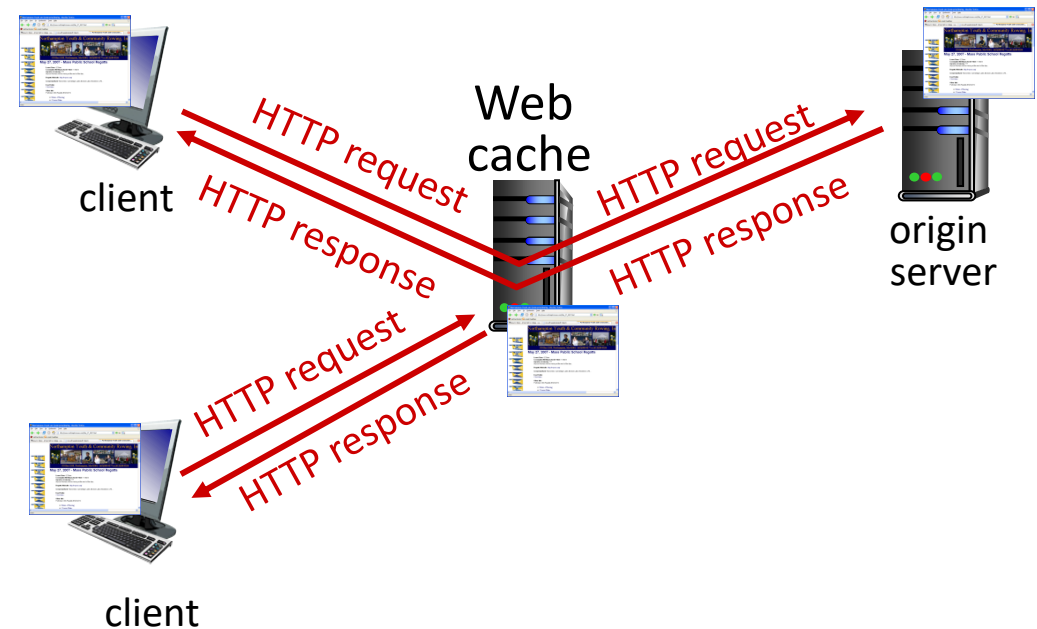


# *Web and HTTP - II*

# Web caches

*Goal:* satisfy client requests without involving origin server

- user configures browser to point to a (local) *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



# Web caches (aka proxy servers)

- Web cache acts as both client and server
  - server for original requesting client
  - client to origin server
- server tells cache about object's allowable caching in response header:

```
Cache-Control: max-age=<seconds>
```

```
Cache-Control: no-cache
```

## *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

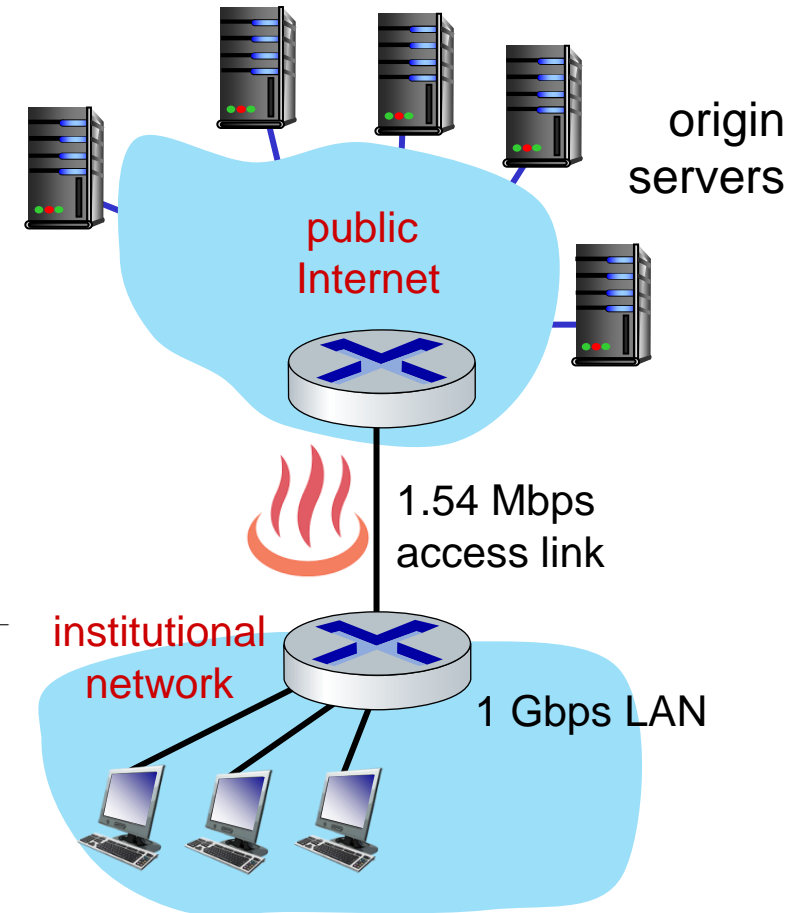
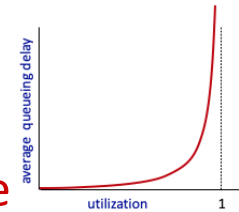
# Caching example

