CSCI 450
Data Communication & Network Programming

Application Layer

Dr. Jiang Li

(Adapted from "Computer Networking: A Top Down Approach Featuring the Internet", 3rd edition. Copyrighted by Jim Kurose, Keith Ross)

Chapter 2: Application Layer

Our goals:

- Conceptual, implementation aspects of network application protocols
  - Transport-layer service models
  - Client-server paradigm
  - Peer-to-peer paradigm
- Learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - SMTP
  - DNS
- Programming 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
- …

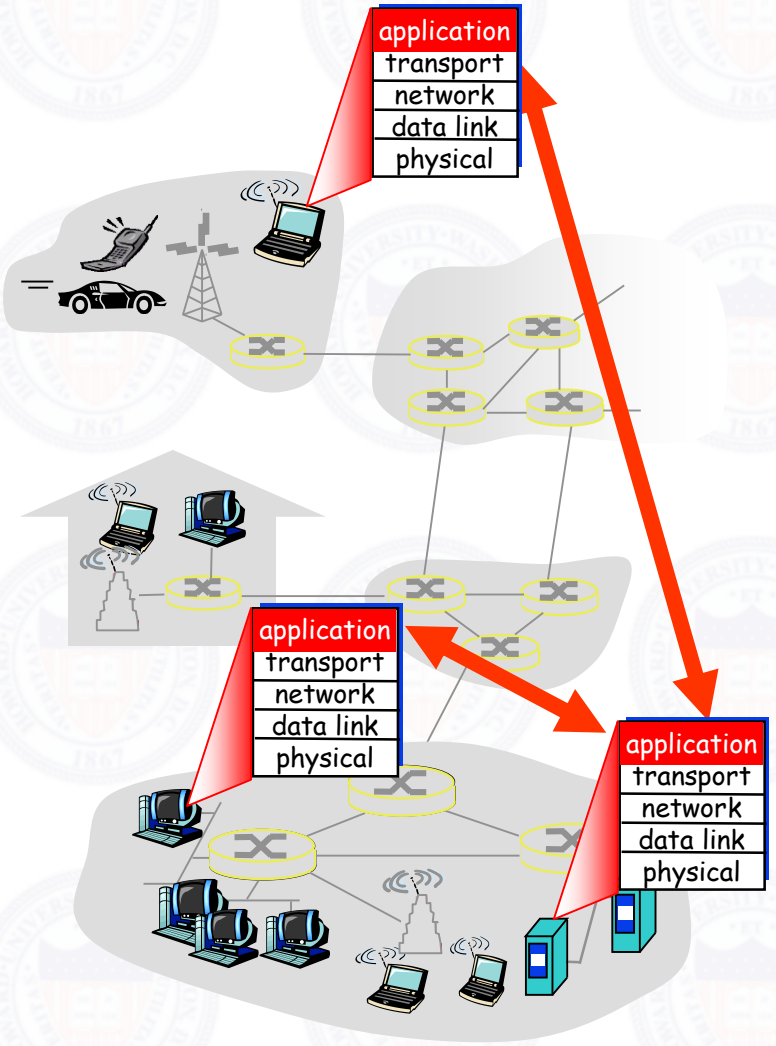
Creating A Network App

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



Chapter 2: Application Layer

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

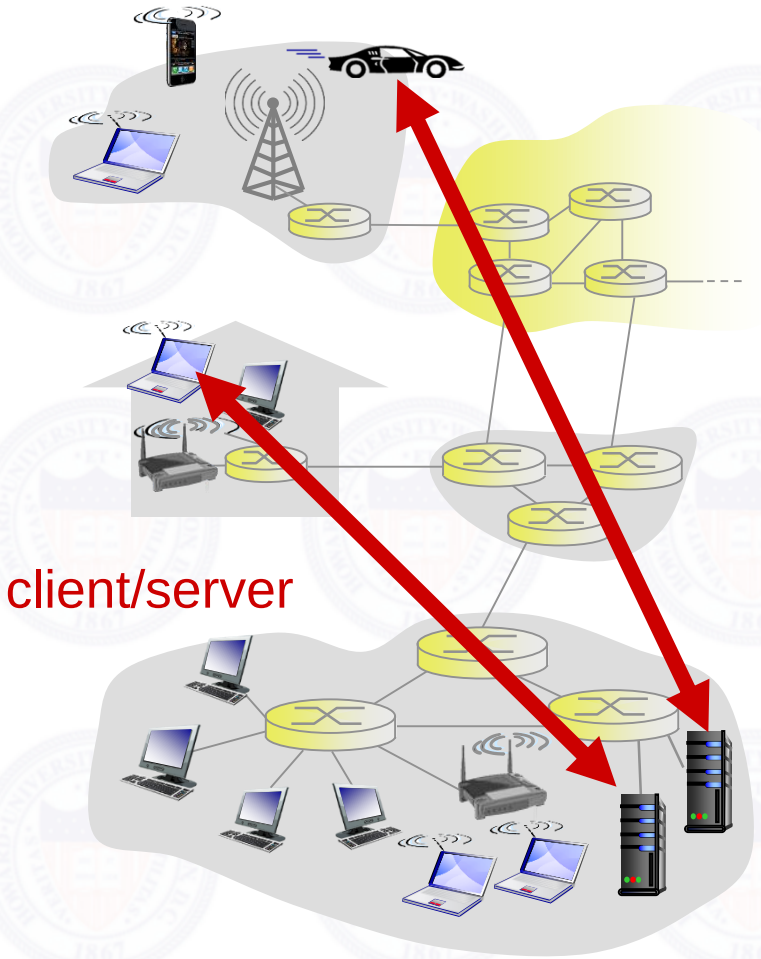
To build network applications …

- We need to know
  - The architecture
  - The protocol
  - How to send/receive data
  - What type of data transport service
    - We need
    - We can have

Application Architectures

- Possible structure of applications
  - Client-server
  - Peer-to-peer (P2P)

Client-Server Architecture



Server:

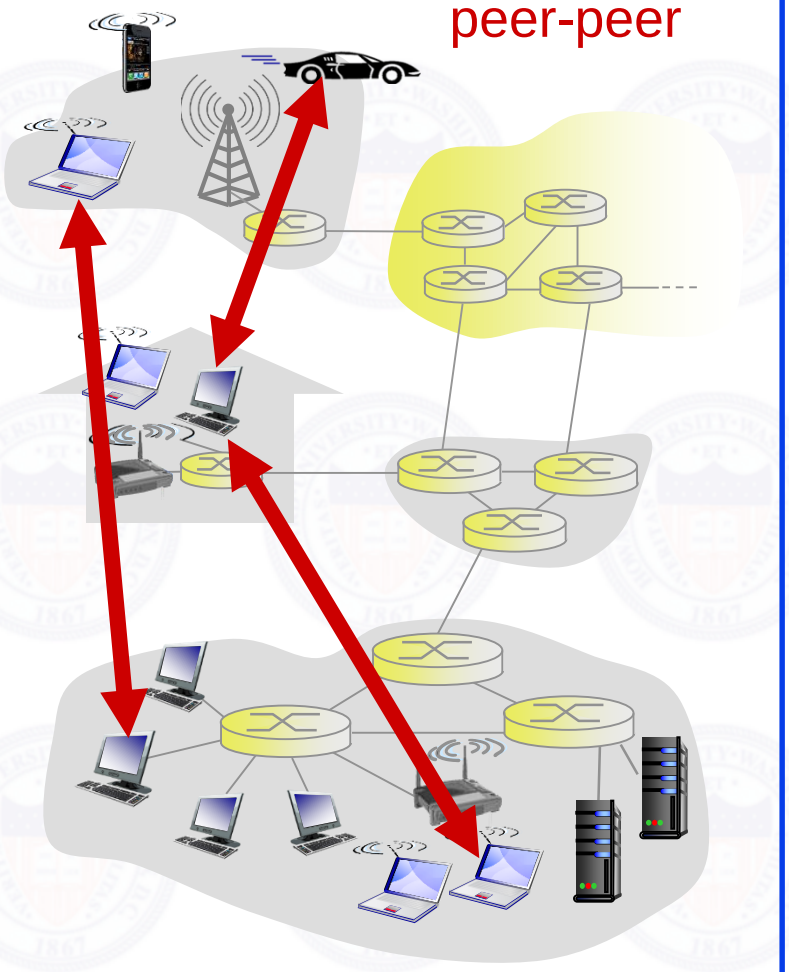
- Always-on host
- Permanent IP address
- Data centers for scaling

Clients:

- Communicate with server
- May be intermittently connected
- May have dynamic IP addresses
- Do not communicate directly with each other

P2P Architecture

- *No* always-on server
- Arbitrary end systems directly communicate
- Peers find other peers and request service from them, 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



App-layer Protocol Defines

- Types of messages exchanged
  - E.g., request, response
- Message syntax
  - What fields in messages & how fields are delineated
- Message semantics
  - Meaning of information in fields
- Rules for when and how processes send & respond to messages

Public-domain protocols:

- Defined in RFCs
- Allows for interoperability
- e.g., HTTP, SMTP

Proprietary protocols:

- e.g., Skype

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

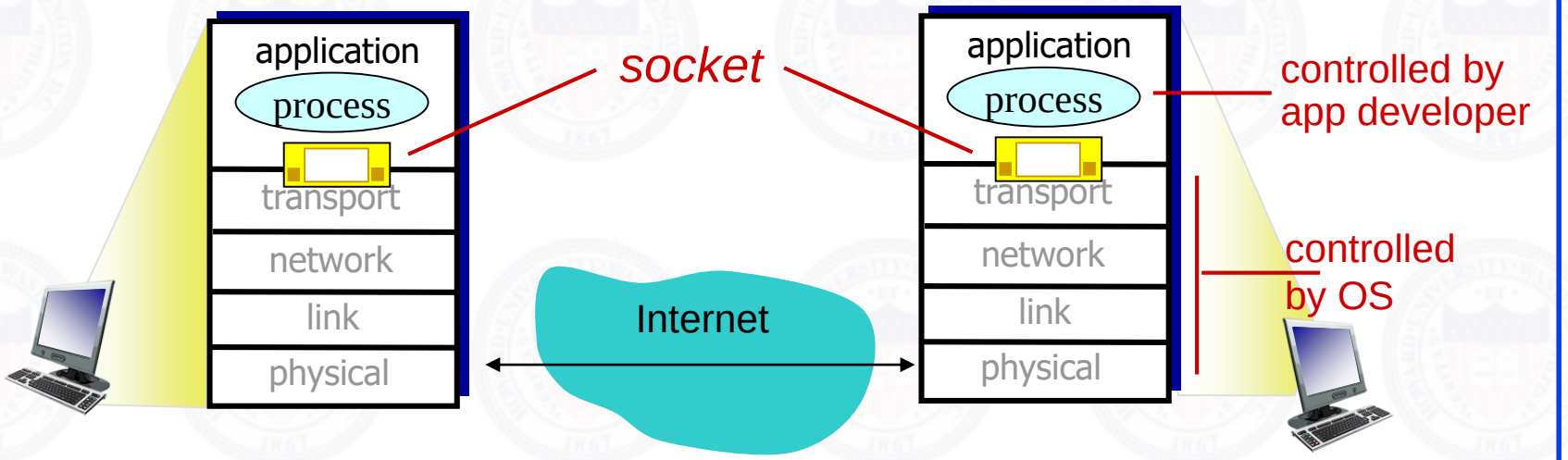
Client process: process that initiates communication

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 to deliver message to socket 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.scs.howard.edu web server:
  - IP address: 138.238.128.178
  - Port number: 80
- more shortly…

What transport service does an app need?

Data loss

- 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, ...

Transport Service Requirements of Common Apps

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
text 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 control, congestion control, timing, or bandwidth guarantee

Q: Why bother? Why is there a UDP?

Internet Apps: Application, 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., Vonage,Skype)	typically UDP

Securing TCP

TCP & UDP

- No encryption
- Clear text passwds sent into socket traverse Internet in clear text

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

- Clear text passwds sent into socket traverse Internet encrypted
- See Chapter 7

Chapter 2: Application layer

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

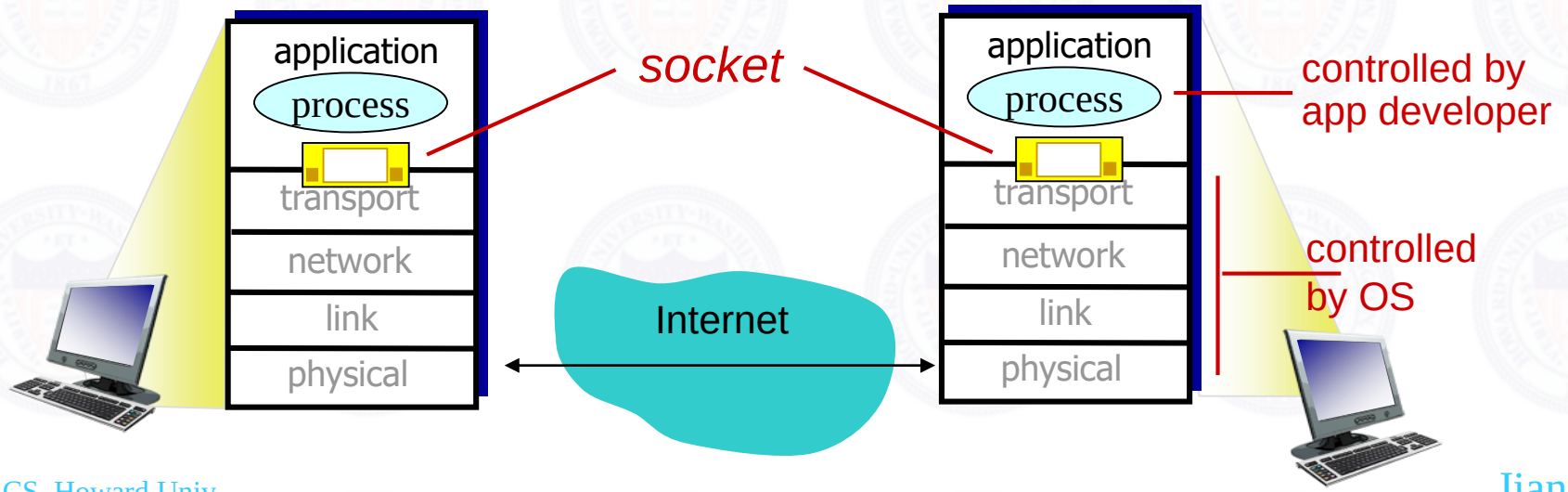
Socket Programming

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

Socket: door between application process and end-end-transport protocol

Two types of transport service via socket API:

- UDP: unreliable datagram
- TCP: reliable, byte stream-oriented



Socket Programming Basics

Two separate programs (server and client) run on different hosts

- Server must be running before client can send anything to it.
- Server must have a socket (door) through which it receives and sends segments
- Similarly client needs a socket
- Socket is locally identified with a port number
  - Analogous to the apt # in a building
- Client needs to know server IP address and socket port number to initiate communication

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:

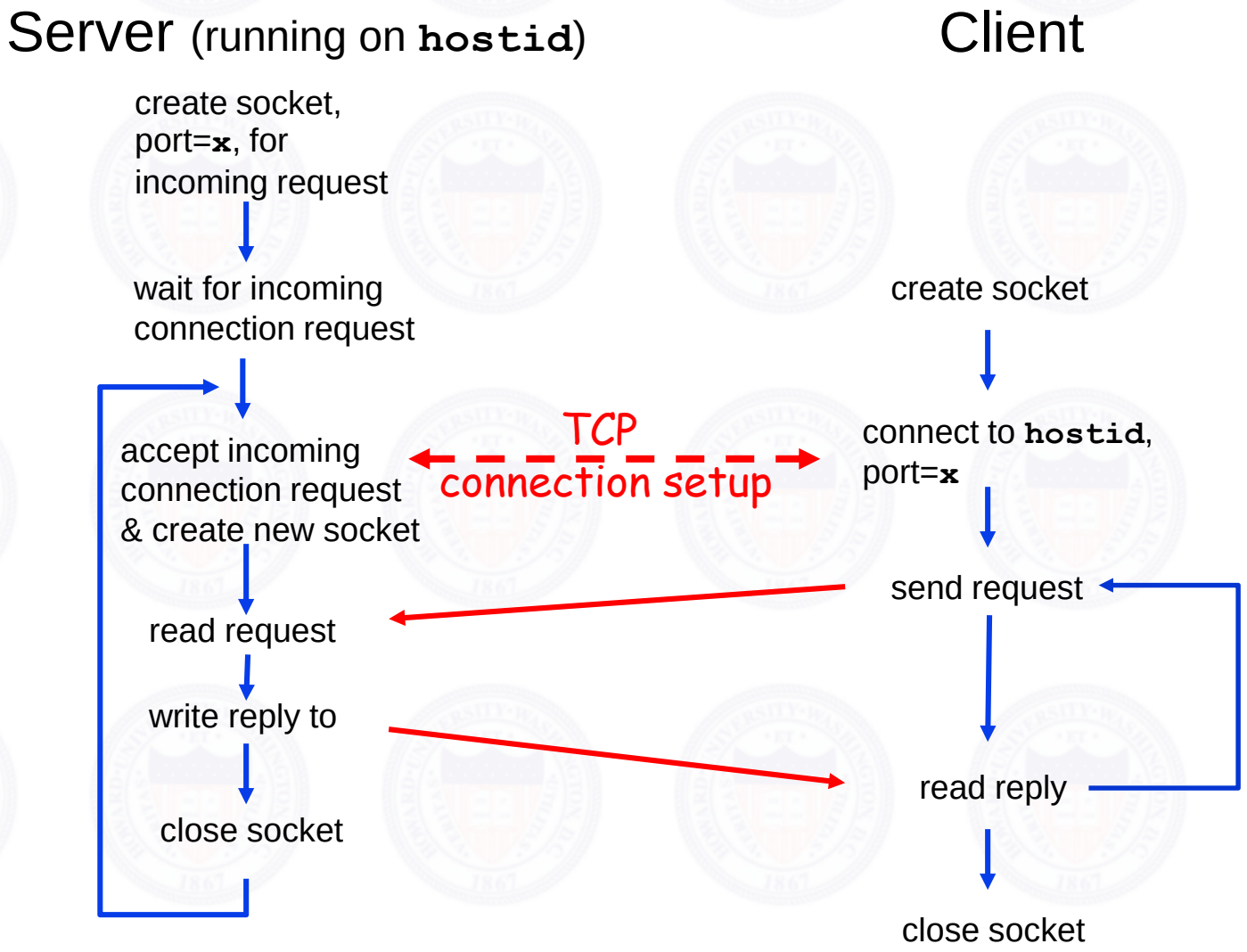
- Creating client-local TCP socket
- Specifying IP address, port number of server process
- Connecting to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - Allows server to talk with multiple clients
  - Source IP and port numbers used to distinguish clients (more in Chap 3)

Application viewpoint

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

Client/Server Socket Interaction: TCP



Socket Programming on *NIX

- We will only talk about socket programming on Linux/Unix/BSD
- Programming assignment must be done on Cygwin or Linux/Unix
- Tutorial for Cygwin
  - http://digiassn.blogspot.com/2006/01/socket-programming-under-cygwin.html

Program Example

1) Client reads one line from standard input, sends to server via socket
2) Server reads one line from socket
3) Server sends the line back to client
4) Client reads, prints the line from socket

The Echo Application Design

- Architecture
  - Client/server
- Protocol
  - Message types
    - Input/request, echo/response
  - Message syntax
    - One or more ASCII characters
    - Why/how do we distinguish between multiple messages?
  - Message semantics
    - Input/request: string to be echoed
    - Echo/response: the echoed string
  - Processing rules
    - Input/request sent by client
    - Echo/response sent by server
    - One echo by server for each received client message

Echo: A Example Program

- Server: http://www.paulgriffiths.net/program/c/echoserv.php
- Client: http://www.paulgriffiths.net/program/c/echoclnt.php

Server Example Header

```
#include <sys/socket.h>    /*  socket definitions        */
#include <sys/types.h>     /*  socket types              */
#include <arpa/inet.h>     /*  inet (3) funtions         */
#include <unistd.h>        /*  misc. UNIX functions      */

#include "helper.h"        /*  our own helper functions  */
#include <stdlib.h>
#include <stdio.h>

/*  Global constants  */

#define ECHO_PORT          (2002)
#define MAX_LINE           (1000)
```

Server Example Skeleton

```
int main(int argc, char *argv[]) {
  if ((list_s = socket (AF_INET, SOCK_STREAM, 0)) < 0) {…}
  if (bind (list_s, (struct sockaddr *) &servaddr, sizeof(servaddr))
      < 0) {…}
  if (listen (list_s, LISTENQ) < 0) {…}
  while ( 1 ) {
    if ((conn_s = accept (list_s, NULL, NULL)) < 0) {…}
    Readline (conn_s, buffer, MAX_LINE-1);
    Writeline( conn_s, buffer, strlen(buffer));
    if ( close (conn_s) < 0 ) {…}
  }
}
```

Creating a Socket

```
int socket(int family,int type,int proto);
```

- `family` specifies the protocol family (**`PF_INET`** for TCP/IP).
- `type` specifies the type of service (**`SOCK_STREAM`**, **`SOCK_DGRAM`**).
- `protocol` specifies the specific protocol (usually 0, which means *the default*).
- Returns a socket descriptor (small integer) or -1 on error.
- Allocates resources needed for a communication endpoint
- But it does not deal with endpoint addressing.

- Used by both client and server

Server Example Skeleton

```
int main(int argc, char *argv[]) {
  if ((list_s = socket (PF_INET, SOCK_STREAM, 0)) < 0) {…}
  if (bind (list_s, (struct sockaddr *) &servaddr, sizeof(servaddr))
      < 0) {…}
  if (listen (list_s, LISTENQ) < 0) {…}
  while ( 1 ) {
    if ((conn_s = accept (list_s, NULL, NULL)) < 0) {…}
    Readline (conn_s, buffer, MAX_LINE-1);
    Writeline( conn_s, buffer, strlen(buffer));
    if ( close (conn_s) < 0 ) {…}
  }
}
```

Assigning An Address to A Socket

```
int bind( int sockfd,
          const struct sockaddr *myaddr,
          int addrlen);
```

const!

