Discussion Session #4

EE450: Computer Networks
Topic: Network Applications
(HTTP, SMTP & P2P)

Some network apps

- e-mail/Text Messaging
- web-based applications
- Social Networking
- Internet Search
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)

- voice over IP
- real-time video conferencing
- cloud computing

Etc…

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
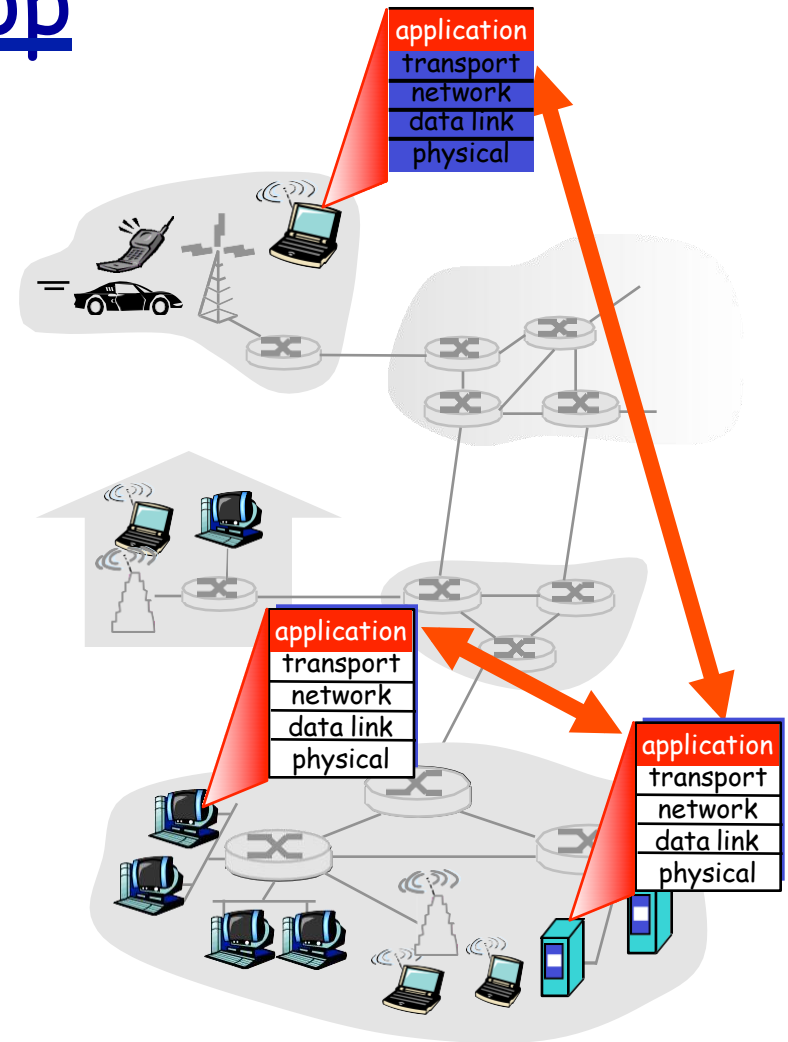
Processes communicating

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

❖ aside: 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
    - two sockets involved: one on each side

5
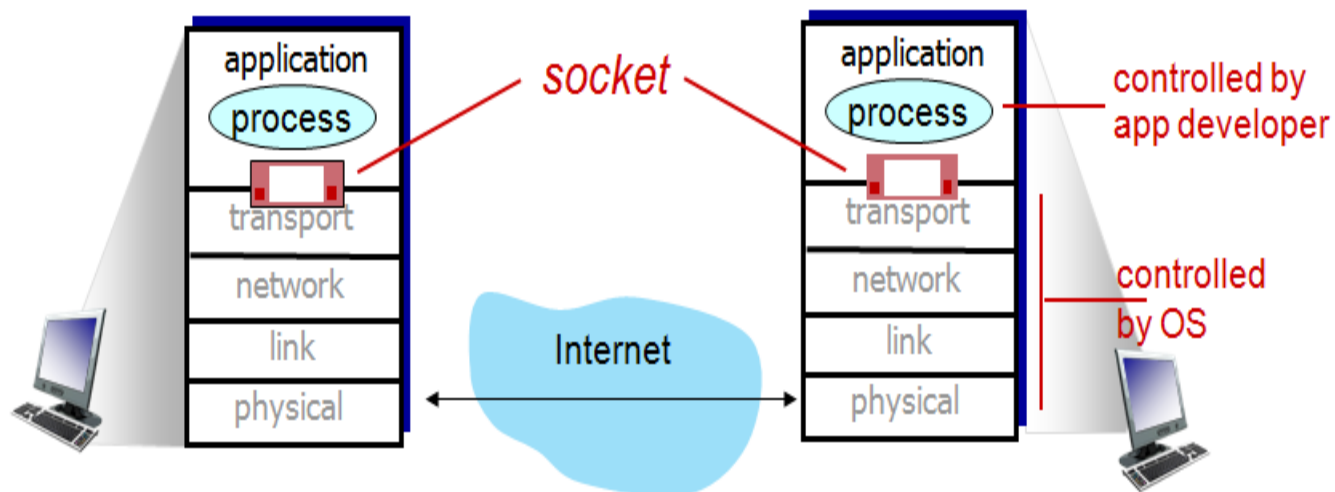
An application-layer protocol defines:

- **types of messages exchanged,**
  - e.g., request, response
- **message syntax:**
  - what fields in messages & how fields are delineated
- **message semantics**
  - meaning of information in fields
- **rules** for when and how processes send & respond to messages

open protocols:
- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:
- e.g., Skype

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process

- *flow control:* sender won't overwhelm receiver

- *congestion control:* throttle sender when network overloaded

- *does not provide:* timing, minimum throughput guarantee, security

- *connection-oriented:* setup required between client and server processes

UDP service:

- *unreliable data transfer* between sending and receiving process

- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Web and HTTP

First, a review...

❖ web page consists of objects

❖ object can be HTML file, JPEG image, Java applet, audio file,...

❖ web page consists of base HTML-file which includes several referenced objects

❖ each object is addressable by a URL
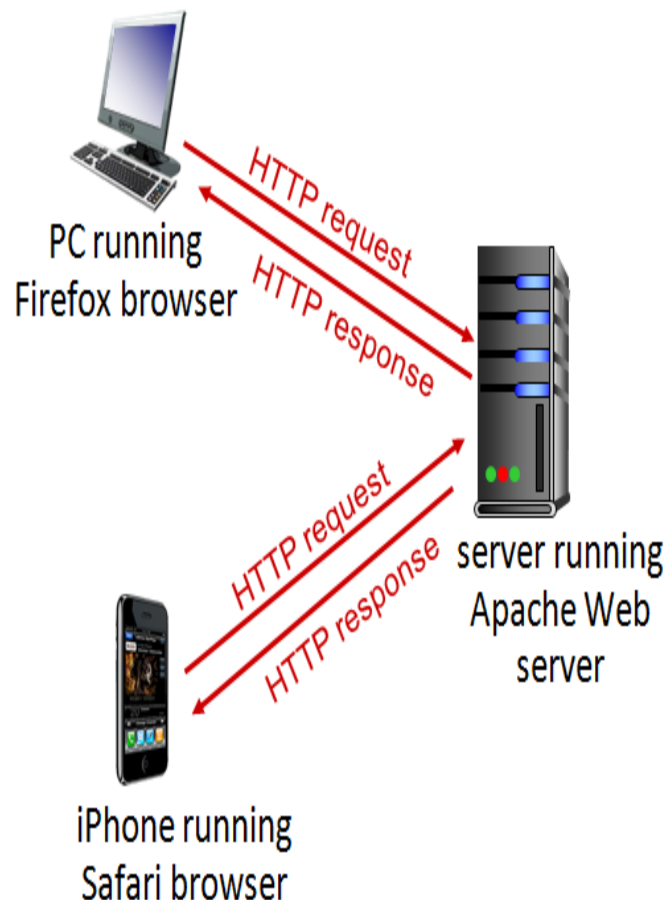
❖ example URL:

www.someschool.edu/someDept/pic.gif

host name   path name

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model:
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests

PC running
Firefox browser

HTTP request

HTTP response

server running
Apache Web
server

HTTP request

HTTP response

iPhone running
Safari browser

9

HTTP overview (continued)

Uses TCP:

❖ client initiates TCP connection (creates socket) to server, port 80

❖ server accepts TCP connection from client

❖ HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)

❖ TCP connection closed

HTTP is "stateless"

❖ server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

❖ past history (state) must be maintained

❖ if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP connections: two types