## 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
  - avg data rate to browsers: 1.50 Mbps

## Performance:

- access link utilization = **.97** *problem: large queueing delays at high utilization!*
- LAN utilization: .0015
- end-end delay = Internet delay + access link delay + LAN delay  
= 2 sec + **minutes** + usecs



# Option 1: buy a faster access link

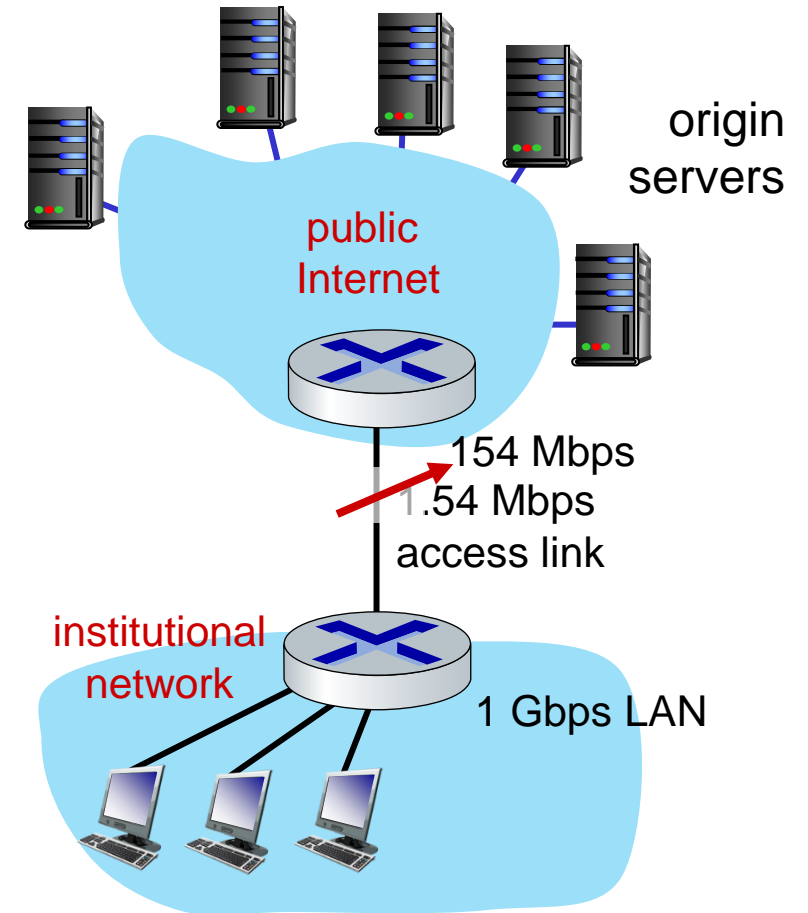
## Scenario:

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

## Performance:

- access link utilization = ~~.97~~ .0097
- LAN utilization: .0015
- end-end delay = Internet delay +  
access link delay + LAN delay  
= 2 sec + ~~minutes~~ + usecs