- Returns 0 if successful or -1 on error.
- Can be used for both server and client

Generic Socket Address

```
struct sockaddr {
    uint8_t         sa_len;
    sa_family_t     sa_family;
    char            sa_data[14];
};
```

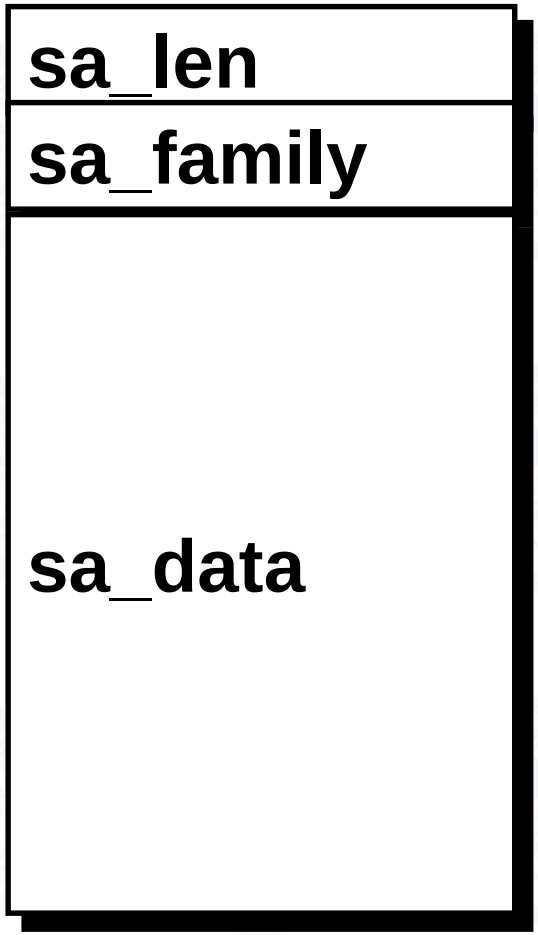
- **sa_family** specifies the address type.
  - AF_INET for (current) Internet addresses
- **sa_data** specifies the address value.
  - For AF_INET we need:
    16 bit port number
    32 bit IP(v4) address

A Special Lind of sockaddr Structure for IPv4

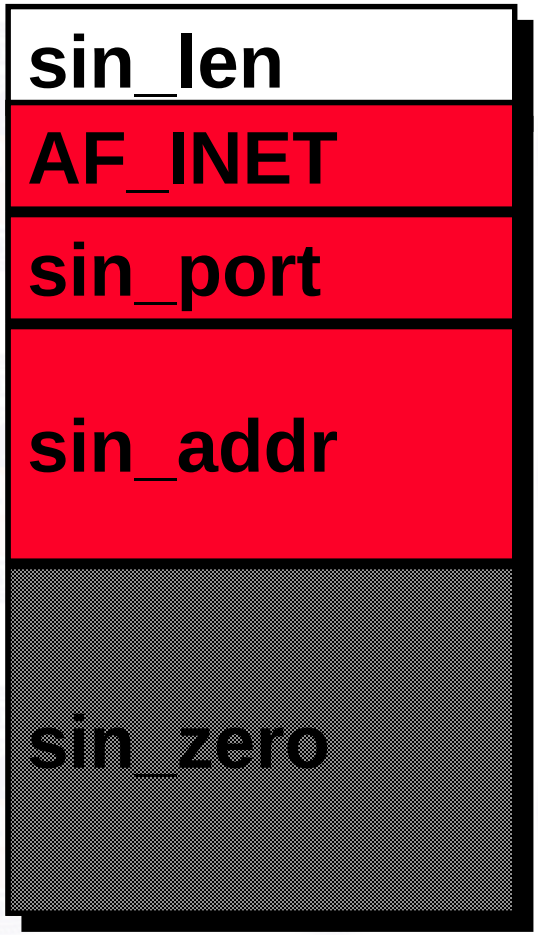
- We don't need to deal with **sockaddr** structures since we will only deal with a real protocol family.
- We can use the **sockaddr_in** structure.
  - But the C functions that make up the sockets API expect structures of type **sockaddr.**

```
struct sockaddr_in {
    uint8_t         sin_len;
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

sockaddr



sockaddr_in



struct in_addr

```
struct in_addr {
    in_addr_t       s_addr;
};
```

in_addr just provides a name for the 'C' type associated with IP addresses.

- POSIX data types

sa_family_t	address family
socklen_t	length of struct
in_addr_t	IPv4 address
in_port_t	IP port number

Network Byte Order

- All values stored in a **`sockaddr_in`** must be in network byte order.
  - **`sin_port`** a TCP/IP port number.
  - **`sin_addr`** an IP address.

Little and Big Endian

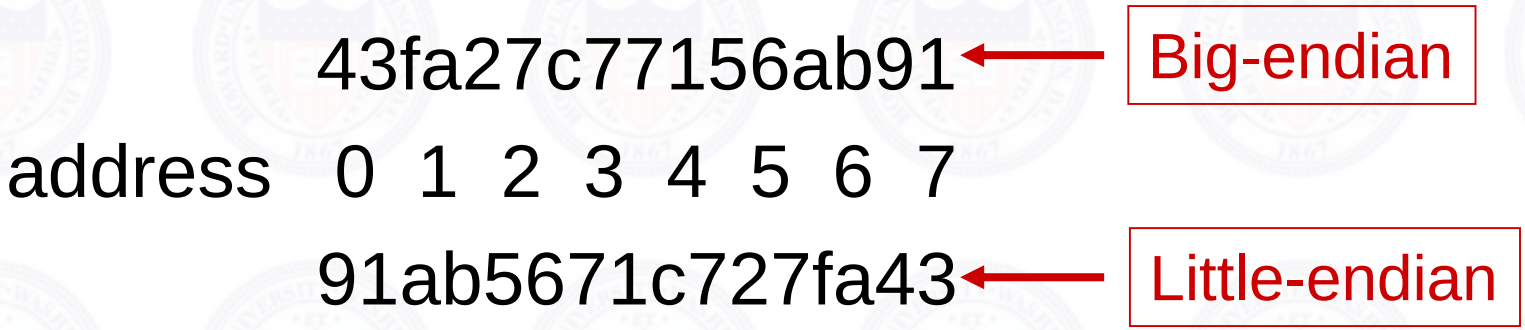
- Consider a 64-bit quantity, composed of bytes 0-7 (LSB-MSB)
- Little-Endian
  - Memory address A will contain byte 0, address A+1 will contain byte 1,….address A+7 will contain byte 7
- Big-Endian
  - Memory address A will contain byte 7, address A+1 will contain byte 6,… address A+7 will contain byte 0
- Network byte order: big-endian

Endianness Example

- Consider the hexadecimal number:

  MSB 0x 43fa27c77156ab91 LSB

- Two options:



Network Byte Order Functions

'h' : host byte order 'n' : network byte order

's' : short (16bit) 'l' : long (32bit)

```
uint16_t htons(uint16_t);
uint16_t ntohs(uint16_t);

uint32_t htonl(uint32_t);
uint32_t ntohl(uint32_t);
```

What is my IP address ?

- How can you find out what your IP address is so you can tell **`bind()`** ?
- There is no realistic way for you to know the right IP address to give bind() - what if the computer has multiple network interfaces?
  - Usually programmers leave that job to users
- specify the IP address as: **`INADDR_ANY`**, this tells the OS to take care of things.

Server Example Skeleton

```
int main(int argc, char *argv[]) {
  if ((list_s = socket (PF_INET, SOCK_STREAM, 0)) < 0) {…}
  if (bind (list_s, (struct sockaddr *) &servaddr, sizeof(servaddr))
      < 0) {…}
  if (listen (list_s, LISTENQ) < 0) {…}
  while ( 1 ) {
    if ((conn_s = accept (list_s, NULL, NULL)) < 0) {…}
    Readline (conn_s, buffer, MAX_LINE-1);
    Writeline( conn_s, buffer, strlen(buffer));
    if ( close (conn_s) < 0 ) {…}
  }
}
```

Listening for Connection Requests

```
int listen (int sockfd, int backlog);
```

- **sockfd** is the TCP socket (already bound to an address)
- **backlog** is the number of incoming connections the kernel should be able to keep track of (queue for us).
- Returns -1 on error (otherwise 0).

Accepting An Incoming Connection

- Once we call **`listen()`**, the O.S. will queue incoming connections
  - Handles the 3-way handshake
  - Queues up multiple connections.

- When our application is ready to handle a new connection, we need to ask the O.S. for the next connection.

Server Example Skeleton

```
int main(int argc, char *argv[]) {
  if ((list_s = socket (PF_INET, SOCK_STREAM, 0)) < 0) {…}
  if (bind (list_s, (struct sockaddr *) &servaddr, sizeof(servaddr))
      < 0) {…}
  if (listen (list_s, LISTENQ) < 0) {…}
  while ( 1 ) {
    if ((conn_s = accept (list_s, NULL, NULL)) < 0) {…}
    Readline (conn_s, buffer, MAX_LINE-1);
    Writeline( conn_s, buffer, strlen(buffer));
    if ( close (conn_s) < 0 ) {…}
  }
}
```

Start Handling A Connection

```
int accept( int sockfd,
            struct sockaddr* cliaddr,
            socklen_t *addrlen);
```

- **sockfd** is the TCP socket.
- **cliaddr** is a pointer to *allocated* space.
- **addrlen** is a *value-result* argument
  - must be set to the size of **cliaddr**
  - on return, will be set to be the number of used bytes in **cliaddr**.
- Returns a new socket descriptor (small positive integer) or -1 on error

Server Example Skeleton

```
int main(int argc, char *argv[]) {
  if ((list_s = socket (PF_INET, SOCK_STREAM, 0)) < 0) {…}
  if (bind (list_s, (struct sockaddr *) &servaddr, sizeof(servaddr))
      < 0) {…}
  if (listen (list_s, LISTENQ) < 0) {…}
  while ( 1 ) {
    if ((conn_s = accept (list_s, NULL, NULL)) < 0) {…}
    Readline (conn_s, buffer, MAX_LINE-1);
    Writeline( conn_s, buffer, strlen(buffer));
    if ( close (conn_s) < 0 ) {…}
  }
}
```

Read/Write Sockets

After **`accept`** returns a new socket descriptor, I/O can be done using the **`read()`** and **`write()`** system calls.

`read()` and **`write()`** operate a little differently on sockets (vs. file operation)!

- Might input or output fewer bytes than requested.
- Make sure to check the actual number of bytes read or written on the socket.

Read Subroutine Skeleton

```
ssize_t Readline(int sockd, void *vptr, size_t maxlen) {
  for ( n = 1; n < maxlen; n++ ) {
    if ( (rc = read(sockd, &c, 1)) == 1 ) {…}
    else if ( rc == 0 ) {…}
    else {
        if (errno == EINTR)
          continue;
        return -1;
    }
  }

  …
}
```

Reading from A TCP Socket

```
int read( int fd, char *buf, int max);
```

- By default **`read()`** will block until data is available.
- Reading from a TCP socket may return less than max bytes (whatever is available).
- You must be prepared to read data 1 byte at a time!

Read Subroutine Skeleton

```
ssize_t Readline(int sockd, void *vptr, size_t maxlen) {
  for ( n = 1; n < maxlen; n++ ) {
    if ( (rc = read(sockd, &c, 1)) == 1 ) {…}
    else if ( rc == 0 ) {…}
    else {
      if (errno == EINTR)
        continue;
      return -1;
    }
  }

  …
}
```

System Calls and Errors

- In general, systems calls return a negative number to indicate an error.
  - We often want to find out what error.
  - Servers generally add this information to a log.
  - Clients generally provide some information to the user.

`extern int errno;`

- Whenever an error occurs, system calls set the value of the global variable **`errno`**.
  - You can check errno for specific errors.
  - You can use support functions to print out or log an ASCII text error message.

When is `errno` valid?

- **`errno`** is valid only after a system call has returned an error.
  - System calls don't *clear* **`errno`** on success.
  - If you make another system call you may lose the previous value of **`errno`**.
    - **`printf`** makes a call to **`write`** (a system call)!

Error codes

```
#include <errno.h>
```

- Error codes are defined in errno.h

```
EAGAIN   EBADF      EACCESS
EBUSY    EINTR      EINVAL
EIO      ENODEV     EPIPE
...
```

Support Routines

```
#include <stdio.h>
void perror(const char *s);
```

- Print the string pointed to by **s**, then ": ", and the error message corresponding to **errno**.

```
#include <string.h>
char *strerror(int errnum);
```

- Map the error number in *errnum* to an error message string and return a pointer to it.

Write Subroutine Skeleton

```
ssize_t Writeline(int sockd, const void *vptr, size_t n) {
  while (nleft > 0) {
    if ((nwritten = write(sockd, buffer, nleft)) <= 0 ) {
      if (errno == EINTR)
        nwritten = 0;
      else
        return -1;
    }
    …
  }

  …
}
```

Writing to A TCP Socket

```
int write( int fd, char *buf, int num);
```

- write might not be able to write all `num` bytes (on a nonblocking socket).

Server Example Skeleton

```
int main(int argc, char *argv[]) {
  if ((list_s = socket (PF_INET, SOCK_STREAM, 0)) < 0) {…}
  if (bind (list_s, (struct sockaddr *) &servaddr, sizeof(servaddr))
      < 0) {…}
  if (listen (list_s, LISTENQ) < 0) {…}
  while ( 1 ) {
    if ((conn_s = accept (list_s, NULL, NULL)) < 0) {…}
    Readline (conn_s, buffer, MAX_LINE-1);
    Writeline( conn_s, buffer, strlen(buffer));
    if ( close (conn_s) < 0 ) {…}
  }
}
```

Terminating a TCP connection

```
int close(int fd);
```

- Both ends of the connection must call the `close()` system call to close the connection
  - Either end can close first
    - Means “no more data to send”
- If the other end has closed the connection, and there is no buffered data, reading from a TCP socket returns 0 to indicate EOF.
- Return 0 if successful, -1 otherwise.

Client Example Skeleton

```
int main(int argc, char *argv[]) {
  ...
  if ((conn_s = socket (PF_INET, SOCK_STREAM, 0)) < 0) {...}
  ...
  if (connect (conn_s, (struct sockaddr *) &servaddr,
    sizeof(servaddr)) < 0 ) {...}
  ...
  Writeline (conn_s, buffer, strlen(buffer));
  Readline (conn_s, buffer, MAX_LINE-1);
  close (conn_s);
  ...
}
```

Connect to Server TCP

```
int connect( int sockfd,
             const struct sockaddr *server,
             socklen_t addrlen);
```

- **`sockfd`** is an already created TCP socket.
- **`server`** contains the endpoint address of the server (IP Address and TCP port number)
- Returns 0 if OK, -1 on error
- The O.S. will automatically assign the local endpoint address (TCP port number, IP address).
  - If connection is wanted from a specific endpoint, need to `bind()` first.
- Use by client

Client Example Skeleton

```
int main(int argc, char *argv[]) {
  ...
  if ((conn_s = socket (AF_INET, SOCK_STREAM, 0)) < 0) {...}
  ...
  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family      = AF_INET;
  servaddr.sin_port        = htons(port);

  if (inet_aton (szAddress, &servaddr.sin_addr) <= 0) {...}

  if (connect (conn_s, (struct sockaddr *) &servaddr,
    sizeof(servaddr)) < 0 ) {...}
  ...
}
```

IPv4 Address Conversion

```
int inet_aton( char *, struct in_addr *);
```

Convert ASCII dotted-decimal IP address to network byte order 32 bit value. Returns 1 on success, 0 on failure.

```
char *inet_ntoa(struct in_addr);
```

Convert network byte ordered value to ASCII dotted-decimal (a string).

Which port to use?

- Clients typically don't care what port they are assigned.
- When you call bind you can tell it to assign you any available port:

```
myaddr.port = htons(0);
```

Name to Address Conversion

- There is a library of functions that act as DNS client (resolver).
  - you don't need to write DNS client code to use DNS!
- With some OSs (e.g. Solaris) you need to explicitly link with the DNS resolver library:

  `-lnsl` (**`nsl`** is "Name Server Library")

DNS library functions

```
gethostbyname

gethostbyaddr

gethostbyname2
```

gethostbyname

```
struct hostent *gethostbyname( const
  char *hostname);
```

struct hostent is defined in netdb.h:

```
#include <netdb.h>
```

struct hostent

```
struct hostent {
  char *h_name;
  char **h_aliases;
  int h_addrtype;
  int h_length;
  char **h_addr_list;
};
```

official name (canonical)

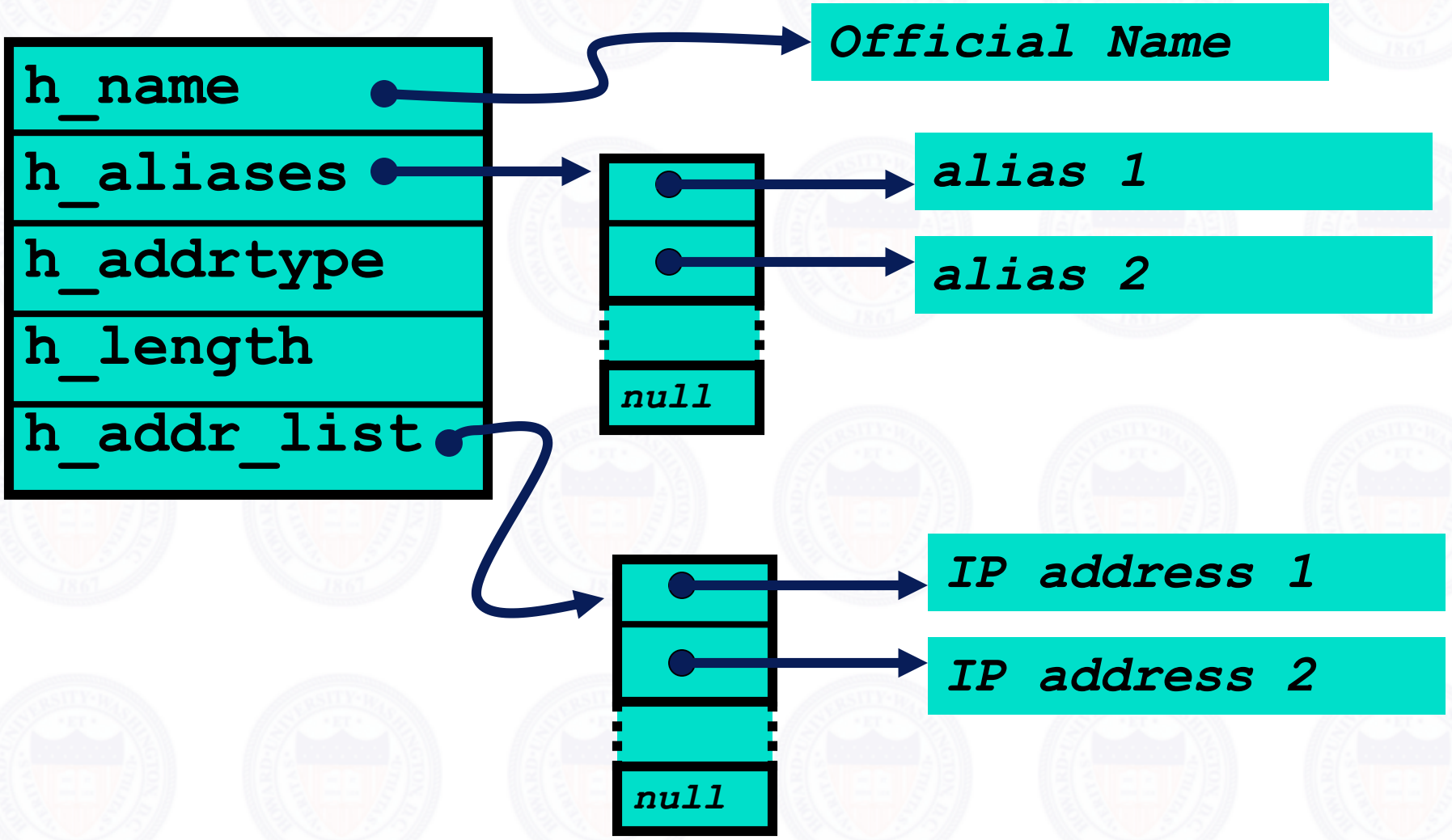
other names

AF_INET or AF_INET6

address length (4 or 16)

array of ptrs to addresses

hostent picture



Which Address?

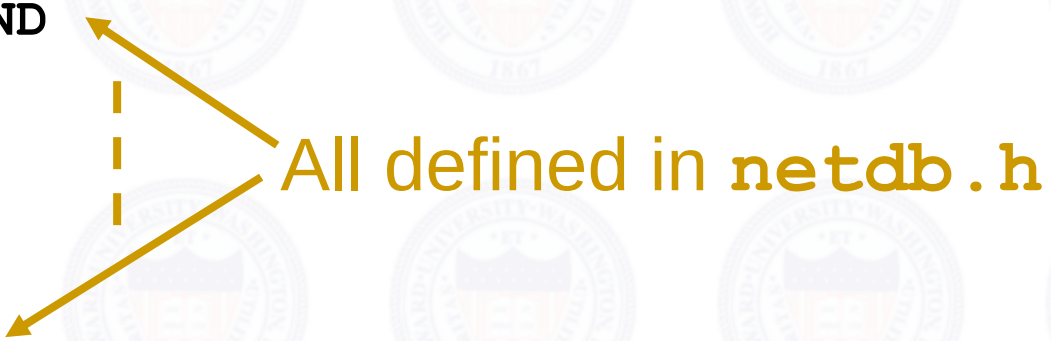
On success, gethostbyname returns the address of a hostent that has been created.

- has an array of ptrs to IP addresses
- Usually use the first one:

```
#define h_addr h_addr_list[0]
```

Gethostbyname and errors

- On error **gethostbyname** return null.
- **Gethostbyname** sets the global variable **h_errno** to indicate the exact error:
  - **HOST_NOT_FOUND**
  - **TRY_AGAIN**
  - **NO_RECOVERY**
  - **NO_DATA**
  - **NO_ADDRESS**



Getting at the address:

```
char **h_addr_list;
```

```
h = gethostbyname("joe.com");
sockaddr.sin_addr.s_addr =
  *(h->h_addr_list[0]);
```

This won't work!!!!

`h_addr_list[0]` is a `char*` !

Using memcpy

- You can copy the 4 bytes (IPv4) directly:

```
h = gethostbyname("joe.com");

memcpy(&sockaddr.sin_addr,
        h->h_addr_list[0],
       sizeof(struct in_addr));
```

Network Byte Order

- All the IP addresses returned via the hostent are in network byte order!

gethostbyaddr

```
struct hostent *gethostbyaddr(
  const char *addr,
  size_t len,
  int family);
```

- **family** is usually **AF_INET**
- **addr** points to a **in_addr** struct
- **len** is **sizeof(struct in_addr)**

Some other functions

`uname` : get hostname of local host

`getservbyname` : get port number for a named service

`getservbyaddr` : get name for service associated with a port number

Client Example Skeleton

```
int main(int argc, char *argv[]) {
  ...
  if ((conn_s = socket (AF_INET, SOCK_STREAM, 0)) < 0) {...}
  ...
  if (connect (conn_s, (struct sockaddr *) &servaddr,
    sizeof(servaddr)) < 0 ) {...}
  ...
  Writeline (conn_s, buffer, strlen(buffer));
  Readline (conn_s, buffer, MAX_LINE-1);
  close (conn_s);
  ...
}
```

Missing in the code online

Socket Programming *with UDP*

UDP: no "connection" between client and server

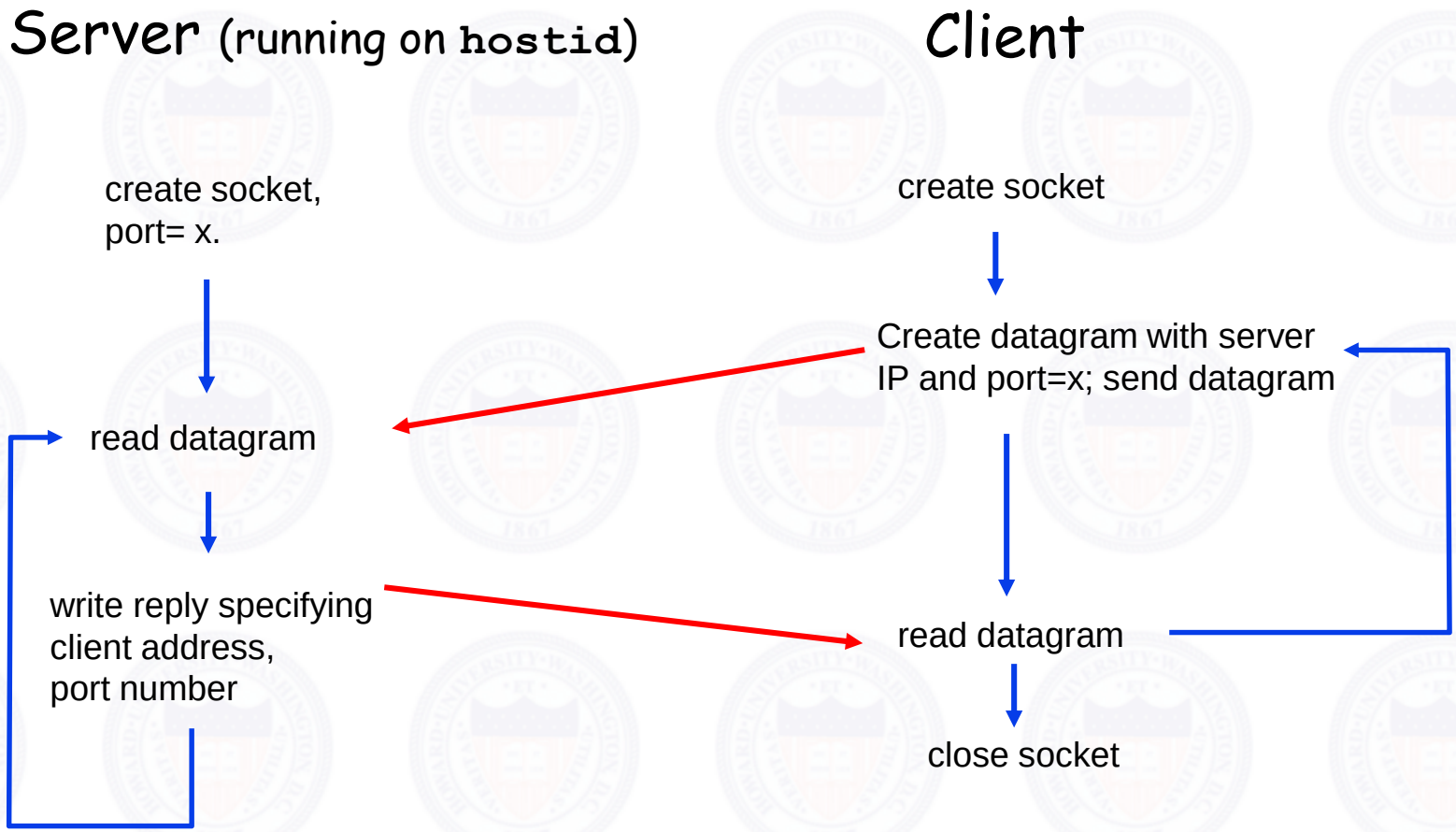
- No handshaking
- Sender explicitly attaches IP address and port of destination to each packet
- Receiver must extract IP address, port of sender from received packet

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



Creating a UDP socket

```
int socket(int family,int type,int proto);


int sock;
sock = socket (PF_INET, SOCK_DGRAM, 0);
if (sock<0) { /* ERROR */ }
```

Binding to Well Known Address (typically done by server only)

```
int mysock;
struct sockaddr_in myaddr;

mysock = socket(PF_INET,SOCK_DGRAM,0);
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons( 1234 );
myaddr.sin_addr = htonl( INADDR_ANY );

bind(mysock, &myaddr, sizeof(myaddr));
```

Sending UDP Datagrams

```
ssize_t sendto( int sockfd,
                void *buff,
                size_t nbytes,
                int flags,
                const struct sockaddr* to,
                socklen_t addrlen);
```

- **sockfd** is a UDP socket
- **buff** is the address of the data (**nbytes** long)
- **to** is the address of a sockaddr containing the destination address.
- Return value is the number of bytes sent, or -1 on error.

sendto()

- You can send 0 bytes of data!

- Some possible errors :
  - **EBADF, ENOTSOCK**: bad socket descriptor
  - **EFAULT**: bad buffer address
  - **EMSGSIZE**: message too large
  - **ENOBUFS**: system buffers are full

More `sendto()`

- The return value of **`sendto()`** indicates how much data was accepted by the O.S. for sending as a datagram - not how much data made it to the destination.
- There is no error condition that indicates that the destination did not get the data!!!

Receiving UDP Datagrams

