport service via socket API:
  - Unreliable datagram
  - Reliable, byte stream-oriented

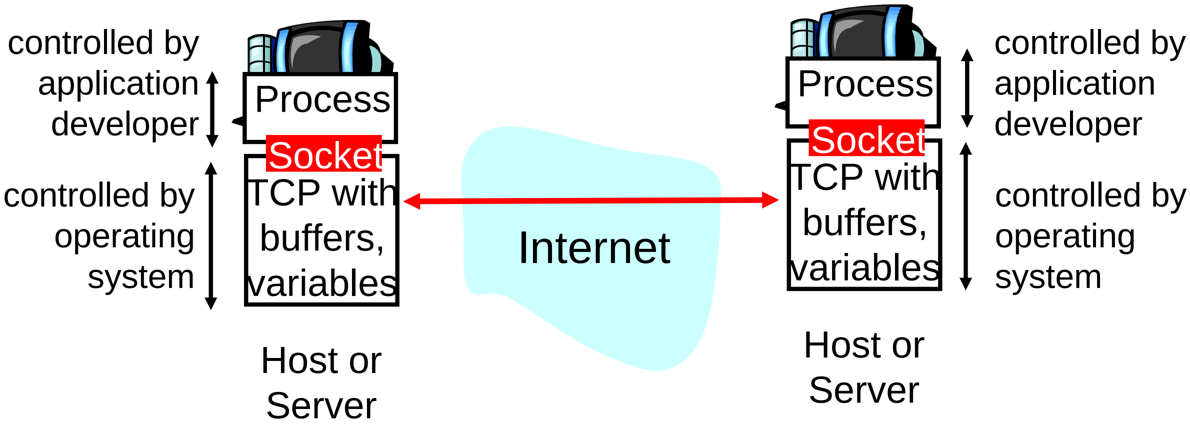
socket

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


controlled by application developer
controlled by operating system
Process
Socket
TCP with buffers, variables
Host or Server
Internet
Process
Socket
TCP with buffers, variables
Host or Server
controlled by application developer
controlled by operating system

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

application viewpoint

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

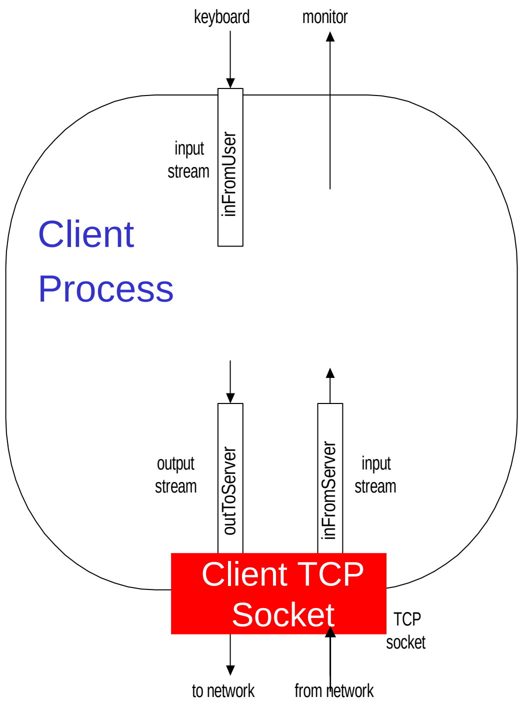
Stream Jargon

- 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, eg, keyboard or socket.
- An output stream is attached to an output source, eg, monitor or socket.

Socket Programming With TCP

Example Client-Server application:

