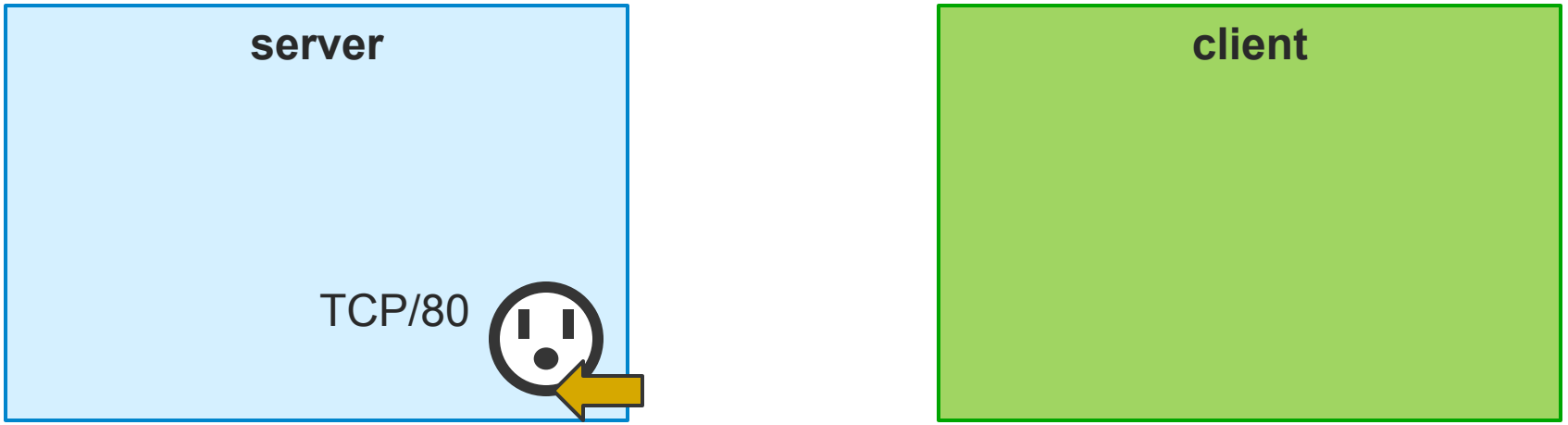
Sockets: send, recv
Network Applications: HTTP

Announcements

- Still using this nifty old slide format...
- MP7 due tomorrow
- Brighten's office hours
  - Tue 3:30 – 5:30, 0220 SC
  - Wed 3:00 – 5:00, 3211 SC

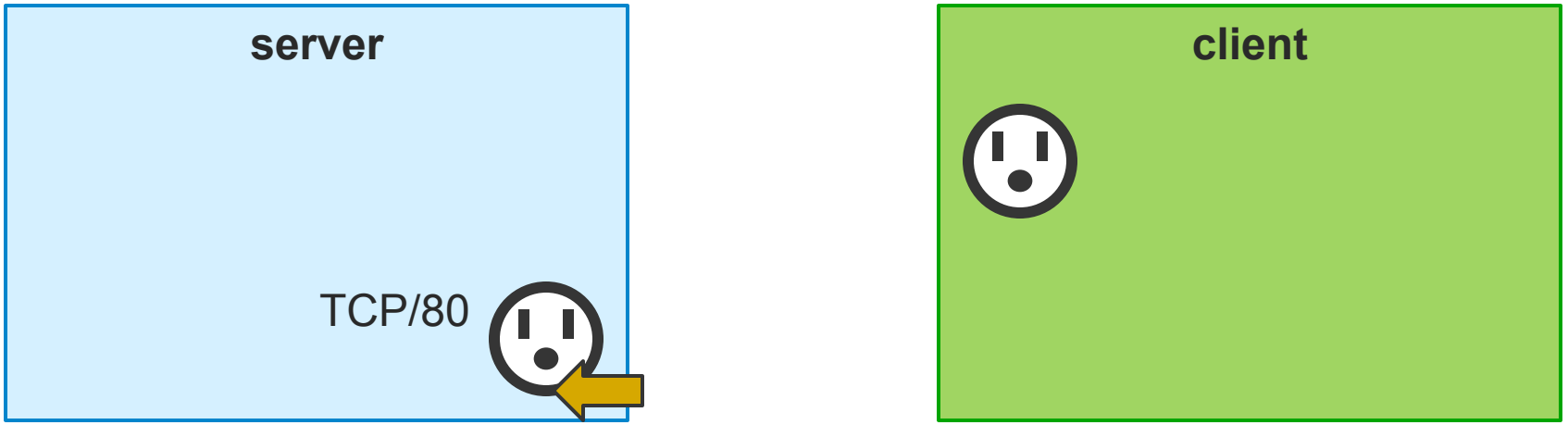
Client-Server Model

- Server:
  - Creates a socket to listen for incoming connections.
  - Must listen on a specific protocol/port.