```
ssize_t recvfrom( int sockfd,
                  void *buff,
                  size_t nbytes,
                  int flags,
                  struct sockaddr* from,
                  socklen_t *fromaddrlen);
```

- **`sockfd`** is a UDP socket
- **`buff`** is the address of a buffer (**`nbytes`** long)
- **`from`** is the address of a sockaddr.
- Return value is the number of bytes received and put into buff, or -1 on error.

recvfrom()

- If buff is not large enough, any extra data is lost forever...
- You can receive 0 bytes of data!
- The **sockaddr** at **from** is filled in with the address of the sender.
- You should set **fromaddrlen** before calling.
- If **from** and **fromaddrlen** are NULL we don't find out who sent the data.
- Same errors as **sendto**, but also:
  - **EINTR**: System call interrupted by signal.
- Unless you do something special - **recvfrom** doesn't return until there is a datagram available.

Typical UDP Client Code

- Create UDP socket.
- Create **`sockaddr`** with address of server.
- Call **`sendto()`**, sending request to the server. **No call to bind() is necessary!**
- Possibly call **`recvfrom()`** (if we need a reply).

Typical UDP Server Code

- Create UDP socket and bind to well known address.
- Call **`recvfrom()`** to get a request, noting the address of the client.
- Process request and send reply back with **`sendto()`.**

UDP Echo Server

```
int mysock;
struct sockaddr_in myaddr, cliaddr;
char msgbuf[MAXLEN];
socklen_t clilen;
int msglen;

mysock = socket(PF_INET,SOCK_DGRAM,0);
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons( S_PORT );
myaddr.sin_addr = htonl( INADDR_ANY );
bind(mysock, &myaddr, sizeof(myaddr));
while (1) {
    len=sizeof(cliaddr);
    msglen=recvfrom(mysock,msgbuf,MAXLEN,0,cliaddr,&clilen);
    sendto(mysock,msgbuf,msglen,0,cliaddr,clilen);
}
```

NEED TO CHECK FOR ERRORS!!!

Timeout When Calling `recvfrom()`

- It might be nice to have each call to **`recvfrom()`** return after a specified period of time even if there is no incoming datagram.
- We can do this by using **`SIGALRM`** and wrapping each call to **`recvfrom()`** with a call to **`alarm()`**

Wrapping recvfrom() with alarm()

```
ssize_t my_recvfrom (…) {
   ssize_t n;
   signal(SIGALRM, sig_alrm);
   alarm(max_time_to_wait);
   if ((n = recvfrom(…) < 0)
       if (errno==EINTR)
            /* timed out */
       else
            /* some other error */
   else
       /* no error or time out
       - turn off alarm */
       alarm(0);
   return n;
}
```

Asynchronous Errors

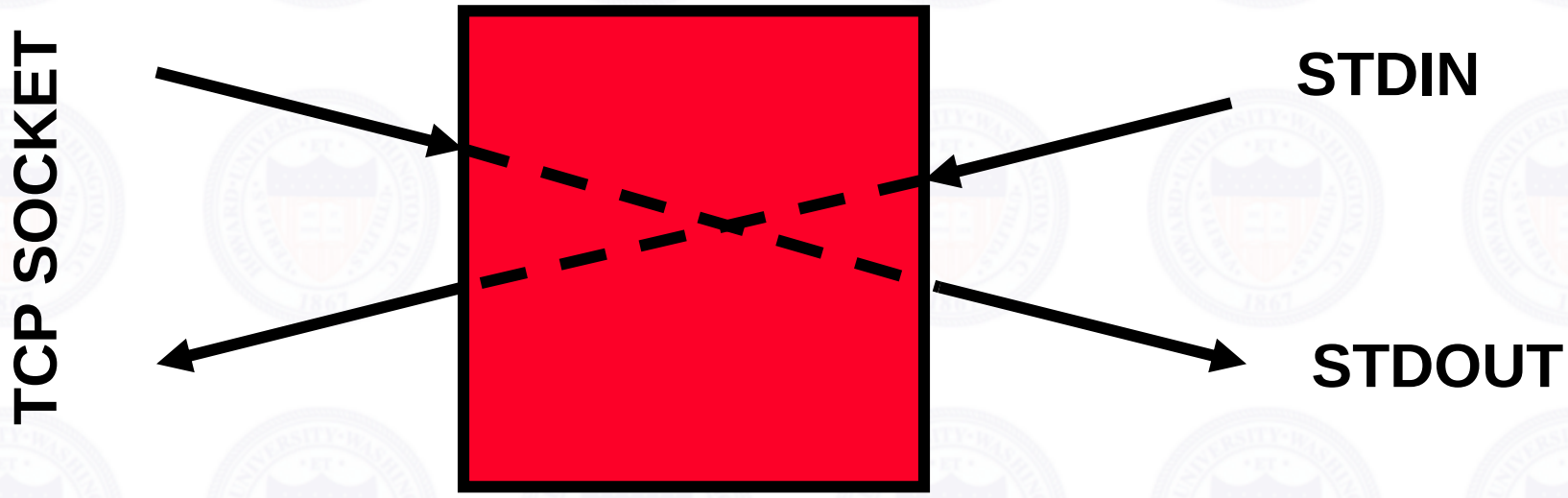
- What happens if a client sends data to a server that is not running?
  - ICMP "port unreachable" error is generated by receiving host and sent to sending host.
  - The ICMP error may reach the sending host after sendto() has already returned!
  - The next call dealing with the socket could return the error.

I/O Multiplexing

- We often need to be able to monitor multiple descriptors:
  - a generic TCP client (like telnet)
  - A server that handles both TCP and UDP
  - Client that can make multiple concurrent requests (browser?).

Example - Generic TCP Client

- Input from standard input should be sent to a TCP socket.
- Input from a TCP socket should be sent to standard output.



- How do we know when to check for input from each source?

Options

- Use nonblocking I/O.
  - use fcntl() to set O_NONBLOCK
- Use alarm and signal handler to interrupt slow system calls.
- Use multiple processes/threads.
- Use functions that support checking of multiple input sources at the same time.

Nonblocking I/O

- use `fcntl()` to set `O_NONBLOCK`:

```
int flags;
flags = fcntl(sock,F_GETFL,0);
fcntl(sock,F_SETFL,flags | O_NONBLOCK);
```

- Now calls to `read()` (and other system calls) will return an error and set `errno` to `EWOULDBLOCK`.

```
while (! done) {
   if ( (n=read(STDIN_FILENO,…)<0)){
      if (errno != EWOULDBLOCK)
         /* ERROR */
   }
   else write(tcpsock,…)

   if ( (n=read(tcpsock,…)<0)) {
      if (errno != EWOULDBLOCK)
         /* ERROR */
   }
   else write(STDOUT_FILENO,…)
}
```

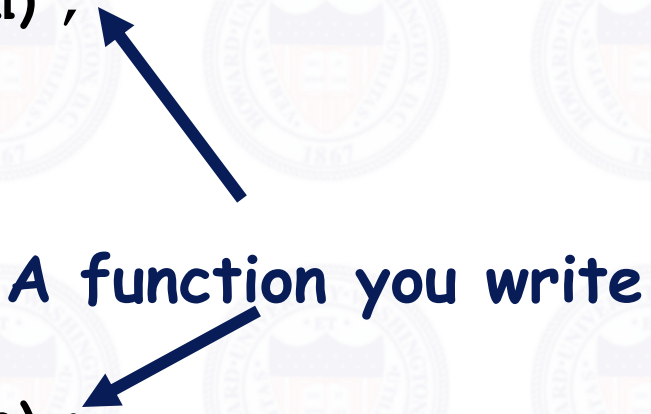
The Problem With Nonblocking I/O

- Using blocking I/O allows the Operating System to put your process to sleep when nothing is happening (no input). Once input arrives, the OS will wake up your process and read() (or whatever) will return.
- With nonblocking I/O, the process will chew up all available processor time!!!

Using Alarms

```
signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(STDIN_FILENO,…);
...

signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(tcpsock,…);
...
```



Alarming Problem

What will happen to the response time ?

What is the ‘right’ value for MAX_TIME?

`select()`

- The `select()` system call allows us to use blocking I/O on a *set* of descriptors (file, socket, …).

- For example, we can ask `select` to notify us when data is available for reading on either STDIN or a TCP socket.

select()

```
int select( int maxfd,
            fd_set *readset,
            fd_set *writeset,
            fd_set *excepset,
          const struct timeval *timeout);
```

maxfd: highest number assigned to a descriptor.

readset: set of descriptors we want to read from.

writeset: set of descriptors we want to write to.

excepset: set of descriptors to watch for exceptions.

timeout: maximum time select should wait

struct timeval

```
struct timeval {
   long tv_sec;      /* seconds */
   long tv_usec;     /* microseconds */
}

struct timeval max = {1,0};
```

fd_set

- Implementation is not important
- Operations you can use with an **fd_set**:

```
void FD_ZERO( fd_set *fdset);
void FD_SET( int fd, fd_set *fdset);
void FD_CLR( int fd, fd_set *fdset);
int FD_ISSET( int fd, fd_set *fdset);
```

Using `select()`

- Create **`fd_set`**

```
fd_set inputfds;
```

- Clear the whole thing with **`FD_ZERO`**

```
FD_ZERO(&inputfds);
```

- Add each descriptor you want to watch using **`FD_SET`**.

```
FD_SET(tcpsock,&inputfds);
```

- Call **`select`**

```
if (select(max+1,&inputfds,NULL,NULL,NULL)) < 0
```

- when **`select`** returns, use **`FD_ISSET`** to see if I/O is possible on each descriptor.

```
if (FD_ISSET(tcpsock,&inputfds)) ...
```

Example Using `select()`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <stdio.h>

…
/* s1 and s2 are two file descriptors for sockets */
fd_set fds;
struct timeval timeout;
int rc, result;
timeout.tv_sec = 3;   timeout.tv_usec = 0;
FD_ZERO(&fds);   FD_SET(s1, &fds);   FD_SET(s2, &fds);
rc = select(sizeof(fds)*8, &fds, NULL, NULL, &timeout);
if (rc == -1) {
  perror("select failed"); return -1;
}
else if (rc > 0)  {
  if (FD_ISSET(s1, &fds)) { /* Read from socket s1 */}
  if (FD_ISSET(s2, &fds)) { /* Read from socket s2 */}
}
```

Man Page

- http://www.freebsd.org/cgi/man.cgi
  - System calls

Before Developing a Net. Application

- Know the architecture
- Know the application layer protocol
  - Message types, syntax, semantics, processing rules
  - Read protocol document (e.g RFC) or complete the design carefully.
- Choose the right transport layer service
  - Is data reliability required?
  - Is there any time constraints?
  - Will the transport layer service interfere with the application layer protocol?
  - And so on …

The Echo Application Design

- Architecture
  - Client/server
- Protocol
  - Message type
    - Request
    - Response
  - Message syntax
    - Request: [body]

      [body] : >= 2 bytes of ASCII, last byte is null.
    - Response : [body]

      [body] : >= 2 bytes of ASCII, last byte is null.
  - Message semantics
    - Request

      [body]: the string to be echoed
    - Response

      [body]: the echoed string
  - Rules
    - Request sent by client
    - Response sent by server
    - One response for one request

Echo+

- Client send one non-empty string
- Client requests the server to send back one or more (<= 1000) copies of each string

The Echo+ Protocol 1

- Message type
  - Request
  - Response
- Message syntax
  - Request: [count][body]

    [count]: 2 bytes of unsigned int. 0 ~ 1000

    [body]: >= 2 bytes of ASCII, last byte is null.
  - Response: [body]…[\n]

    [body]: >= 2 bytes of ASCII, last byte is null.
- Message semantics
  - Request:

    [count]: number of copies

    [body]: the string to be echoed.
  - Response:

    [body]: A sequence of echoed strings.
- Rules
  - Request sent by client
  - Response sent by server
  - One response for one request

The Echo+ Protocol 2

- Message type
  - Request
  - Response
- Message syntax
  - Request: [count][body]

    [count]: 2 bytes of unsigned int. 0 ~ 1000

    [body]: >= 2 bytes of ASCII, last byte is null.
  - Response: [body]

    [body]: >= 1 byte of ASCII, last byte is null.
- Message semantics
  - Request:

    [count]: number of copies

    [body]: the string to be echoed
  - Response:

    [body]: one echoed string. An empty string means the responses end for a request.
- Rules
  - Request sent by client
  - Response sent by server
  - Two or more (count + 1) responses for one request.

Chapter 2: Application Layer

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

Web and HTTP

First some jargon

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

`www.someschool.edu/someDept/pic.gif`

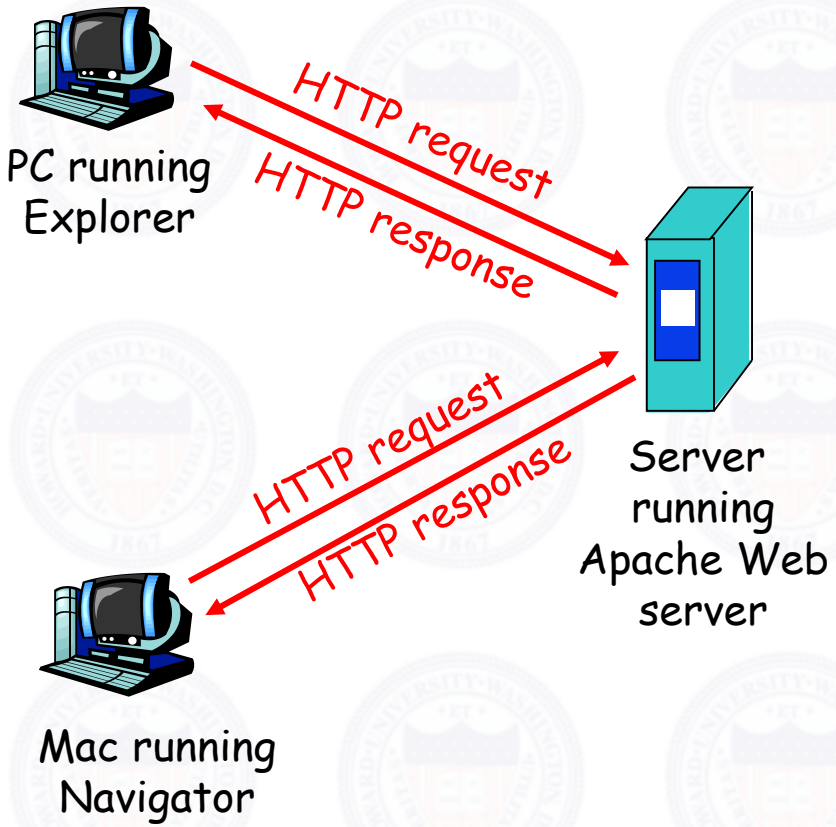
host name
case insensitive

path name
case sensitivity depends on server OS

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 0.9: by Tim Berners-Lee, deprecated
- HTTP 1.0: RFC 1945, 05/1996
- HTTP 1.1: RFC 2068, 07/1999



HTTP Overview (cont'd)

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. It handles each request independently.

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

How HTTP Uses TCP 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

time

Nonpersistent HTTP (cont'd)

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

4. HTTP server closes TCP connection.

6. Steps 1-5 repeated for each of 10 jpeg objects

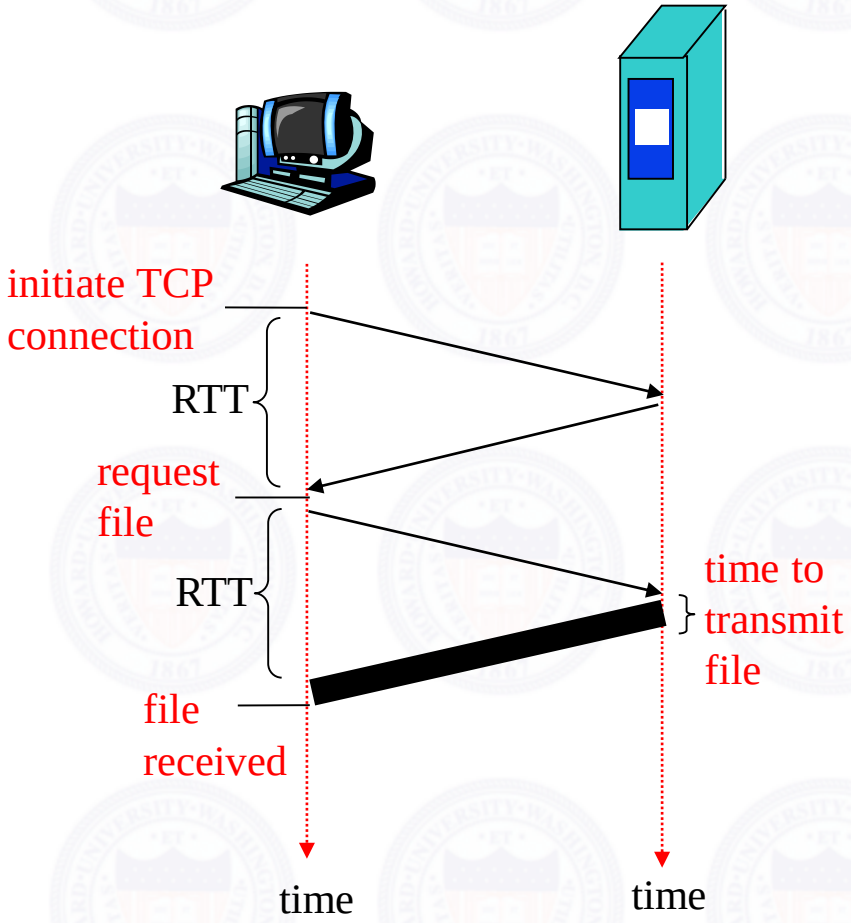
Nonpersistent HTTP Response Time

Definition of RTT: time to send a *very 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
  - Single hop? Multi hop?

total = 2RTT+transmit 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

Persistent HTTP (cont'd)

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

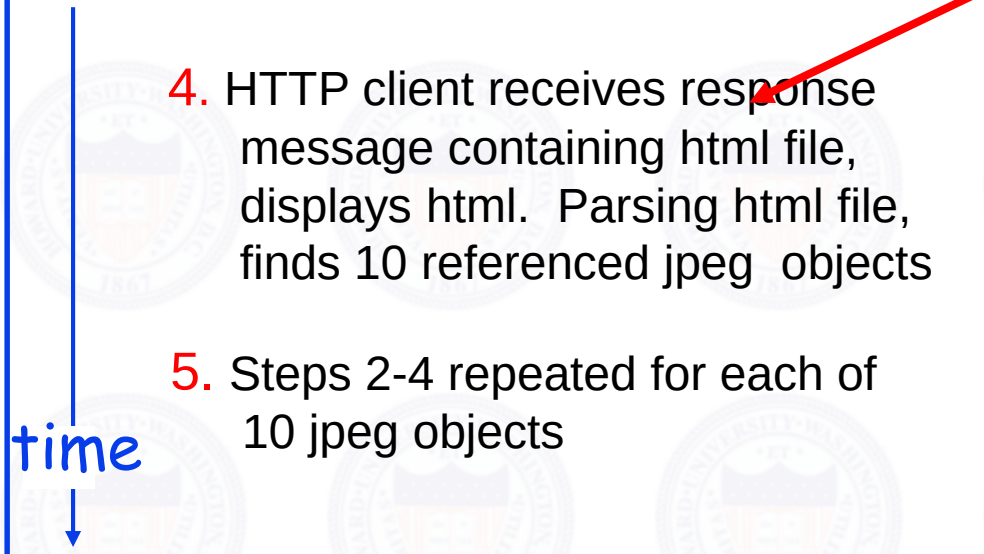
1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

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

time

Persistent HTTP (cont'd)



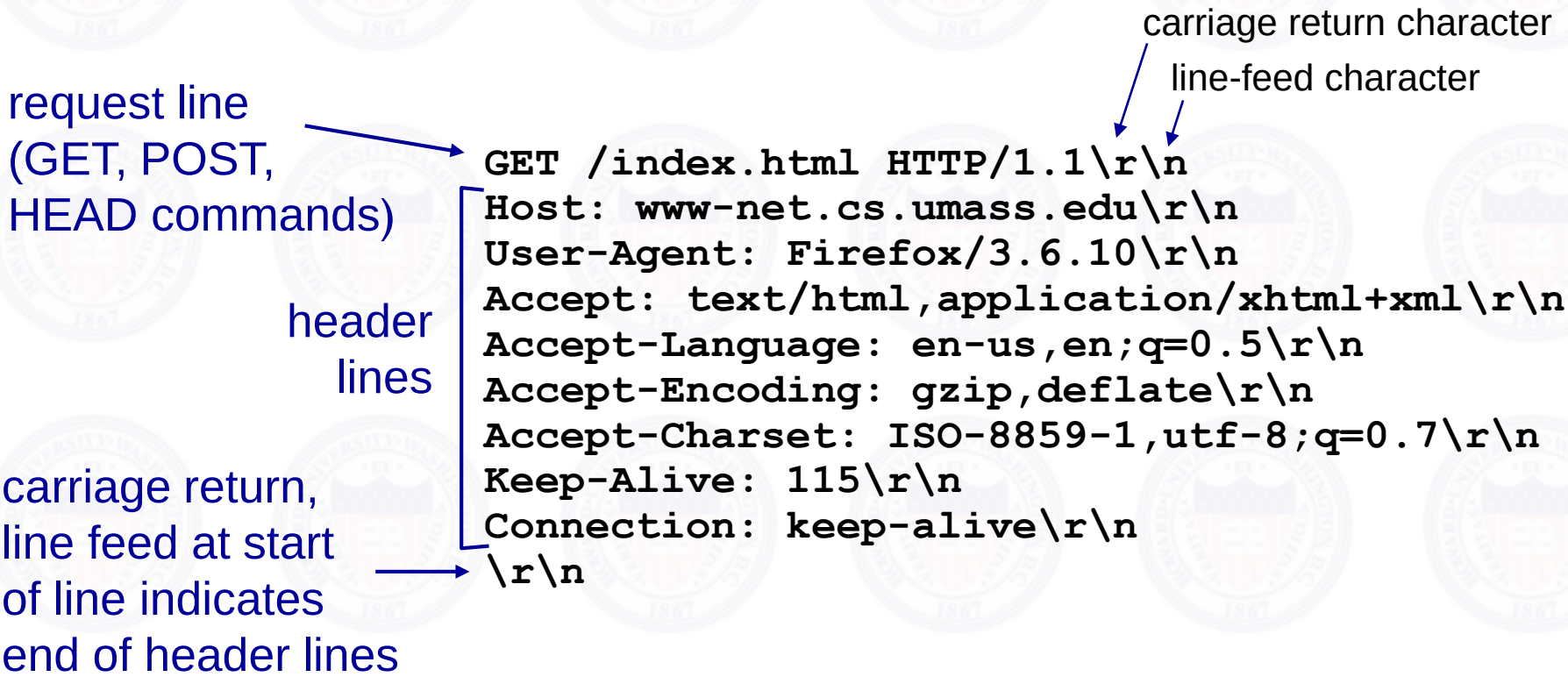
4. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

5. Steps 2-4 repeated for each of 10 jpeg objects

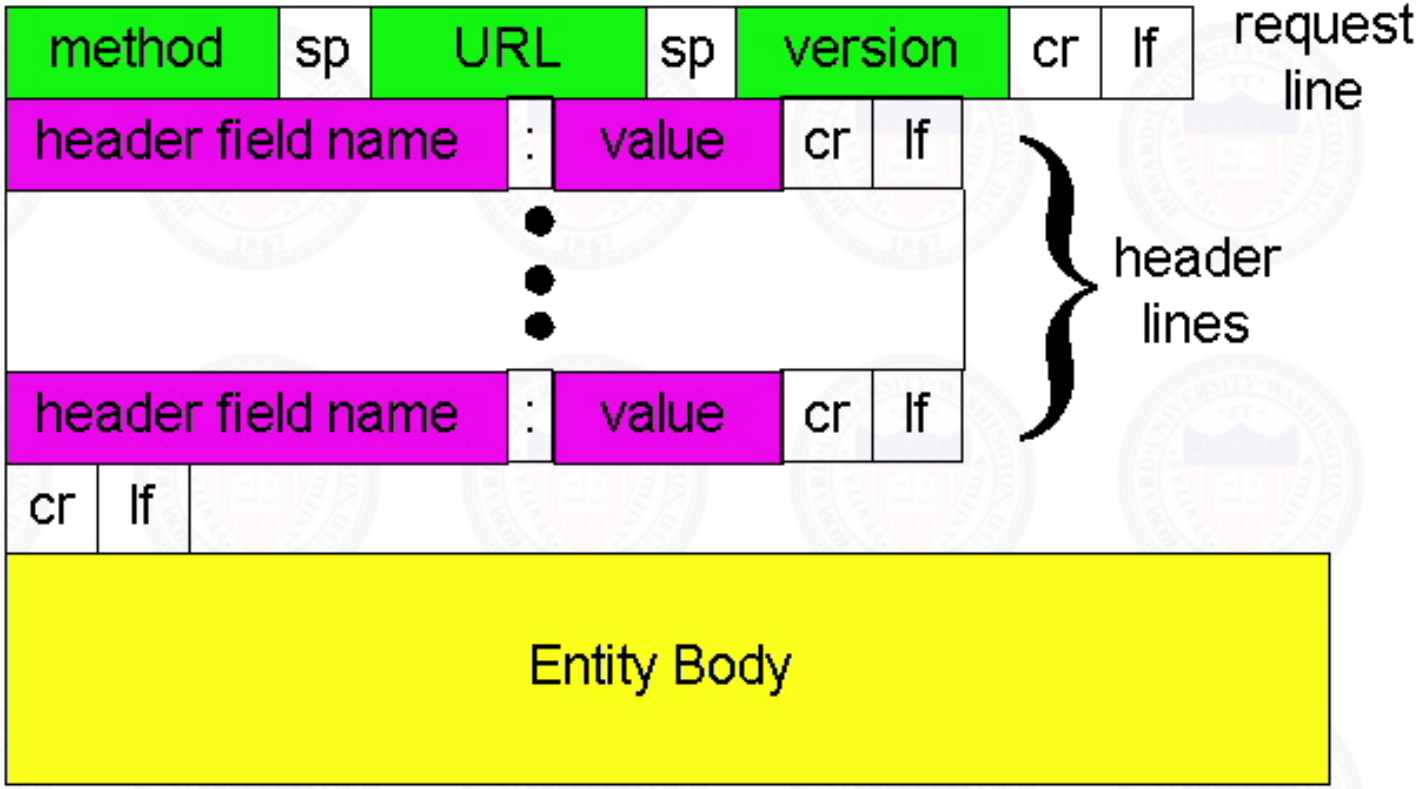
6. HTTP server closes TCP connection.

