COMP9121: Design of Networks and Distributed Systems

Week 9: Application layer 1

Wei Bao

School of Computer Science

THE UNIVERSITY OF SYDNEY

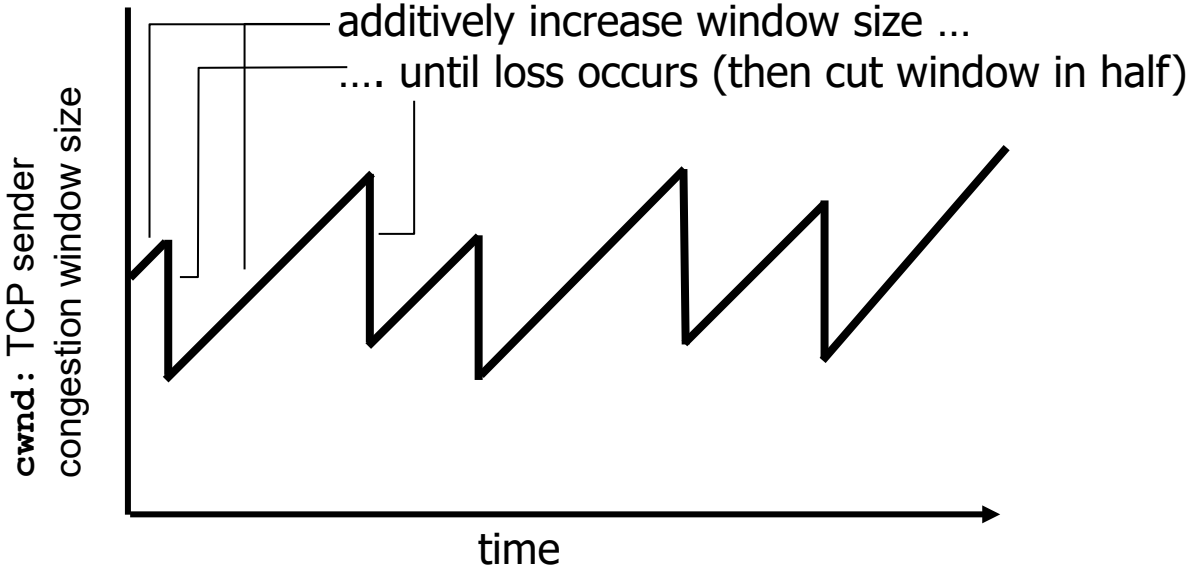
different from flow control

TCP congestion control

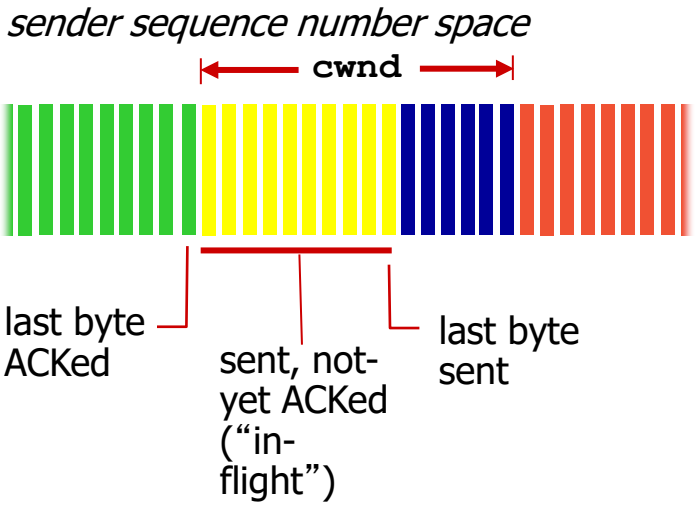
Additive increase multiplicative decrease (AIMD)

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** by 1 MSS (maximum segment size) every RTT until loss detected
  - *multiplicative decrease:* cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control: details



- sender limits transmission:

LastByteSent- LastByteAcked ≤ cwnd

- **cwnd** is dynamic, function of perceived network congestion

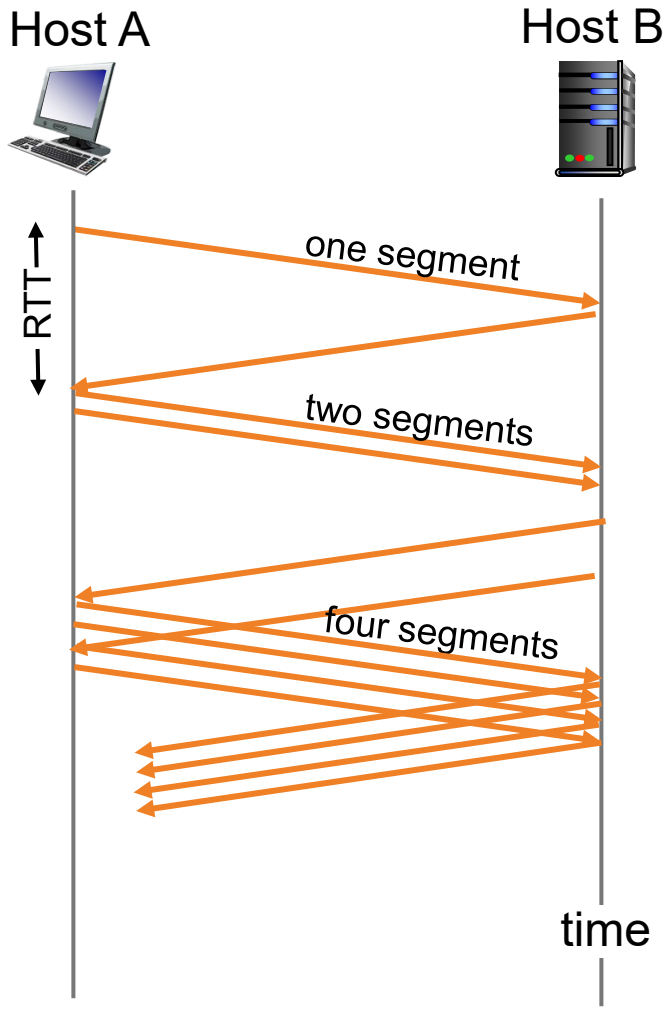
TCP sending rate:

- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP Slow Start

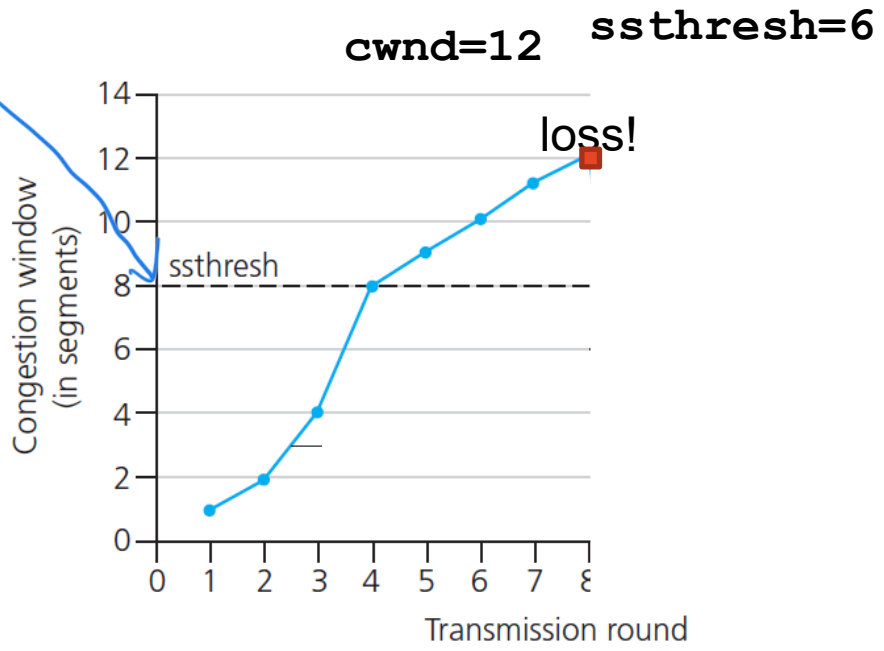
- when connection begins, increase rate exponentially:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast
- when should the exponential increase switch to linear (additive increase)?



TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: cwnd reaches **`ssthresh`**



Implementation:

- At beginning **`ssthresh`**, specified in different versions of TCP
- (In this example **`ssthresh=8 segment`**)

› on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

TCP: detecting, reacting to loss

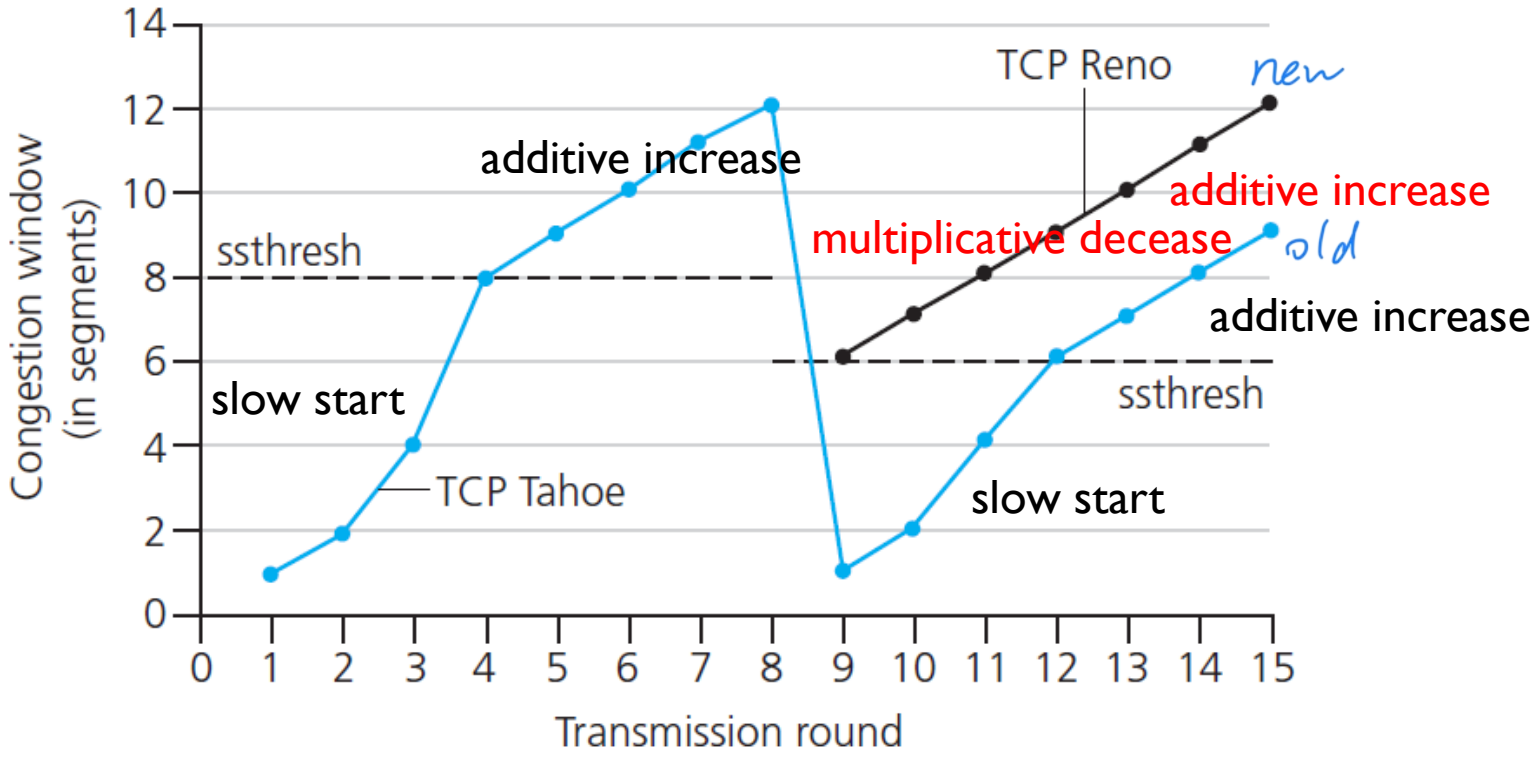
Case 1

› loss indicated by timeout:
  - **cwnd** set to 1 MSS;
  – window then grows exponentially (as in slow start) to **ssthresh**, then grows linearly