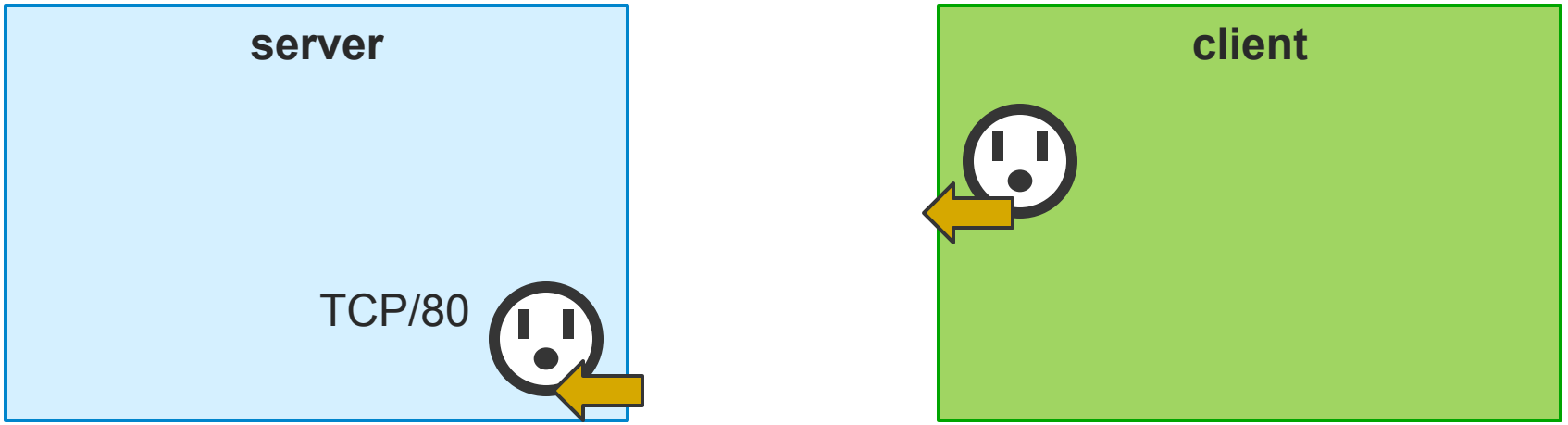
Client-Server Model

- Client:
  - Creates a socket to connect to a remote computer.



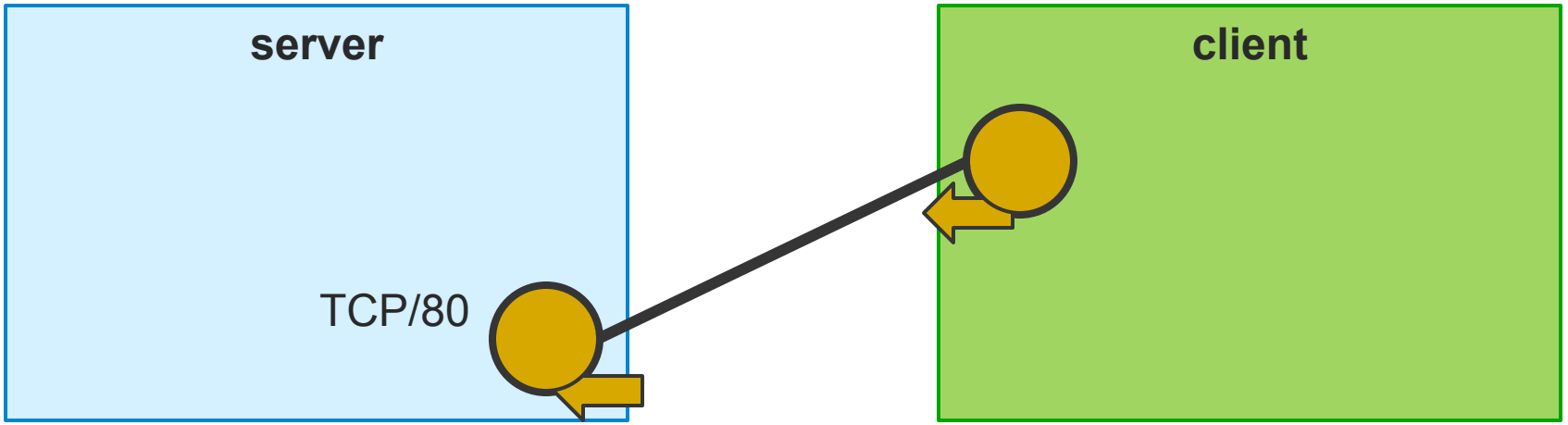
Client-Server Model

- Client:
  - Requests a connection to TCP port 80 on 74.125.225.70.