HTTP Request Message

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



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

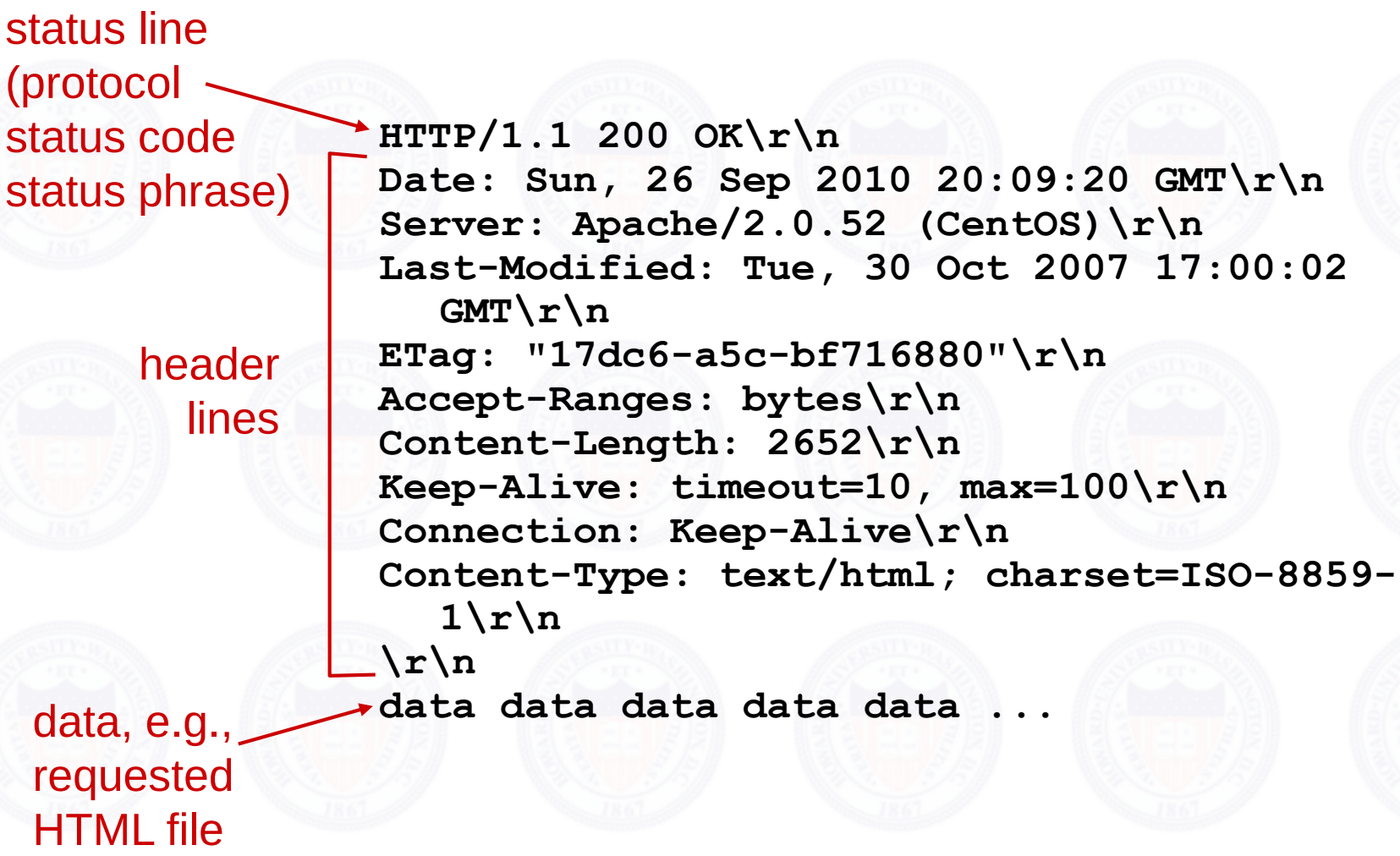
HTTP/1.0

- GET
- POST
- HEAD
  - asks server to leave requested object out of response, i.e. no message body

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
- TRACE
  - Echoes back the received request, so that a client can see what intermediate servers are adding or changing in the request.
- OPTIONS
  - Returns the HTTP methods that the server supports. This can be used to check the functionality of a web server.
- CONNECT
  - For use with a proxy that can change to being an SSL tunnel.

HTTP Response Message



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 message

301 Moved Permanently

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

400 Bad Request

- request message not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (Client Side) for Yourself