Non-persistent HTTP

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

Persistent HTTP

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

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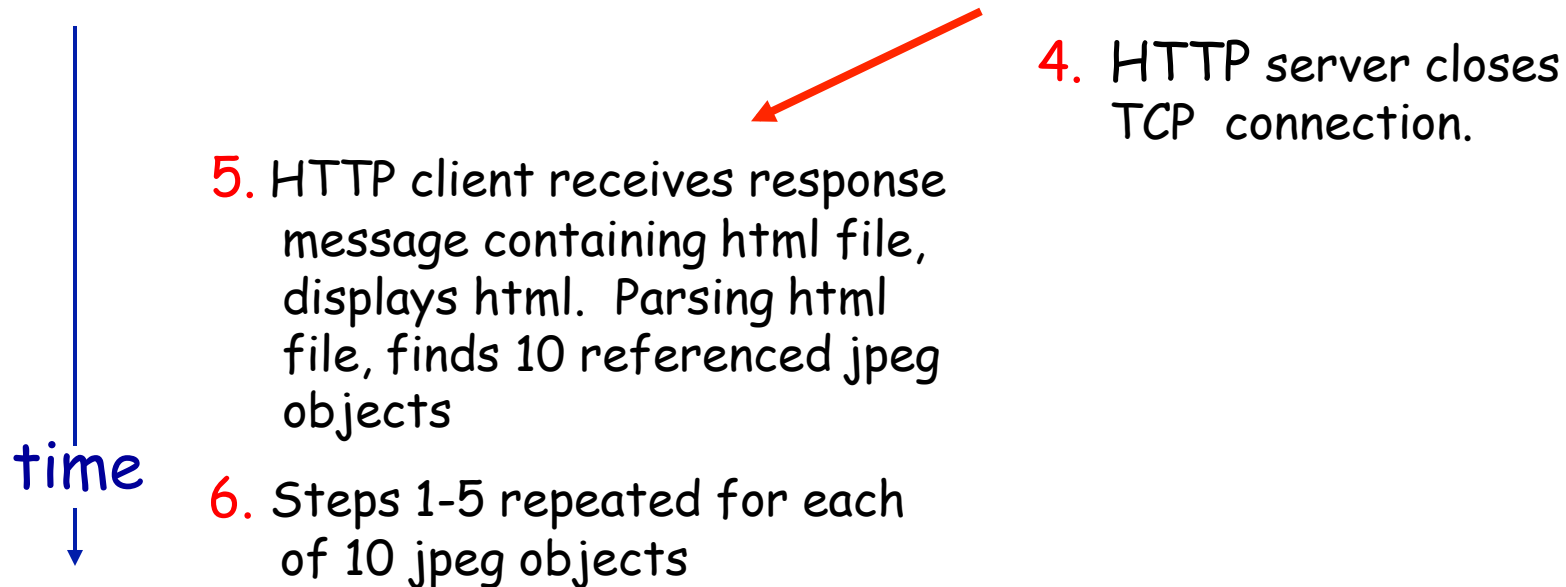
1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

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

12

Nonpersistent HTTP (cont.)

4. HTTP server closes TCP connection.

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

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

time

Non-Persistent HTTP: Response time

definition of RTT: time for a small packet to travel from client to server and back.

response time:

* one RTT to initiate TCP connection
* one RTT for HTTP request and first few bytes of HTTP response to return
* file transmission time

total = 2RTT+transmit time

initiate TCP connection

RTT

request file

RTT

time to transmit file

file received

time   time
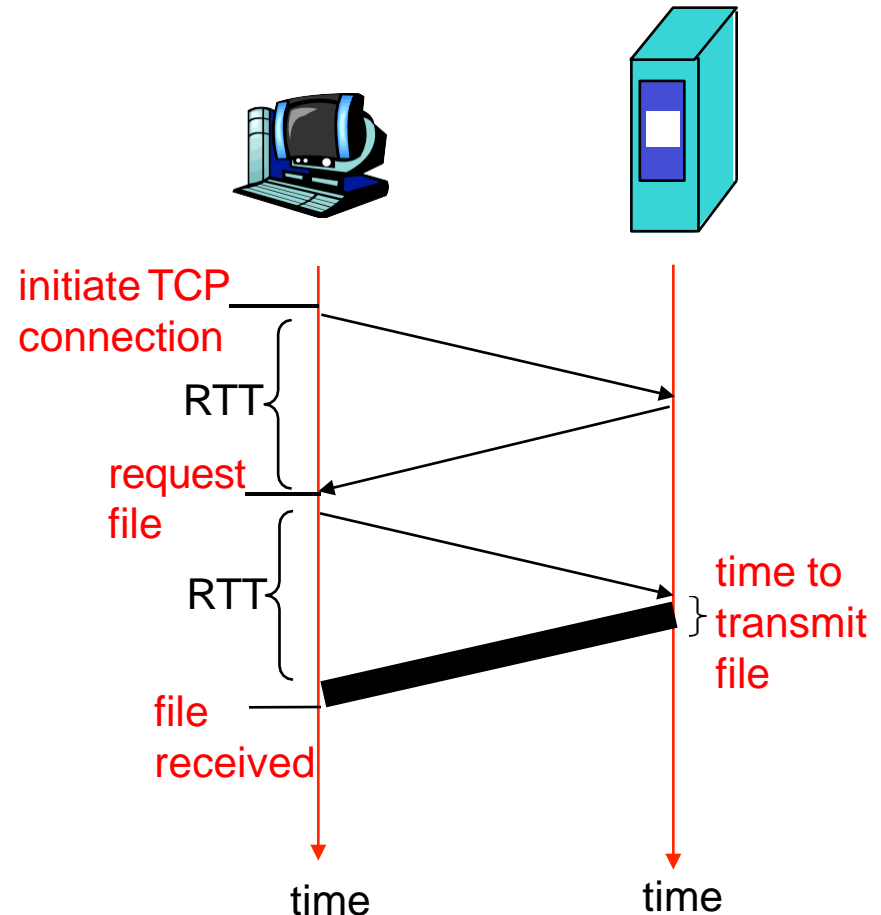
Persistent HTTP

non-persistent HTTP issues:
- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

persistent HTTP
- server leaves connection open after sending response
- subsequent HTTP messages between same client/ server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

HTTP request message

❖ two types of HTTP messages: *request, response*

❖ HTTP request message:

  ■ ASCII (human-readable format)

carriage return character

line-feed character

request line
(GET, POST,
HEAD commands)

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

header
lines

carriage return,
line feed at start
of line indicates
end of header lines

HTTP response message

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT
   \r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html;
   charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

header
lines

data, e.g.,
requested
HTML file

HTTP response status codes

❖ status code appears in 1st line in server->client response message.

❖ some sample codes:

200 OK
- request succeeded, requested object later in this msg

301 Moved Permanently
- requested object moved, new location specified later in this msg (Location:)

400 Bad Request
- request msg not understood by server

404 Not Found
- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

telnet cis.poly.edu 80 opens TCP connection to port 80
(default HTTP server port) at cis.poly.edu.
anything typed in sent
to port 80 at cis.poly.edu

2. type in a GET HTTP request:

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

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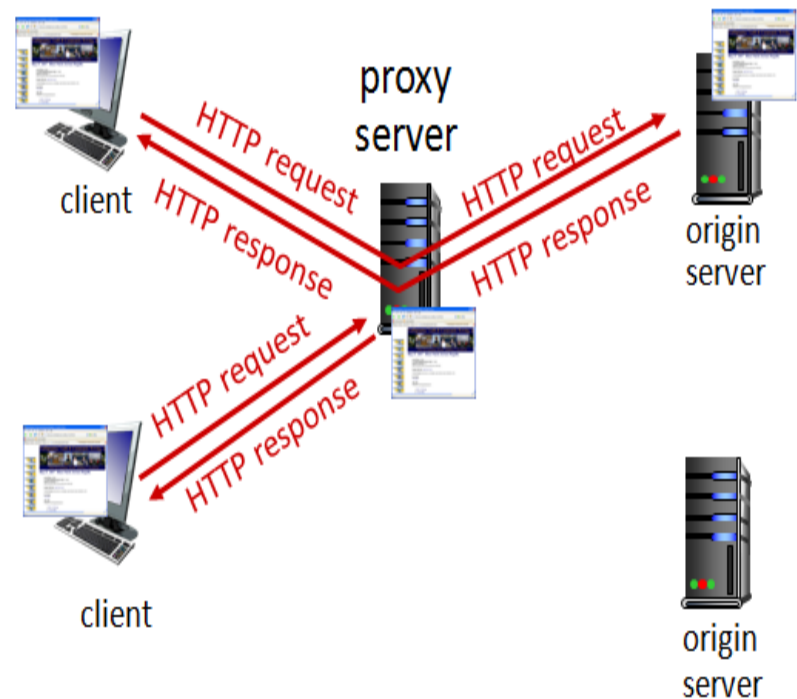this minimal (but complete)
GET request to HTTP server

3. look at response message sent by HTTP server!

(or use Wireshark!)