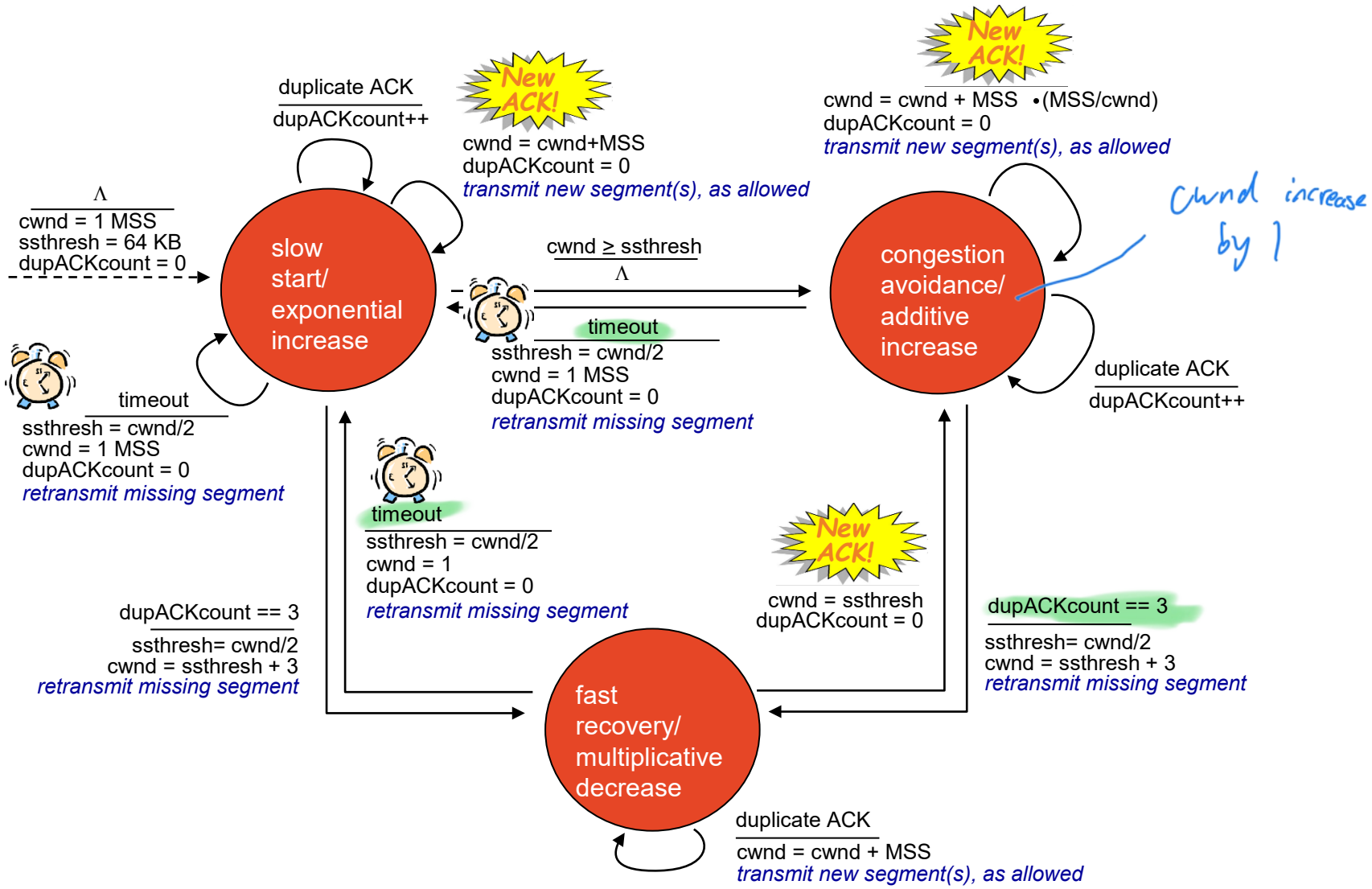
Case 2:

› loss indicated by 3 duplicate ACKs:
  › TCP Tahoe, same as loss indicated by timeout, always sets **cwnd** to 1 (timeout or 3 duplicate acks) old
  › TCP RENO new
    • **cwnd** is cut in half window then grows linearly (additive increase)
    • fast recovery

TCP: switching from slow start to CA

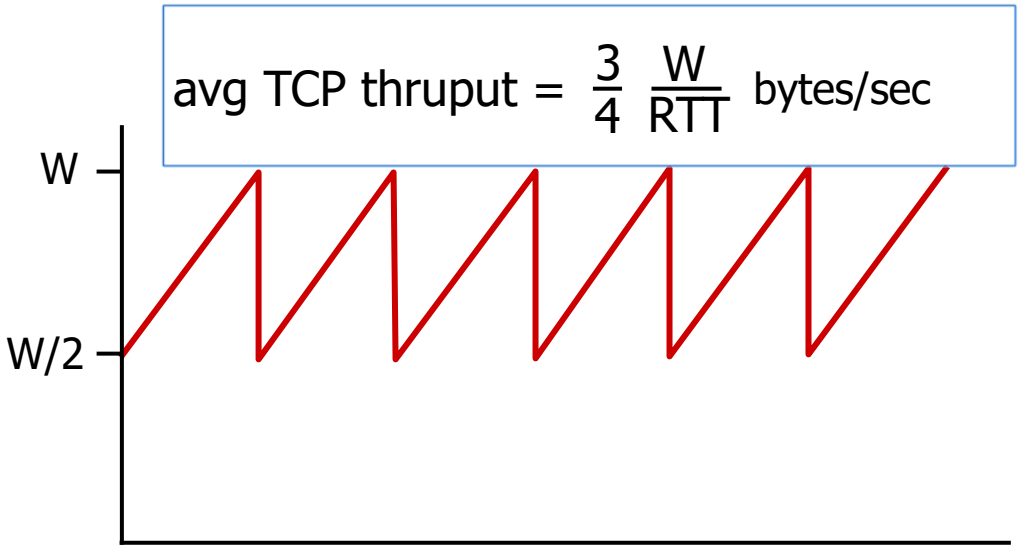


Summary: TCP Reno Congestion Control



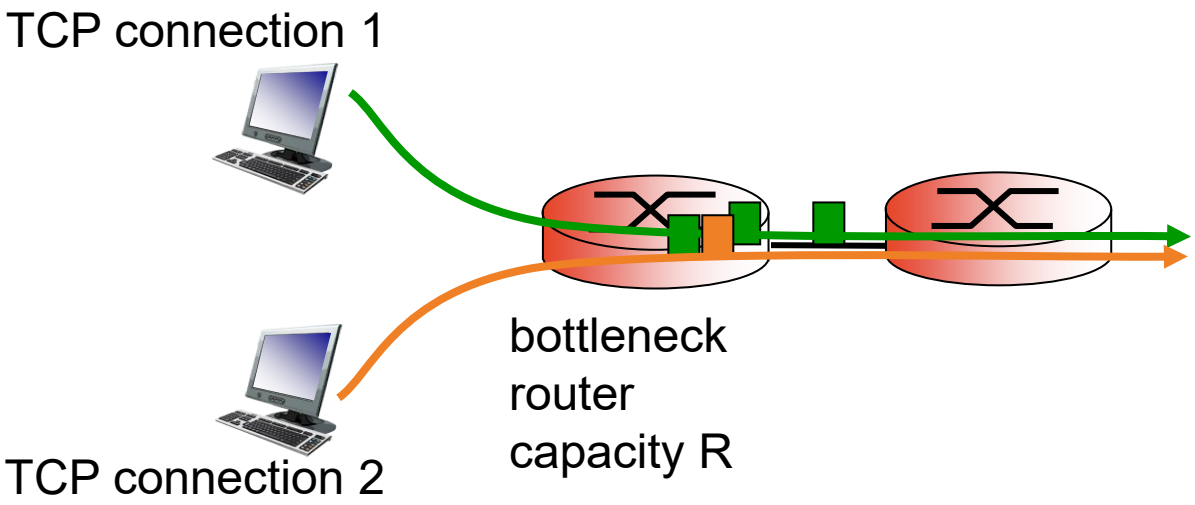
TCP Reno throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT



TCP Fairness

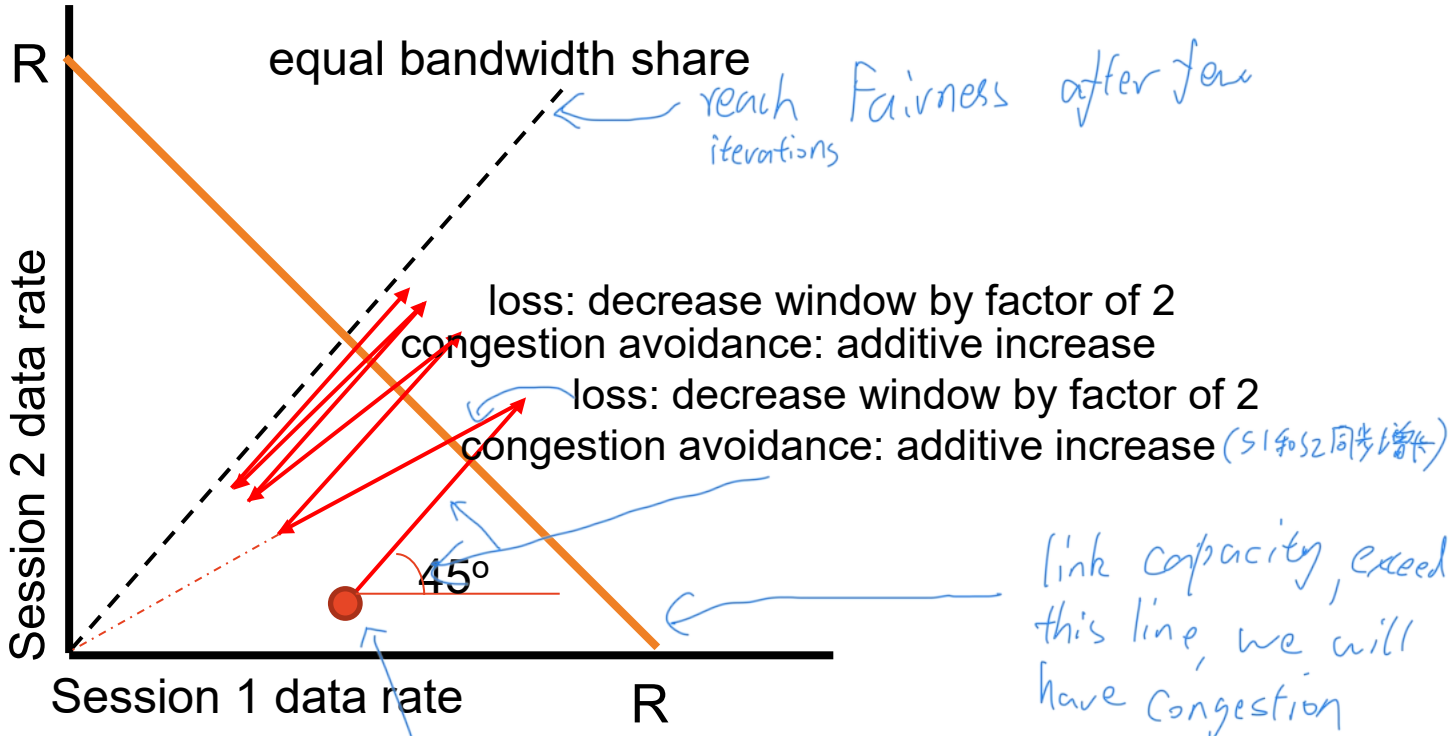
Fairness: K TCP sessions share same bottleneck link of bandwidth R, each has average rate of R/K



Why is TCP fair?

two competing sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

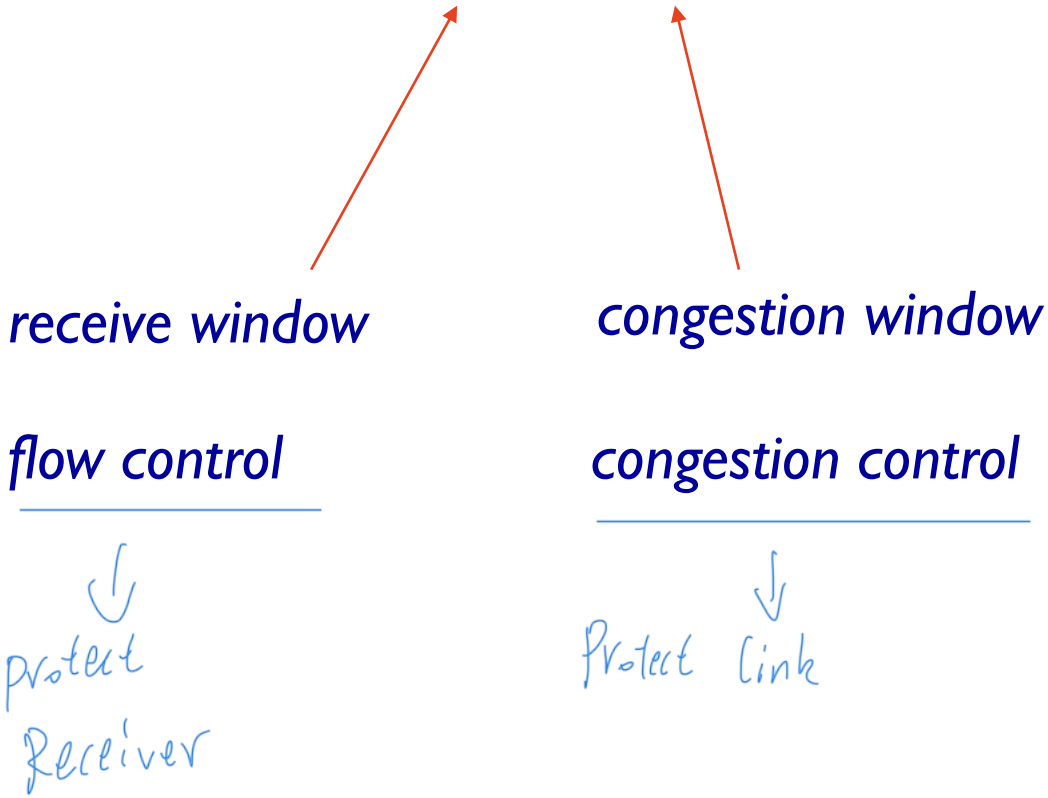
- application can open multiple parallel connections between two hosts
- e.g., link of rate R
  - App 1 asks for 1 TCP, gets 0.1R
  - App 2 asks for 9 TCPs, gets 0.9R

App2 起9个TCP 则不公平
1 开1个TCP

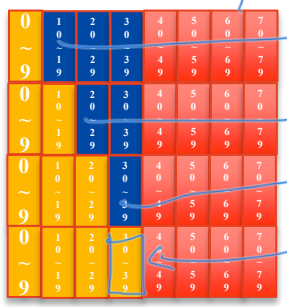
Solve: 限制 port 开TCP的数量.