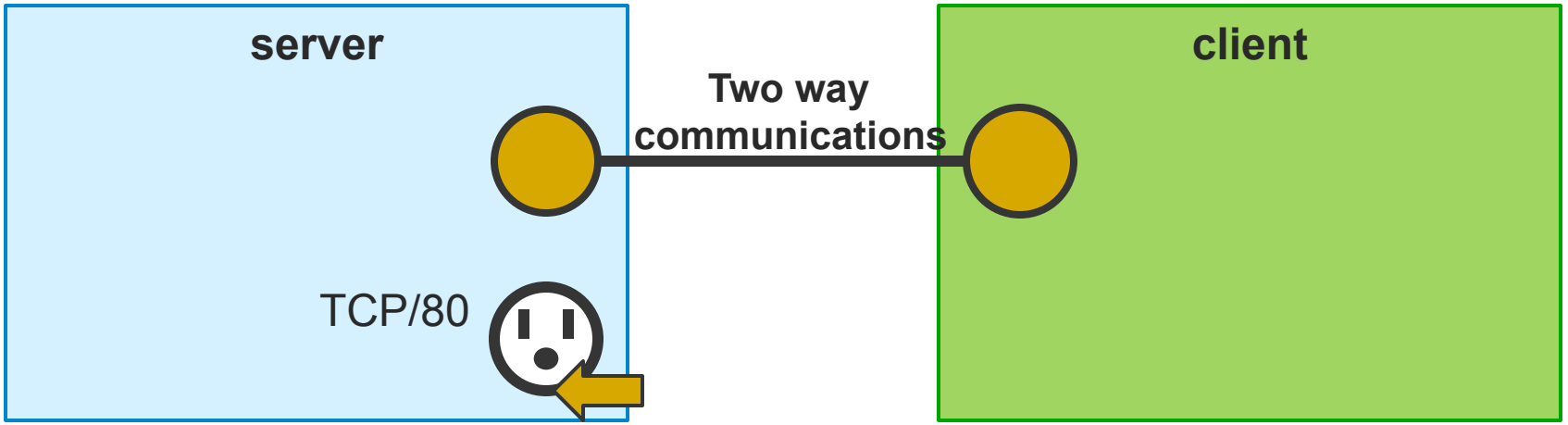
Client-Server Model

- Server:
  - Accepts the connection.



Client-Server Model

- Server:
  - Spawns a new socket to communicate directly with the newly connected client.
  - Allows other clients to connect.



Creating a "Server Socket"

socket(): Creates a new socket for a specific protocol (eg: TCP)

bind(): Binds the socket to a specific port (eg: 80)

listen(): Moves the socket into a state of listening for incoming connections.

accept(): Accepts an incoming connection.

Creating a “Client Socket”

socket(): Creates a new socket for a specific protocol (eg: TCP)

connect():

Makes a network connection to a specified IP address and port.

Functions: accept

- Notes
  - After **`accept()`** returns a new socket descriptor, I/O can be done using **`read()`** and **`write()`**
  - Why does **`accept()`** need to return a new descriptor?

Sending and Receiving Data

```
int send(int sockfd, const void * buf,
   size_t nbytes, int flags);
```

- Write data to a stream (TCP) or "connected" datagram (UDP) socket.
  - Returns number of bytes written or -1.

```
int recv(int sockfd, void *buf, size_t
   nbytes, int flags);
```

- Read data from a stream (TCP) or "connected" datagram (UDP) socket.
  - Returns number of bytes read or -1.

Functions: send

```
int send(int sockfd, const void * buf, size_t
   nbytes, int flags);
```

- Send data un a stream (TCP) or “connected” datagram (UDP) socket
  - Returns number of bytes written or -1 and sets **`errno`** on failure
  - **`sockfd`**: socket file descriptor (returned from **`socket`**)
  - **`buf`**: data buffer
  - **`nbytes`**: number of bytes to try to write
  - **`flags`**: control flags
    - MSG_PEEK: get data from the beginning of the receive queue without removing that data from the queue

Functions: send

```
int send(int sockfd, const void * buf, size_t
   nbytes, int flags);
```

- Example

```
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
```


Functions: recv

```
int recv(int sockfd, void *buf, size_t nbytes,
    int flags);
```

- Read data from a stream (TCP) or "connected" datagram (UDP) socket
  - Returns number of bytes read or -1, sets **`errno`** on failure
  - Returns 0 if socket closed
  - **`sockfd`**: socket file descriptor (returned from **`socket`**)
  - **`buf`**: data buffer
  - **`nbytes`**: number of bytes to try to read
  - **`flags`**: see man page for details; typically use 0

Functions: recv

```
int recv(int sockfd, char* buf, size_t nbytes);
```

- Notes
  - **`read`** blocks waiting for data from the client but does not guarantee that **`sizeof(buf)`** is read
  - Example

```
if((r = read(newfd, buf, sizeof(buf))) < 0) {
   perror("read"); exit(1);
}
```


Sending and Receiving Data

- Datagram sockets aren't connected to a remote host
  - What piece of information do we need to give before we send a packet?
  - The destination/source address!