Web caches (proxy servers)

Goal: satisfy client request without involving origin server

- user configures browser to point to a *Web cache*
- browser sends all HTTP requests to cache
  - *if* object in cache: cache returns object to client
  - *else* cache requests object from origin server, caches received object, then returns object to client

proxy server

client

HTTP request
HTTP response
HTTP request
HTTP response

origin server

HTTP request
HTTP response

client

origin server

Web caches (proxy servers)

- Web 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
  - cache is closer to client
- reduce traffic on an institution's access link
- Internet is dense with caches
  - enables "poor" content providers to more effectively deliver content
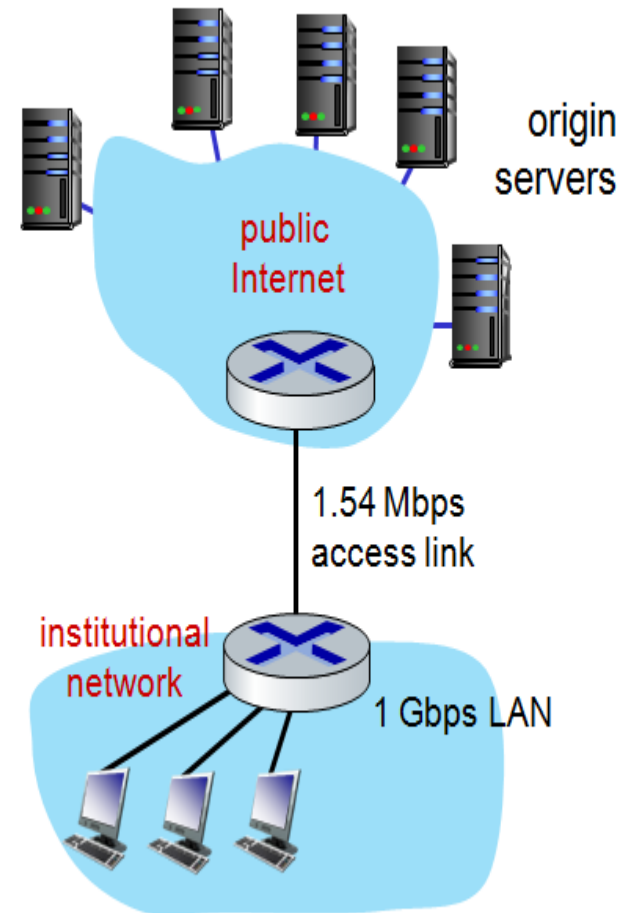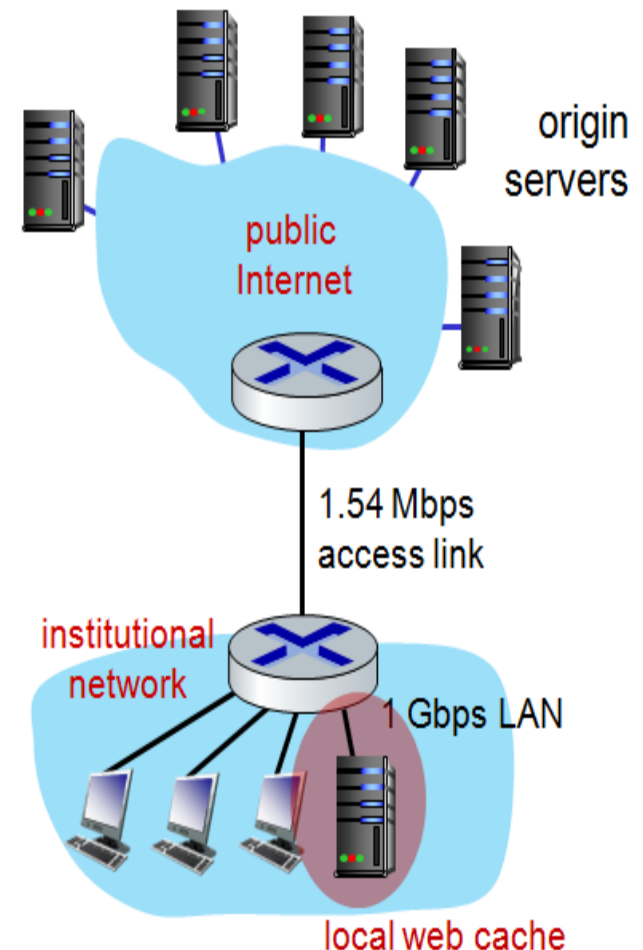
Caching example

Scenario:

- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Average request rate from browsers to origin servers: 15/sec
  - average data rate to browsers: 1.50 Mbps

Performance:

- LAN utilization: .0015
- access link utilization = .97
- end-end delay = Internet delay + access link delay + LAN delay

  = 2 sec + minutes + usecs

problem: large delays at high utilization!

origin servers

public Internet

1.54 Mbps access link

institutional network

1 Gbps LAN

Caching example: install a web cache

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

Performance:
- LAN utilization: .?
- access link utilization = ?
- average end-end delay = ?
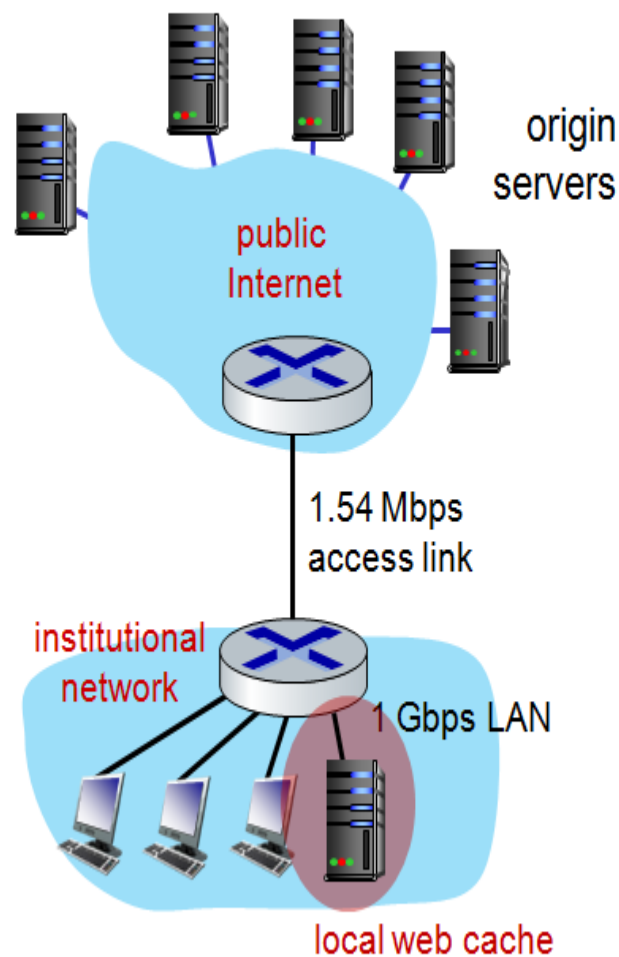
How to compute link utilization, delay?

Cost: web cache (cheap!)

origin servers

public Internet

1.54 Mbps access link

institutional network

1 Gbps LAN

local web cache

Caching example: install a web cache

Calculating access link utilization, end-end delay with cache:

- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 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
- average end-end delay
    = 0.6 * (delay from origin servers)
        + 0.4 * (delay when satisfied at cache)
    = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs

lower average end-end delay than with 154 Mbps link (and cheaper too!)