Final word

Window size = min (rwnd, cwnd)



TCP in action (3 duplicate ACKs)



sender window (cwnd=40 bytes)

(cwnd=20 bytes)

sender

receiver

send seq0

send seq10

send seq20

send seq30

X *loss*

receive 0~9, deliver 0~9, send ack10

receive 10~19, deliver 10~19, send ack20

receive 30~39, buffer, (re)send ack20

rcv ack10, send seq40

rcv ack20, send seq50

receive 40~49, buffer, (re)send ack20

receive 50~59, buffer, (re)send ack20

duplicate ACK++

duplicate ACK++

duplicate ACK++

(re)send seq20

receive 20~29, deliver 20~59 send ack60

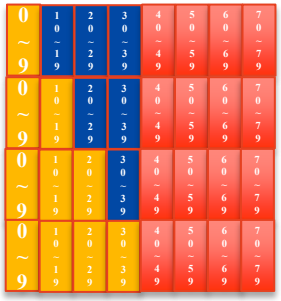
rcv ack60

send following bytes

TCP in action (timeout)

4 segment

sender window (cwnd=40 bytes)



sender

send seq0
send seq10
send seq20
send seq30

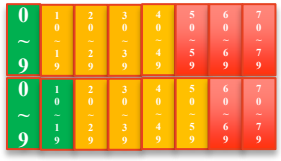
receiver

receive 0~9, deliver 0~9, send ack10

X *loss*

receive 10~19, deliver 10~19, send ack20

receive 30~39, buffer, (re)send ack20



rcv ack10, send seq40
rcv ack20, send seq50

receive 40~49, buffer, (re)send ack20

receive 50~59, buffer, (re)send ack20

duplicate ACK++

timeout

(re)send seq20

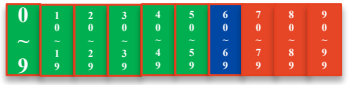
1 segment bcs Timeout

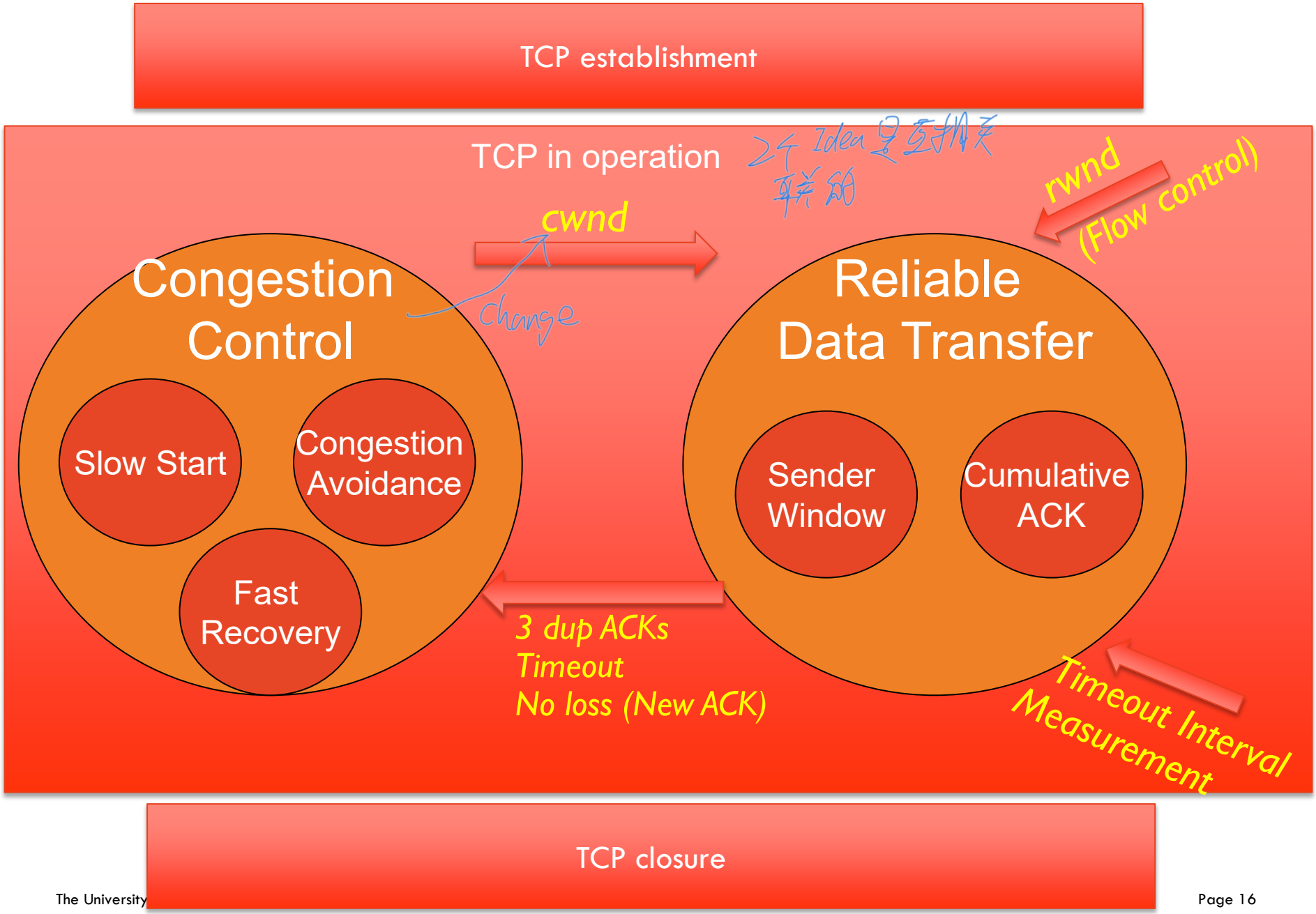
(cwnd=10 bytes)

rcv ack60

send following bytes

receive 20~29, deliver 20~59 send ack60




TCP establishment
TCP in operation
cwnd
rwnd
(Flow control)
Congestion Control
Slow Start
Congestion Avoidance
Fast Recovery
Reliable Data Transfer
Sender Window
Cumulative ACK
3 dup ACKs
Timeout
No loss (New ACK)
Timeout Interval Measurement
TCP closure

The Application Layer

Outline

- The Application layer
- Web and HTTP
- FTP
- Email

Some network applications

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

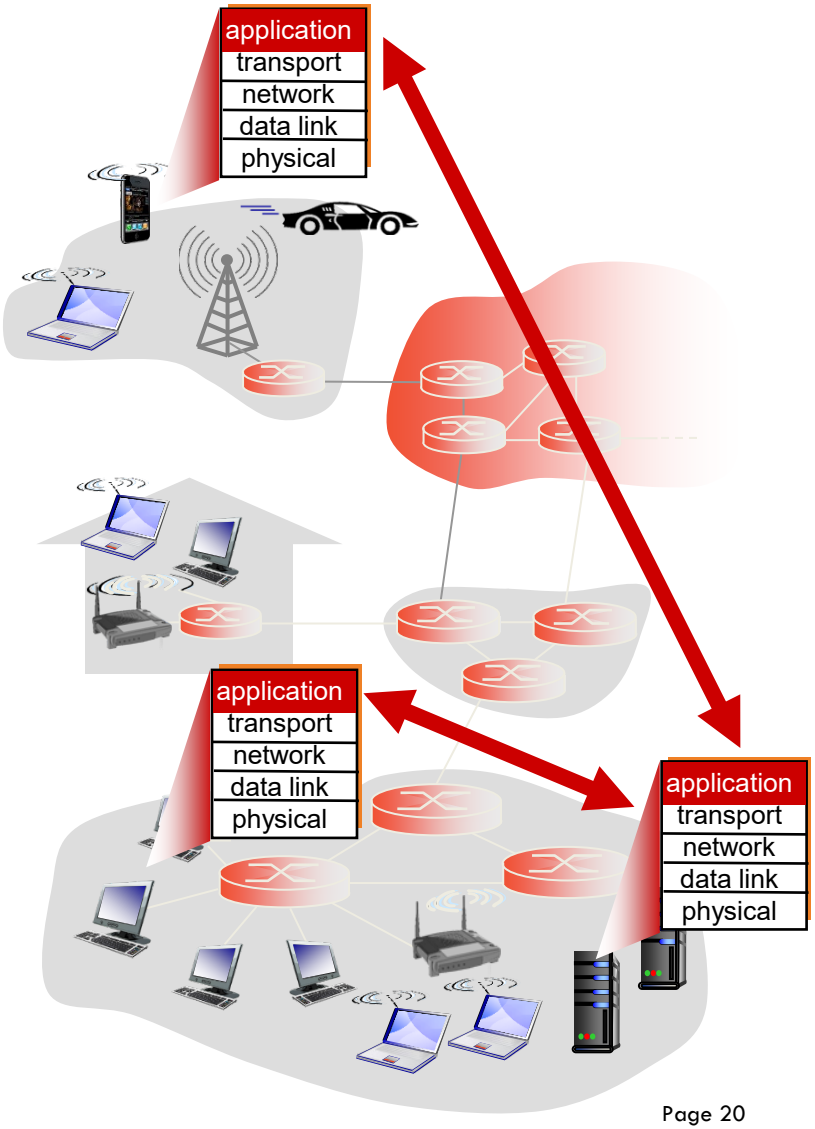
Creating a network app

write programs that:

- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

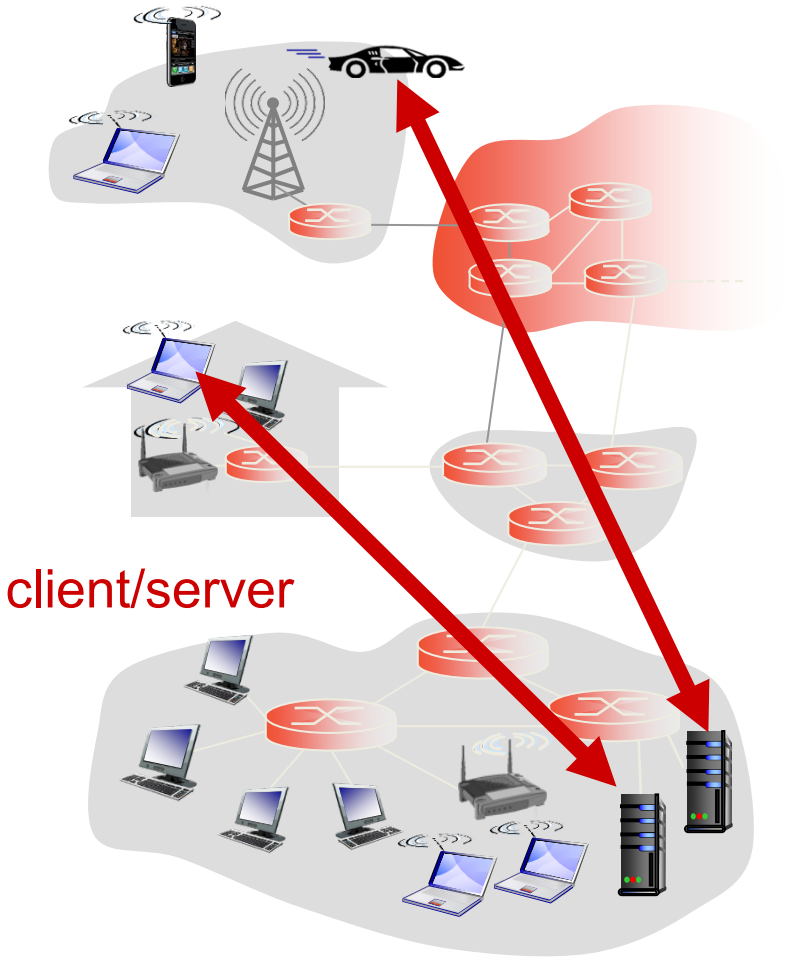


Application architectures

Possible structure of applications

- Client-server
- Peer-to-peer (P2P)