Cost: faster access link (expensive!) → msecs



# Option 2: install a web cache

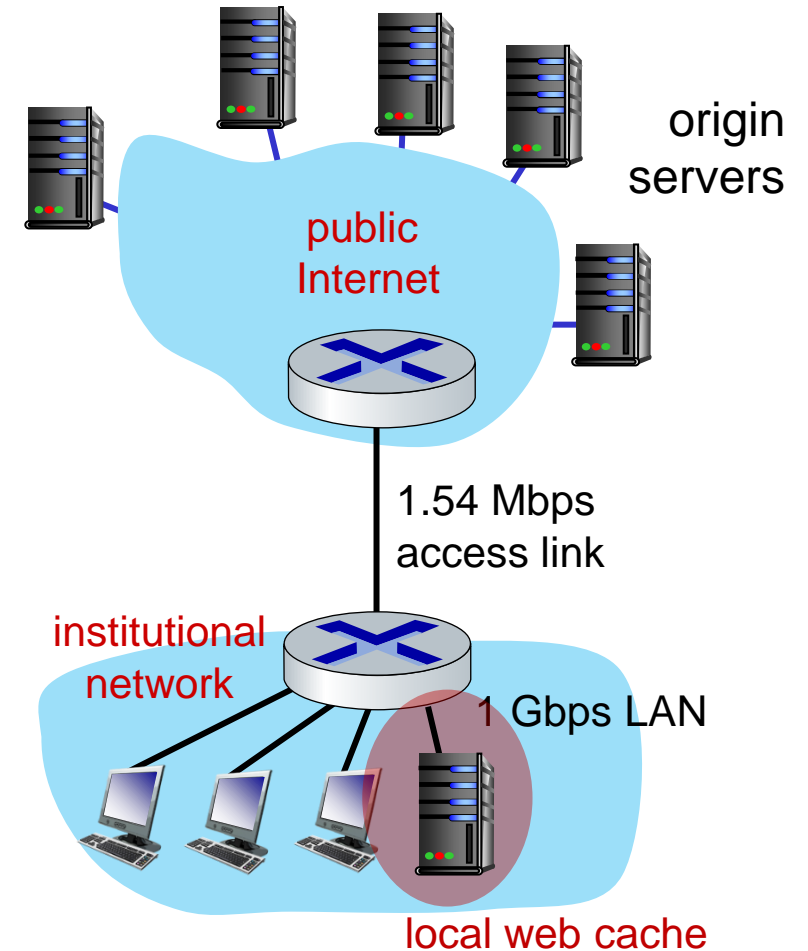
## *Scenario:*

- 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
  - avg data rate to browsers: 1.50 Mbps

*Cost:* web cache (cheap!)

## *Performance:*

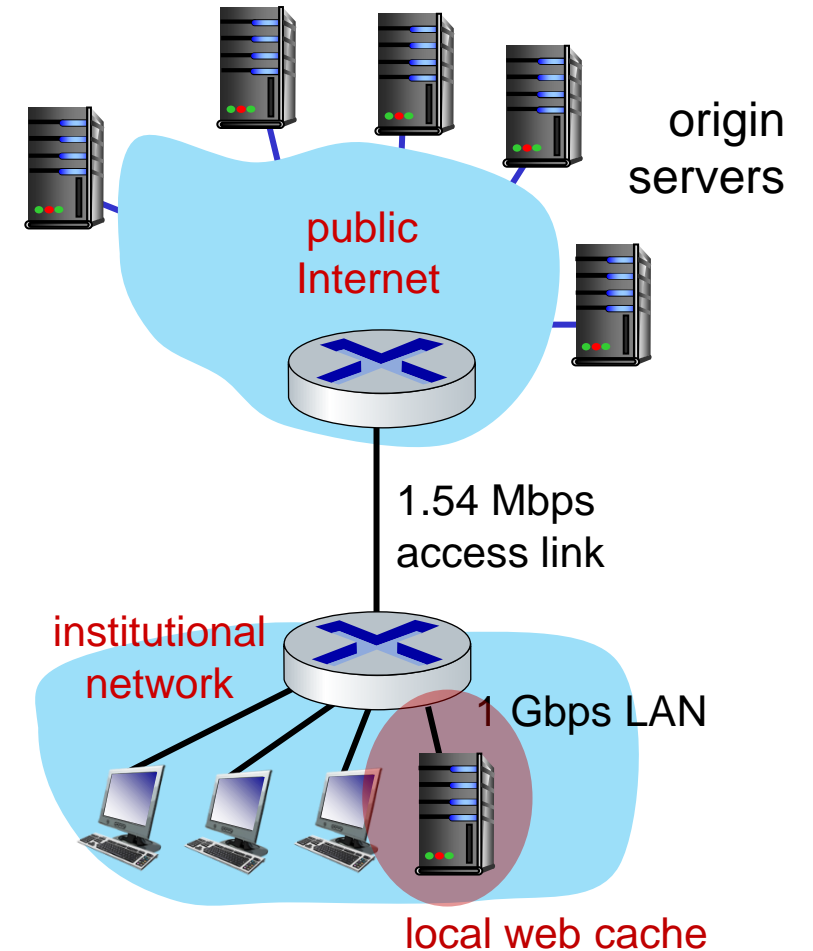
- LAN utilization: .?
  - access link utilization = ?
  - average end-end delay = ?
- How to compute link utilization, delay?*