origin servers

public Internet

1.54 Mbps access link

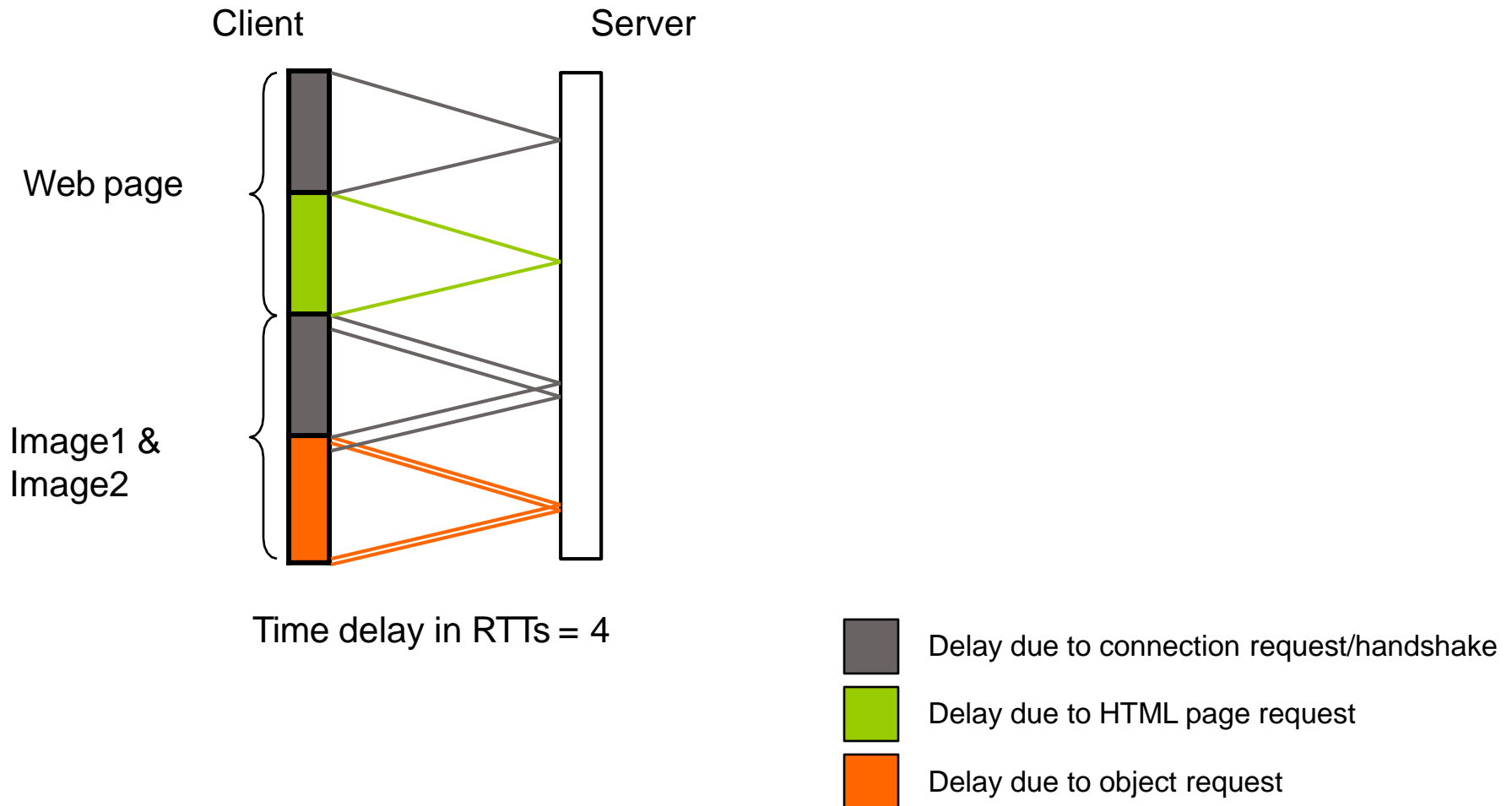institutional network

1 Gbps LAN

local web cache

Variations of HTTP

- Nonpersistent connections
  - with serial connections
  - with parallel connections

- Persistent connections
  - without pipelining
  - with pipelining
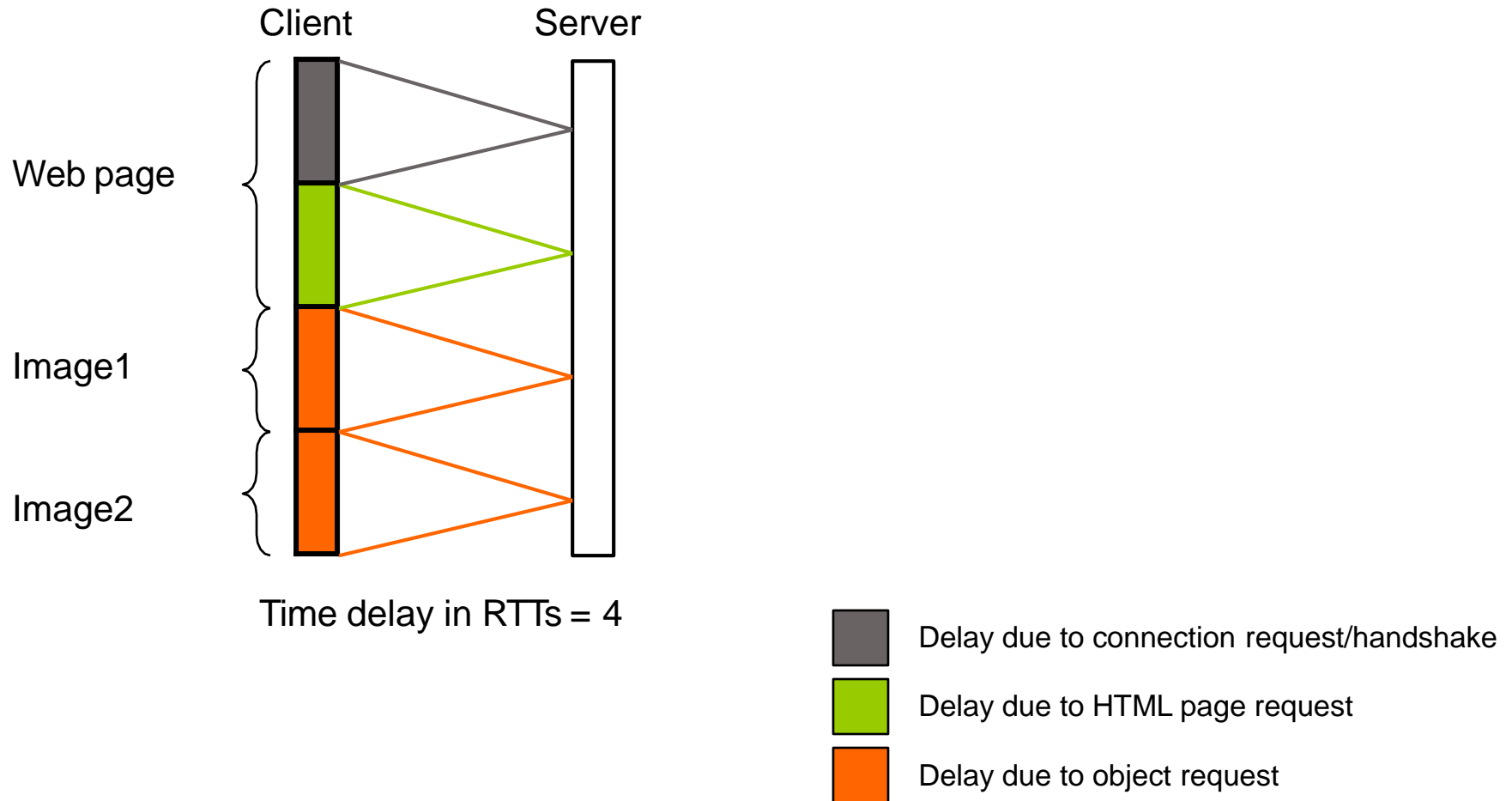
  ❖ Example: an HTML page with 2 objects (images)

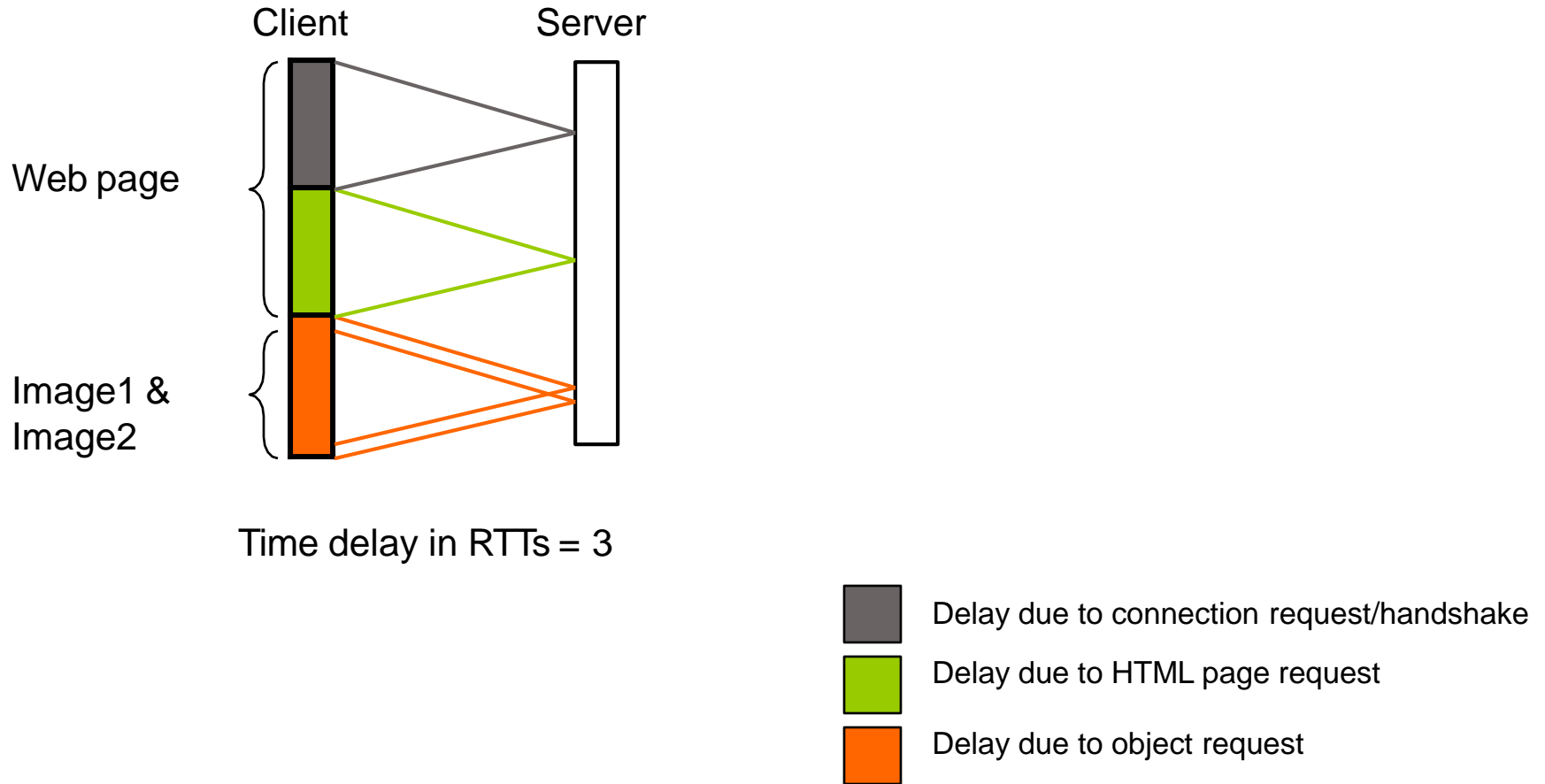# Non-Persistent: Rough calculation for number of RTTS