Client-server architecture



server:

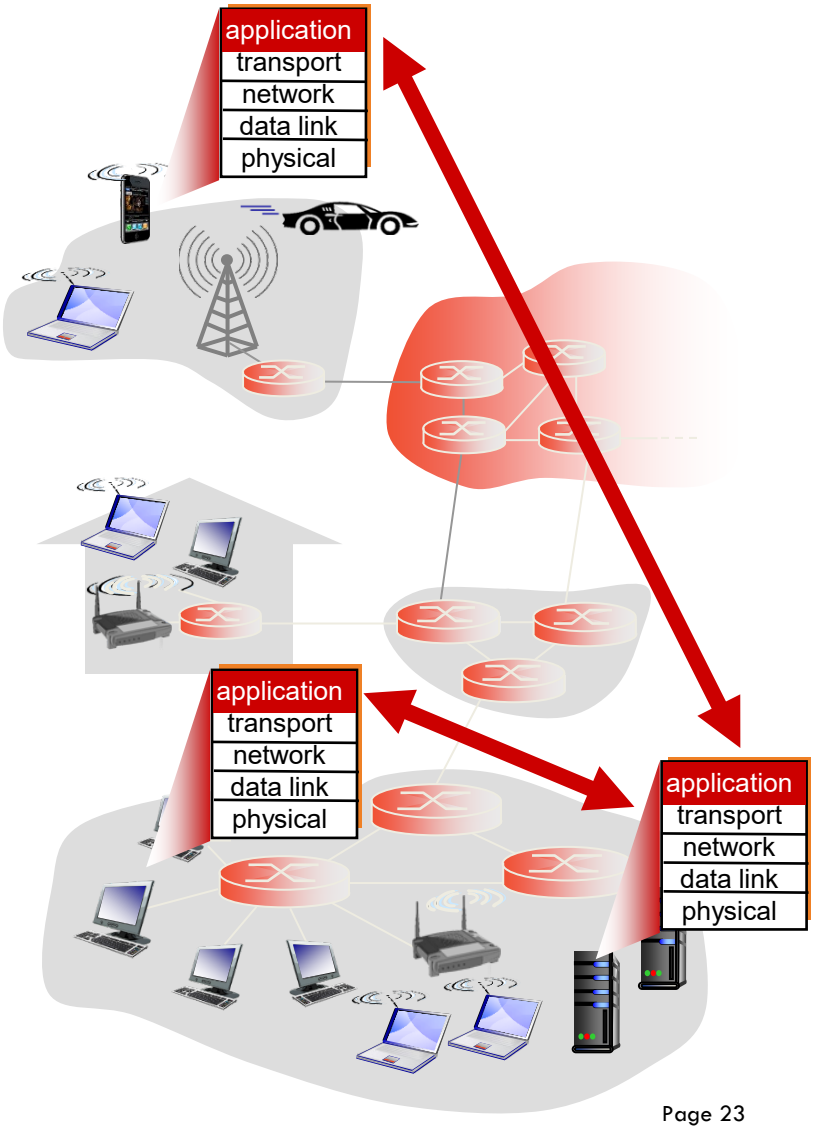
- always-on
- permanent IP address
- data centers for scaling

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 request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management



Process communicating

process: program running within a host

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

clients, servers

client process: process that initiates communication

server process: process that waits to be contacted

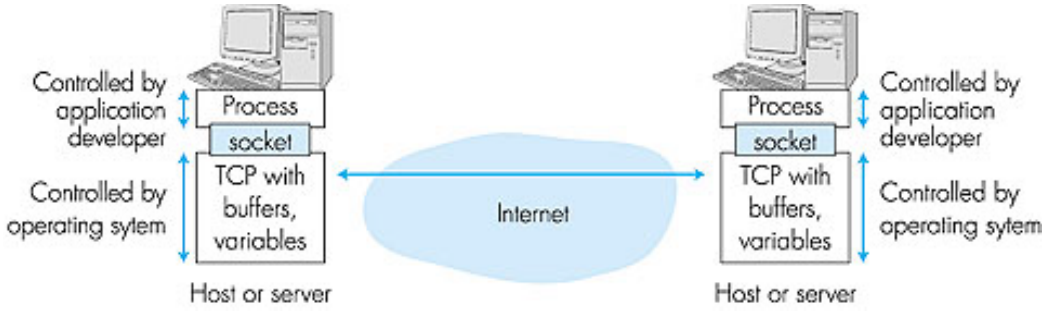
- aside: applications with P2P architectures have client processes & server processes

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 (or 128 in IPv6)
- *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 gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
  - e.g. First line: method. Second line: URL
  - message semantics
  - meaning of information in fields
  - e.g. 404 means "not found"
- rules for when and how processes send & respond to messages

open protocols:

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

proprietary protocols:

- e.g., Skype

What transport service does an app need?

data integrity

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

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

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
- *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, or connection setup,

Internet apps: application, transport protocols

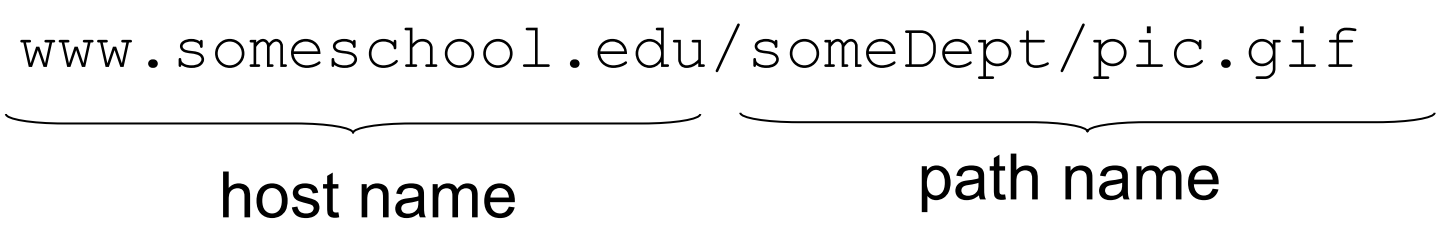
application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

Web and HTTP

Web and HTTP

First, a review…

- web page consists of *base HTML-file* which includes *several referenced objects*
  - *HTML: HyperText Markup Language*
- object can be JPEG image, Java applet, audio file,…
- each object is addressable by a *URL (Uniform Resource Locator),* e.g.,



Web and HTTP

File: usually base-html file
(HyperText Markup Language)

xxxxxxxxx
www.aaa.edu/Obj1.jpg
yyyyyyyyyyy
www.aaa.edu/Obj2.jpg
zzzzzzzzz

Browser shows

xxxxxxxxx

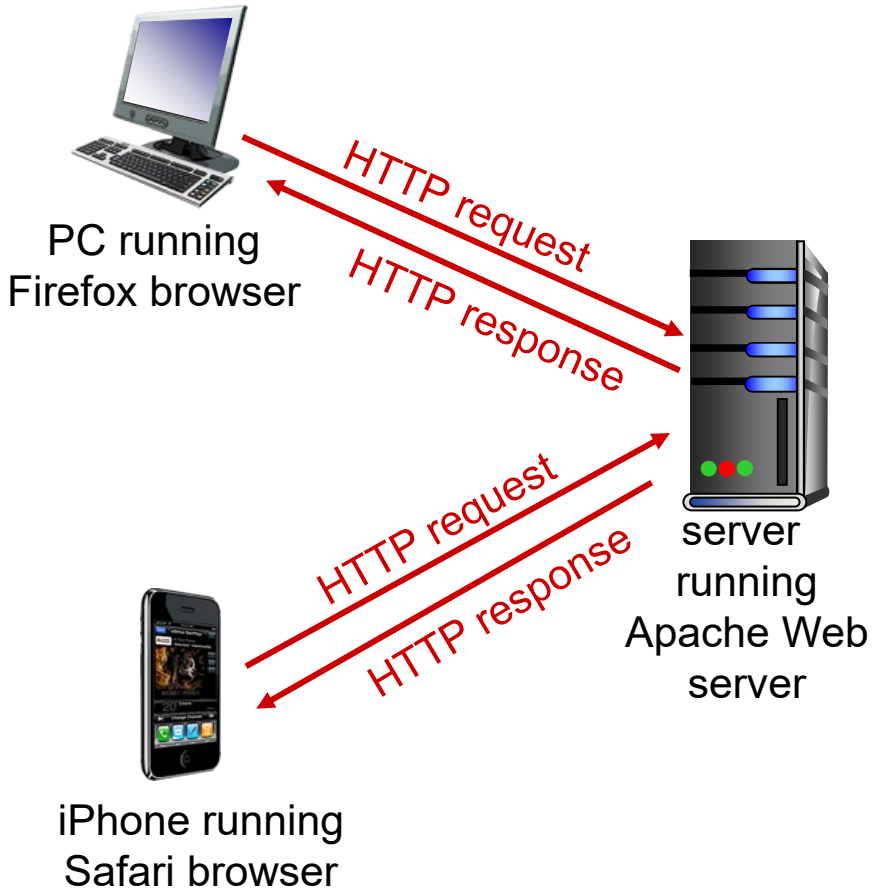
yyyyyyyyyyy

zzzzzzzzz

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



HTTP overview (cont'd)

uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
  - How to know IP address?
  - DNS (Domain Name System)
- 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