1. Telnet to your favorite Web server:

```
telnet www.howard.edu 80
```

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

2. Type in a GET HTTP request:

```
GET /index.html HTTP/1.1
Host: www.howard.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!

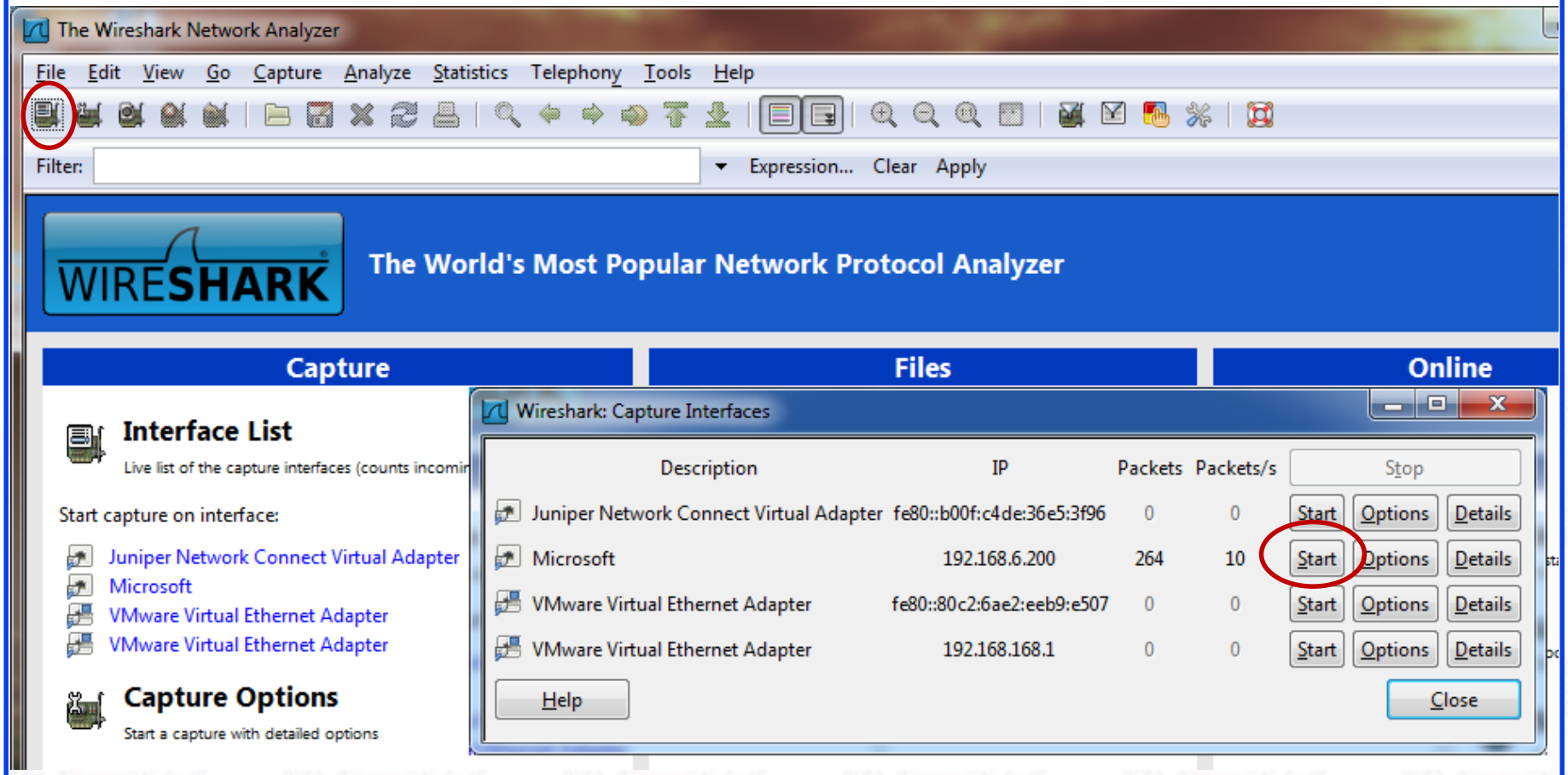
(or use Wireshark to look at captured HTTP request/response)

Wireshark

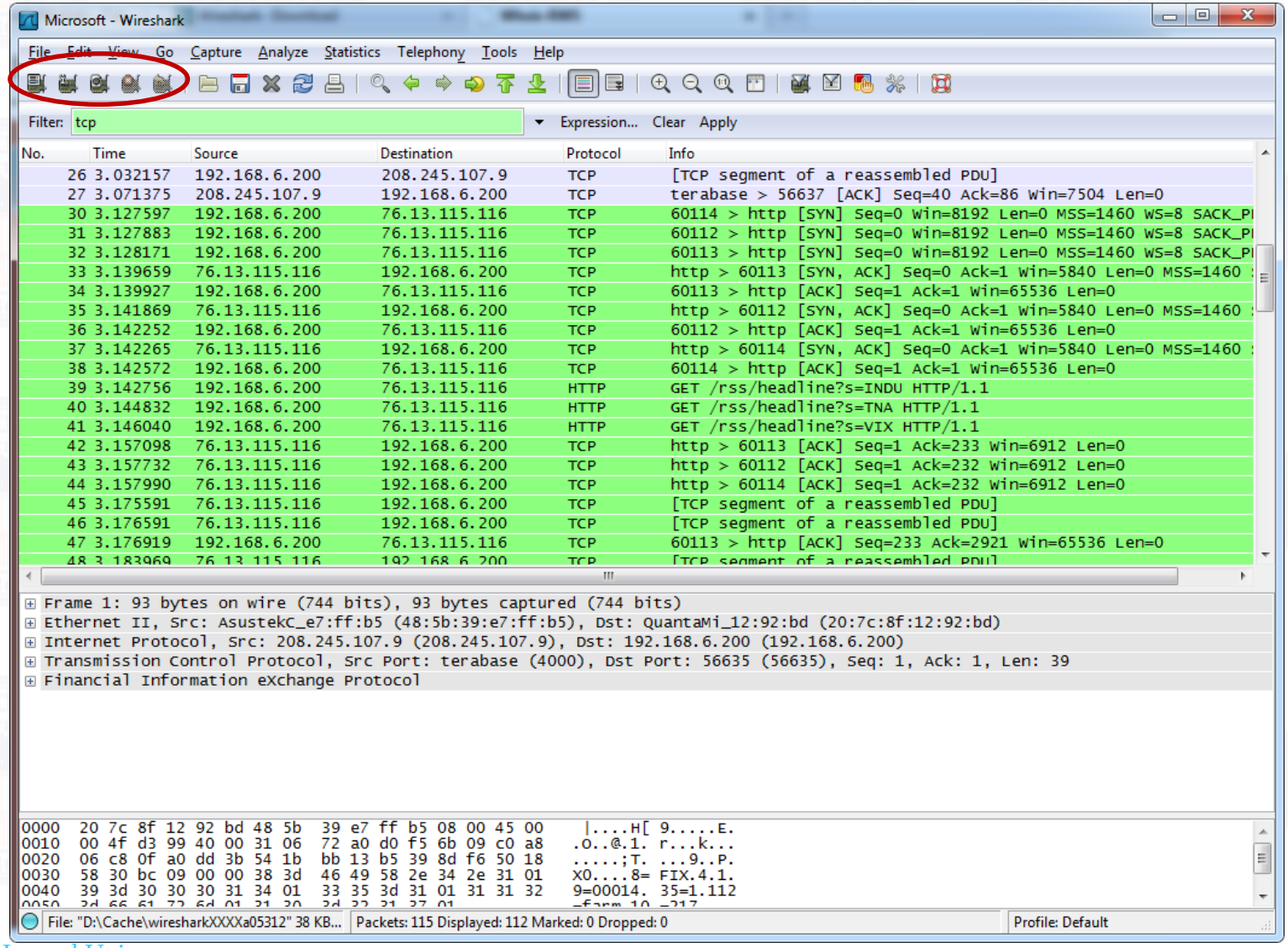
- http://www.wireshark.org/
- Formerly known as Ethereal
- Quote from the web site

  “Wireshark® is a network protocol analyzer. It lets you capture and interactively browse the traffic running on a computer network.”

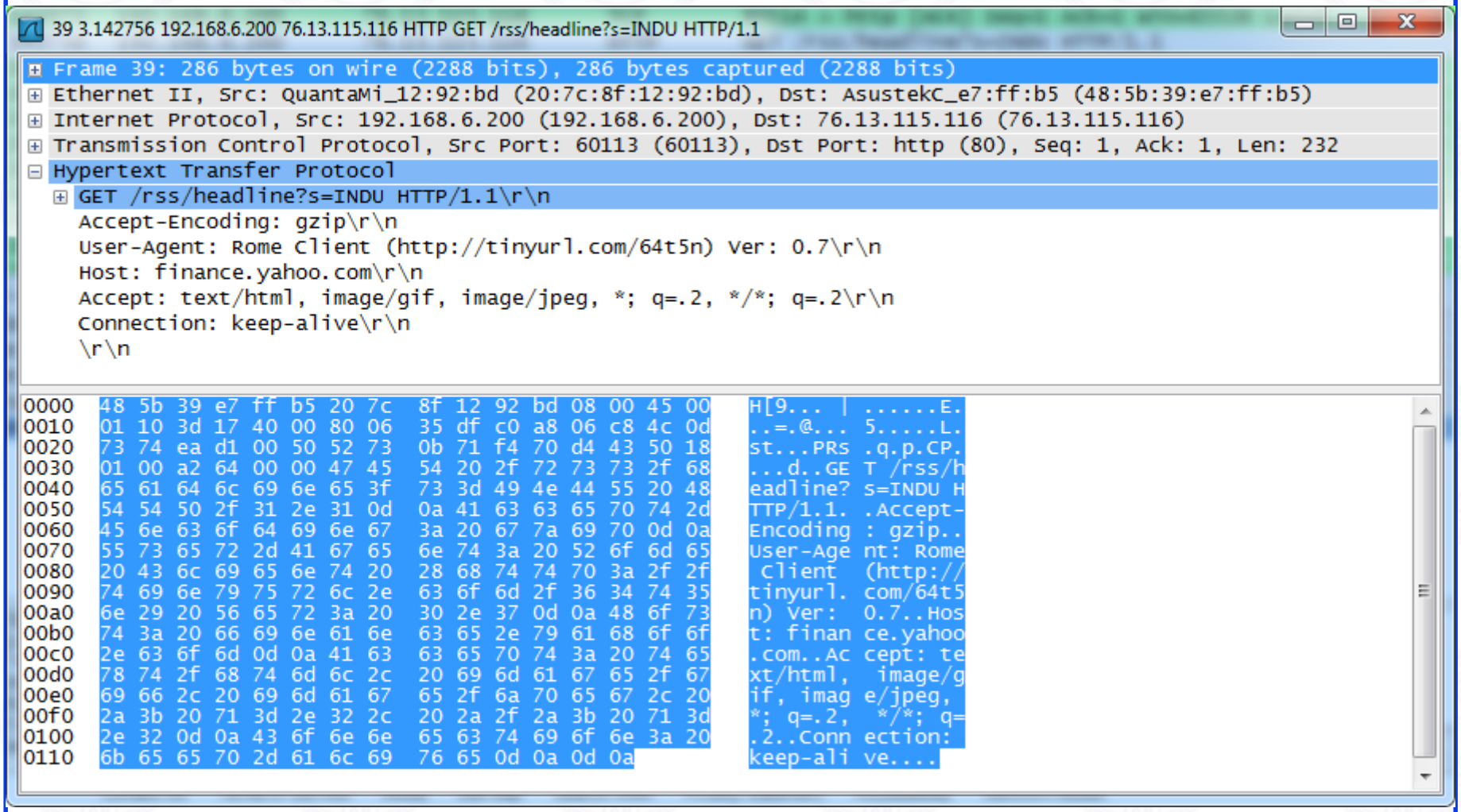
Start Capturing



View Captured Packets



Details of A Packet



User-Server State: Cookies

Many major Web sites use cookies

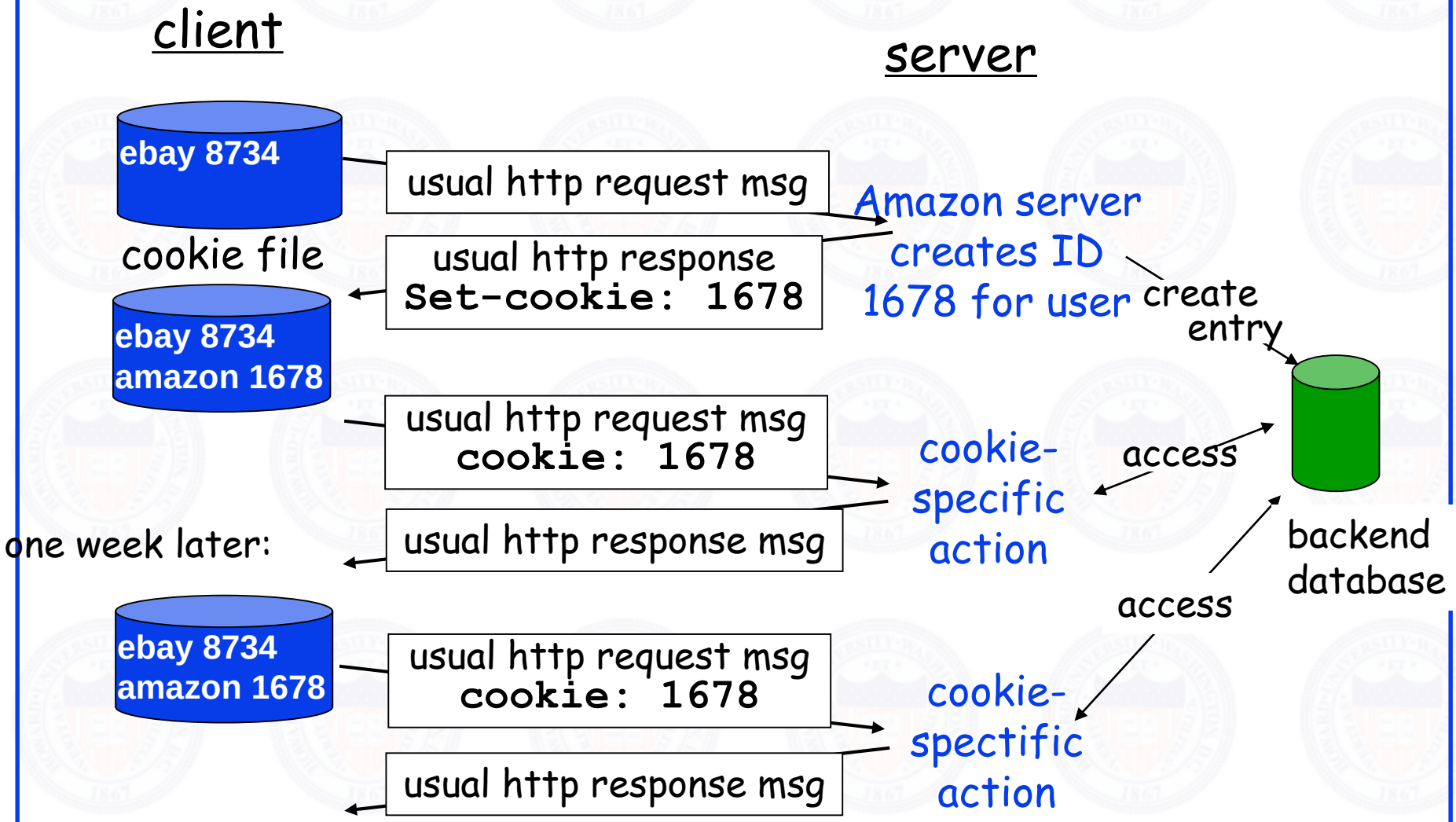
Four components:

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

Example:

- Susan 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" (cont'd)



Cookies (cont'd)

How to keep "state":

- Protocol endpoints: maintain state at sender/receiver over multiple transactions
- Cookies: http messages carry state

What cookies can be used for:

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

aside

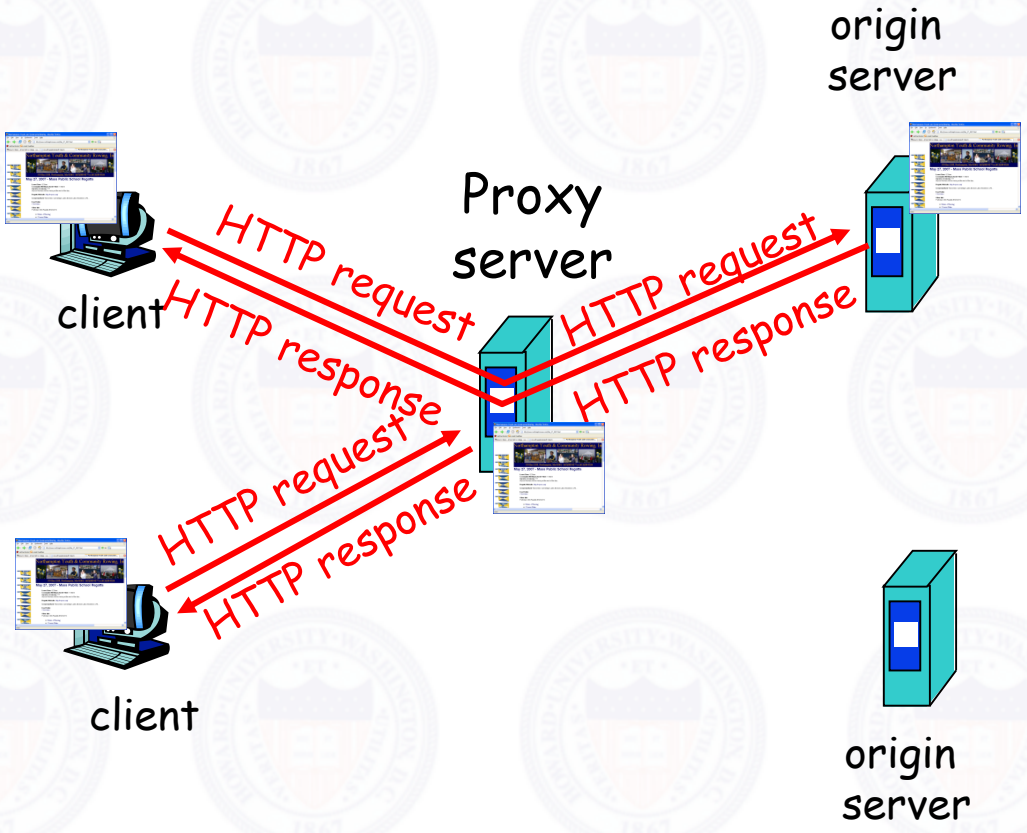
Cookies and privacy:

- Cookies permit sites to learn a lot about you
- You may supply name and e-mail to sites

Web caches (proxy server)

Goal: satisfy client request without involving origin server

- User sets browser: Web accesses via cache
- Browser sends all HTTP requests to cache
  - Object in cache: cache returns object
  - Else cache requests object from origin server, then returns object to client



More about Web caching

- Cache acts as both client and server
  - Server for original requesting client
  - Client to origin 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 (so does P2P file sharing)

Caching example:

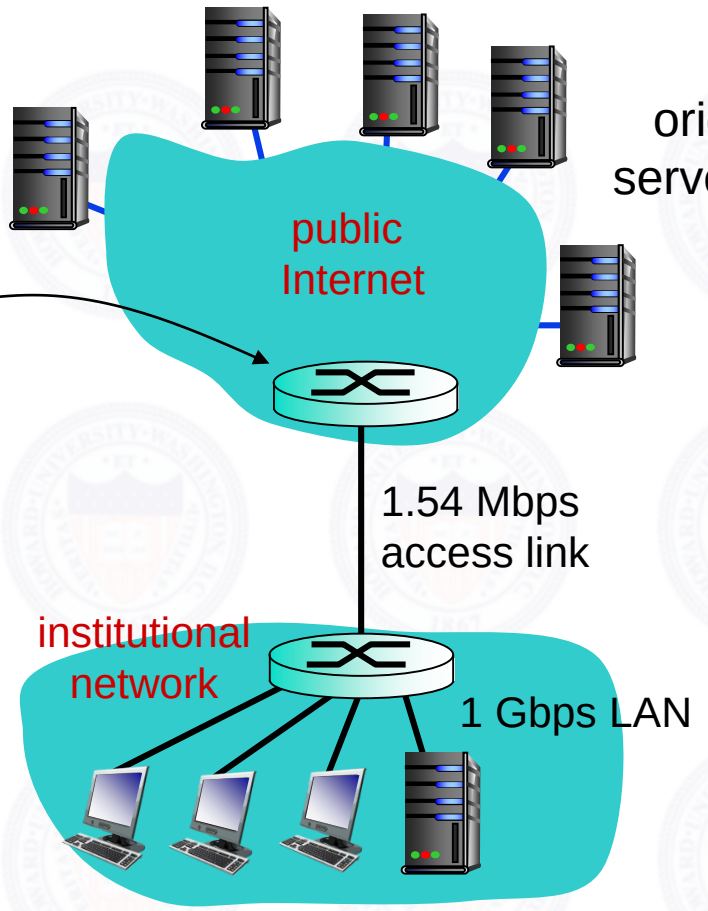
Assumptions:

- Avg object size: 100K bits
- Avg request rate from browsers to origin servers:15/sec
- Avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- Access link rate: 1.54 Mbps

Consequences:

- LAN utilization: 0.15%
- Access link utilization = 99%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + usecs

problem!



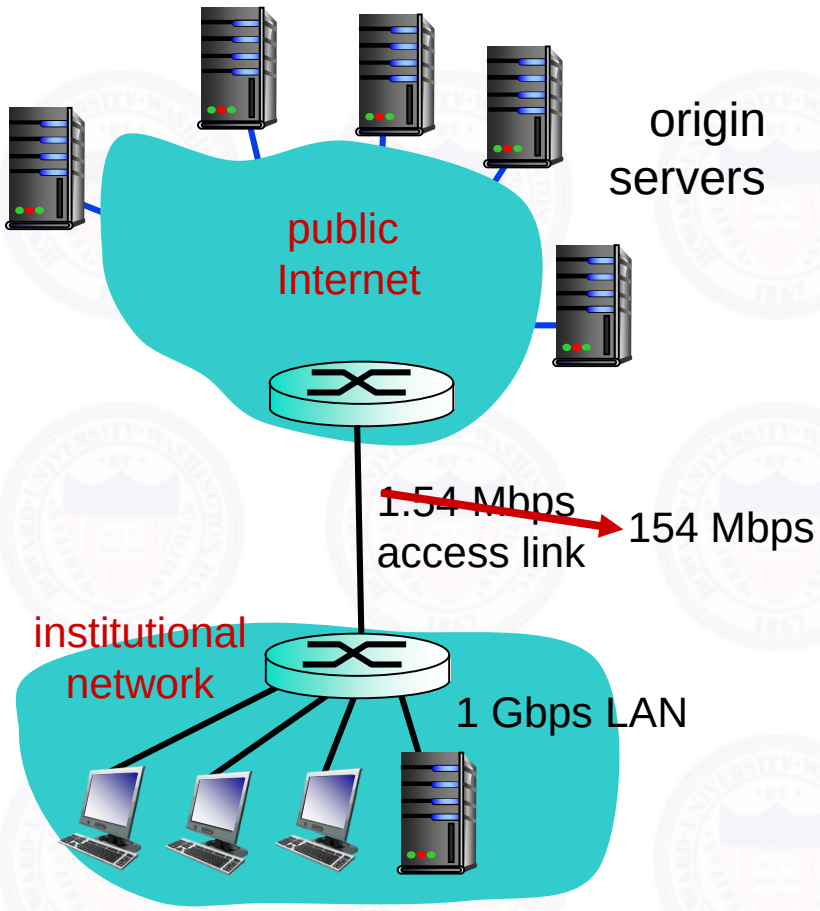
Caching example: fatter access link

Assumptions:

- Avg object size: 100K bits
- Avg request rate from browsers to origin servers:15/sec
- Avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- Access link rate: ~~1.54 Mbps~~ 154 Mbps

Consequences:

- LAN utilization: 0.15%
- Access link utilization = ~~99%~~ 9.9%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + ~~minutes~~ msecs + usecs



Cost: increased access link speed (not cheap!)

Caching example: install local cache

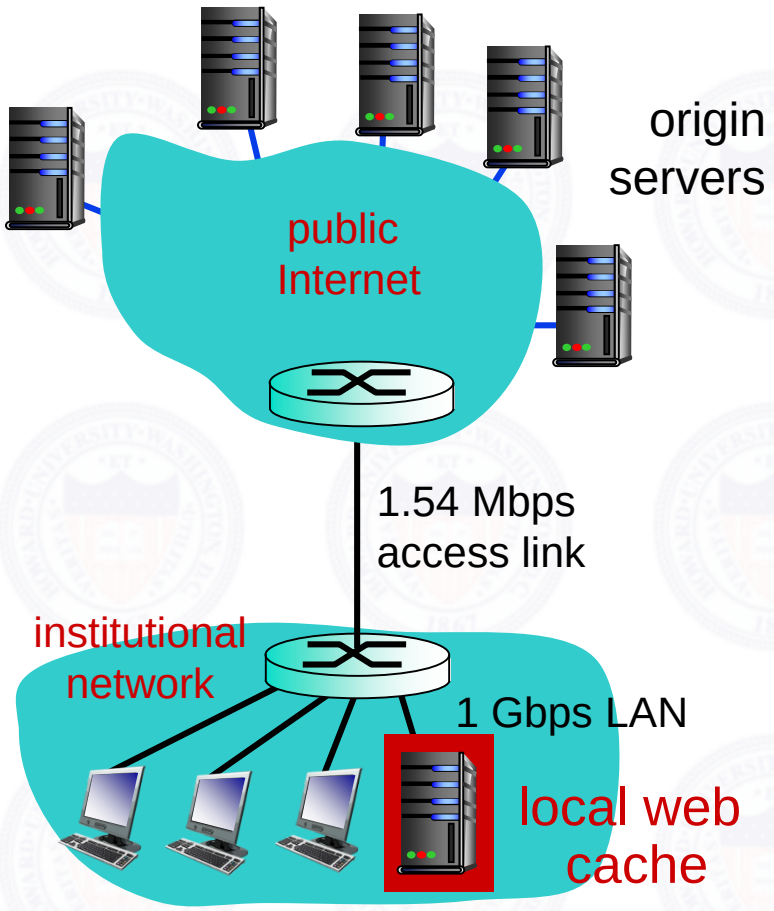
Assumptions:

- Avg object size: 100K bits
- Avg request rate from browsers to origin servers:15/sec
- Avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- Access link rate: 1.54 Mbps

Consequences:

- LAN utilization: 0.15%
- Access link utilization = ?
- Total delay = ?

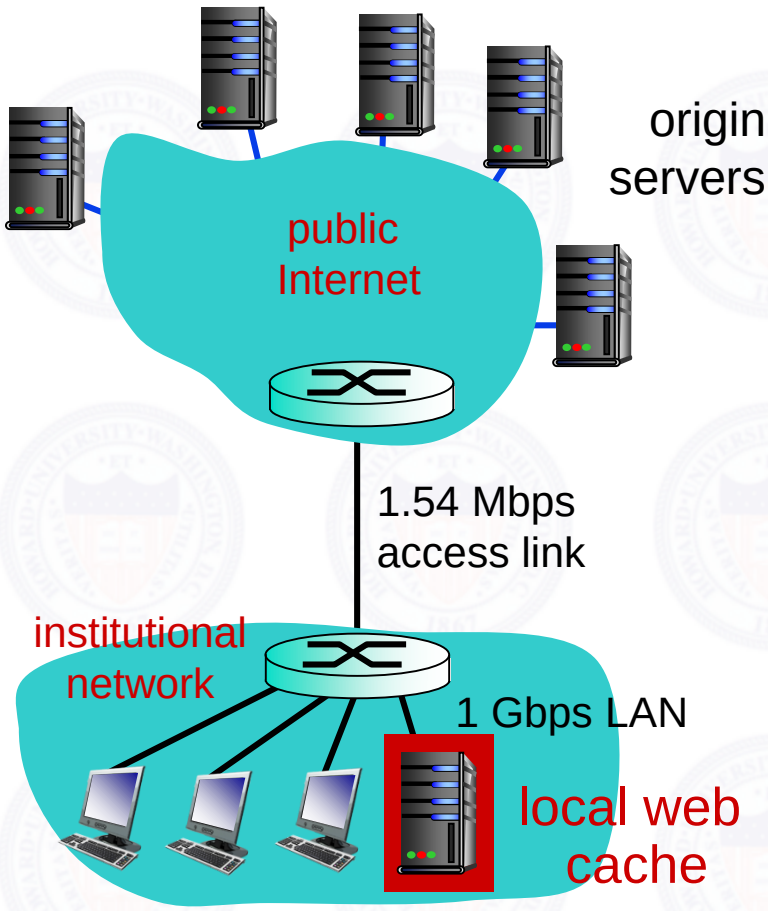
How to compute link utilization, delay?



Caching example: install local cache

Calculating access link utilization, delay with cache:

- Suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin
- Access link utilization:
  - 60% of requests use access link
  - data rate to browsers over access link = 0.6*1.50 Mbps = .9 Mbps
  - utilization = 0.9/1.54 = .58
  - → less access delay (say 10 ms)
- Total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~msecs)
  - = ~ 1.2 secs
  - less than with 154 Mbps link (and cheaper too!)

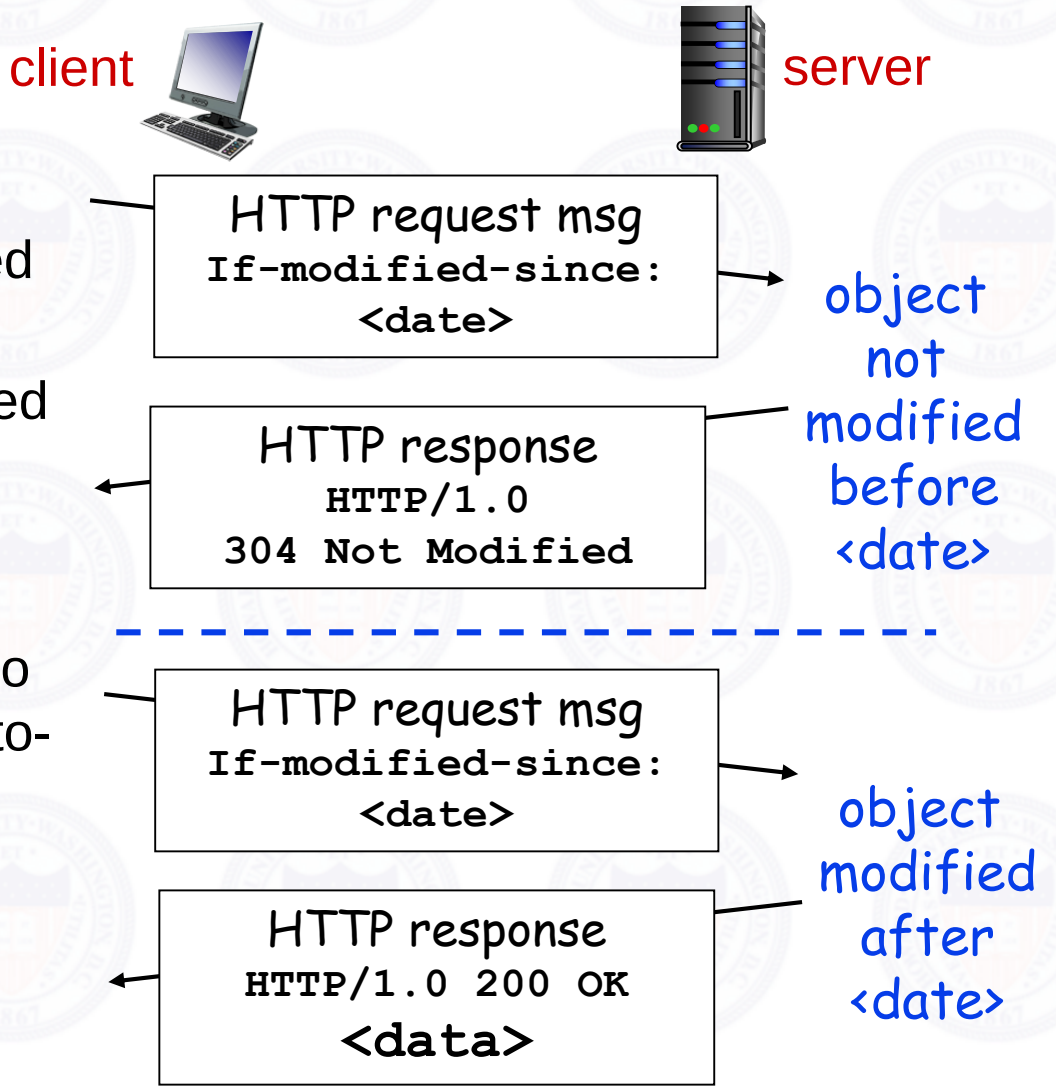


Cache Update

- Conditional GET
- Goal: don't send object if cache has up-to-date cached version
- cache: specify date of cached copy in HTTP request

  If-modified-since: <date>
- server: response contains no object if cached copy is up-to-date:

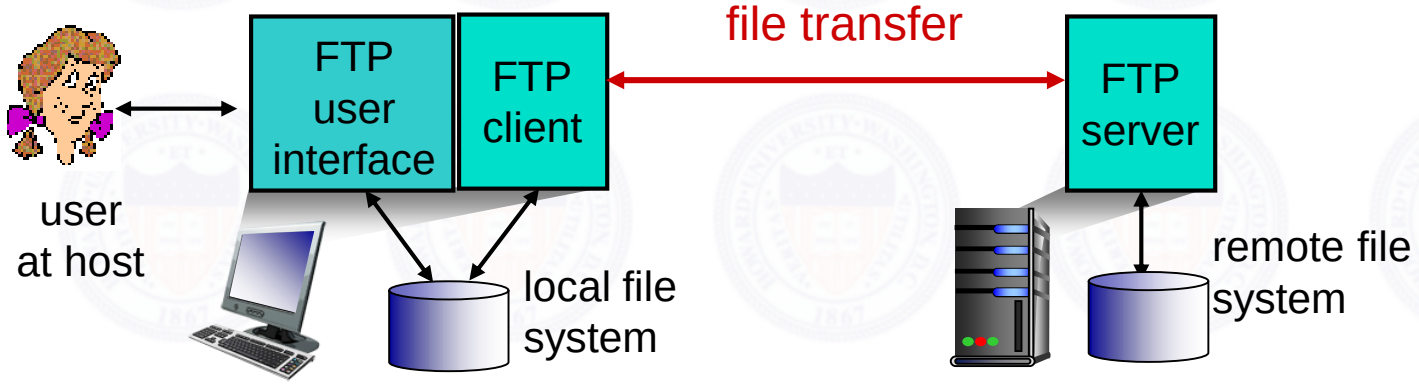
  HTTP/1.0 304 Not Modified



Chapter 2: Application layer

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

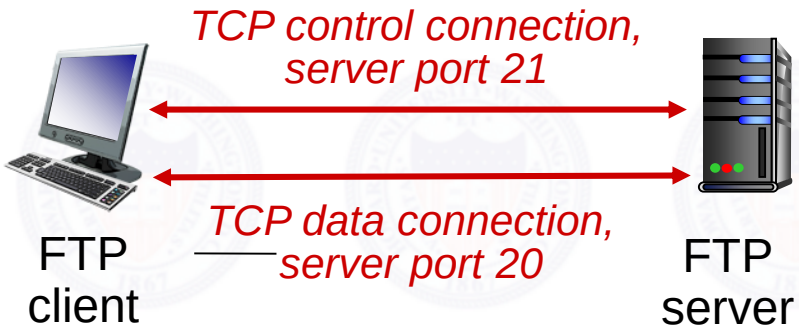
FTP: the file transfer protocol



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

FTP: separate control, data connections

- FTP client contacts FTP server at port 21, using TCP
- Client authorized over control connection
- Client browses remote directory, sends commands over control connection
- When server receives file transfer command, *server* opens *2nd* 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:

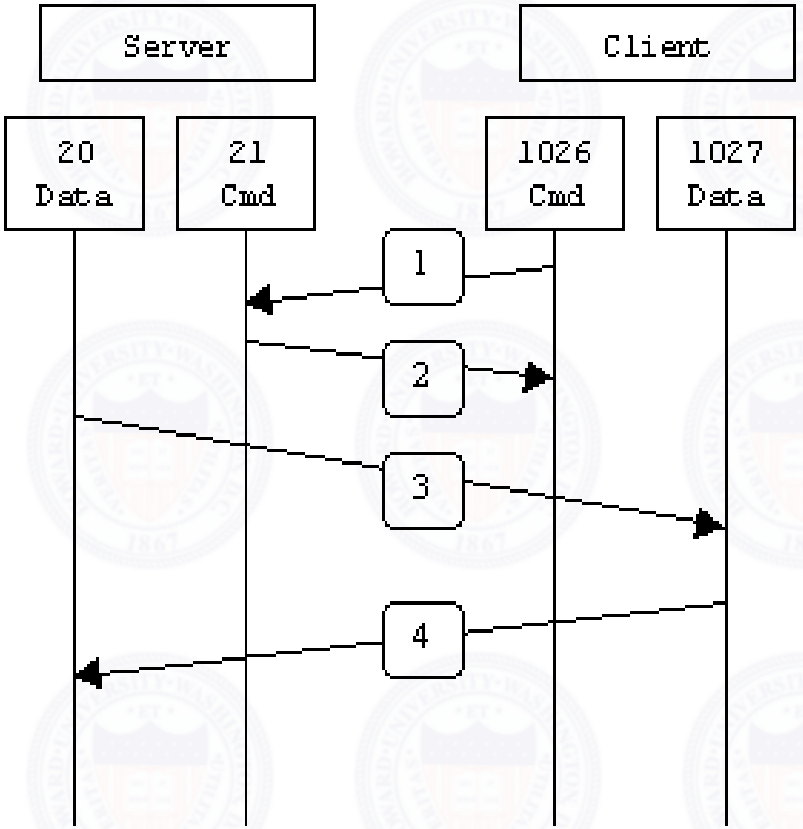
- sent as ASCII text over control channel
- **USER *username***
- **PASS *password***
- **LIST** return list of file in current directory
- **RETR filename** retrieves (gets) file
- **STOR filename** stores (puts) file onto remote host

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**

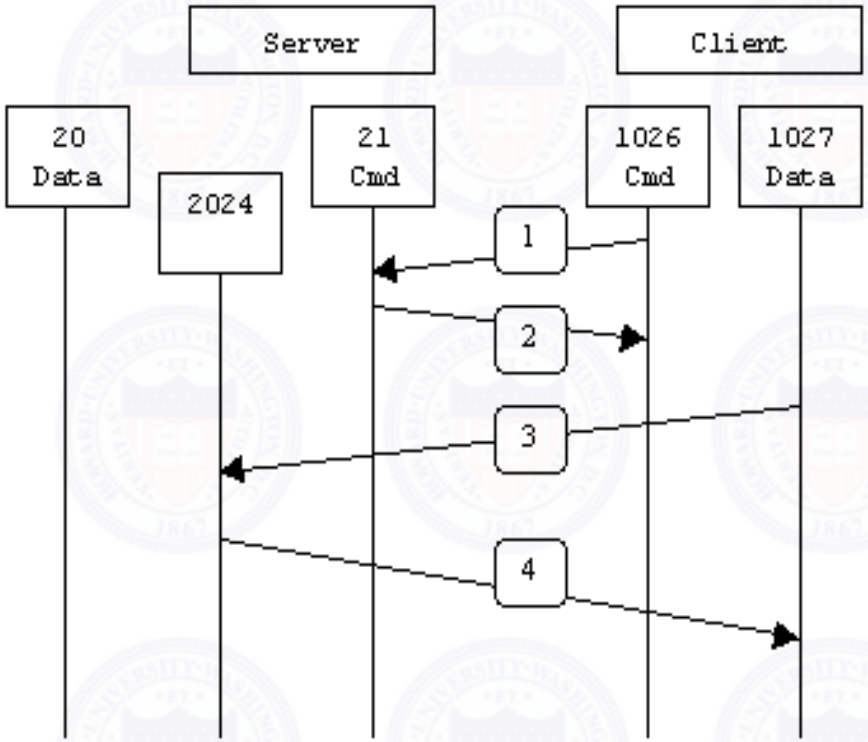
FTP: active mode

- Step 1: the client's cmd port contacts the server's cmd port and sends the cmd port 1026.
- Step 2: the server sends an ACK back to the client's cmd port.
- Step 3: the server initiates a connection on its local data port to the data port the client specified earlier (cmd port # + 1).
- Step 4: the client sends an ACK back.



FTP: passive mode

- Step 1: the client contacts the server on the cmd port and issues the PASV cmd.
- Step 2: the server replies with PORT 2024, telling the client which port it is listening to for the data connection.
- Step 3: the client initiates the data connection from its data port to the specified server data port.
- Step 4: the server sends back an ACK to the client's data port.



Chapter 2: Application layer

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

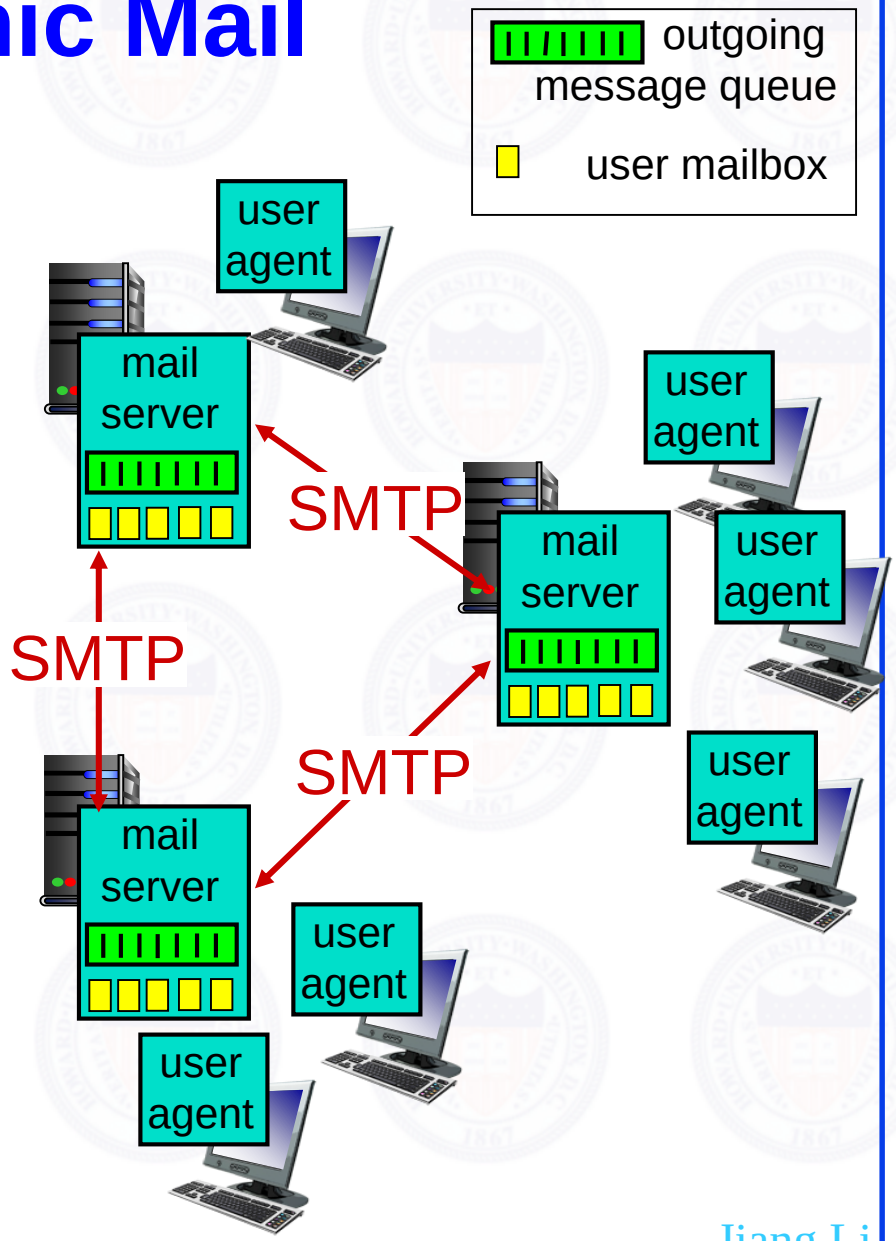
Electronic Mail

Three major components:

- User agents
- Mail servers
- Protocols: Simple mail transfer protocol (SMTP), mail access protocols

User Agent

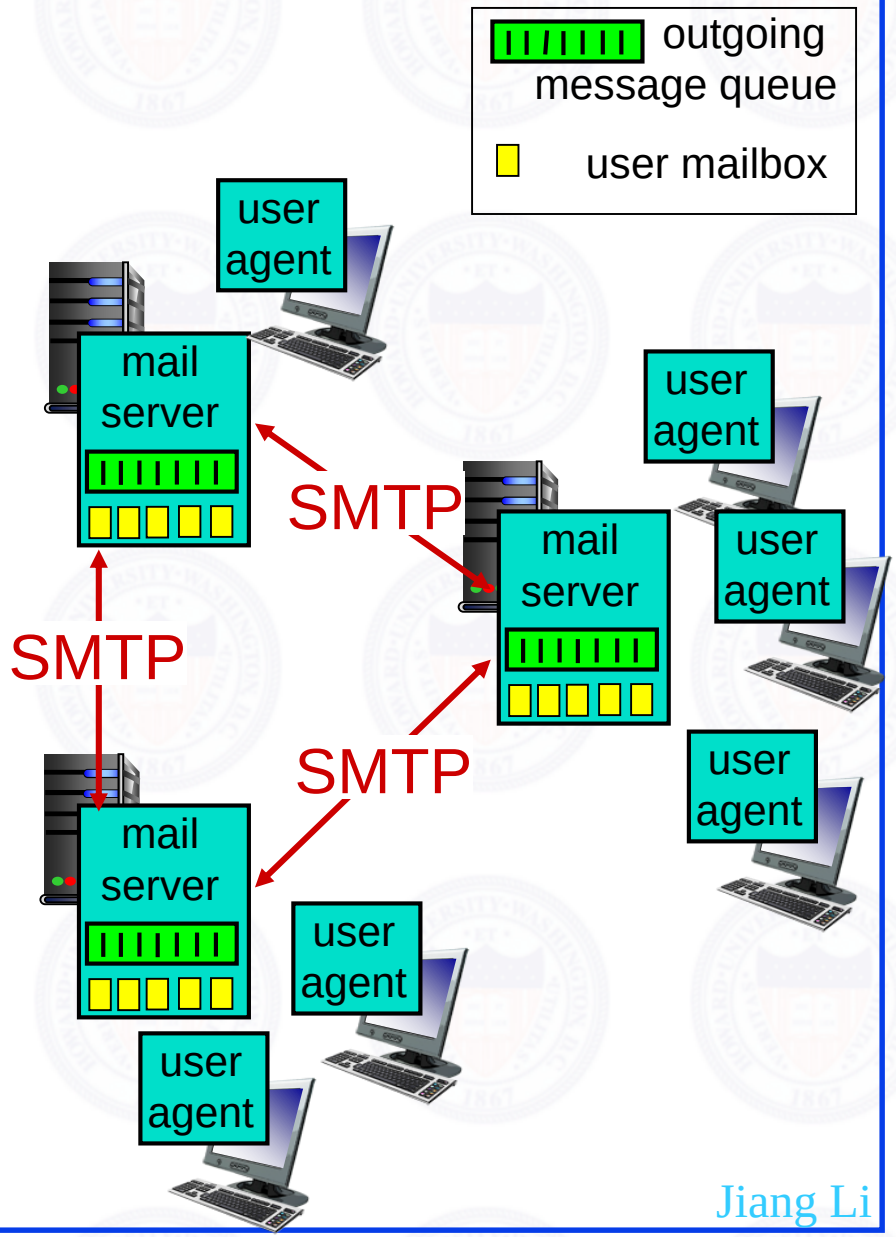
- A.k.a. "mail reader"
- Composing, editing, reading mail messages
- E.g., Eudora, Outlook, elm, Mozilla Thunderbird
- Outgoing, incoming messages stored on server



Electronic Mail: mail servers

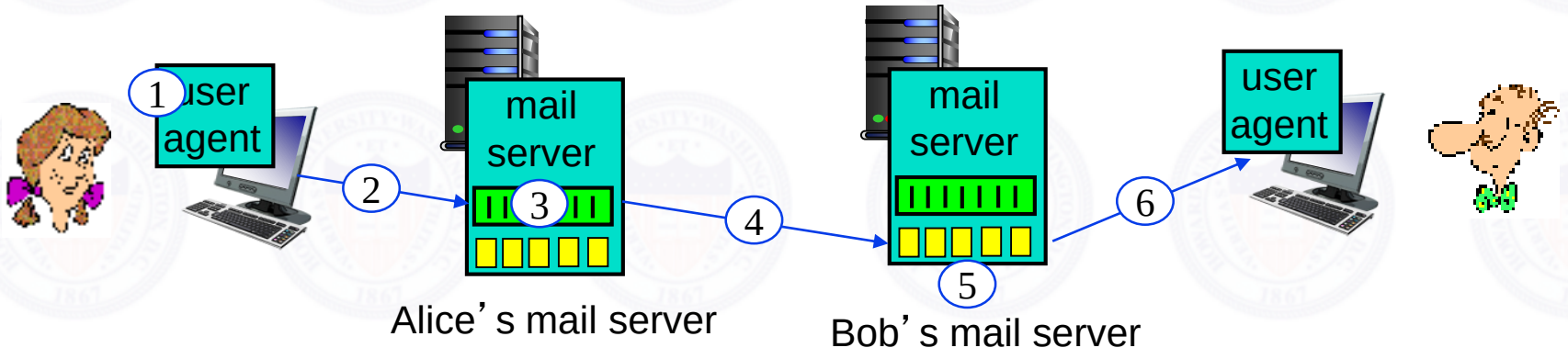
Mail Servers

- Mailbox contains incoming messages for user
- Message queue of outgoing (to be sent) mail messages
- SMTP protocol between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server



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



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

Sample SMTP interaction