Sending and Receiving Data

```
int sendto (int sockfd, char* buf,
   size_t nbytes, int flags, struct
   sockaddr* destaddr, int addrlen);
```

- Send a datagram to another UDP socket.
  - Returns number of bytes written or -1.

```
int recvfrom (int sockfd, char* buf,
   size_t nbytes, int flags, struct
   sockaddr* srcaddr, int* addrlen);
```

- Read a datagram from a UDP socket.
  - Returns number of bytes read or -1.

Functions: sendto

```
int sendto (int sockfd, char* buf, size_t nbytes,
   int flags, struct sockaddr* destaddr, int
   addrlen);
```

- Send a datagram to another UDP socket
  - Returns number of bytes written or -1 and sets **`errno`** on failure
  - **`sockfd`**: socket file descriptor (returned from **`socket`**)
  - **`buf`**: data buffer
  - **`nbytes`**: number of bytes to try to read
  - **`flags`**: see man page for details; typically use 0
  - **`destaddr`**: IP address and port number of destination socket
  - **`addrlen`**: length of address structure
    - **`= sizeof (struct sockaddr_in)`**

Functions: sendto

```
int sendto (int sockfd, char* buf, size_t nbytes,
   int flags, struct sockaddr* destaddr, int
   addrlen);
```

- Example

```
n = sendto(sock, buf, sizeof(buf), 0,(struct
    sockaddr *) &from,fromlen);
if (n < 0)
    perror("sendto");
    exit(1);
}
```


Functions: recvfrom

```
int recvfrom (int sockfd, char* buf, size_t
   nbytes, int flags, struct sockaddr* srcaddr,
   int* addrlen);
```

- Read a datagram from a UDP socket.
  - Returns number of bytes read (0 is valid) or -1 and sets **`errno`** on failure
  - **`sockfd`**: socket file descriptor (returned from **`socket`**)
  - **`buf`**: data buffer
  - **`nbytes`**: number of bytes to try to read
  - **`flags`**: see man page for details; typically use 0
  - **`srcaddr`**: IP address and port number of sending socket (returned from call)
  - **`addrlen`**: length of address structure = pointer to **`int`** set to **`sizeof (struct sockaddr_in)`**

Functions: recvfrom

```
int recvfrom (int sockfd, char* buf, size_t
   nbytes, int flags, struct sockaddr* srcaddr,
   int* addrlen);
```

- Example

```
n = recvfrom(sock, buf, 1024, 0, (struct sockaddr
    *)&from,&fromlen);
if (n < 0) {
    perror("recvfrom");
    exit(1);
}
```


Tearing Down a Connection

```
int close (int sockfd);
```

- Close a socket.
  - Returns 0 on success, -1 and sets `errno` on failure.

```
int shutdown (int sockfd, int howto);
```

- Force termination of communication across a socket in one or both directions.
  - Returns 0 on success, -1 and sets `errno` on failure.

Functions: close

```
int close (int sockfd);
```

- Close a socket
  - Returns 0 on success, -1 and sets **`errno`** on failure
  - **`sockfd`**: socket file descriptor (returned from **`socket`**)
- Closes communication on socket in both directions
  - All data sent before **`close`** are delivered to other side (although this aspect can be overridden)
- After **`close`**, **`sockfd`** is not valid for reading or writing

Functions: shutdown

```
int shutdown (int sockfd, int howto);
```

- Force termination of communication across a socket in one or both directions
  - Returns 0 on success, -1 and sets **`errno`** on failure
  - **`sockfd`**: socket file descriptor (returned from **`socket`**)
  - **`howto`**:
    - **`SHUT_RD`** to stop reading
    - **`SHUT_WR`** to stop writing
    - **`SHUT_RDWR`** to stop both

- **`shutdown`** overrides the usual rules regarding duplicated sockets, in which TCP teardown does not occur until all copies have closed the socket

Note on `close` vs. `shutdown`

- **`close()`**: closes the socket but the connection is still open for processes that shares this socket
  - The connection stays opened both for read and write
- **`shutdown()`**: breaks the connection for all processes sharing the socket
  - A read will detect **`EOF`**, and a write will receive **`SIGPIPE`**
  - **`shutdown()`** has a second argument how to close the connection:
    - 0 means to disable further reading
    - 1 to disable writing
    - 2 disables both

Application Layer

Networked Applications

- All networked applications use "application level" protocols to communicate
- Examples
  - HTTP
  - FTP
  - SMTP
  - …