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

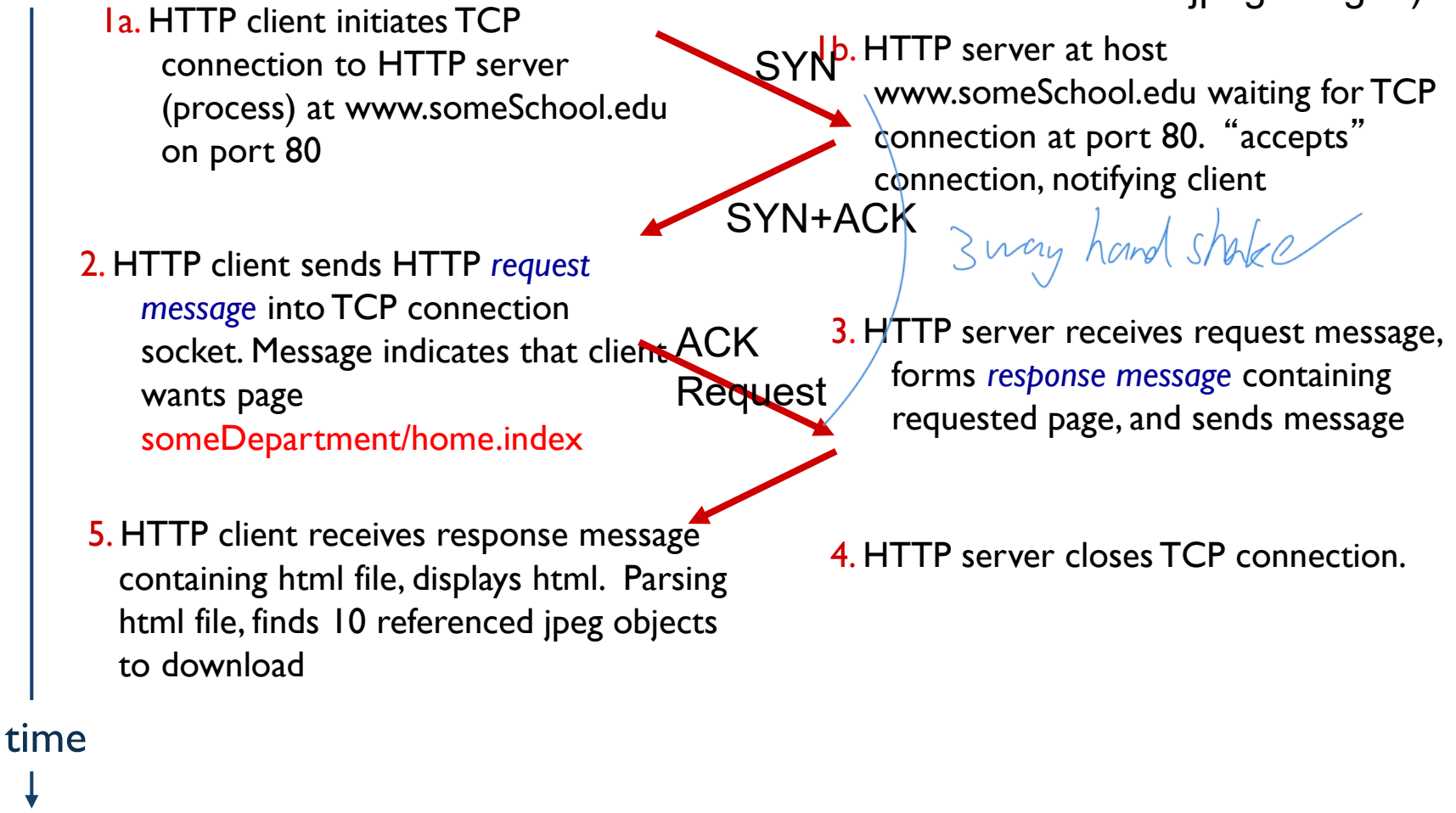
persistent HTTP

- multiple objects can be sent over single TCP connection between client, server

Non-persistent HTTP

suppose user enters URL:
`www.someSchool.edu/someDepartment/home.index`

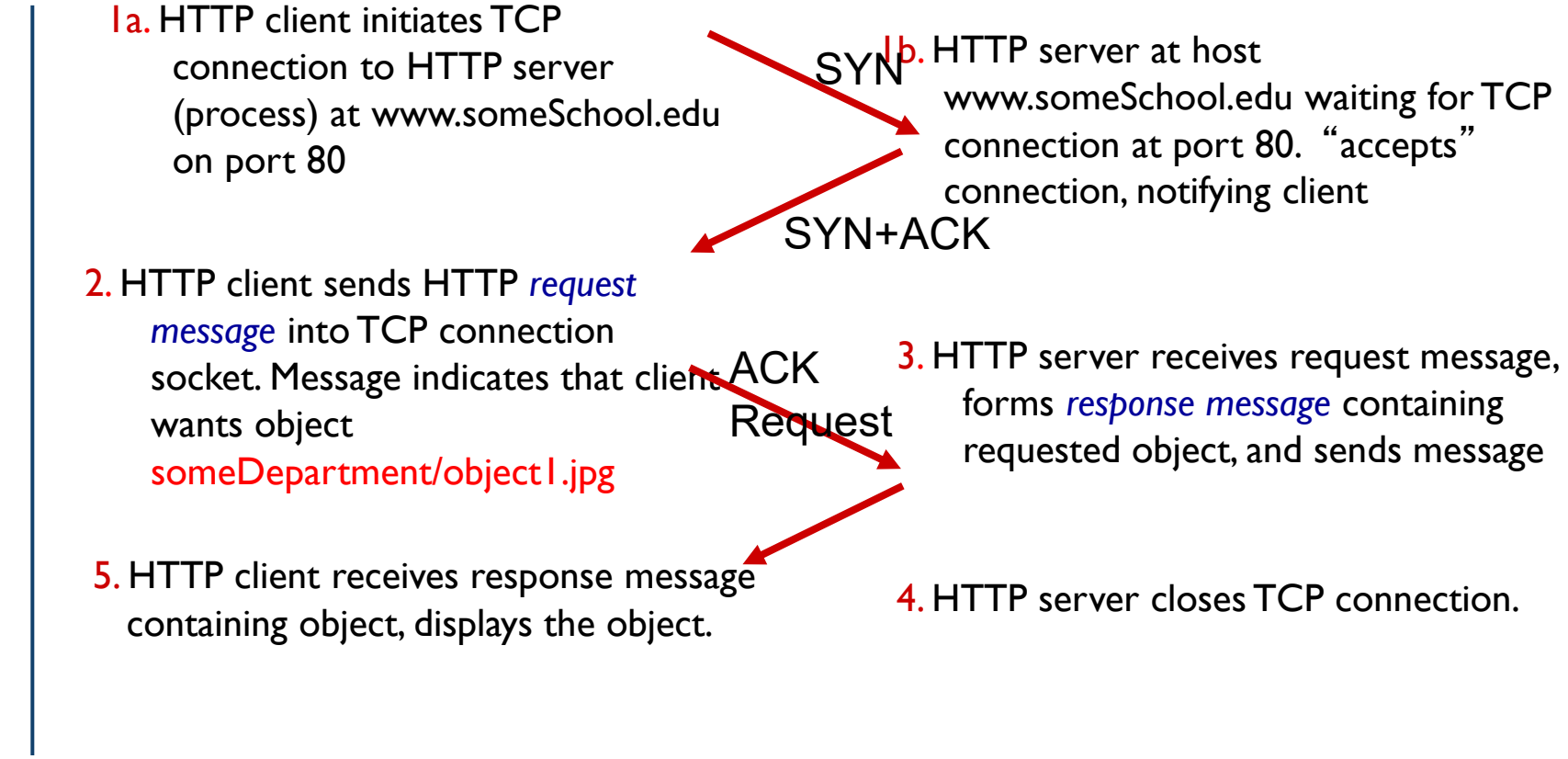
(contains text, references to 10 jpeg images)



Non-persistent HTTP

(contains text, references to 10 jpeg images, each object fits into one TCP packet)

suppose user enters URL:
`www.someSchool.edu/someDepartment/home.index`



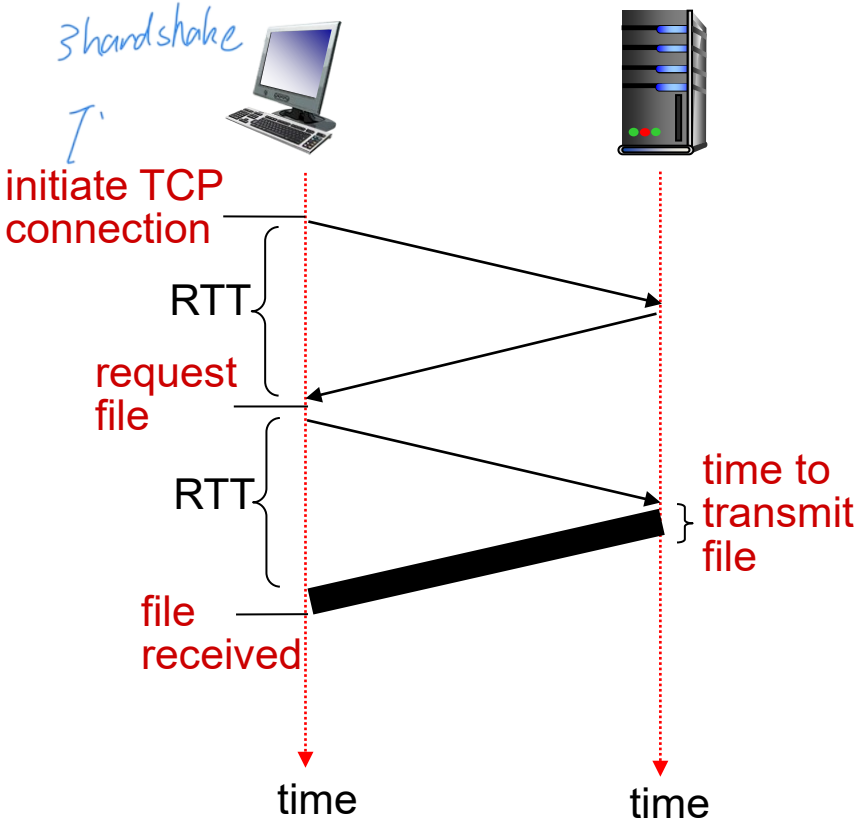
HTTP: response time

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

HTTP response time:

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2RTT+ file transmission time



Persistent HTTP

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* into TCP connection socket. Message indicates that client wants page someDepartment/home.index

3. HTTP server receives request message, forms *response message* containing requested page, and sends message

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

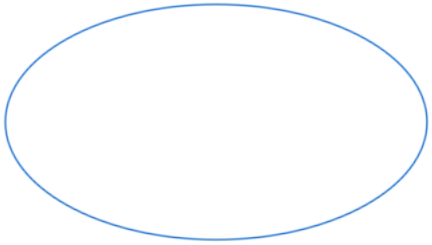
TCP is still on

time

Persistent HTTP

suppose user enters URL:
`www.someSchool.edu/someDepartment/home.index`

(contains text, references to 10 jpeg images)



2. HTTP client sends HTTP *request message* into TCP connection socket. Message indicates that client wants object someDepartment/object1.jpg

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

4. HTTP client receives response message containing object, displays the object.

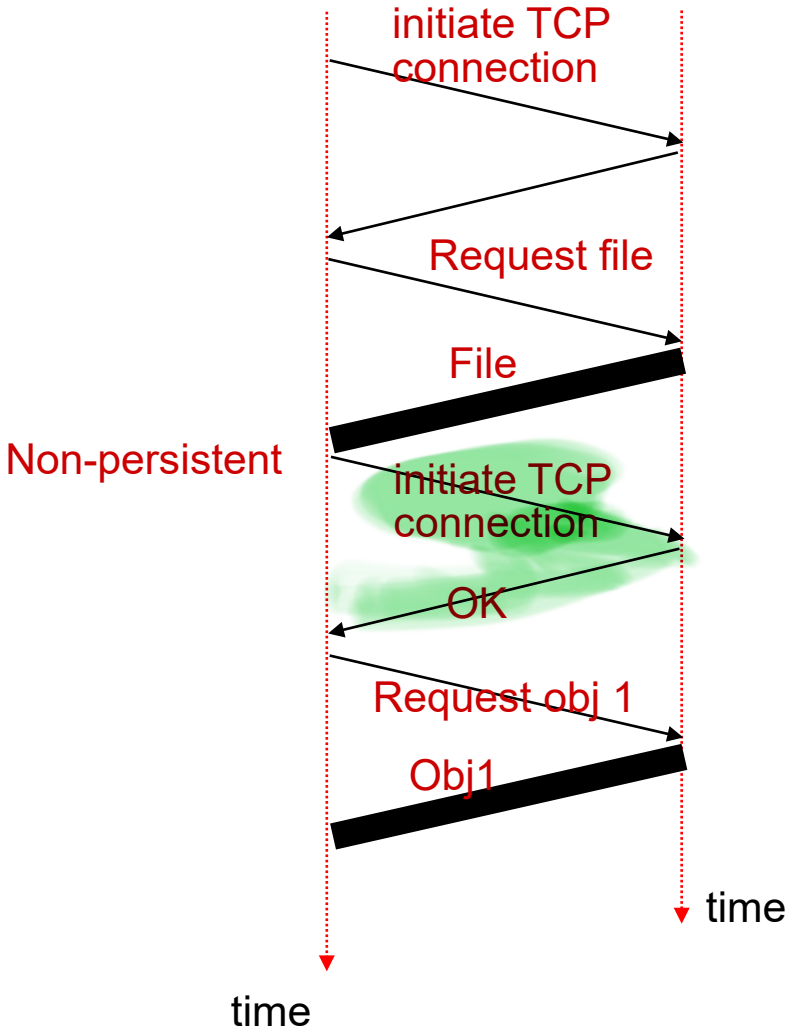
Repeated for each of 10 jpeg objects

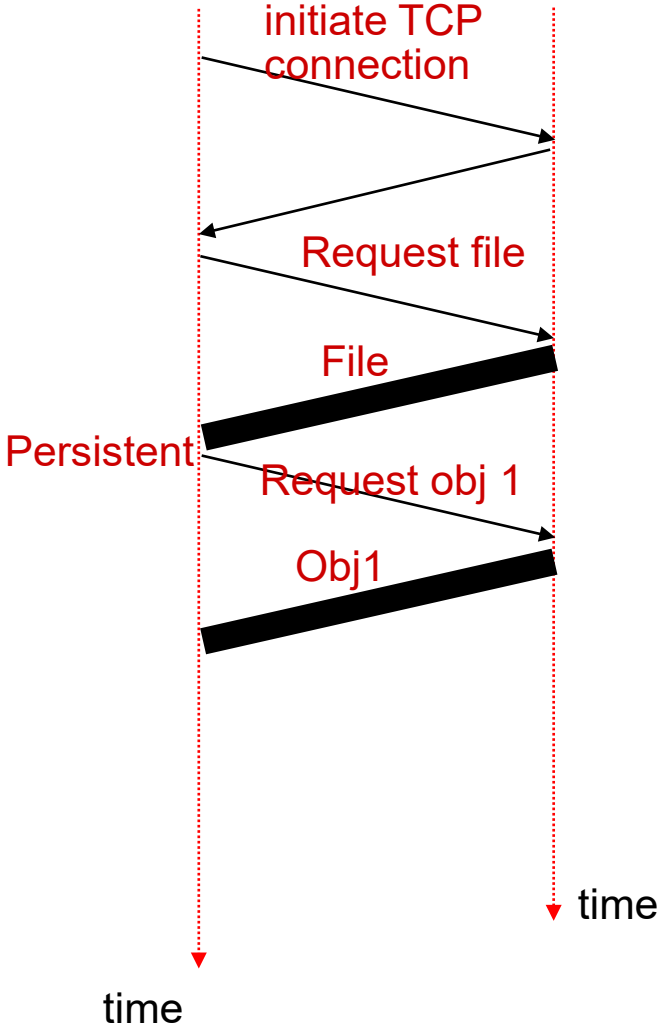
10 rounds later HTTP server closes TCP connection.