Client   Server

Web Page

Image 1

Can we reduce the number of RTTS?

Image 2

Time delay in RTTs = 6

Delay due to connection request/handshake

Delay Due to HTML Page Request

Delay Due to Object Request

26
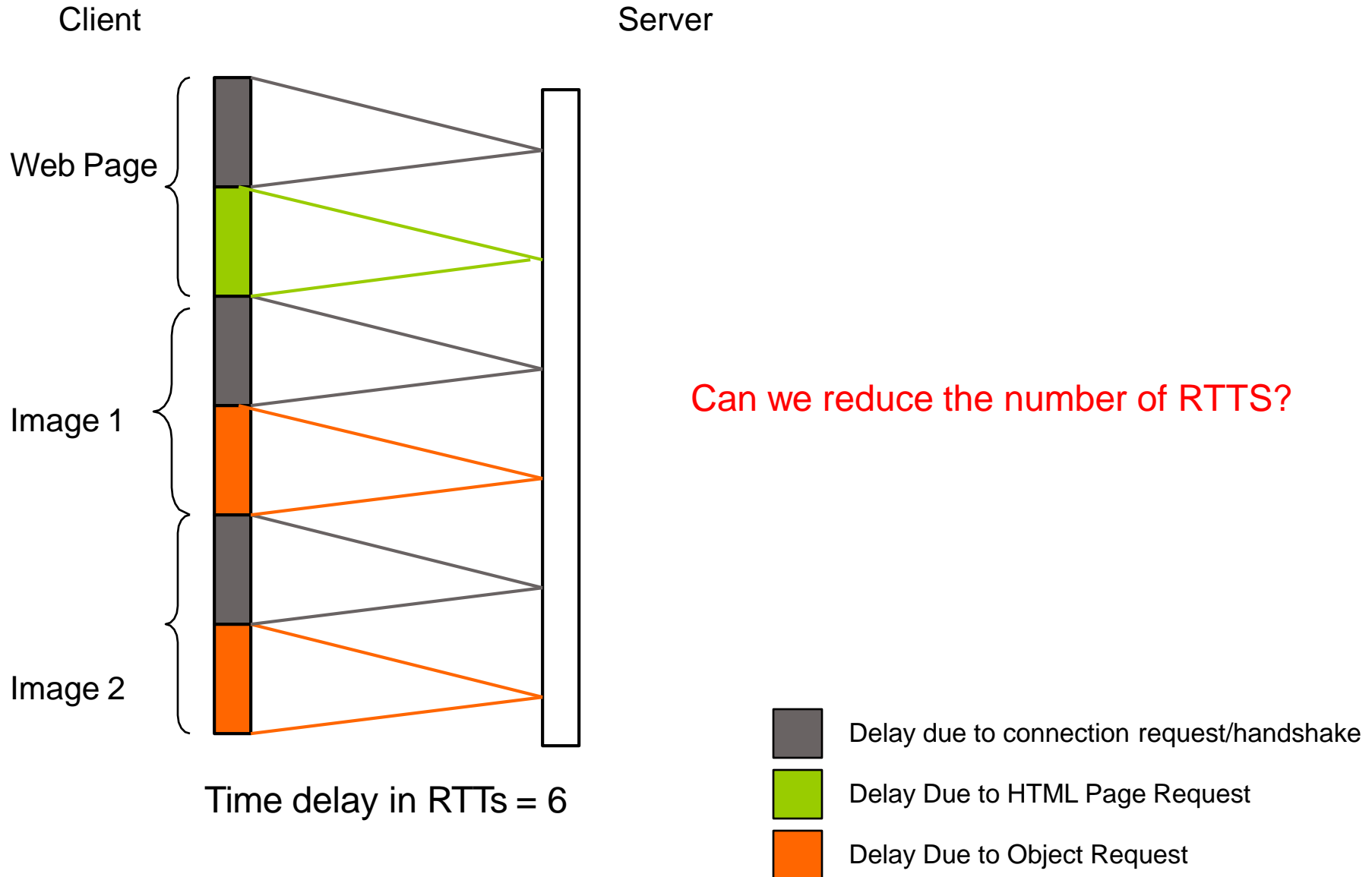
Non-Persistent with Parallel Connections: Rough calculation

Client   Server

Web page

Image1 & Image2

Time delay in RTTs = 4

■ Delay due to connection request/handshake

■ Delay due to HTML page request

■ Delay due to object request

Persistent Connection without Pipelining: Rough calculation

Client   Server

Web page

Image1

Image2

Time delay in RTTs = 4

■ (gray) Delay due to connection request/handshake

■ (green) Delay due to HTML page request

■ (orange) Delay due to object request

Persistent Connection with Pipelining: Rough calculation

Client   Server

Web page

Image1 & Image2

Time delay in RTTs = 3

■ Delay due to connection request/handshake

■ Delay due to HTML page request

■ Delay due to object request

Electronic Mail

Three major components:
- user agents
- mail servers
- simple mail transfer protocol: SMTP

User Agent
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, elm, Mozilla Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server