Web and HTTP

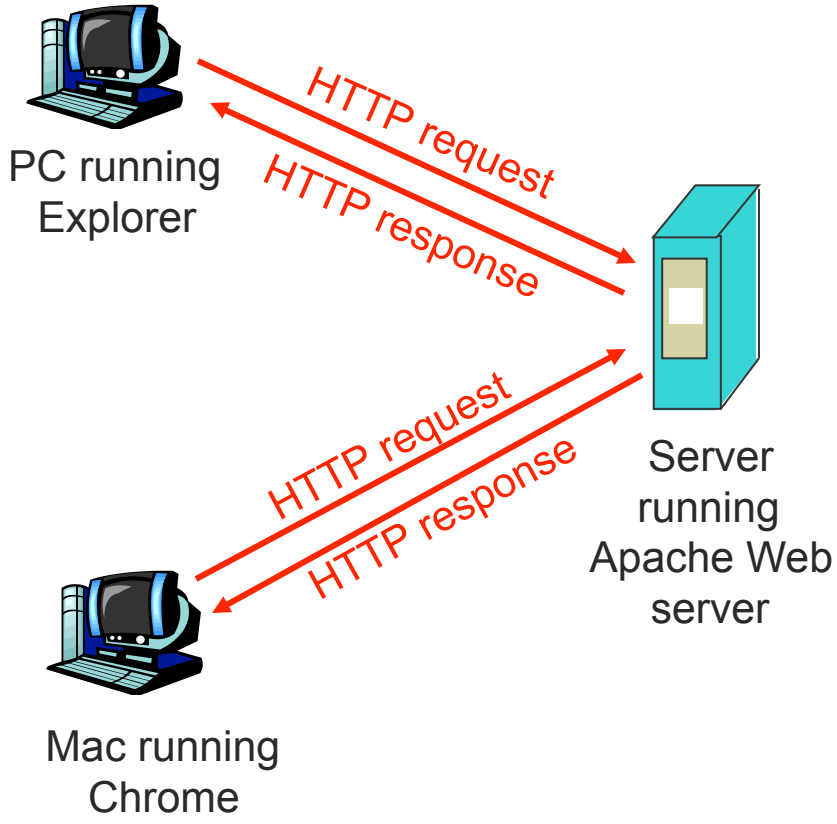
- Web pages consist of
  - Objects
    - HTML files, JPEG images, Java applets, audio files,…
  - Base HTML-file
    - Includes several referenced objects
- Each object is addressable by a URL
- Example URL:

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

host name — path name

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

- 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
- Stateless
  - Server maintains no information about past client requests

HTTP Connections

- Nonpersistent HTTP
  - At most one object is sent over a TCP connection
- Persistent HTTP
  - Multiple objects can be sent over single TCP connection between client and server

Nonpersistent HTTP

- User enters URL
  - Text plus references to 10 jpeg images

`www.someschool.edu/someDepartment/home.index`

1a. HTTP client initiates TCP connection to HTTP server 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 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

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

4. HTTP server closes TCP connection.

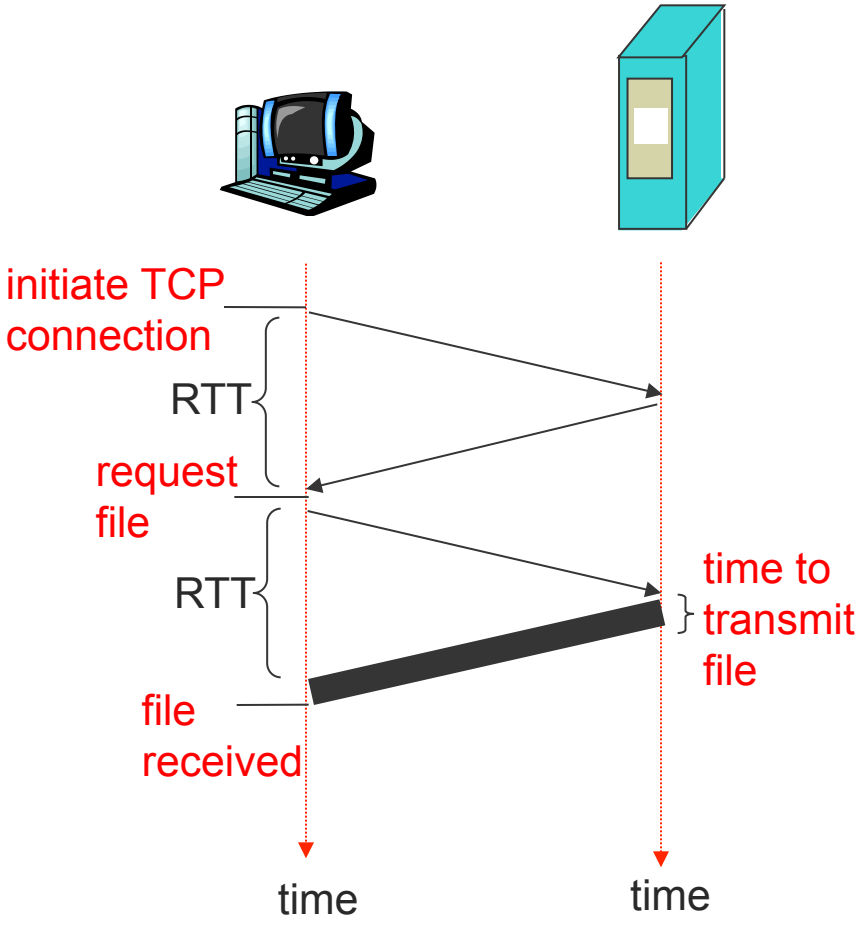
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