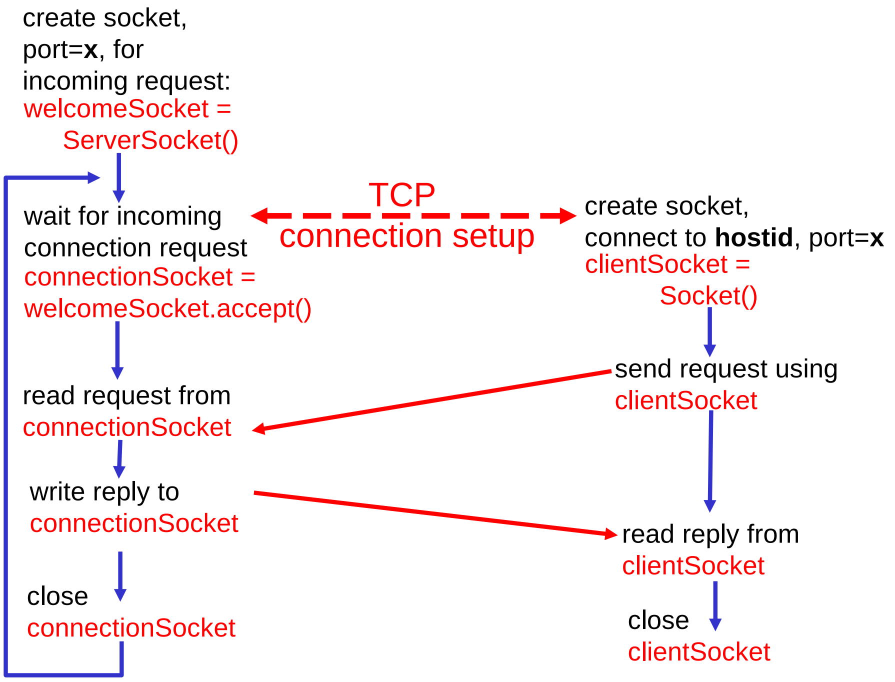
1) Client reads line from standard input (**`inFromUser`** stream) , sends to server via socket (**`outToServer`** stream)
2) Server reads line from socket
3) Server converts line to uppercase, sends back to client
4) Client reads, prints modified line from socket (**`inFromServer`** stream)



Client/Server Socket Interaction: TCP

Server (running on **hostid**)

Client



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

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

TCP connection setup

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

send request using clientSocket

read request from connectionSocket

write reply to connectionSocket

read reply from clientSocket

close connectionSocket

close clientSocket

Example: Java Client (TCP)

```
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());
```

Create input stream → BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

Create client socket, connect to server → Socket clientSocket = new Socket("hostname", 6789);

Create output stream attached to socket → DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());

Example: Java Client (TCP, continued)

Create input stream attached to socket

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

 sentence = inFromUser.readLine();
```

Send line to server

```
 outToServer.writeBytes(sentence + '\n');
```

Read line from server

```
 modifiedSentence = inFromServer.readLine();

 System.out.println("FROM SERVER: " + modifiedSentence);

 clientSocket.close();

   }
}
```

Example: Java Server (TCP)

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

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;
```

Create welcoming socket at port 6789

```
     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {
```

Wait, on welcoming socket for contact by client

```
        Socket connectionSocket = welcomeSocket.accept();
```

Create input stream, attached to socket

```
       BufferedReader inFromClient =
         new BufferedReader(new
         InputStreamReader(connectionSocket.getInputStream()));
```

Example: Java Server (TCP, continued)

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

      clientSentence = inFromClient.readLine();