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

suppose cache hit rate is 0.4:

- 40% requests served by cache, with low (msec) delay
- 60% requests satisfied at origin
  - rate to browsers over access link  
 $= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
  - access link utilization  $= 0.9 / 1.54 = .58$  means low (msec) queueing delay at access link
- average end-end delay:  
 $= 0.6 * (\text{delay from origin servers})$   
 $+ 0.4 * (\text{delay when satisfied at cache})$   
 $= 0.6 (2.01) + 0.4 (\sim \text{msecs}) = \sim 1.2 \text{ secs}$

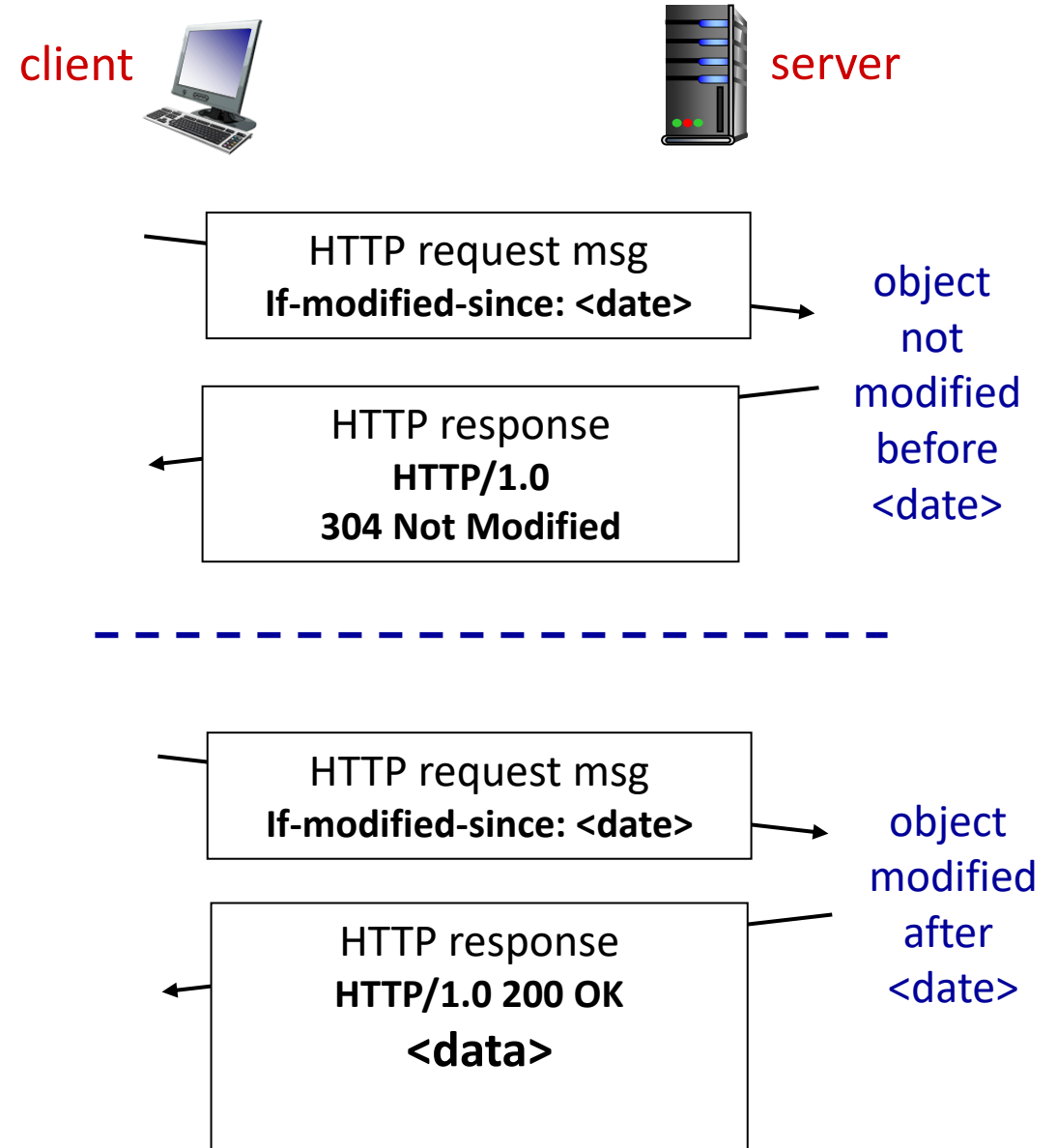


*lower average end-end delay than with 154 Mbps link (and cheaper too!)*

# Conditional GET

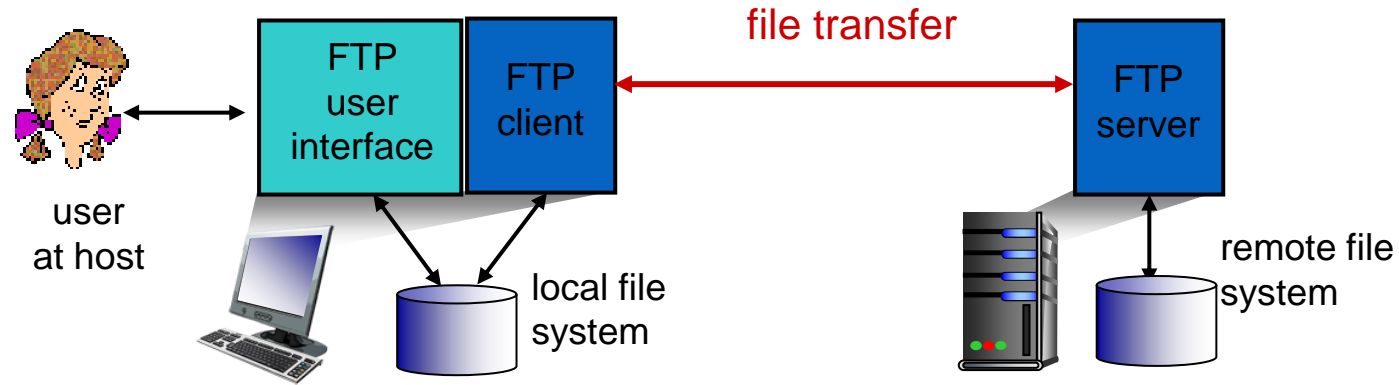