time

Non-persistent vs. persistent

Assume object is very small





Persistent HTTP

non-persistent HTTP issues:

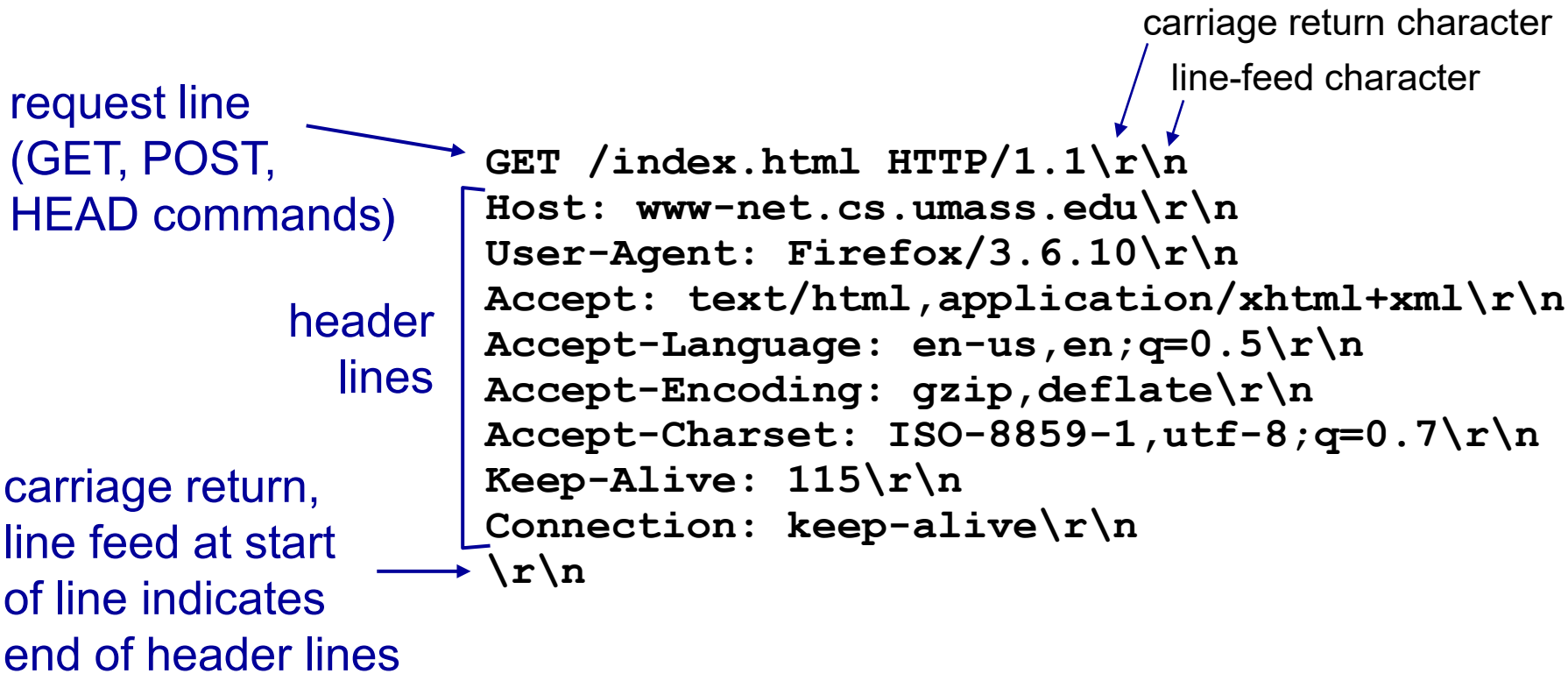
- requires 2 RTTs + file transmission time per object

persistent HTTP:

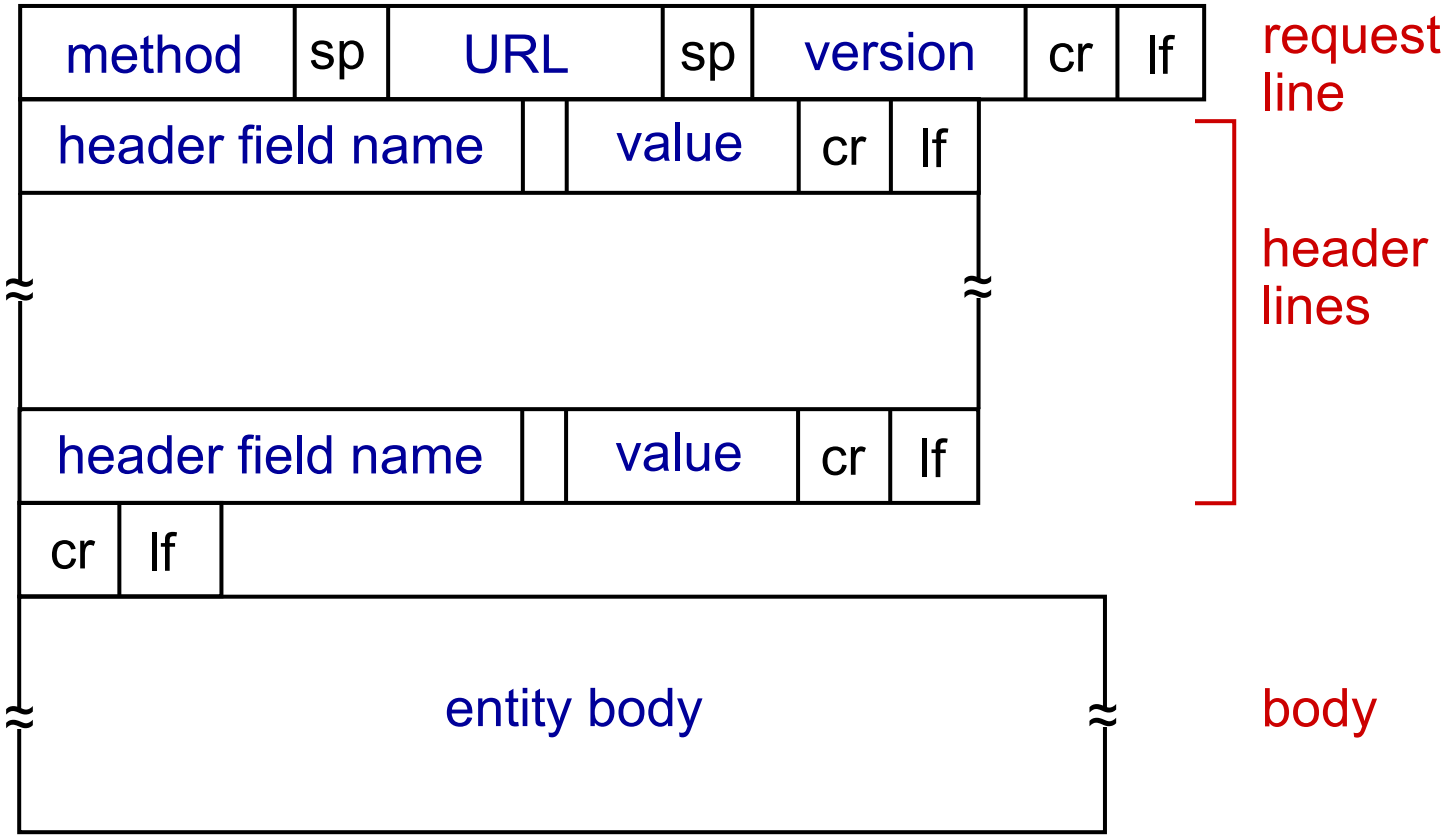
- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT + file transmission time for all the referenced objects

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

GET method tell server I want to get

POST method: tell server we want to upload

- web page often includes form input
- input is uploaded to server in entity body

For example: google search

Method types

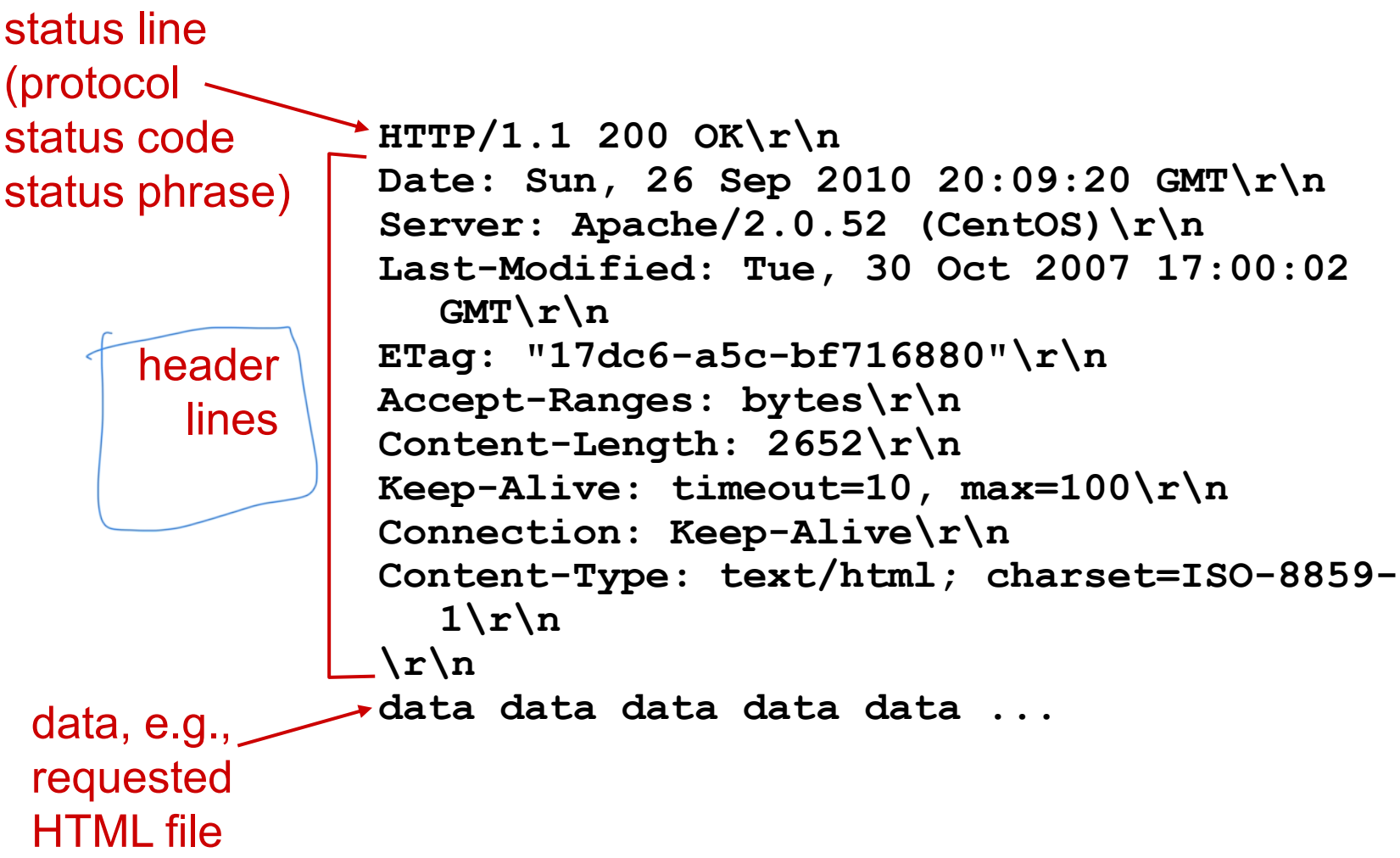
HTTP/1.0:

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

HTTP response message



HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK

- request succeeded, requested object later in this msg

301 Moved Permanently

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

400 Bad Request

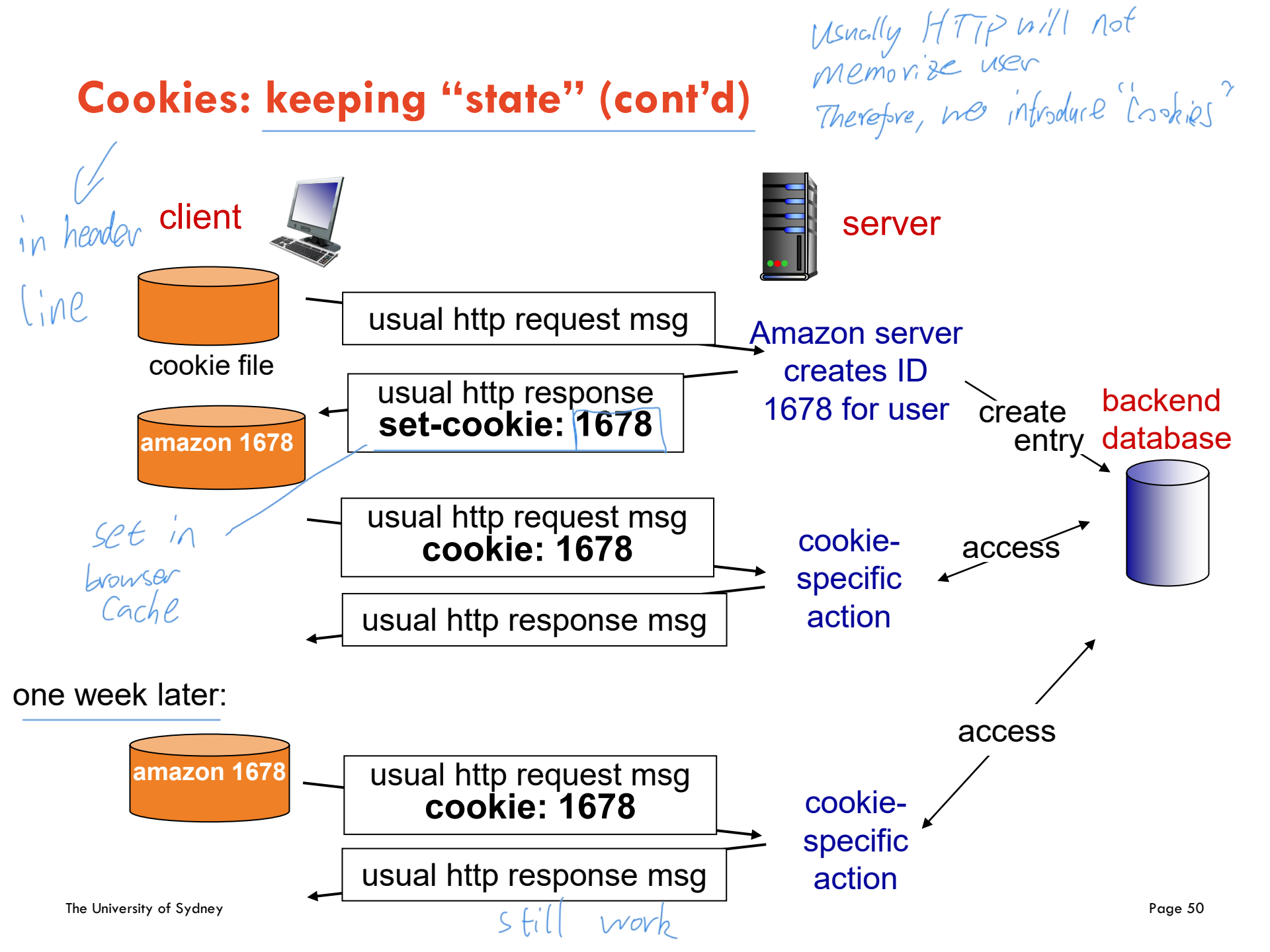
- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Cookies: keeping "state" (cont'd)



User-server state: cookies

many Web sites use cookies

four components:

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

Cookies (cont'd)

what cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

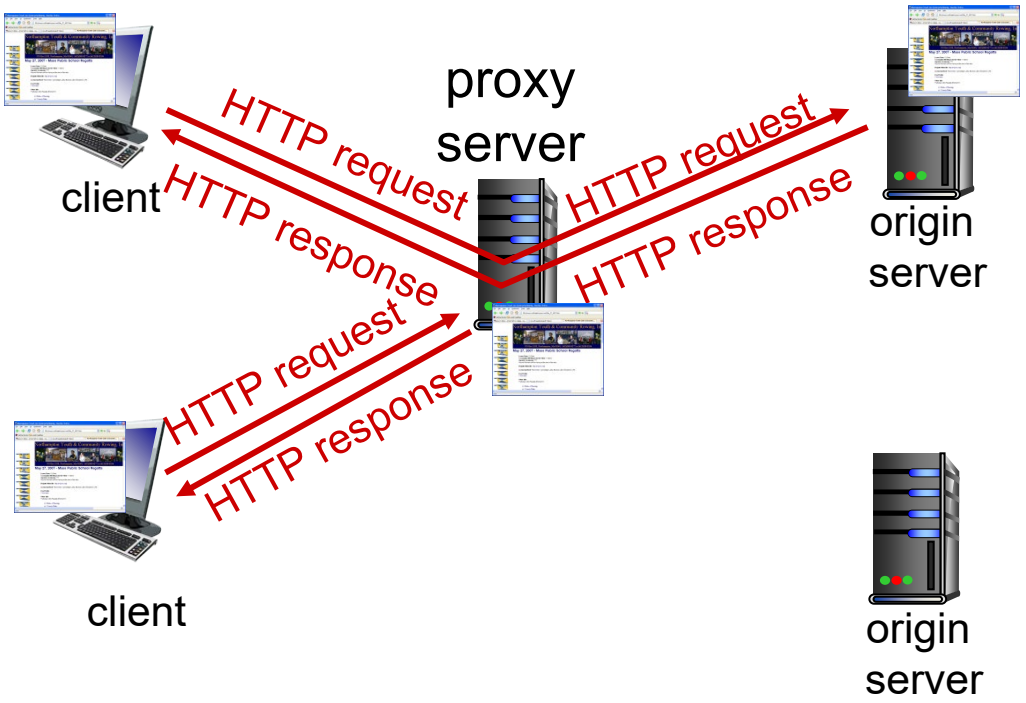
how to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Web caches (proxy server)

goal: satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
- if object in cache:
  - then cache returns object
  - else cache requests object from origin server, then returns object to client



More about Web caching

- Q: Does the cache act as a client or a server?

More about Web caching

- R: 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?

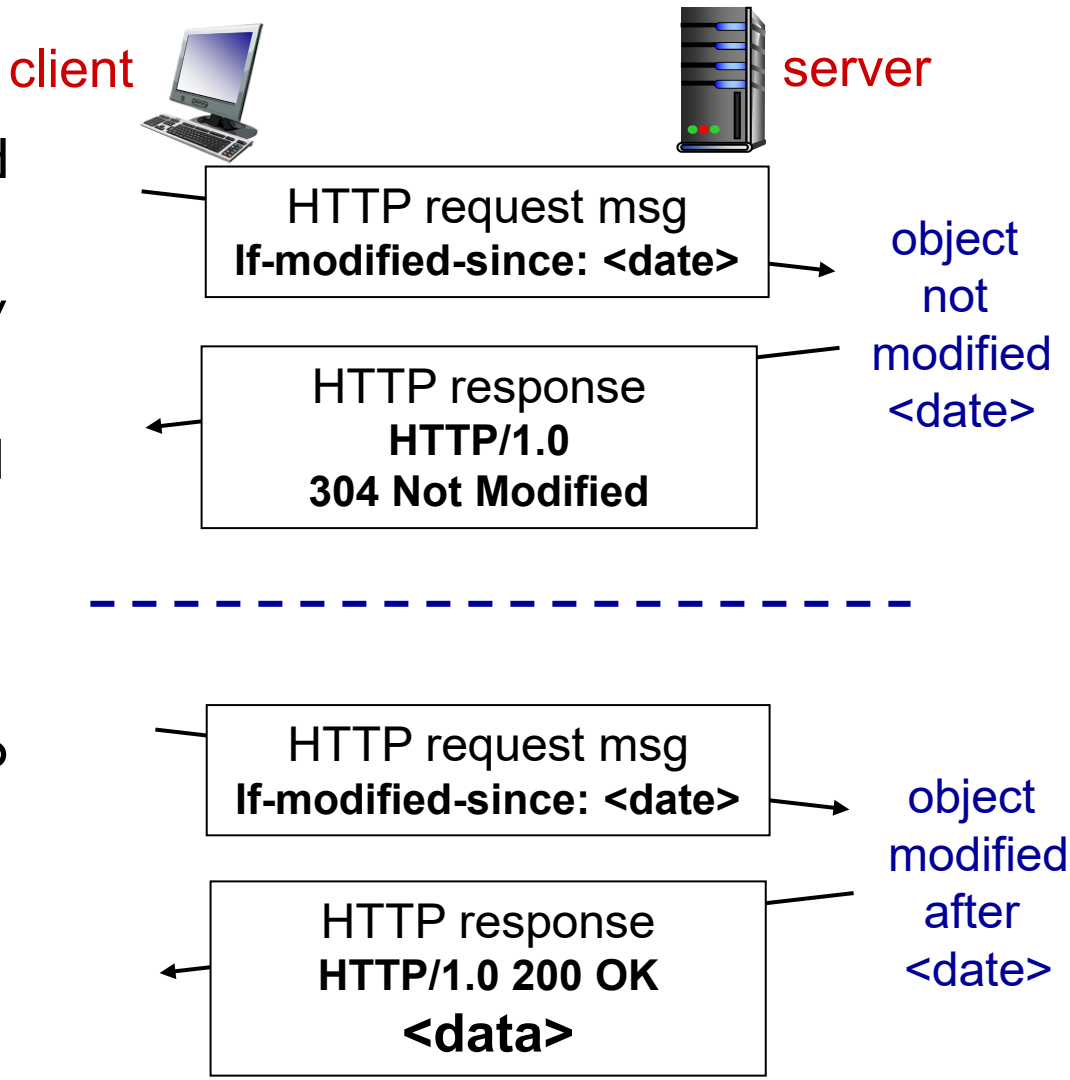
- reduce response time for client request
- reduce traffic on an institution's access link

Conditional GET

- *Goal:* don't send object if client has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- *client:* specify date of cached copy in HTTP request

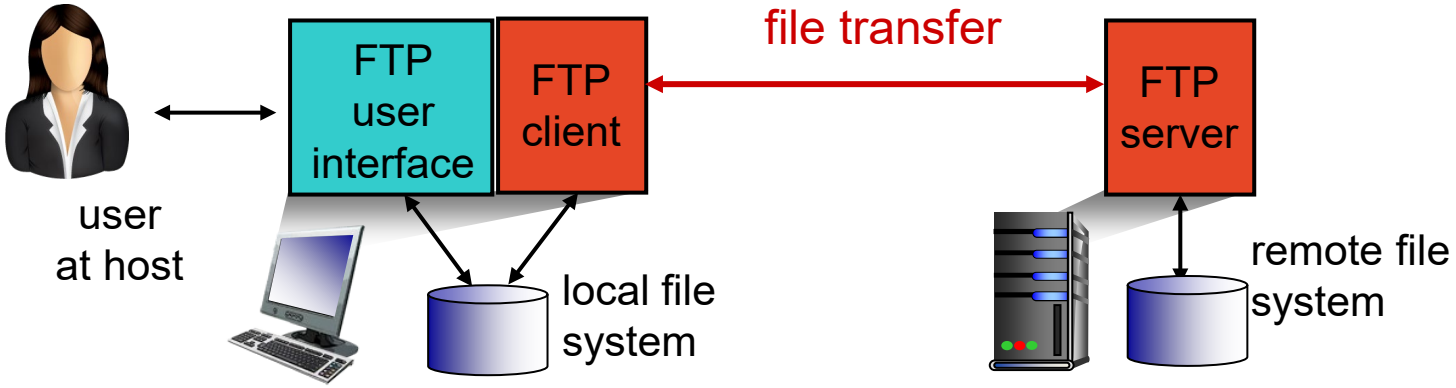
  `If-modified-since: <date>`
- *server:* response contains no object if cached copy is up-to-date:

  `HTTP/1.0 304 Not Modified`



FTP

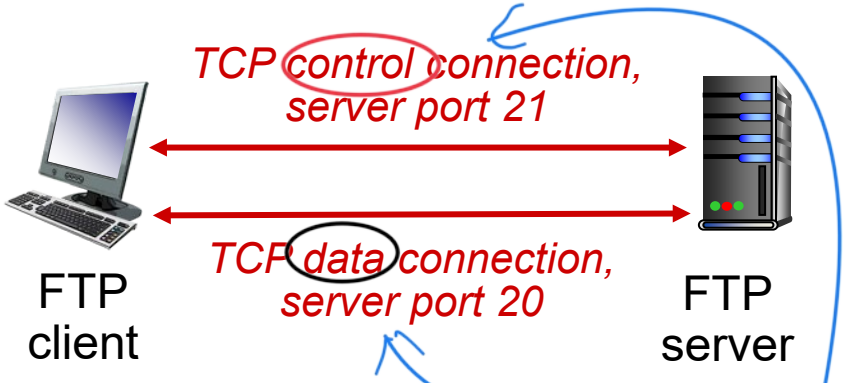
FTP: the file transfer protocol



- 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, 20

FTP: separate control, data connections

- FTP client contacts FTP server at port 21, using TCP
- client authorized over control connection
- client browses remote directory, sends commands over control connection
- when server receives file transfer command, *server* opens 2^{nd} TCP data connection (for file) *to* client
- after transferring one file, server closes data connection



› server opens another TCP data connection to transfer another file

› control connection: *"out of band"*

› FTP server maintains "state": current directory, earlier authentication

FTP commands, responses

sample commands:

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

sample return codes

- status code and phrase (as in HTTP)
- **331 Username OK, password required**
- **125 data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

Email

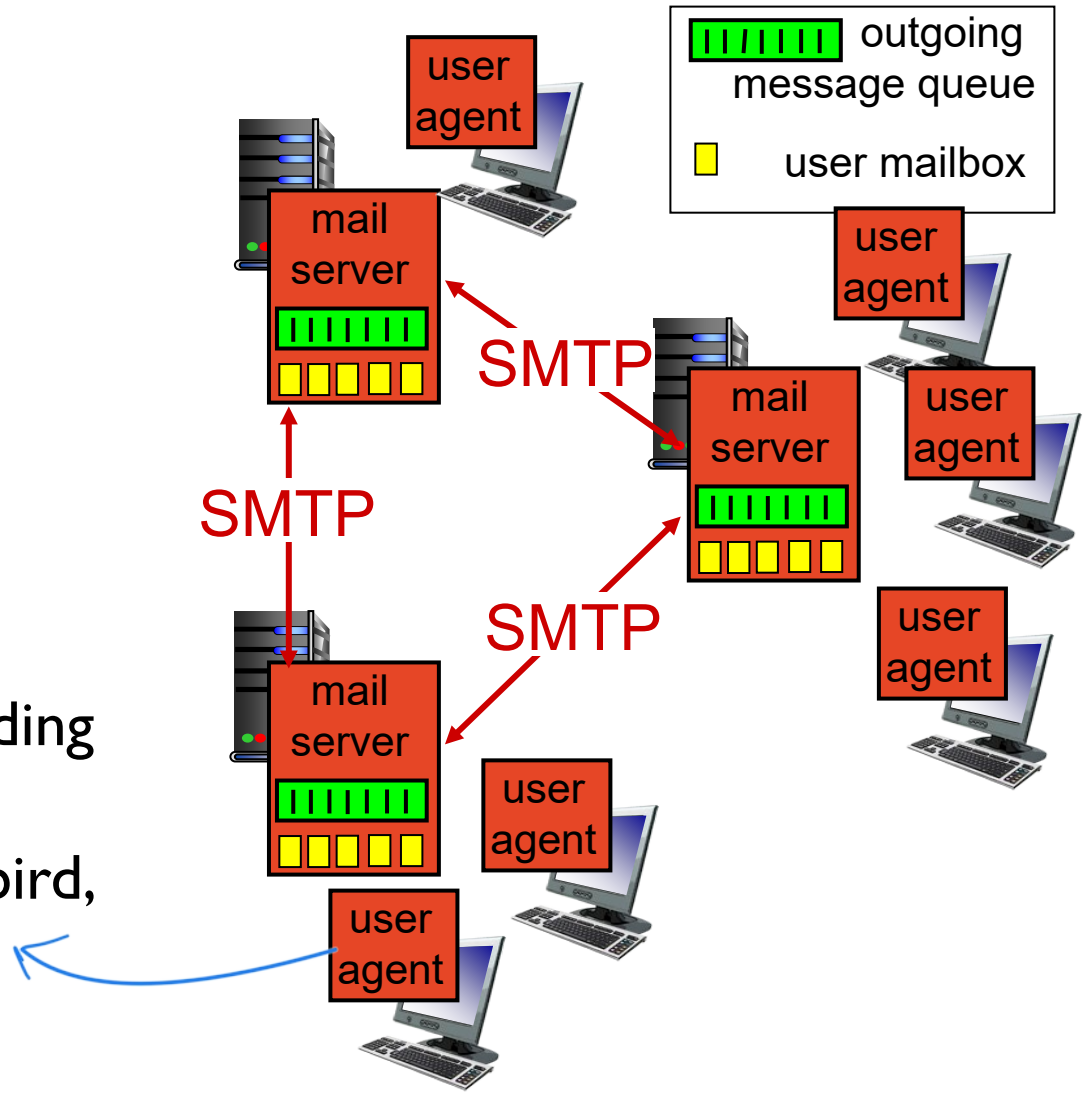
Electronic mail

Three major components:

- user agents (clients)
- mail servers
- simple mail transfer protocol: SMTP

User Agent

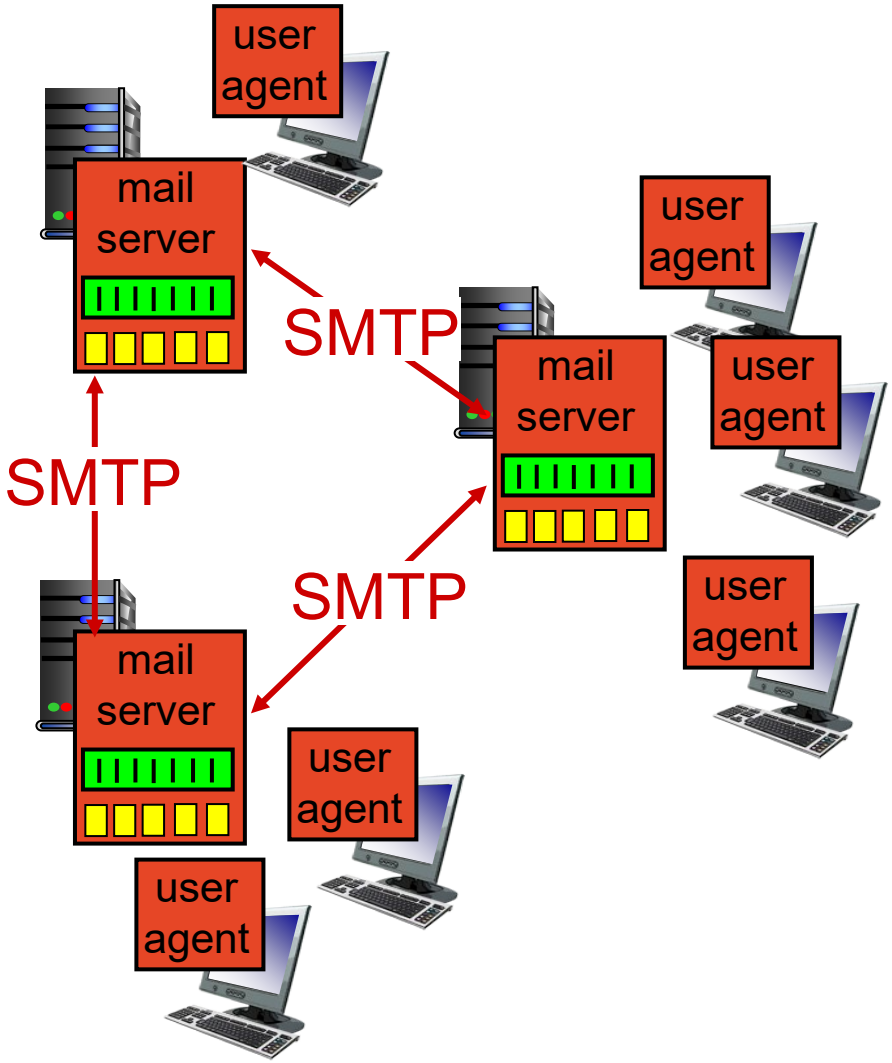
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client



Electronic mail: mail servers

mail servers:

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



Electronic Mail: SMTP [RFC 2821]

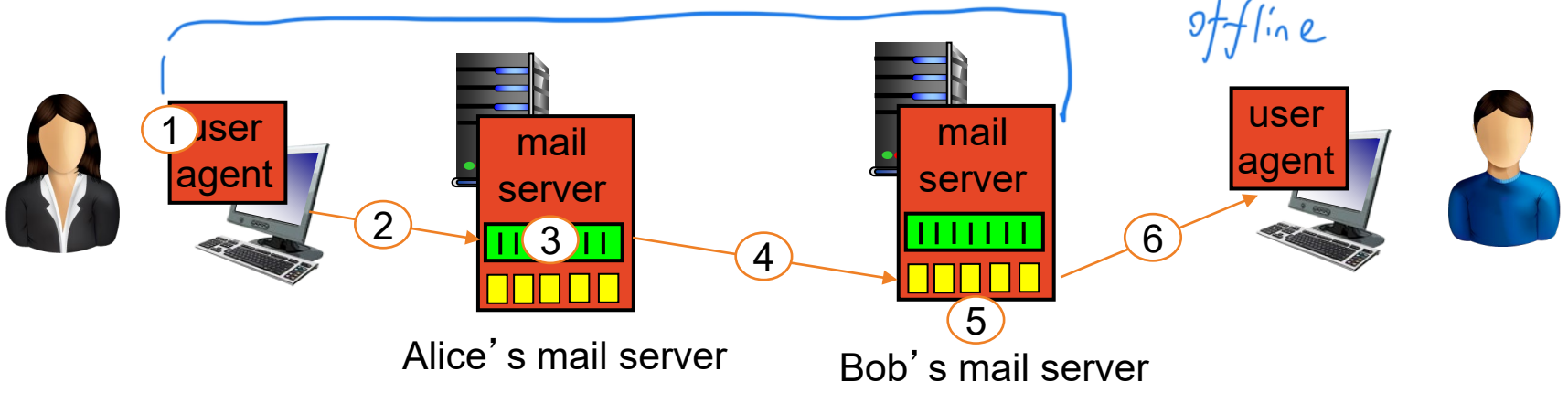
- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP, FTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII
- Q: is SMTP stateful or stateless?
  - Stateful

Scenario: Alice sends message to Bob

User agent

1) Alice uses UA to compose message "to" `bob@someschool.edu`
2) Alice's UA sends message to her mail server; message placed in message queue
3) client side of SMTP opens TCP connection with Bob's mail server
4) SMTP client sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's mailbox
6) Bob invokes his user agent to read message

6 could take long time because Bob could be offline



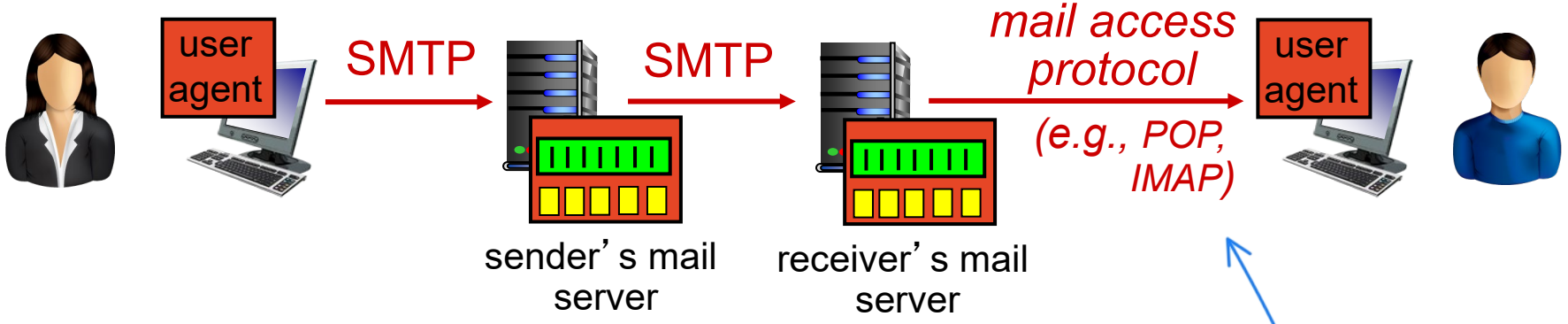
SMTP: final words

- 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
  - Carriage return
  - Line feed

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 msg

Mail access protocols



- SMTP: delivery/storage to receiver's server
- mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]: authorization, download
  - IMAP: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
  - HTTP: Using a browser to access a webmail
    `https://webmail.sydney.edu.au`

解决Bob不在线导致的长时间delay

POP3 (more) and IMAP

more about POP3

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

IMAP

- keeps all messages in one place: at server
- allows user to organize messages in folders
- keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name