outgoing message queue

user mailbox

mail server

user agent
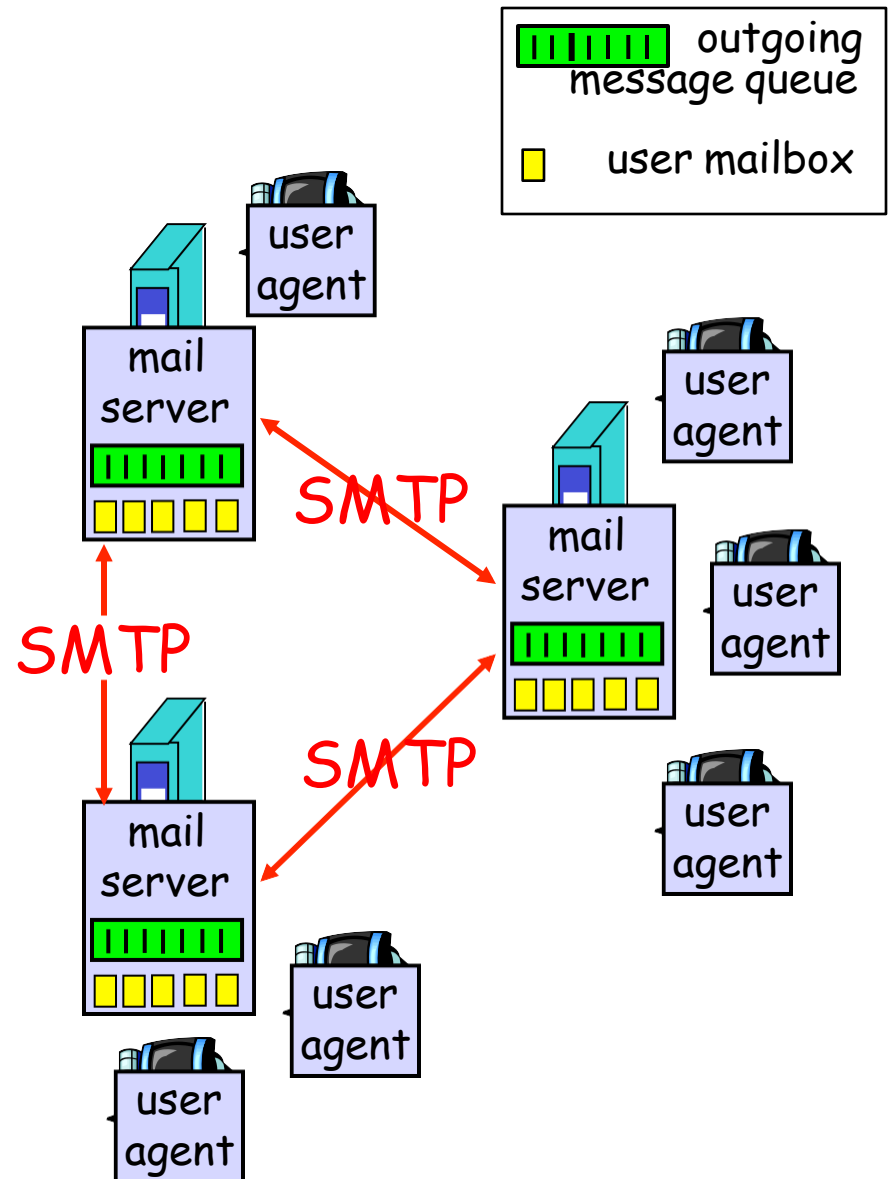
SMTP

SMTP

SMTP

Electronic Mail: mail servers

Mail Servers

❖ **mailbox** contains incoming messages for user

❖ **message queue** of outgoing (to be sent) mail messages

❖ **SMTP protocol** between mail servers to send email messages
- client: sending mail server
- "server": receiving mail server
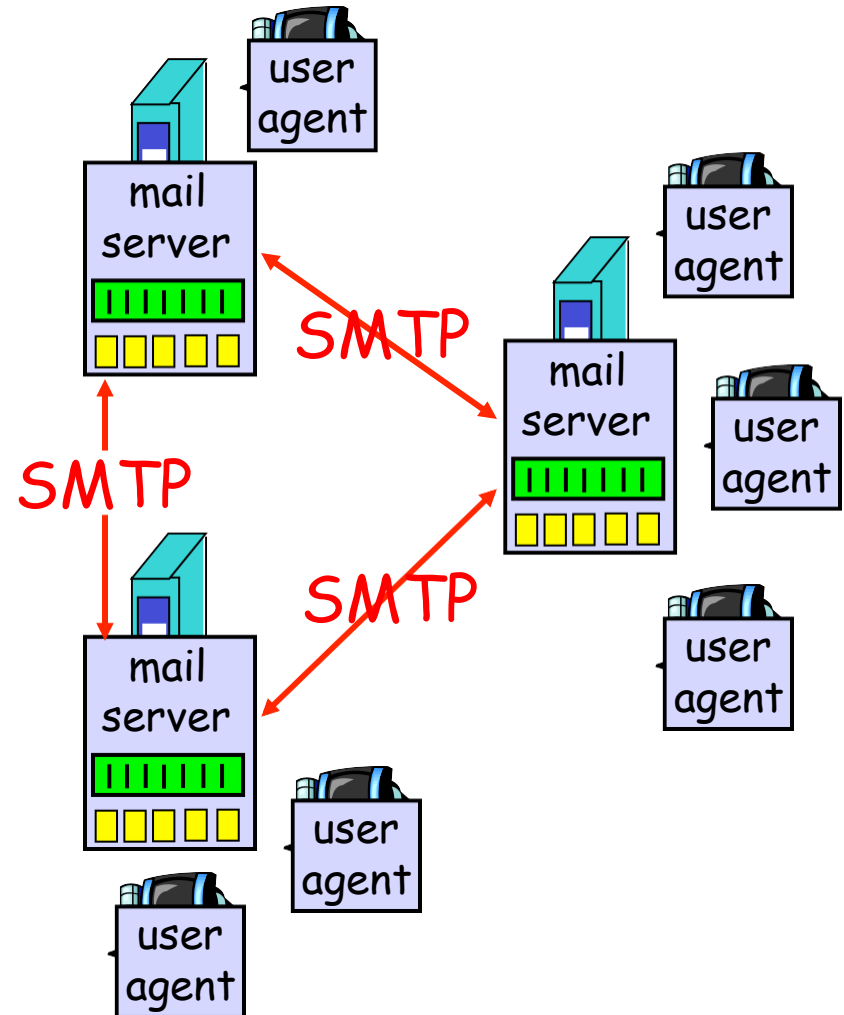
Electronic Mail: SMTP [RFC 5321]

❖ uses TCP to reliably transfer email message from client to server, port 25

❖ direct transfer: sending server to receiving server

❖ three phases of transfer
  ▪ handshaking (greeting)
  ▪ transfer of messages
  ▪ closure

❖ command/response interaction
  ▪ commands: ASCII text
  ▪ response: status code and phrase

❖ messages must be in 7-bit ASCII

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message and "to" bob@someschool.edu

2) Alice's UA sends message to her mail server; message placed in message queue

3) Client side of SMTP opens TCP connection with Bob's mail server

4) SMTP client sends Alice's message over the TCP connection

5) Bob's mail server places the message in Bob's mailbox
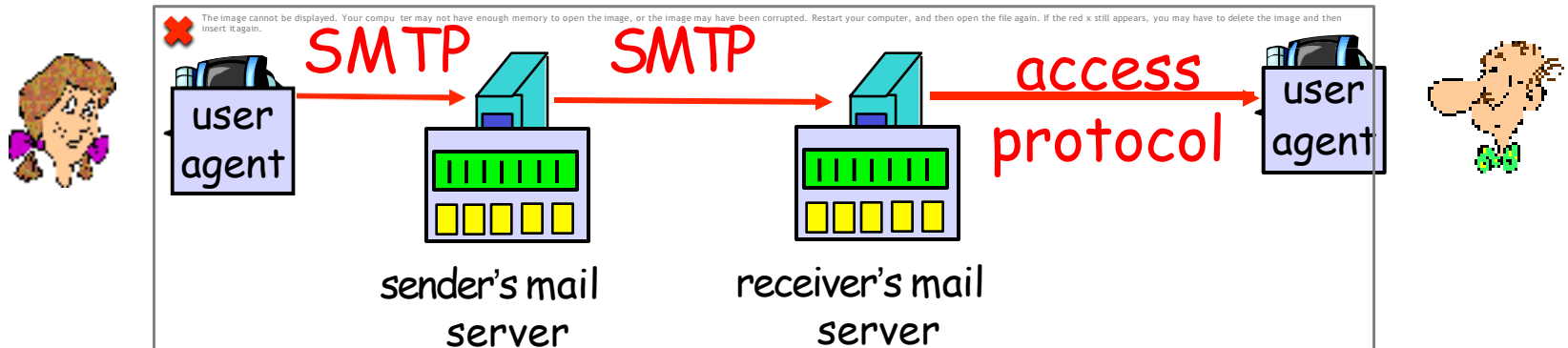
6) Bob invokes his user agent to read message
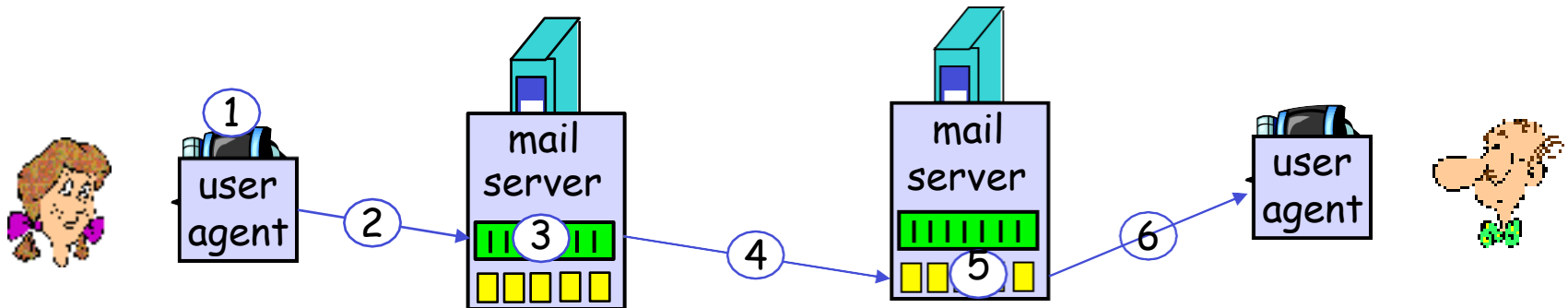
Try SMTP interaction for yourself:

❖ `telnet servername 25`
❖ see 220 reply from server
❖ enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

above lets you send email without using email client (reader)

Mail access protocols

SMTP   SMTP   access protocol

user agent   sender's mail server   receiver's mail server   user agent

❖ 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 3501]
    - more features (more complex)
    - manipulation of stored msgs on server
  - HTTP: gmail, Hotmail, Yahoo! Mail, etc.