```
S: 220 mx.google.com
C: HELO yahoo.com
S: 250  Hello yahoo.com, pleased to meet you
C: MAIL FROM: <alice@yahoo.com>
S: 250 alice@yahoo.com... Sender ok
C: RCPT TO: <bob@gmail.com>
S: 250 bob@gmail.com ... 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 mx.google.com closing connection
```

Try SMTP interaction for yourself

- **`telnet servername 25`**
  - e.g. **`telnet aspmx.l.google.com 25`**
- See 220 reply from server
- Enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

- Above lets you send email without using email client (reader)

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

Comparison with HTTP:

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

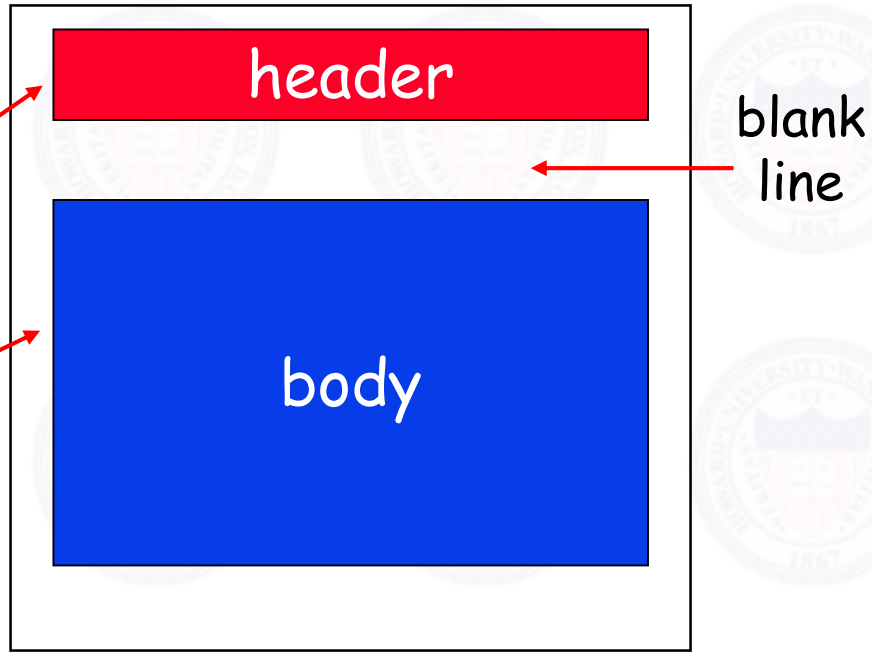
Mail Message Format

SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format:

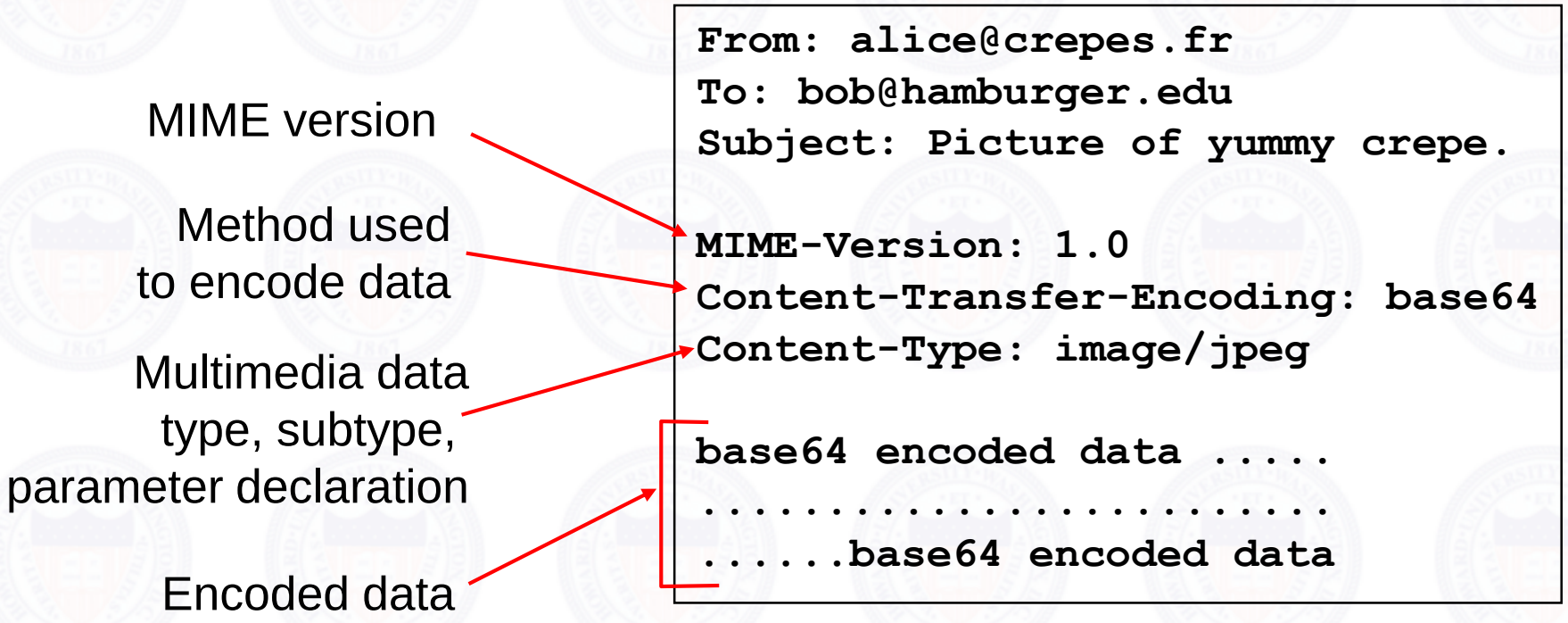
- Header lines (for email clients), e.g.,
  - To:
  - From:
  - Subject:

  different from SMTP MAIL FROM, RCPT TO: *commands*!
- Body
  - the "message", ASCII characters only



Message Format: Multimedia Extensions

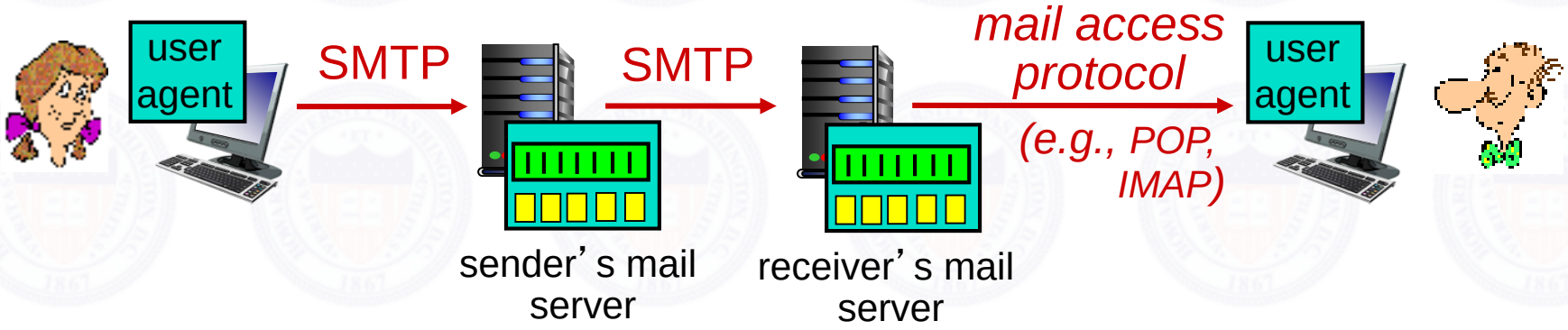
- MIME: multimedia mail extension, RFC 2045, 2056
- Additional lines in msg header declare MIME content type



Base64 Encoding

- Arbitrary sequence of bytes => A-Z,a-z,0-9,+,/
  - Also use = as a special suffix code
- Encoding
  - Take three bytes, i.e. 24 bits
    - Pad with 0 at the end if fewer than 3 bytes
  - Take 6 bits each time, map them to “A-Za-z0-9+/”
  - If there are only one or two input bytes, only the first two or three characters of the output are used and are padded with two or one "=" characters respectively.
- Example
  - Man (0x4D616E)
  - Binary: 0100 1101 0110 0001 0110 1110
  - 19 => T, 22 => W, 5 => F, 46 => u
  - Man => TWFu
  - Ma => TWE=

Mail Access Protocols



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

POP3 Protocol

Authorization phase

- Client commands:
  - **user:** declare username
  - **pass:** password
- Server responses
  - **+OK**
  - **-ERR**

Transaction phase, client:

- **list:** list message numbers
- **retr:** retrieve message by number
- **dele:** delete
- **quit**

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

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

POP3 (more) and IMAP

More about POP3

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

IMAP

- Keep all messages in one place: the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - Names of folders and mappings between message IDs and folder name

IMAP Session Example

telnet: > telnet imap.example.com imap
telnet: Trying 192.0.2.2...
telnet: Connected to imap.example.com.
telnet: Escape character is '^]'.
server: * OK Dovecot ready.
client: a1 LOGIN MyUsername MyPassword
server: a1 OK Logged in.
client: a2 LIST "" "*"
server: * LIST (\HasNoChildren) "." "INBOX"
server: a2 OK List completed.
client: a3 EXAMINE INBOX
server: * FLAGS (\Answered \Flagged \Deleted \Seen \Draft)
server: * OK [PERMANENTFLAGS ()] Read-only mailbox.
server: * 1 EXISTS
server: * 1 RECENT
server: * OK [UNSEEN 1] First unseen.
server: * OK [UIDVALIDITY 1257842737] UIDs valid
server: * OK [UIDNEXT 2] Predicted next UID
server: a3 OK [READ-ONLY] Select completed.
client: a4 FETCH 1 BODY[]
server: * 1 FETCH (BODY[] {405}
server: Return-Path: sender@example.com
server: Received: from client.example.com ([192.0.2.1])
server: by mx1.example.com with ESMTP
server: id <20040120203404.CCCC18555.mx1.example.com@client.example.com>
server: for <recipient@example.com>; Tue, 20 Jan 2004 22:34:24 +0200
server: From: sender@example.com
server: Subject: Test message
server: To: recipient@example.com
server: Message-Id: <20040120203404.CCCC18555.mx1.example.com@client.example.com>
server:
server: This is a test message.
server:)
server: a4 OK Fetch completed.
client: a5 LOGOUT
server: * BYE Logging out
server: a5 OK Logout completed.

http://www.anta.net/misc/telnet-troubleshooting/imap.shtml

Chapter 2: Application layer

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

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: How to map between IP addresses and name ?

Domain Name System:

- *Distributed database* implemented in hierarchy of many *name servers*
- *Application-layer protocol* host, routers, name servers to communicate to *resolve* names (address/name translation)
  - Note: core Internet function, implemented as application-layer protocol
  - Complexity at network's "edge"

DNS : Services, Structure

DNS services

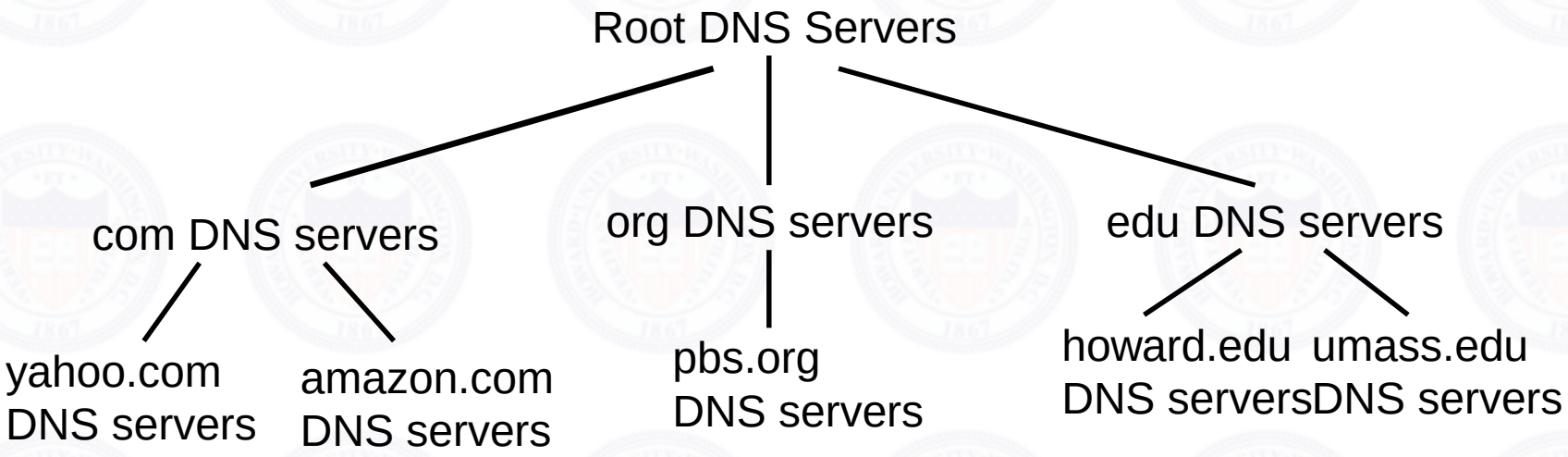
- Hostname to IP address translation
- Host aliasing
  - Canonical and alias names
- Mail server aliasing
- Load distribution
  - Replicated Web servers: multiple IP addresses correspond to one name

Why not centralize DNS?

- Single point of failure
- Traffic volume
- Distant centralized database
- Maintenance

It doesn't *scale!*

Distributed, Hierarchical Database

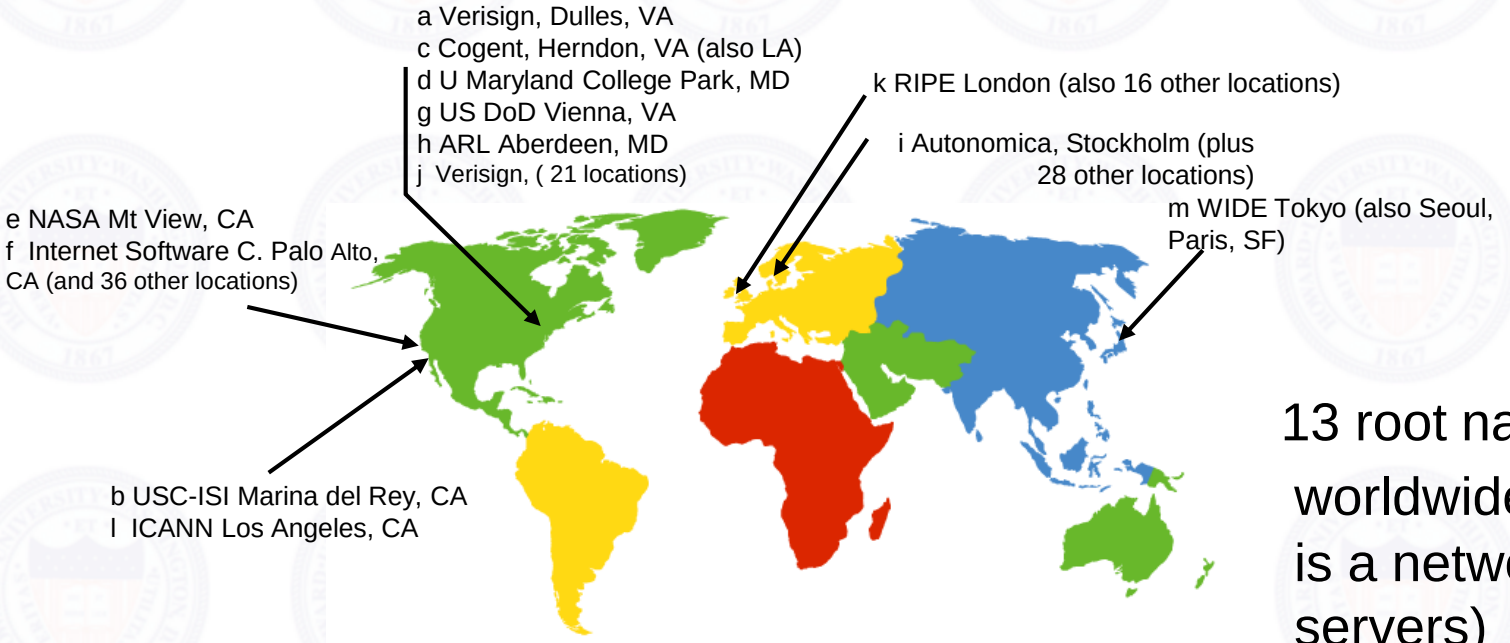


Client wants IP for www.amazon.com; 1st approx:

- Client queries a root server to find com DNS server
- Client queries com DNS server to get amazon.com DNS server
- Client queries amazon.com DNS server to get IP address for www.amazon.com

DNS: Root name servers

- Contacted by local name server that can not resolve name
- Tells the local name server about the TLD servers



13 root nameservers worldwide (each one is a network of servers)

TLD and Authoritative Servers

- Top-level domain (TLD) servers: responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
  - Verisign 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 Name Server

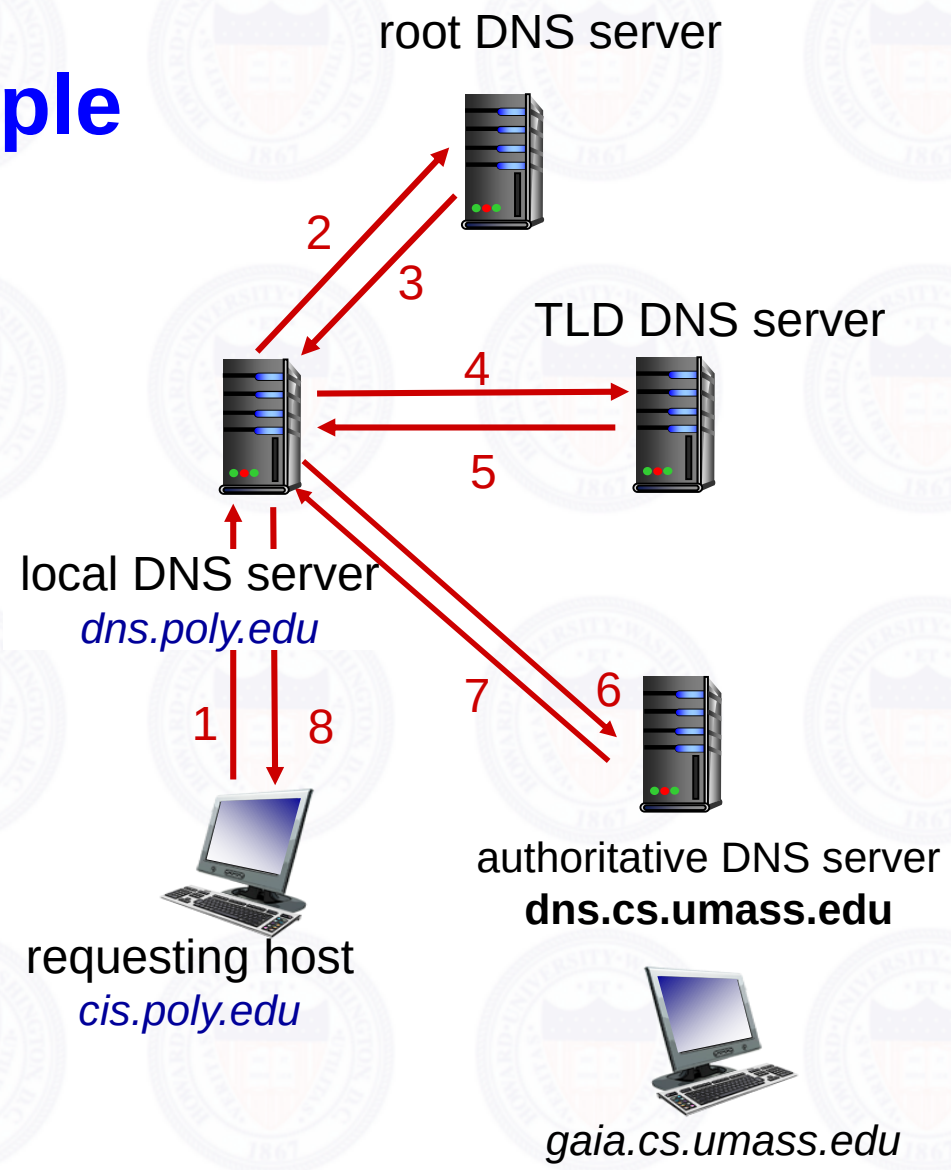
- Does not strictly belong to hierarchy
- Each ISP (residential ISP, company, university) has one.
  - Also called “default name server”
- When a host makes a DNS query, query is sent to its local DNS server
  - Acts as a proxy, forwards query into hierarchy.

DNS Name Resolution Example

- Host at cis.poly.edu wants IP address for gaia.cs.umass.edu

<u>Iterated query:</u>

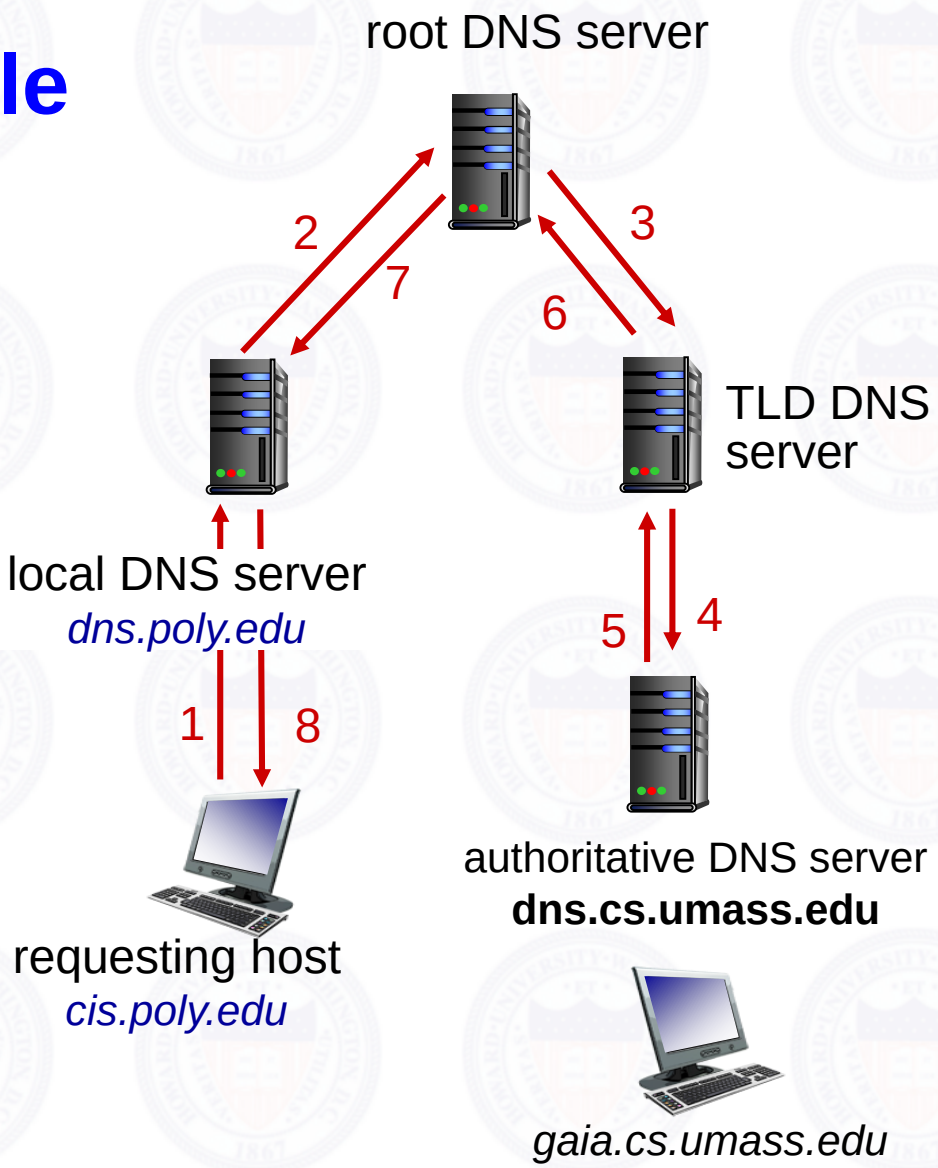
- Contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS Name Resolution Example

Recursive query:

- Puts burden of name resolution on contacted name server
- Heavy load at upper levels of hierarchy?



DNS: Caching and Updating Records

- Once (any) name server learns mapping, it *caches* mapping
  - Cache entries timeout (disappear) after some time
    - Individual configuration
  - TLD servers typically cached in local name servers
    - Thus 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
  - http://www.ietf.org/html.charters/dnsind-charter.html

DNS Records

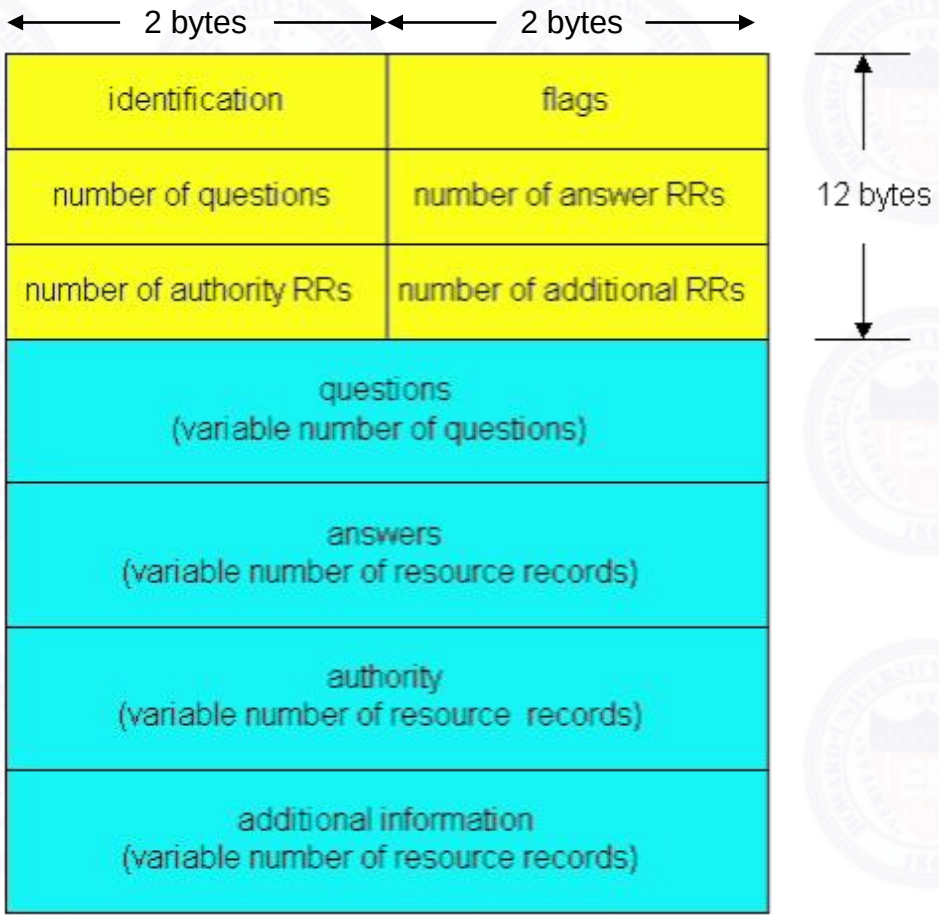
DNS: distributed db storing resource records (RR)

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

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



Message header

- Identification: 16 bit # for query, reply to query uses same #
- Flags:
  - Query or reply
  - Recursion desired
  - Recursion available
  - Reply is authoritative

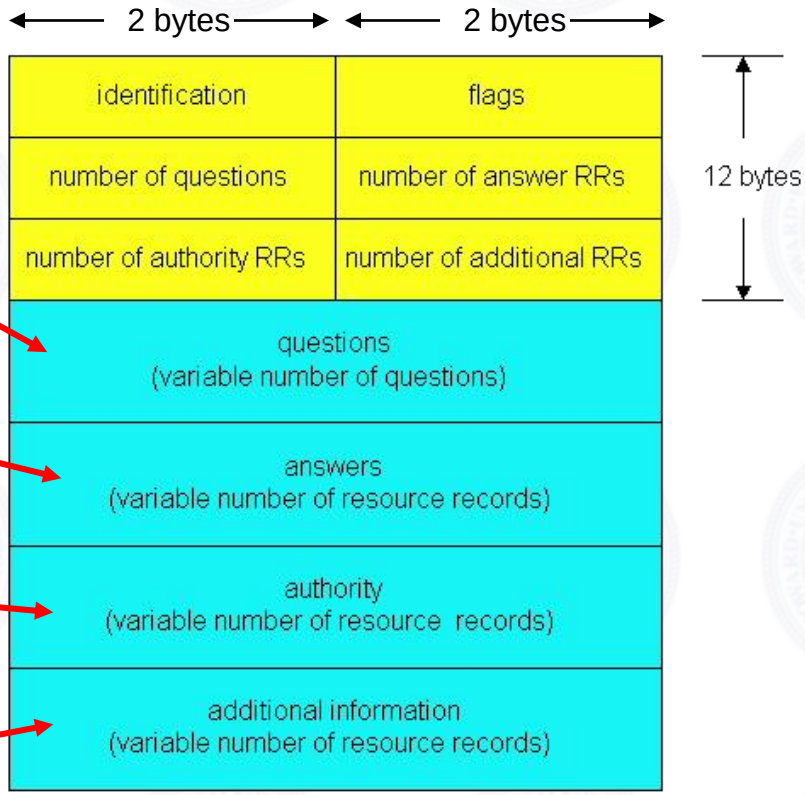
DNS Protocol, Messages

Name, type fields for a query

RRs in response to query

records for authoritative servers

additional "helpful" info that may be used



Inserting Records into DNS

- Example: just created startup “Network Utopia”
- Register name networkuptopia.com at a DNS 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.networkuptopia.com and 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, allowing root server bypass
- Bombard TLD servers
  - Potentially more dangerous

Redirect attacks

- Man-in-middle
  - Intercept queries
- DNS poisoning
  - Send bogus replies to local DNS server, which caches

Exploit DNS for DDoS