**Goal:** don't send object if cache has up-to-date cached version

- no object transmission delay (or use of network resources)
- **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**





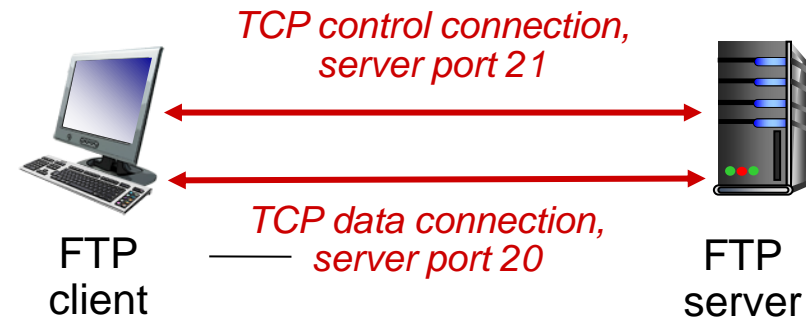
# FTP: the file transfer protocol



- ❖ 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 (**username and password, commands to**
- **change remote directory, and commands to “put” and “get” files)**
- when server receives file transfer command, **server** opens 2<sup>nd</sup> 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
- ❖ **HTTP** “in-band” sends request
- ❖ and response header lines into the same TCP connection that carries the transferred
- ❖ file itself