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

time

Response Time: First request

- RTT
  - Time for a small packet to travel from client to server and back
- Response time
  - One RTT to initiate TCP connection
  - \+ One RTT for HTTP request and first few bytes of HTTP response to return
  - \+ File transmission time
  - = 2RTT+transmit time



Response time for whole web page

- Nonpersistent HTTP
  - 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 total** for all the referenced objects
    - See “HTTP pipelining”

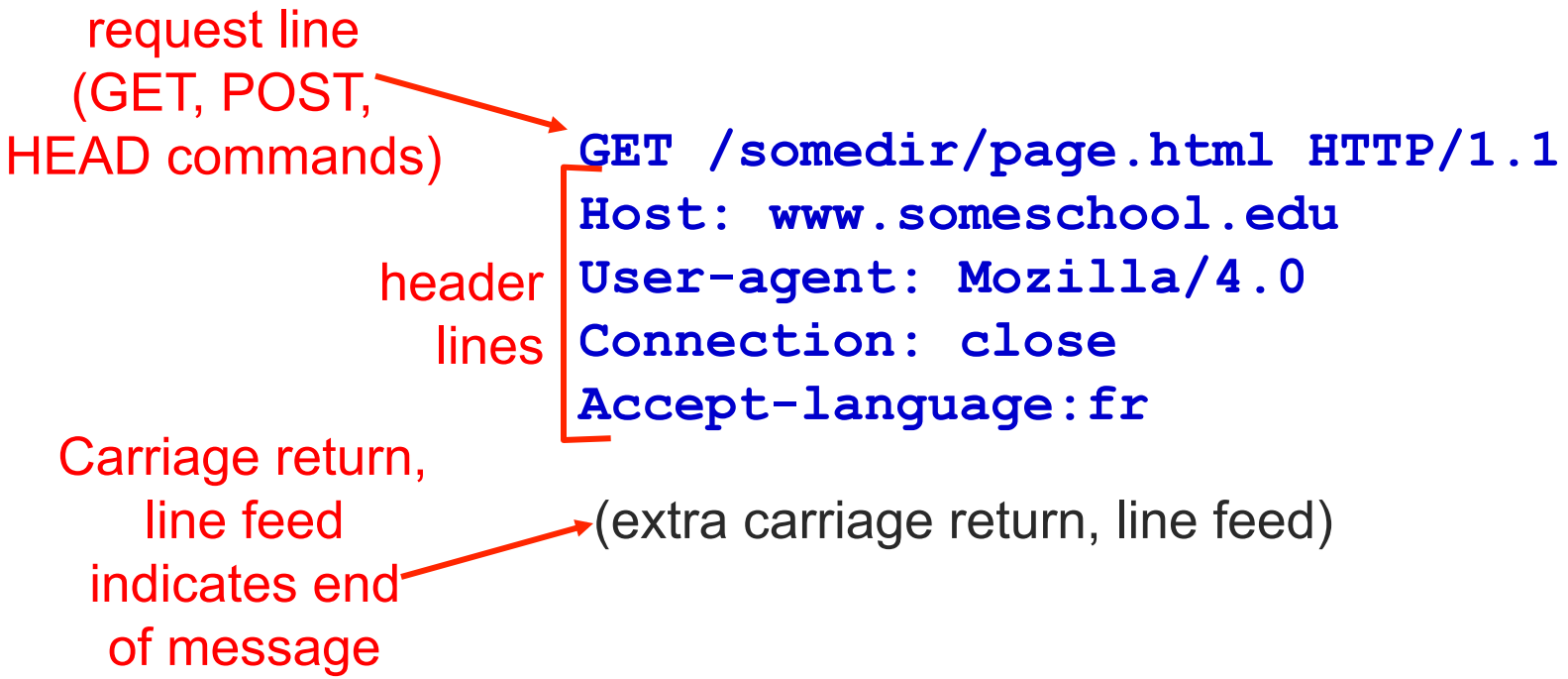
Aside: Do a few RTTs matter?