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

      outToClient.writeBytes(capitalizedSentence);
    }
  }
}
```

Create output stream, attached to socket

Read in line from socket

Write out line to socket

End of while loop, loop back and wait for another client connection

Chapter 1: Application Layer

- ❑ Principles of network applications
- ❑ Web and HTTP
- ❑ FTP
- ❑ Electronic Mail
  - ❑ SMTP, POP3, IMAP
- ❑ DNS
- ❑ P2P file sharing
- ❑ Socket programming with TCP
- ❑ Socket programming with UDP
- ❑ Building a Web server

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

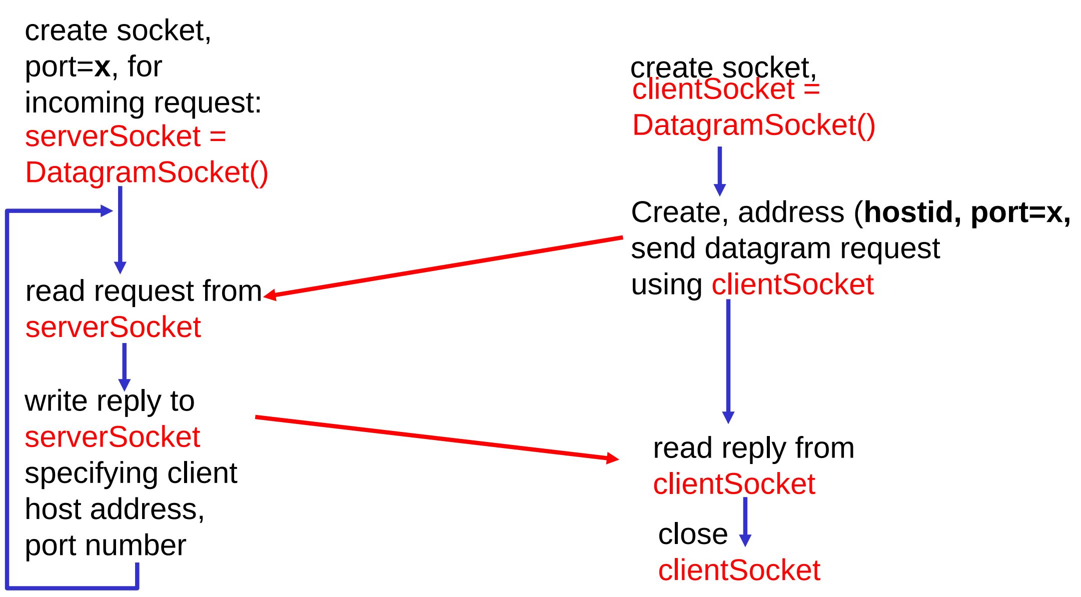
application viewpoint

UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server

Client/Server Socket Interaction: UDP

Server (running on **hostid**)

Client



create socket,
port=**x**, for
incoming request:
serverSocket =
DatagramSocket()

read request from
serverSocket

write reply to
serverSocket
specifying client
host address,
port number

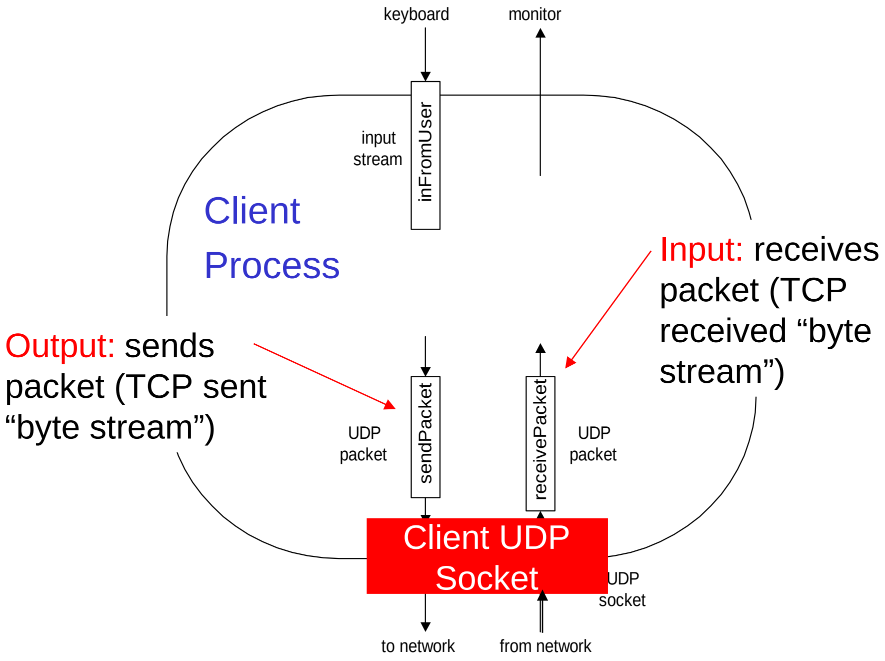
create socket,
clientSocket =
DatagramSocket()

Create, address (**hostid, port=x,**
send datagram request
using clientSocket

read reply from
clientSocket

close
clientSocket

Example: Java Client (UDP)



Output: sends packet (TCP sent “byte stream”)

Input: receives packet (TCP received “byte stream”)

Example: Java Client (UDP)

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

class UDPClient {
   public static void main(String args[]) throws Exception
   {
```

Create input stream

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

Create client socket

```
      DatagramSocket clientSocket = new DatagramSocket();
```

Translate hostname to IP address using DNS

```
      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, continued)

Create datagram with data, length IP addr, port

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

Send datagram to server

```
clientSocket.send(sendPacket);

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

Read datagram from server

```
clientSocket.receive(receivePacket);

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

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}

}
```

Example: Java Server (UDP)

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

class UDPServer {
  public static void main(String args[]) throws Exception
    {
```

Create datagram socket at port 9876