# 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

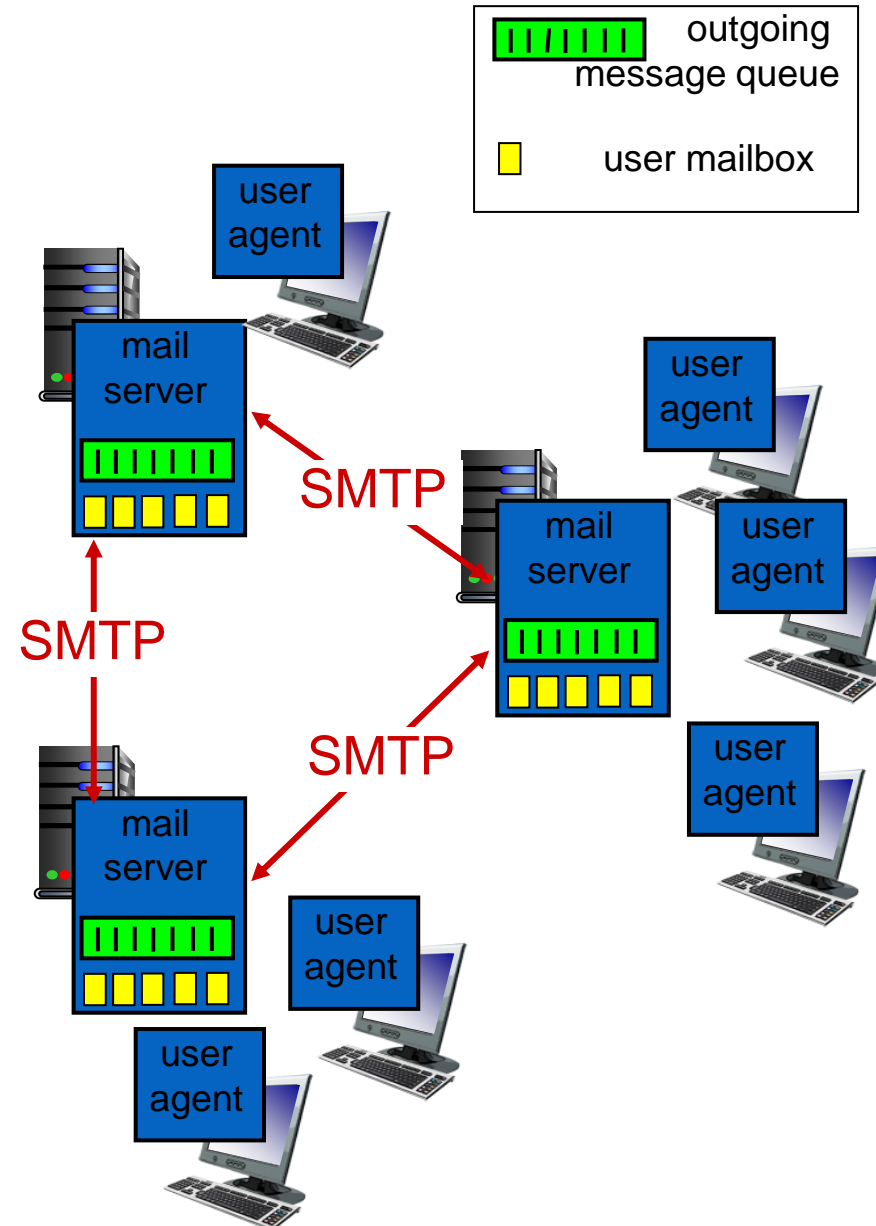
# Electronic mail

## *Three major components:*

- user agents
- mail servers
- simple mail transfer protocol: SMTP

## *User Agent*

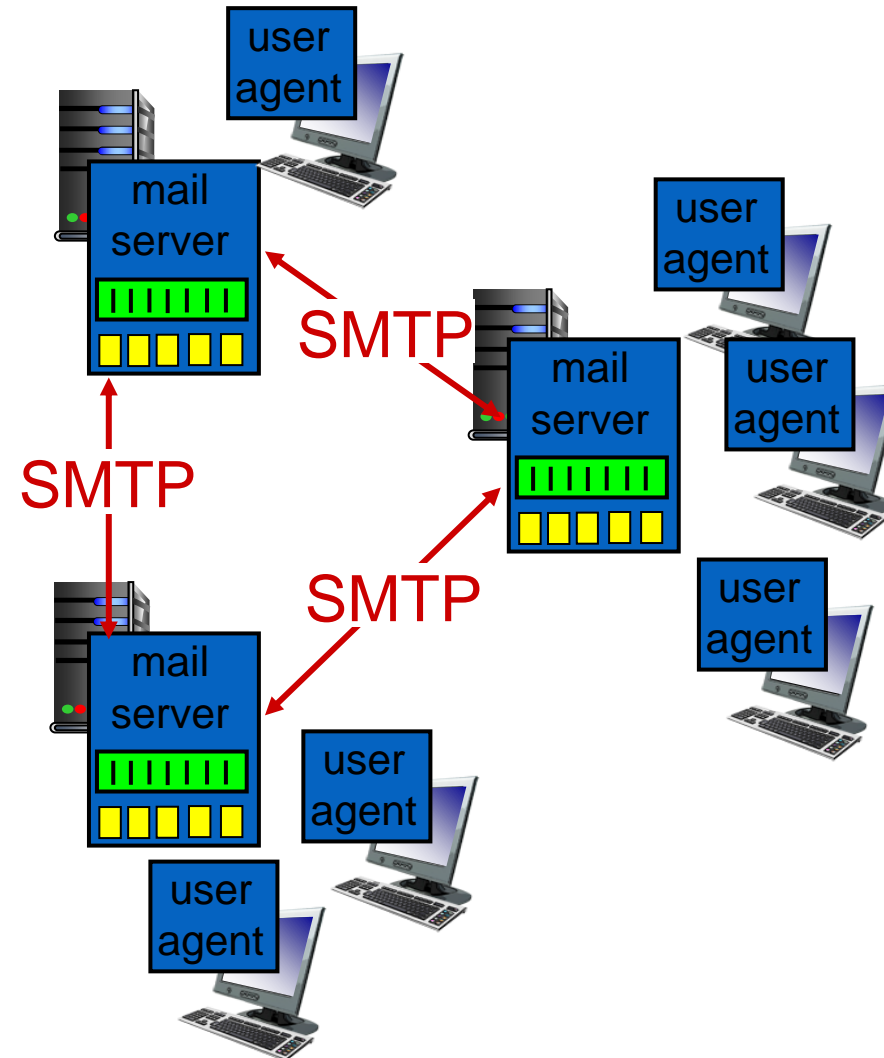