- Collective experiment

```
ping your_favorite_domain.foo
```


HTTP Request Message

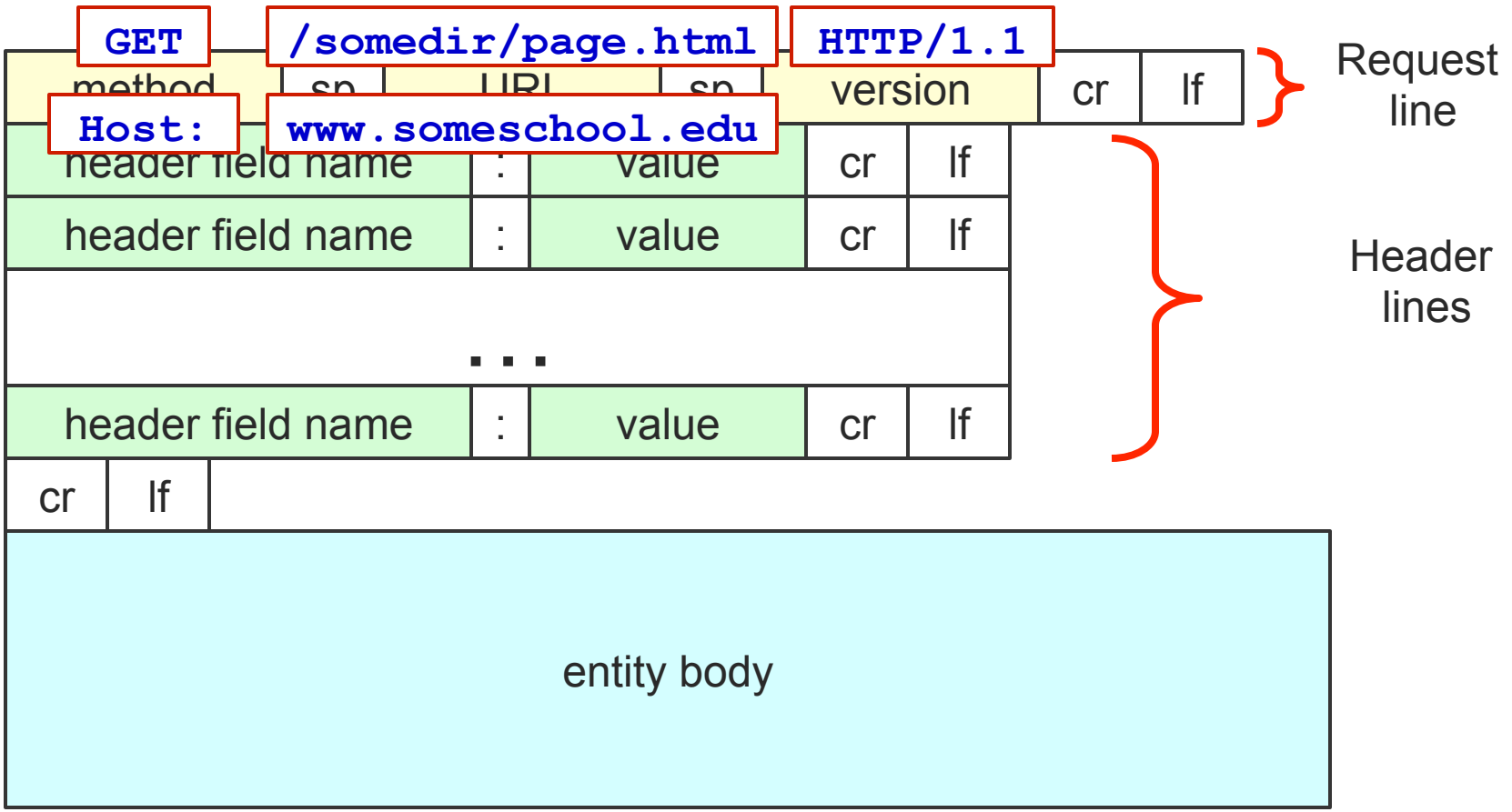
- Two types of HTTP messages: *request*, *response*
  - ASCII (human-readable format)
- HTTP request message:



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 Request Message: General Format