```
      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {
```

Create space for received datagram

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

Receive datagram

```
           serverSocket.receive(receivePacket);
```

Example: Java Server (UDP, continued)

```
        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

Chapter 1: Application Layer

- ❑ Principles of network applications
- ❑ Web and HTTP
- ❑ FTP
- ❑ Electronic Mail
  - ❑ SMTP, POP3, IMAP
- ❑ DNS
- ❑ P2P file sharing
- ❑ Socket programming with TCP
- ❑ Socket programming with UDP
- ❑ Building a Web server

Building a Simple Web Server

- ❑ Handles one HTTP request
- ❑ Accepts the request
- ❑ Parses header
- ❑ Obtains requested file from server's file system
- ❑ Creates HTTP response message:
  - ❑ Header lines + file
- ❑ Sends response to client

- ❑ After creating server, you can request file using a browser (eg IE explorer)
- ❑ See [KR04, chapter 2.8] for details

Chapter 1: Summary

Our study of network applications is now complete!

- ❑ Application architectures
  - ❑ Client/Server
  - ❑ Peer2Peer
  - ❑ 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
- ❑ Socket programming

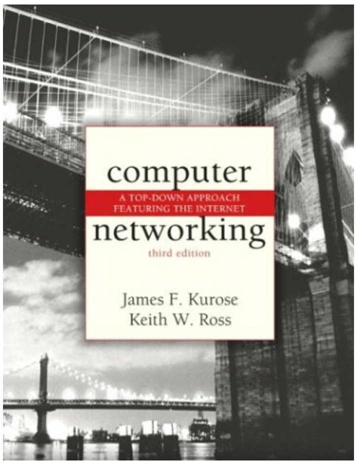
Chapter 1: Summary

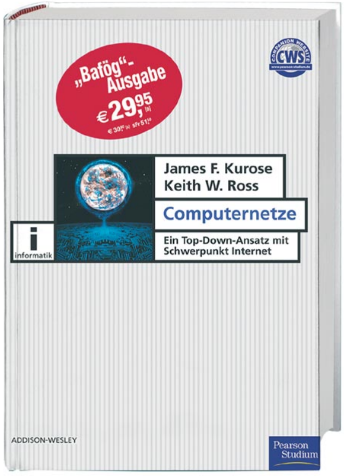
Most importantly: Learned about *application protocols*

- 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
- Control vs. data msgs
  - In-band, out-of-band
- Centralized vs. decentralized
- Stateless vs. stateful
- Reliable vs. unreliable msg transfer
- “Complexity at network edge”

Additional Reference for this Chapter

[KR04] J. F. Kurose & K. W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, 2004, 3rd edition, Addison Wesley.
(chapter 2 covers the application layer)





Appendix: Socket Programming with C

- Because of the prime importance of the C programming language in the area of communications and networking, the following slides show the socket programming examples in C
  - C often looks ugly to the beginner’s eye
  - The programmer has to take care of many things, that the Java programmer does not need to bother about
  - These disadvantages are the price for the big advantage of C over Java and other “higher-level” languages:
    - It allows the programmer to control low-level issues in order to write programs that achieve a better performance (in terms of execution time, resource consumption, etc.)

Example: C Client (TCP)

```
/* client.c  */
void main(int argc, char *argv[])
{
struct sockaddr_in sad; /* structure to hold an IP address */
int clientSocket; /* socket descriptor */
struct hostent *ptrh; /* pointer to a host table entry */

char Sentence[128];
char modifiedSentence[128];

host = argv[1]; port = atoi(argv[2]);

clientSocket = socket(PF_INET, SOCK_STREAM, 0);
           memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
           sad.sin_family = AF_INET; /* set family to Internet */
           sad.sin_port = htons((u_short)port);
           ptrh = gethostbyname(host); /* Convert host name to IP address */
           memcpy(&sad.sin_addr, ptrh->h_addr, ptrh→h_length);
           connect (clientSocket, (struct sockaddr *)&sad, sizeof(sad));
```

Create client socket, connect to server

Example: C Client (TCP, continued)

Get input stream from user

```
gets(Sentence);
```

Send line to server

```
n=write(clientSocket, Sentence, strlen(Sentence)+1);
```