Difference between POP & IMAP: https://help.aol.com/articles/what-is-the-difference-between-pop3-and-imap

Pure P2P architecture

❖ *no* always-on server

❖ arbitrary end systems directly communicate

❖ peers are intermittently connected and change IP addresses

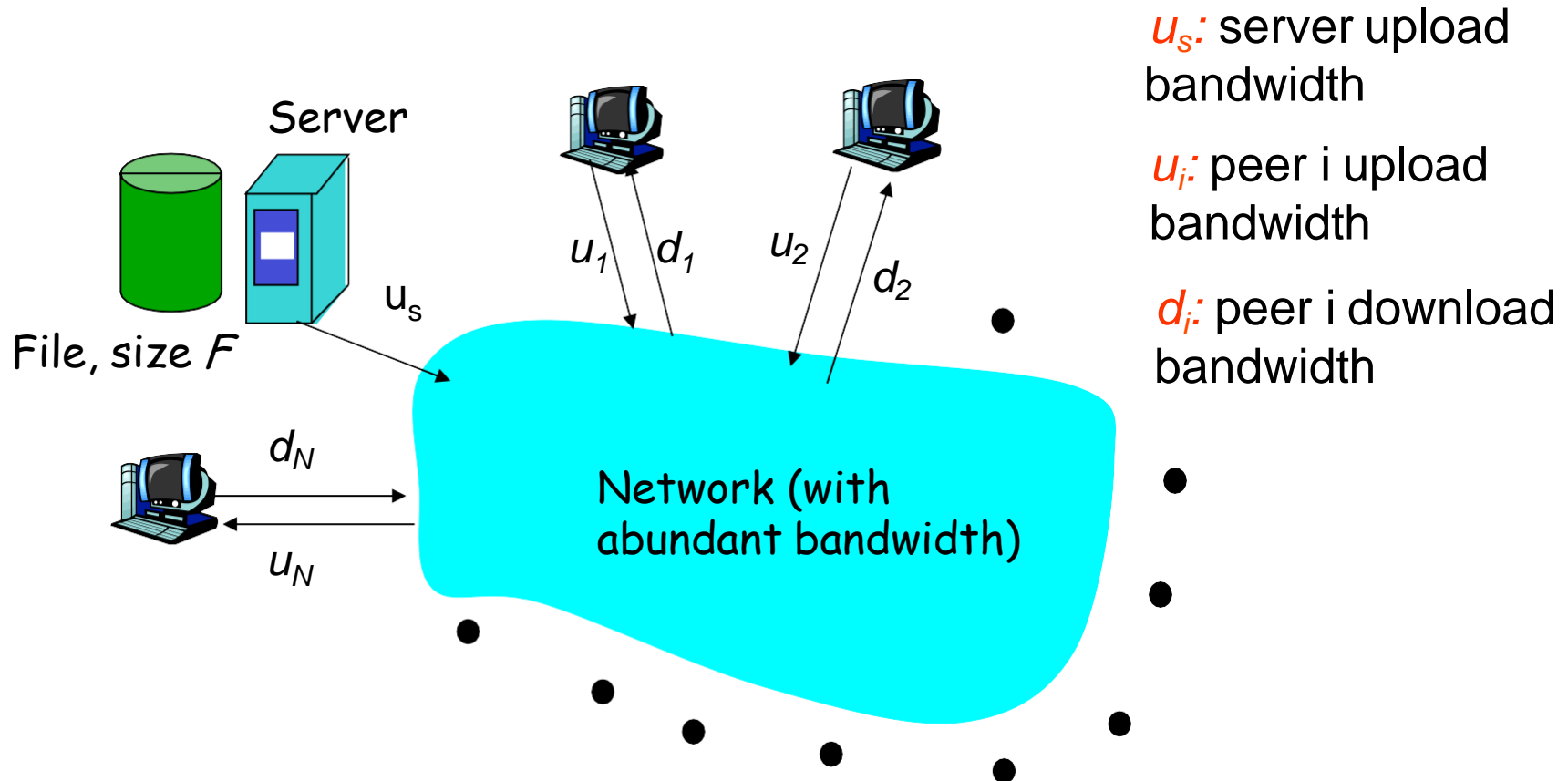Three topics:

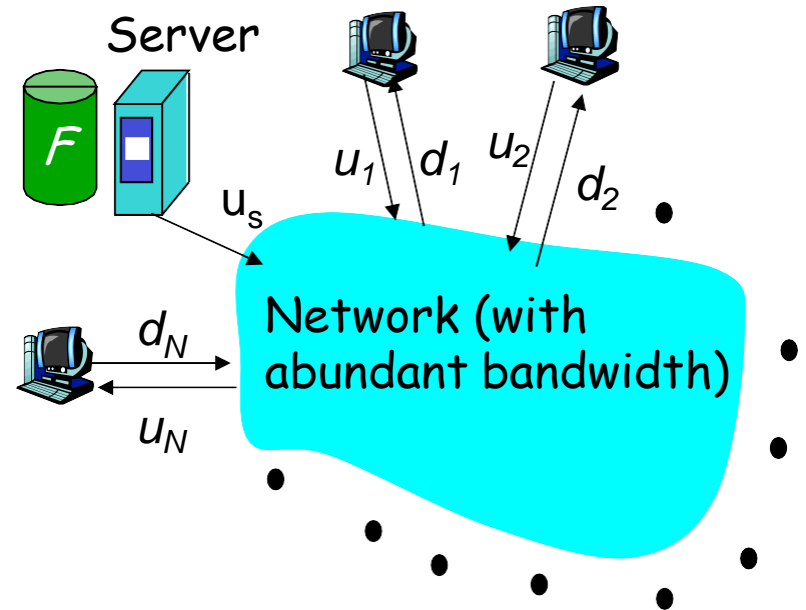- file distribution
- searching for information
- case Study: Skype

peer-peer
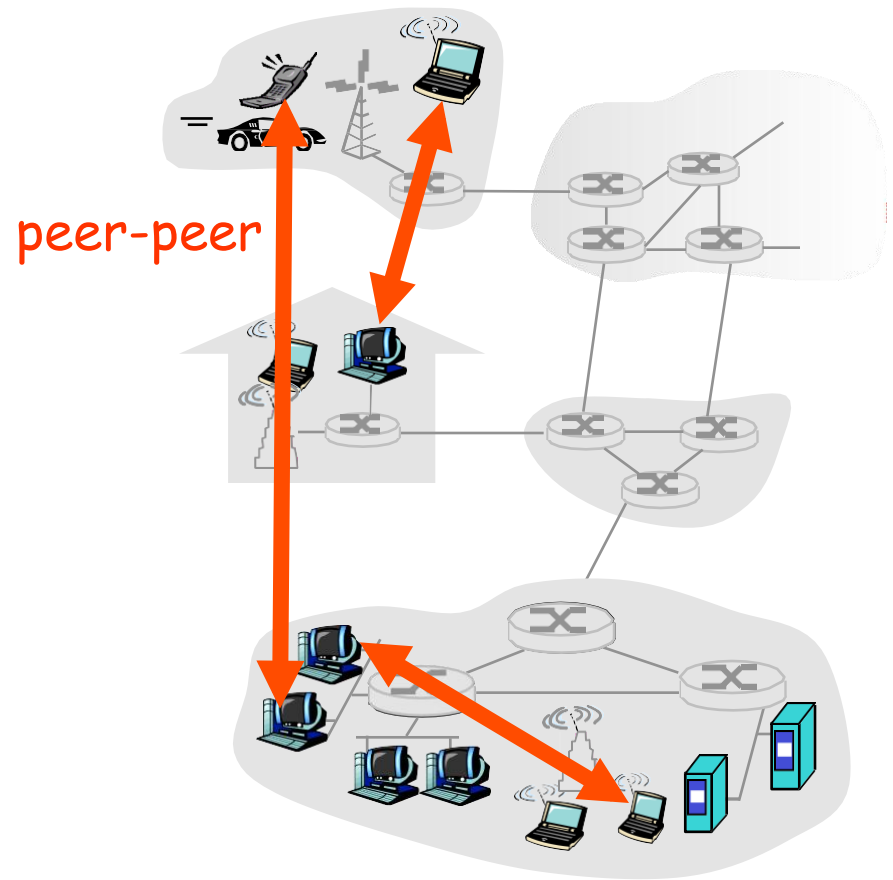
File Distribution: Server-Client vs P2P

Question : How much time to distribute file from one server to N *peers?*

u_s: server upload bandwidth

u_i: peer i upload bandwidth

d_i: peer i download bandwidth

File distribution time: server-client

❖ server sequentially sends N copies:
- NF/u_s time

❖ client i takes F/d_i time to download

Time to distribute F to N clients using client/server approach $= d_{cs} = \max\{ NF/u_s, F/\min_i(d_i)\}$