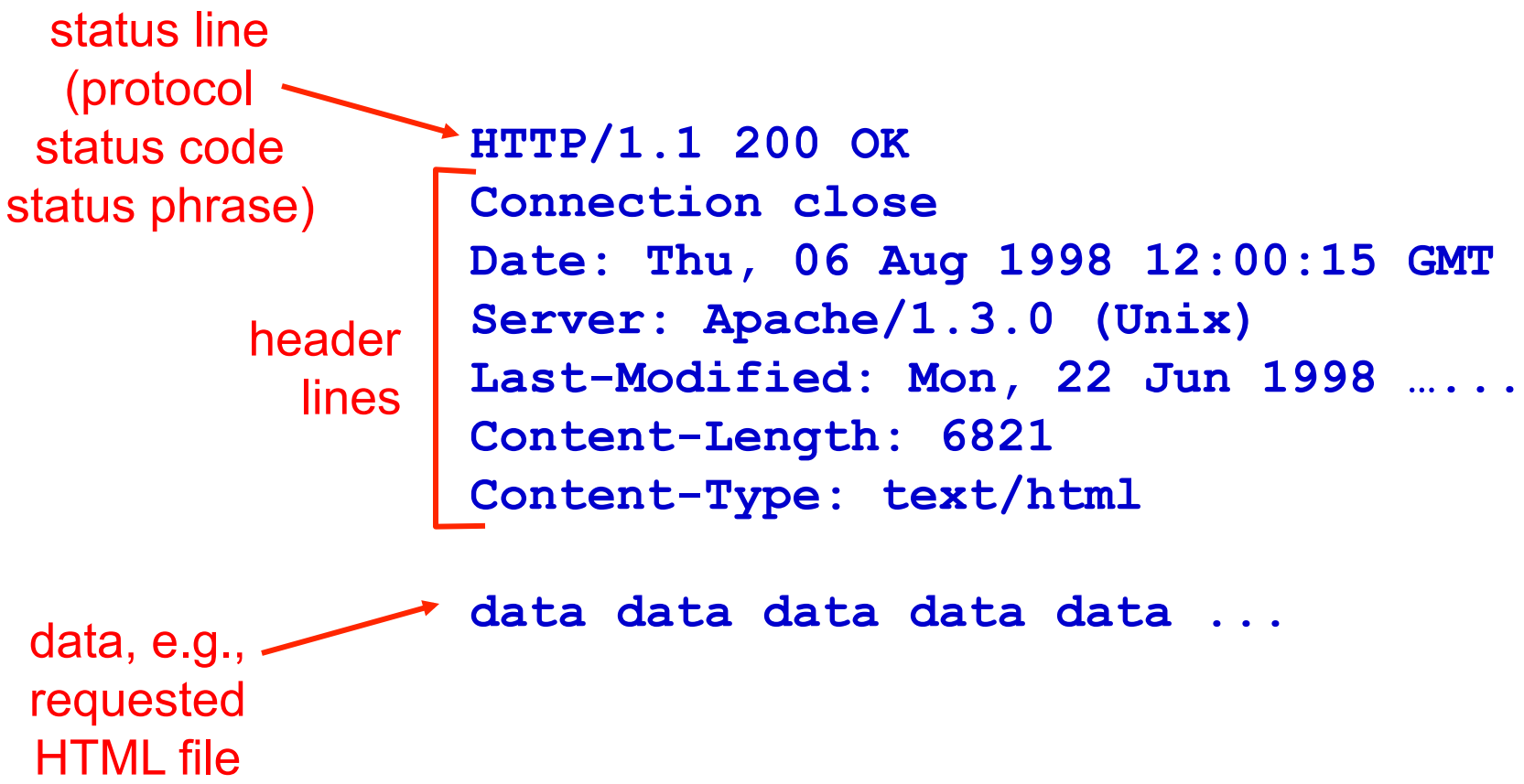


Uploading Form Input

- Post method
  - Web page often includes form of 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`

HTTP Response Message



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:), client automatically retrieves new URL
400	Bad Request	request message not understood by server
404	Not Found	requested document not found on this server
505	HTTP Version Not Supported	

HTTP response status codes

- In first line in server->client response message
- A few sample codes
- More in the illustrated guide...
  - http://tinyurl.com/cvyepwt

Trying out HTTP (client side) For Yourself

1. Telnet to your favorite Web server

```
telnet www.cs.illinois.edu 80
```

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

2. Type in a GET HTTP request

```
GET /class/sp12/cs241/index.html
    HTTP/1.0
```

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

3. Look at response message sent by HTTP server!

User-server State: Cookies

- Many major Web sites use cookies
- Four components
  1. Cookie header line of HTTP response message
  2. Cookie header line in HTTP request message
  3. Cookie file kept on user's host, managed by user's browser
  4. Back-end database at Web site
- Example
  - Alice always accesses Internet from PC
  - Visits specific e-commerce site for first time
  - When initial HTTP requests arrives at site, site creates:
    - unique ID
    - entry in backend database for ID

Cookies

- What cookies can bring
  - Authorization
  - Shopping carts
  - Recommendations
  - User session state (Web e-mail)
- How to keep “state”
  - Protocol endpoints: maintain state at sender/receiver over multiple transactions
  - cookies: http messages carry state
- Cookies and privacy
  - Cookies permit sites to learn a lot about you
  - You may supply name and e-mail to sites

Cookies: Keeping "State"