Read line from server

```
n=read(clientSocket, modifiedSentence,
              sizeof(modifiedSentence));

printf("FROM SERVER: %s\n”,modifiedSentence);
```

Close connection

```
close(clientSocket);

}
```

Example: C Server (TCP)

```
/* server.c  */
void main(int argc, char *argv[])
{
struct sockaddr_in sad; /* structure to hold an IP address */
struct sockaddr_in cad;
int welcomeSocket, connectionSocket; /* socket descriptor */
struct hostent *ptrh; /* pointer to a host table entry */

char clientSentence[128];
char capitalizedSentence[128];

port = atoi(argv[1]);

welcomeSocket = socket(PF_INET, SOCK_STREAM, 0);
            memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
            sad.sin_family = AF_INET; /* set family to Internet */
            sad.sin_addr.s_addr = INADDR_ANY; /* set the local IP address */
            sad.sin_port = htons((u_short)port);/* set the port number */
bind(welcomeSocket, (struct sockaddr *)&sad, sizeof(sad));
```

Create welcoming socket at port
&
Bind a local address

Example: C Server (TCP, continued)

```
/* Specify the maximum number of clients that can be queued */
listen(welcomeSocket, 10)

while(1) {

   connectionSocket=accept(welcomeSocket,
                           (struct sockaddr *)&cad, &alen);

   n=read(connectionSocket, clientSentence, sizeof(clientSentence));

   /* capitalize Sentence and store the result in capitalizedSentence*/


   n=write(connectionSocket, capitalizedSentence,
                strlen(capitalizedSentence)+1);

   close(connectionSocket);
   }
}
```

Wait, on welcoming socket for contact by a client

Write out the result to socket

End of while loop, loop back and wait for another client connection

Example: C Client (UDP)

```
void main(int argc, char *argv[])
{
struct sockaddr_in sad; /* structure to hold an IP address */
int clientSocket; /* socket descriptor */
struct hostent *ptrh; /* pointer to a host table entry */
char Sentence[128];
char modifiedSentence[128];
host = argv[1]; port = atoi(argv[2]);

clientSocket = socket(PF_INET, SOCK_DGRAM, 0);

/* determine the server's address */
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET; /* set family to Internet */
sad.sin_port = htons((u_short)port);
ptrh = gethostbyname(host); /* Convert host name to IP address */
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);
}
```

Create client socket,
NO connection to server

Example: C Client (UDP, continued)

Get input stream from user

```
gets(Sentence);
```

Send line to server

```
addr_len =sizeof(struct sockaddr);
n=sendto(clientSocket, Sentence, strlen(Sentence)+1,
                (struct sockaddr *) &sad, addr_len);
```

Read line from server

```
n=recvfrom(clientSocket, modifiedSentence,
sizeof(modifiedSentence).
                   (struct sockaddr *) &sad, &addr_len);


printf("FROM SERVER: %s\n”,modifiedSentence);
```

Close connection

```
close(clientSocket);
}
```

Example: C Server (UDP)

```
/* server.c  */
void main(int argc, char *argv[])
{
struct sockaddr_in sad; /* structure to hold an IP address */
struct sockaddr_in cad;
int serverSocket; /* socket descriptor */
struct hostent *ptrh; /* pointer to a host table entry */

char clientSentence[128];
char capitalizedSentence[128];

port = atoi(argv[1]);
```

Create welcoming socket at port
&
Bind a local address

```
serverSocket = socket(PF_INET, SOCK_DGRAM, 0);
            memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
            sad.sin_family = AF_INET; /* set family to Internet */
            sad.sin_addr.s_addr = INADDR_ANY; /* set the local IP address */
            sad.sin_port = htons((u_short)port);/* set the port number */
            bind(serverSocket, (struct sockaddr *)&sad, sizeof(sad));
```

Example: C Server (UDP, continued)

```
while(1) {

    n=recvfrom(serverSocket, clientSentence, sizeof(clientSentence), 0
                      (struct sockaddr *) &cad, &addr_len );

   /* capitalize Sentence and store the result in capitalizedSentence*/

   n=sendto(connectionSocket, capitalizedSentence,
                 strlen(capitalizedSentence)+1, 0,
                (struct sockaddr *) &cad, &addr_len);

   close(connectionSocket);
   }
}
```

Receive messages from clients

Write out the result to socket

End of while loop,
loop back and wait for
another client connection