- Send queries with spoofed source address: target IP
- Requires amplification

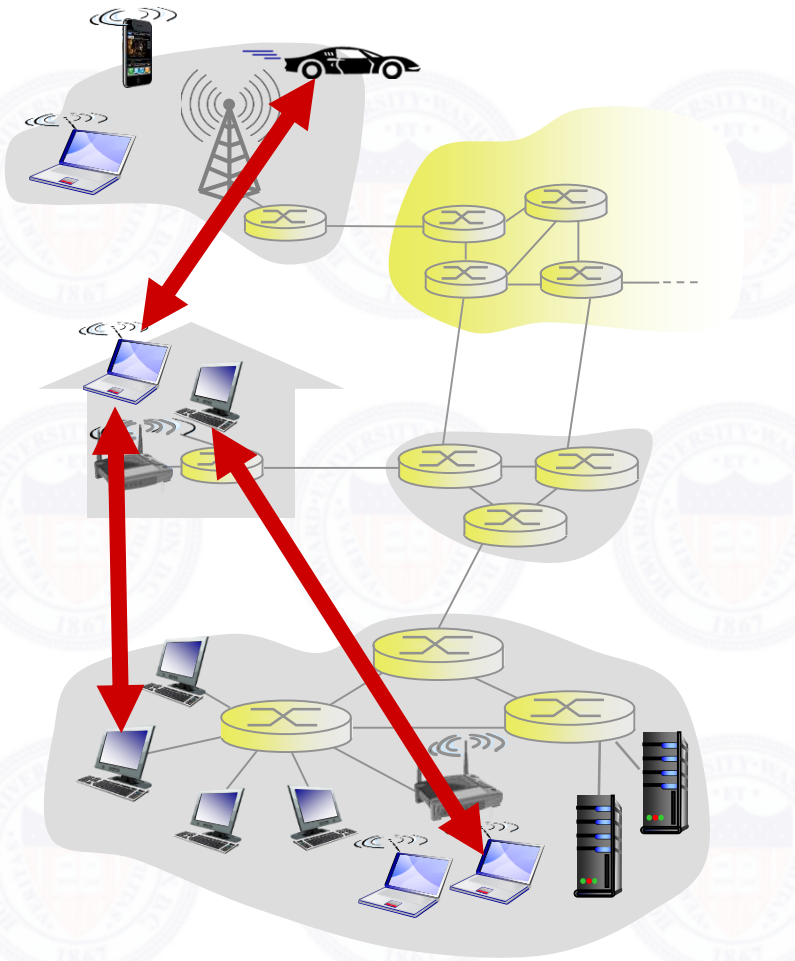
DNS has been surprisingly robust against attacks.

Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P applications
- 2.7 Socket programming with TCP & UDP

Pure P2P architecture

- *No* always-on server
- Arbitrary end systems directly communicate
- Peers are intermittently connected and change IP addresses



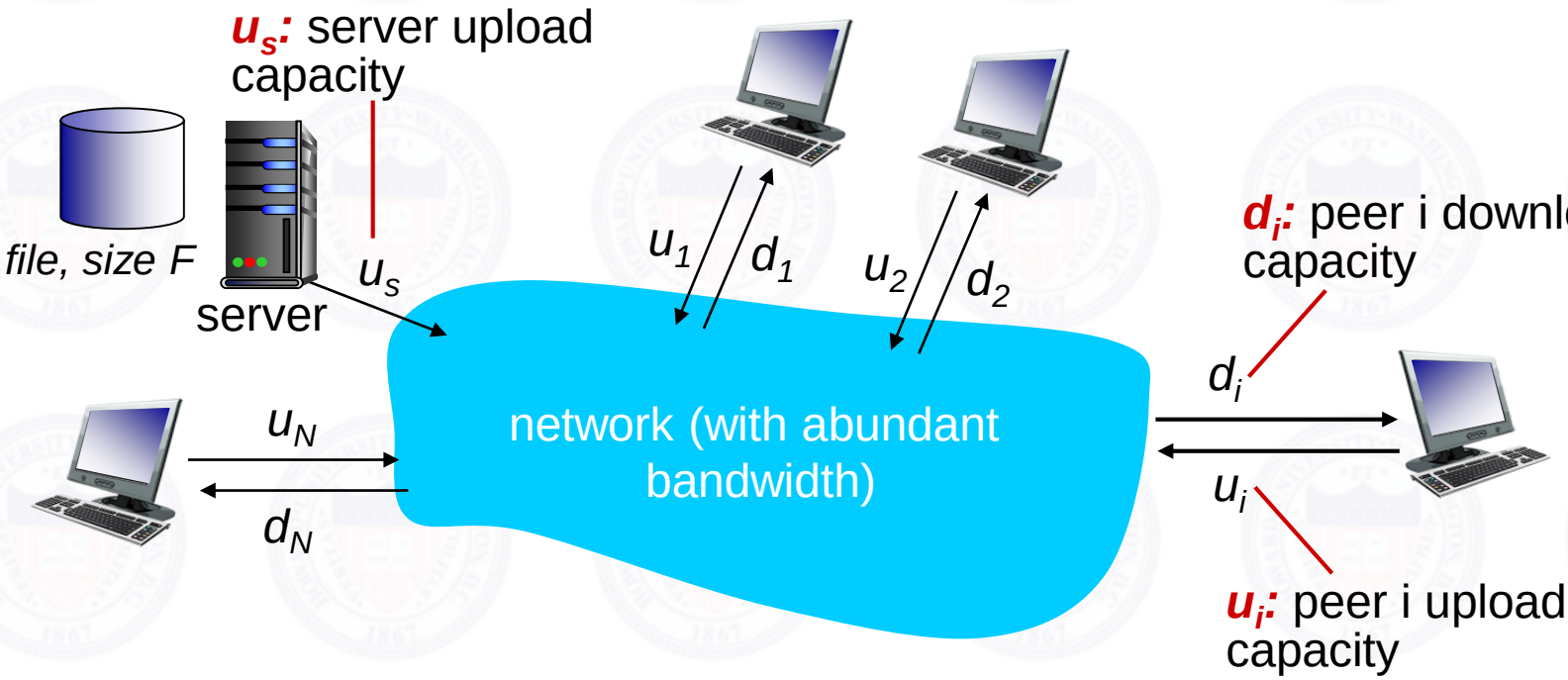
Examples:

- File distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)

File distribution: client-server vs P2P

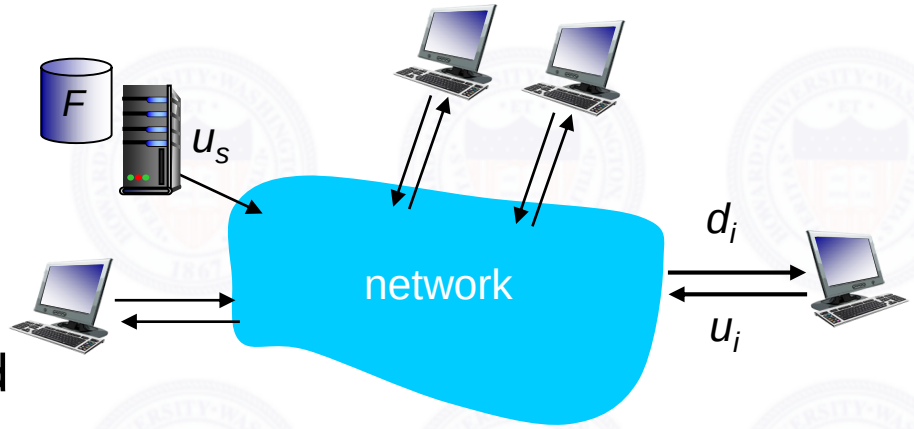
Question: How much time to distribute file (size *F*) from one server to *N peers*?

- Peer upload/download capacity is limited resource



File distribution time: client-server

- *Server transmission:* must 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 client download rate
  - min client download time: F/d_{min}



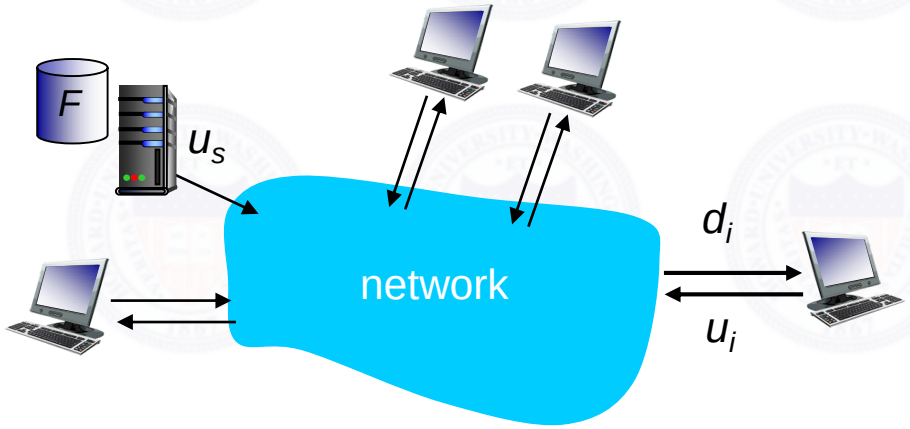
time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} \geq max\{NF/u_s, F/d_{min}\}$$

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_{min}



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

time to distribute F to N clients using P2P approach

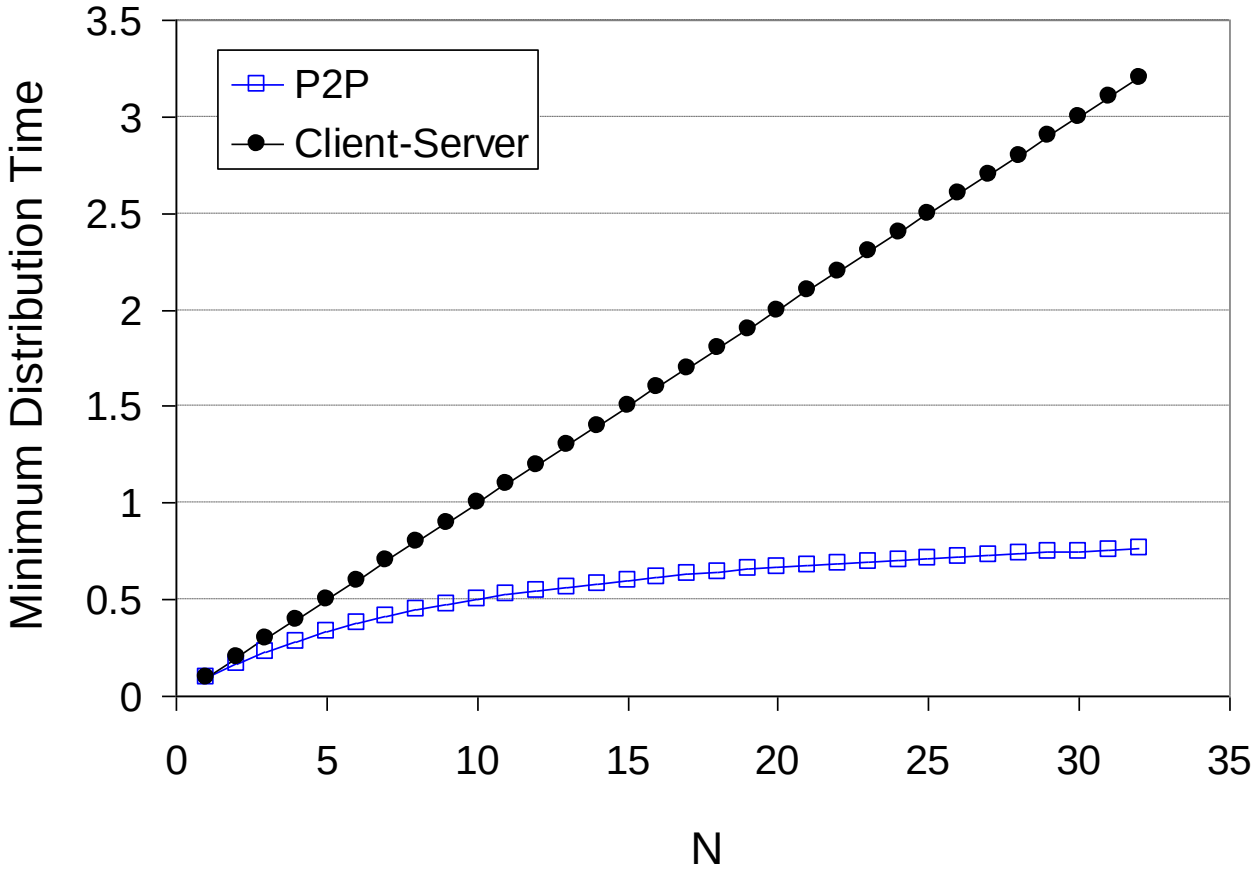
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

… but so does this, as each peer brings service capacity

Server-client vs. P2P: example

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

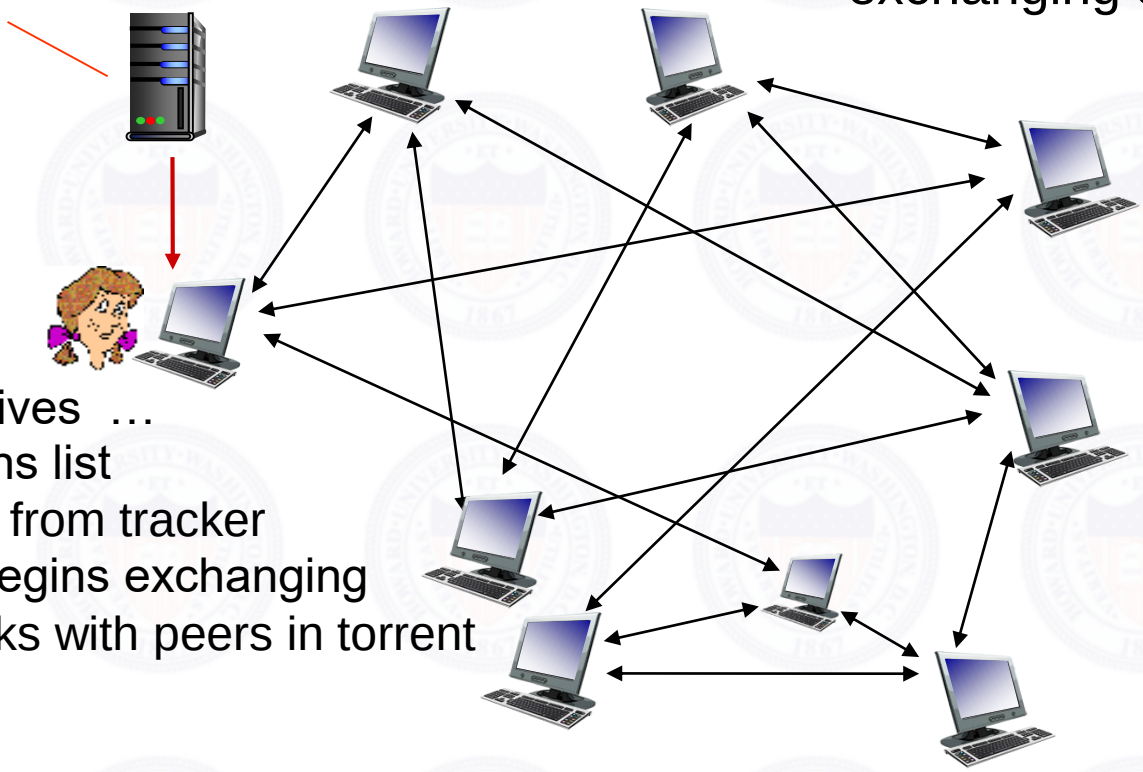


P2P file distribution: BitTorrent

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

tracker: tracks peers participating in torrent

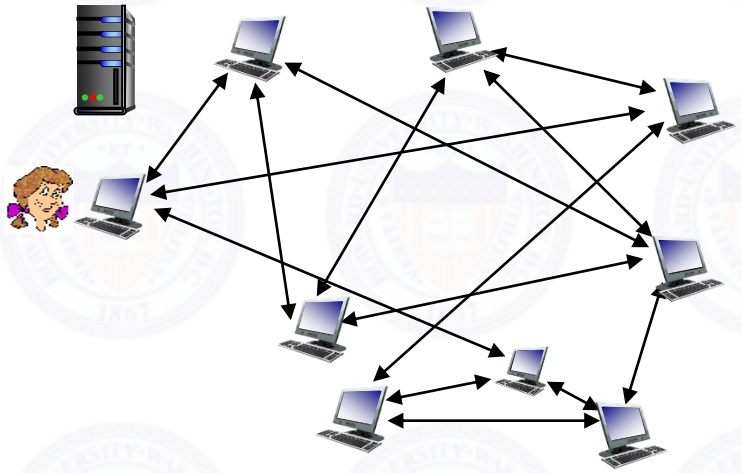
torrent: group of peers exchanging chunks of a file



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 subsets of file chunks
- Periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

Sending chunks: tit-for-tat

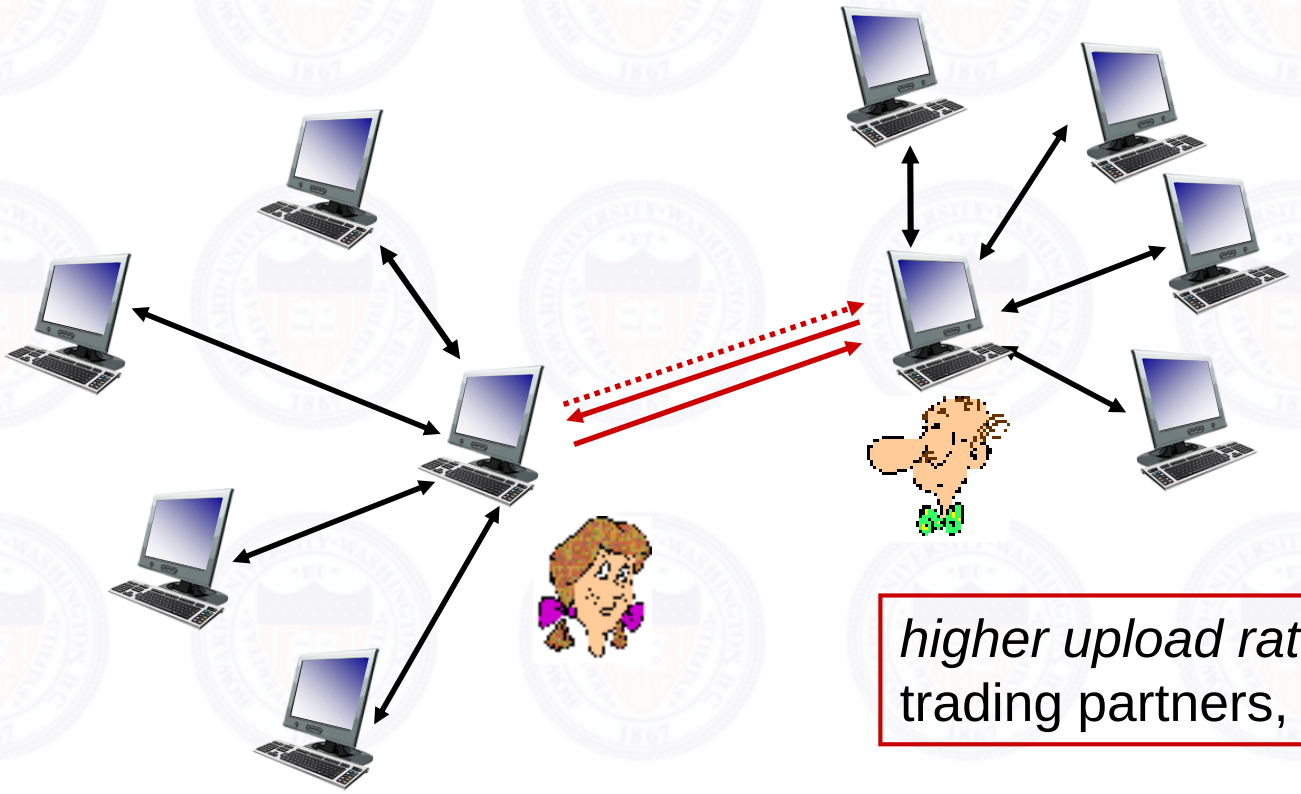
- Alice sends chunks to those four peers currently sending her chunks at highest rate
  - Other peers are choked by Alice (do not receive chunks from her)
  - Re-evaluate top 4 every 10 secs
- Every 30 secs: randomly select another peer, starts sending chunks
  - "Optimistically unchoke" this peer
  - 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)

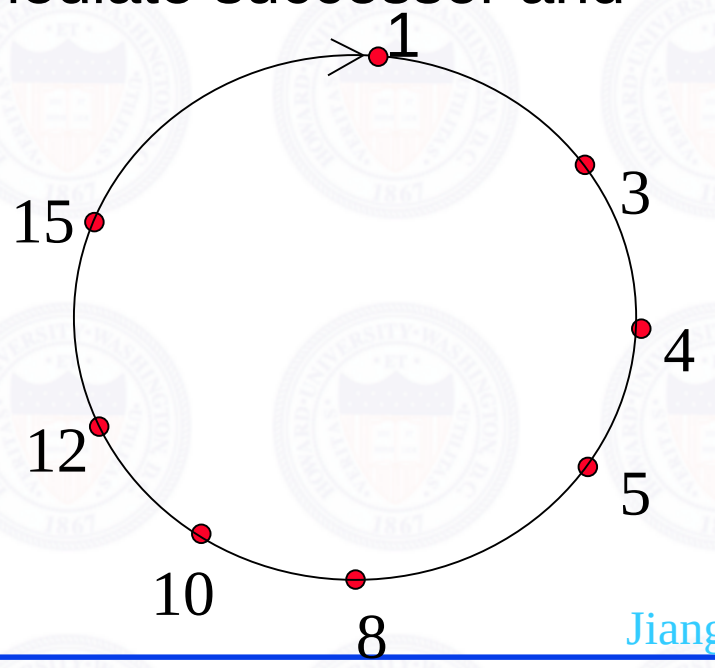
- DHT: a *distributed P2P database*
- Database has (key, value) pairs; examples:
  - Key: ss number; value: human name
  - Key: movie title; value: IP address
- Distribute the (key, value) pairs over the (millions of peers)
- A peer queries DHT with key
  - DHT returns values that match the key
- Peers 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

Peer Identifiers in DHT

- Assign integer identifier to each peer in range [0,2^n-1] for some *n*.
  - Each identifier represented by *n* bits.
- Circular DHT
  - Each peer *only* aware of immediate successor and predecessor.
  - "Overlay network"



DHT Keys

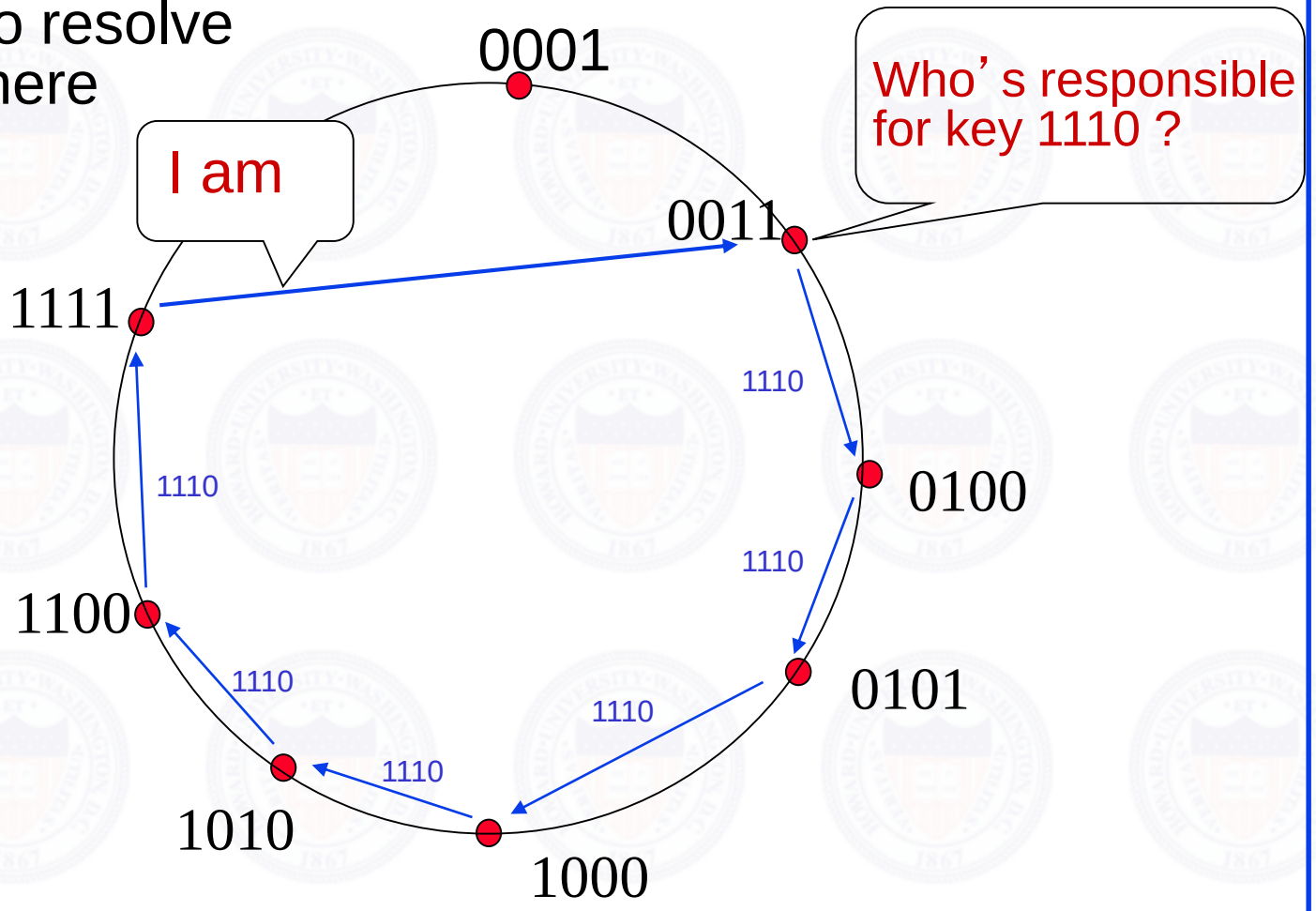
- Require each key to be an integer in same range as peer identifiers
- To get integer key, hash original key
  - *e.g.*, key = hash("Led Zeppelin IV")
  - This is why it is referred to as a *distributed "hash" table*

Assigning Keys to Peers

- Rule: assign key to the peer that has the *closest* ID.
- Convention in lecture: closest is the *immediate successor* of the key.
- 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

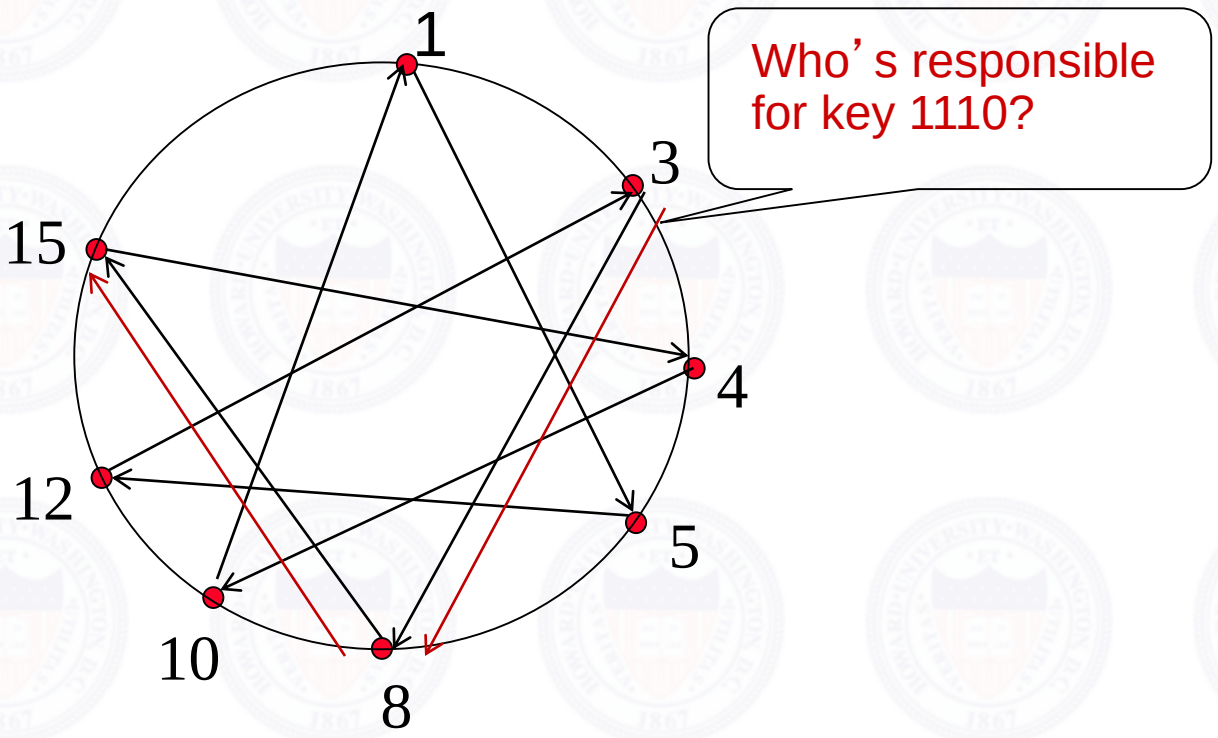
Queries in Circular DHT

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



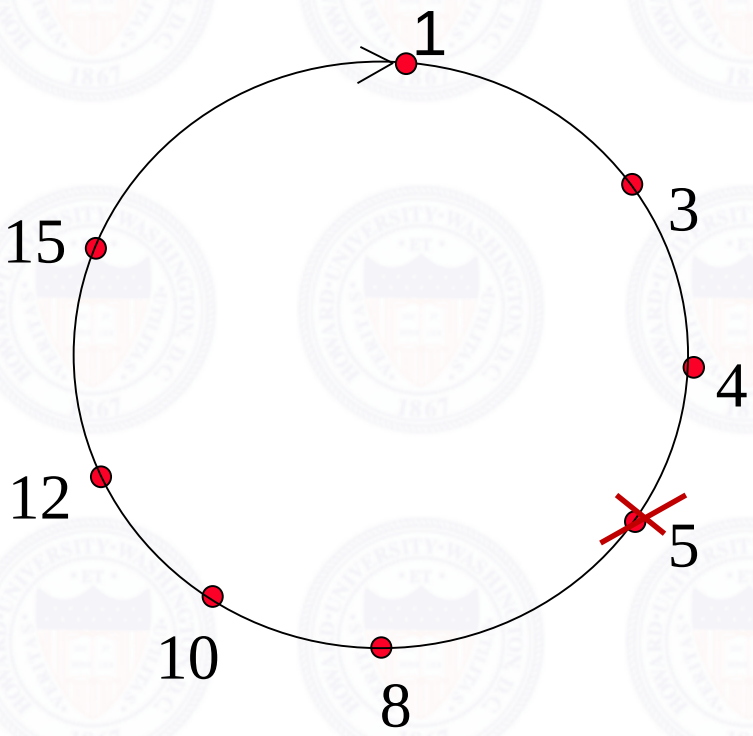
Define closest as closest successor

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

Peer Churn



Handling peer churn:

- Peers may come and go (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?

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks (CDNs)

2.7 socket programming with UDP and TCP

Video Streaming and CDNs: context

- video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube: 37%, 16% of downstream residential ISP traffic
  - ~1B YouTube users, ~75M Netflix users
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution:* distributed, application-level infrastructure

Multimedia: video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image
  - spatial (within image)
  - temporal (from one image to next)

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)

frame *i*

frame *i+1*

temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

Multimedia: video

- CBR: (constant bit rate): video encoding rate fixed
- VBR: (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- examples:
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, < 1 Mbps)

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)

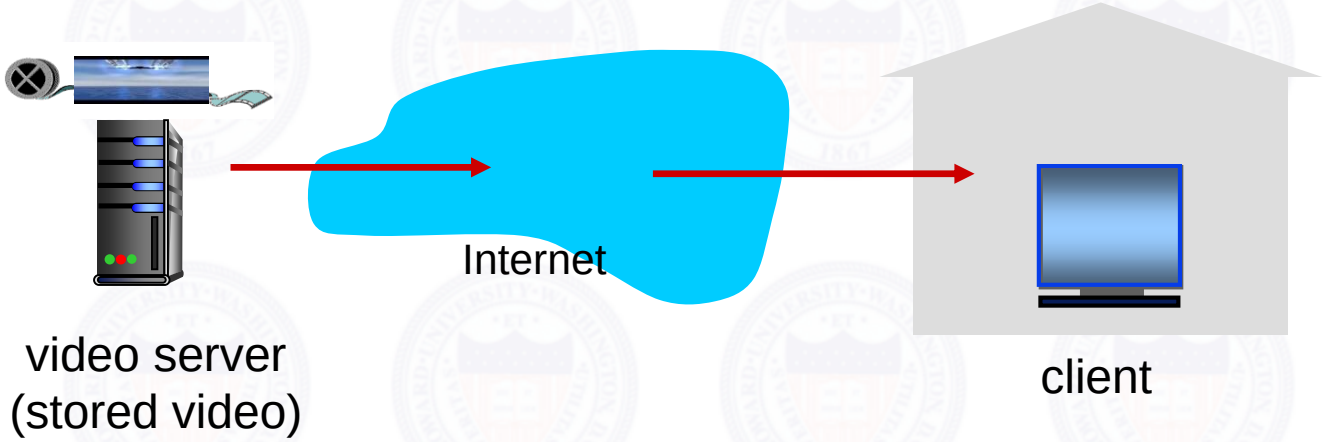
frame *i*

frame *i+1*

temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

Streaming stored video:

simple scenario:



Streaming multimedia: DASH

- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
- *server:*
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - *manifest file:* provides URLs for different chunks
- *client:*
  - periodically measures server-to-client bandwidth
  - consulting manifest, requests one chunk at a time
    - chooses maximum coding rate sustainable given current bandwidth
    - can choose different coding rates at different points in time (depending on available bandwidth at time)

Streaming multimedia: DASH

- *DASH: Dynamic, Adaptive Streaming over HTTP*
- *"intelligence"* at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is "close" to client or has high available bandwidth)

Content distribution networks

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?

- *option 1:* single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of video sent over outgoing link

….quite simply: this solution *doesn't scale*

Content distribution networks

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of simultaneous users?

- *option 2:* store/serve multiple copies of videos at multiple geographically distributed sites *(CDN)*
  - *enter deep:* push CDN servers deep into many access networks
    - close to users
    - used by Akamai, 1700 locations
  - *bring home:* smaller number (10's) of larger clusters in POPs near (but not within) access networks
    - used by Limelight

Content Distribution Networks (CDNs)

- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested

Content Distribution Networks (CDNs)

Internet host-host communication as a service

OTT challenges: coping with a congested Internet

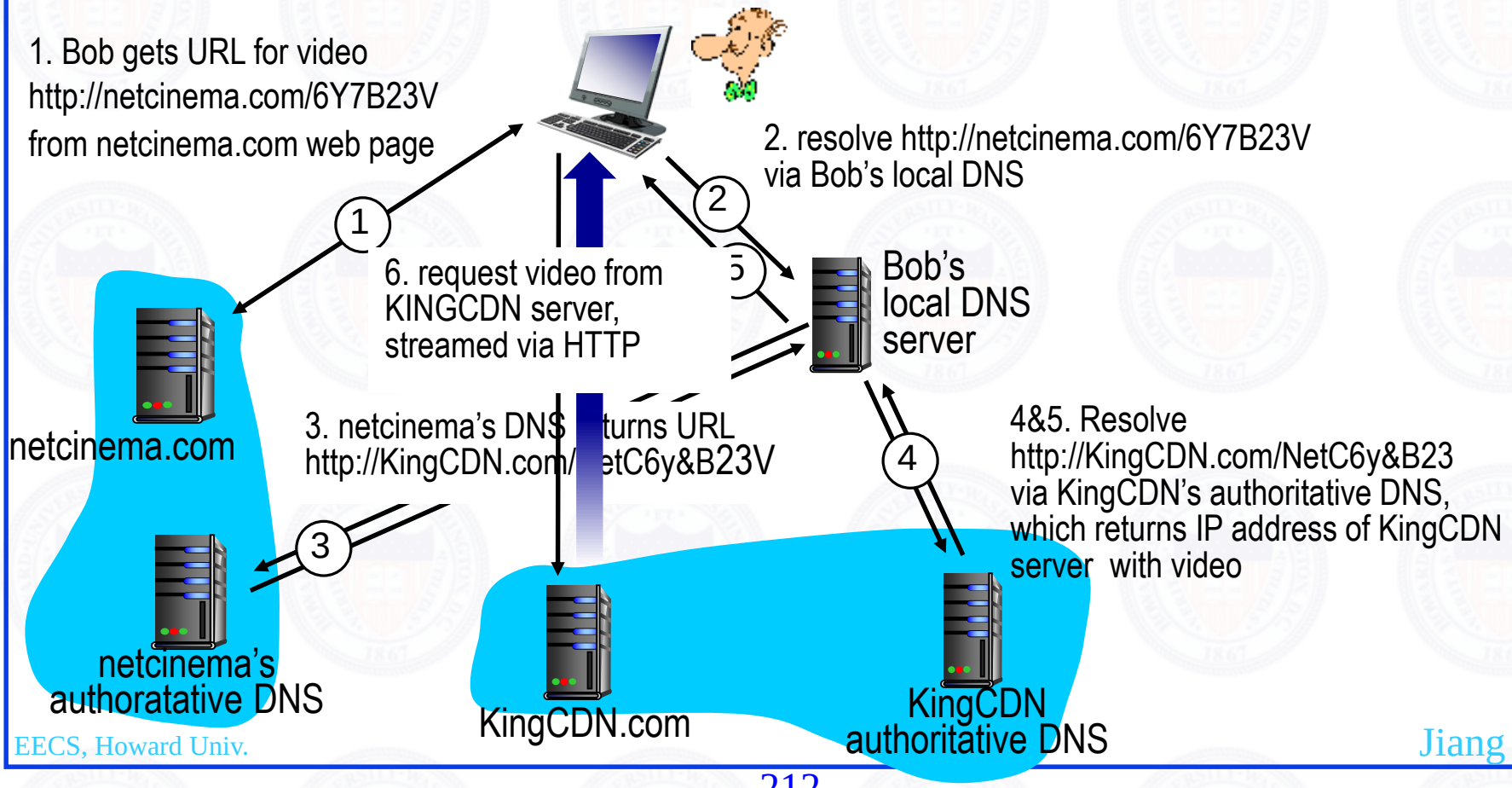
- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

more .. in chapter 7

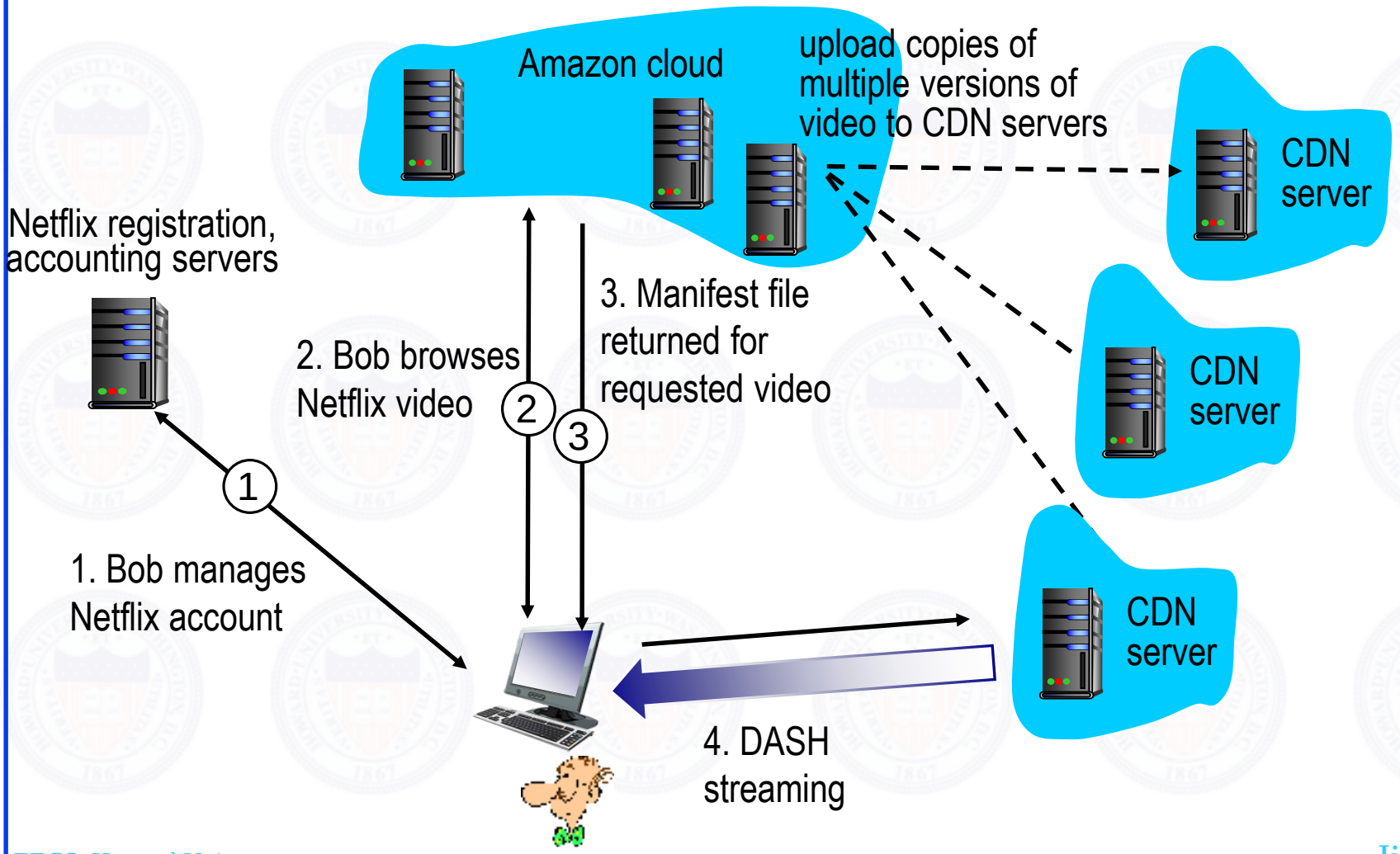
CDN content access: a closer look

Bob (client) requests video http://netcinema.com/6Y7B23V

- video stored in CDN at http://KingCDN.com/NetC6y&B23V



Case study: Netflix



Chapter 2: Summary

Our study of network apps now complete!

- Application architectures
  - Client-server
  - P2P
  - Hybrid
- Application service requirements:
  - Reliability, bandwidth, delay
- Internet transport service model
  - Connection-oriented, reliable: TCP
  - Unreliable, datagrams: UDP
- Specific protocols:
  - **HTTP**
  - FTP
  - SMTP, POP, IMAP
  - **DNS**
  - P2P: BitTorrent, Skype
- Socket programming

Chapter 2: Summary

Most importantly: learned about *protocols*

- Typical request/reply message exchange:
  - Client requests info or service
  - Server responds with data, status code
- Message formats:
  - Headers: fields giving info about data
  - Data: info being communicated

Important themes:

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