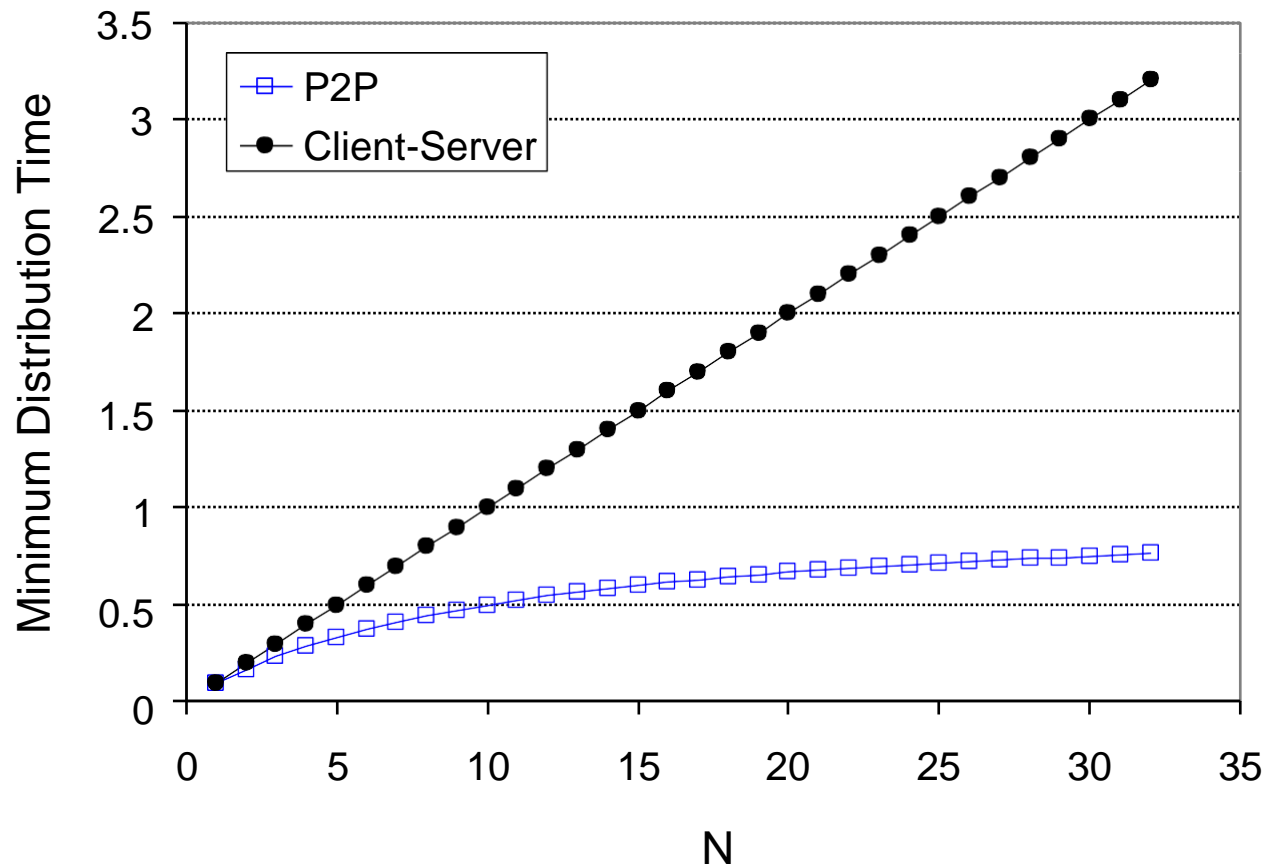increases linearly in N (for large N)

File distribution time: P2P

❖ server must send one copy: F/u_s time
❖ client i takes F/d_i time to download
❖ NF bits must be downloaded (aggregate)
  - fastest possible upload rate: $u_s + \Sigma u_i$

Server

F

u_s

u_1 d_1 u_2 d_2

d_N

u_N

Network (with abundant bandwidth)
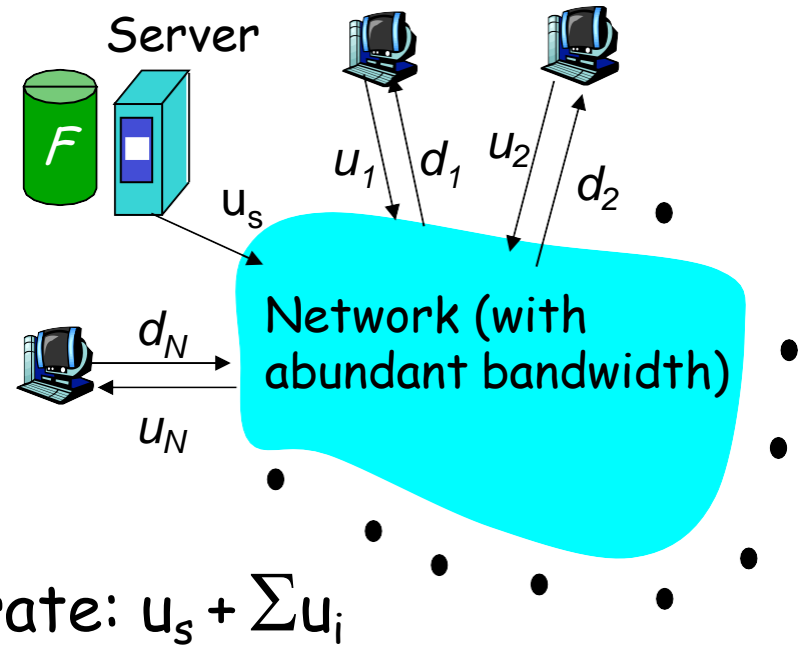
$$d_{P2P} = \max \left\{ F/u_s, F/min(d_i)_i, NF/(u_s + \Sigma u_i) \right\}$$

Server-client vs. P2P: example

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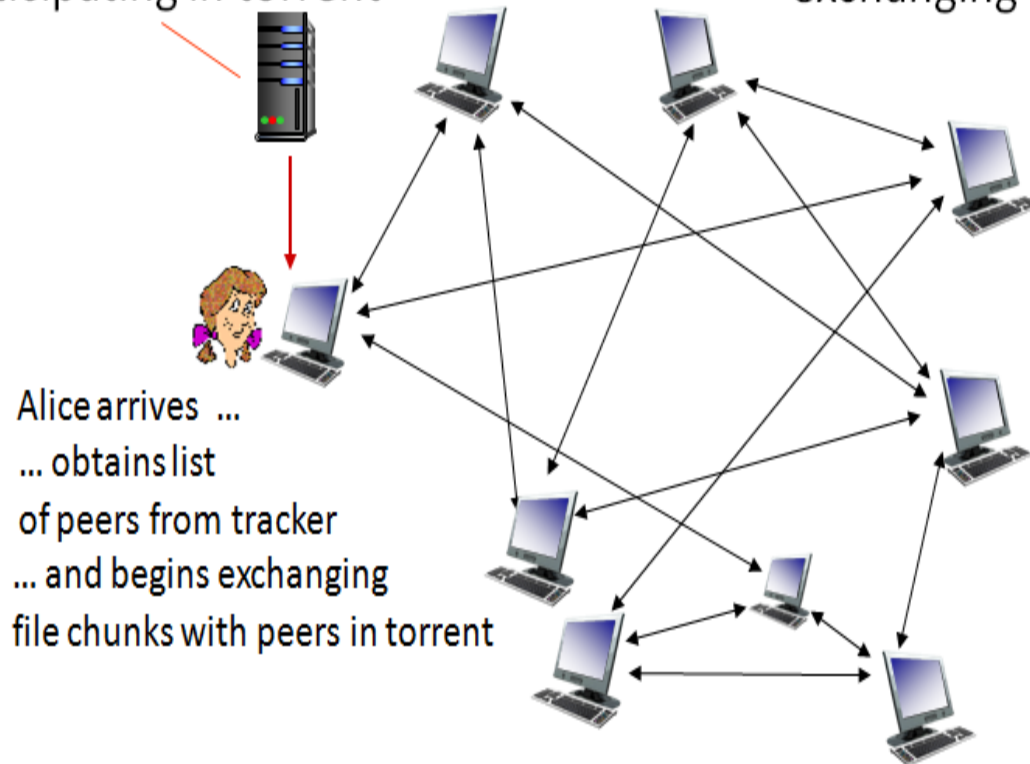
P2P file distribution: BitTorrent

- 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
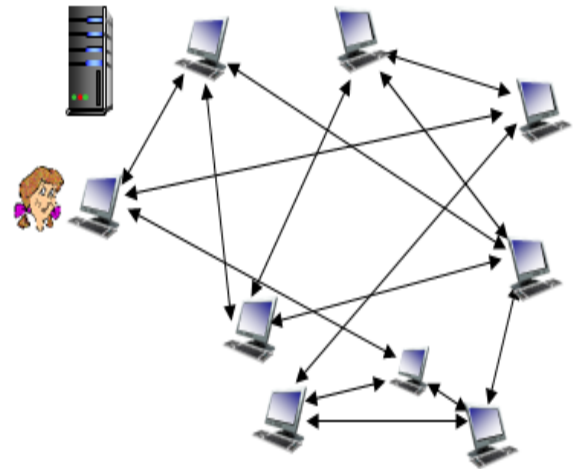
BitTorrent: requesting, sending file chunks

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 every10 secs

- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

43