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



# Electronic mail: mail servers

## mail servers:

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

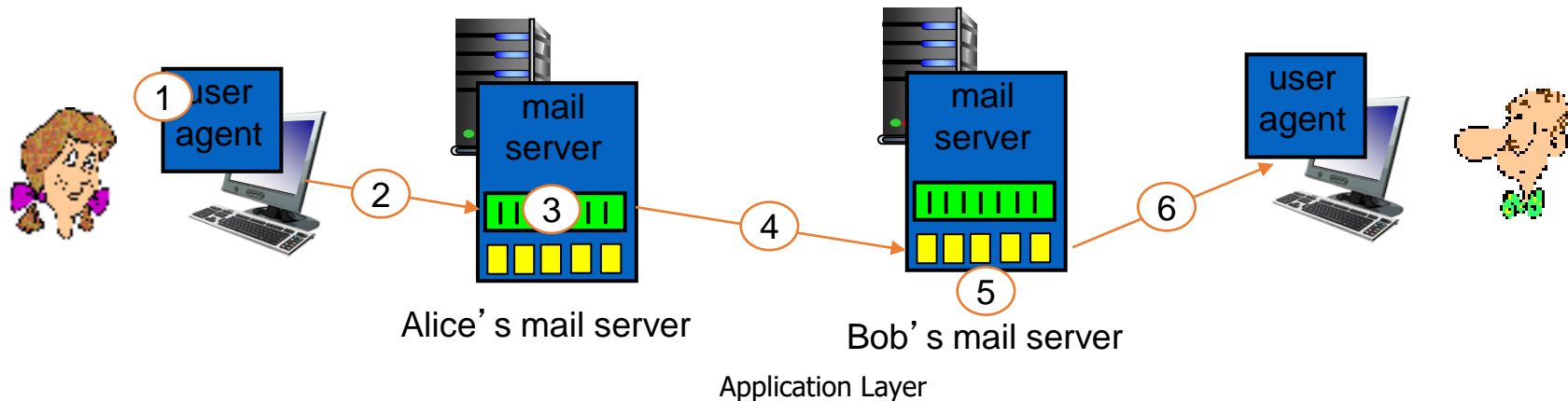


# Electronic Mail: SMTP [RFC 2821]

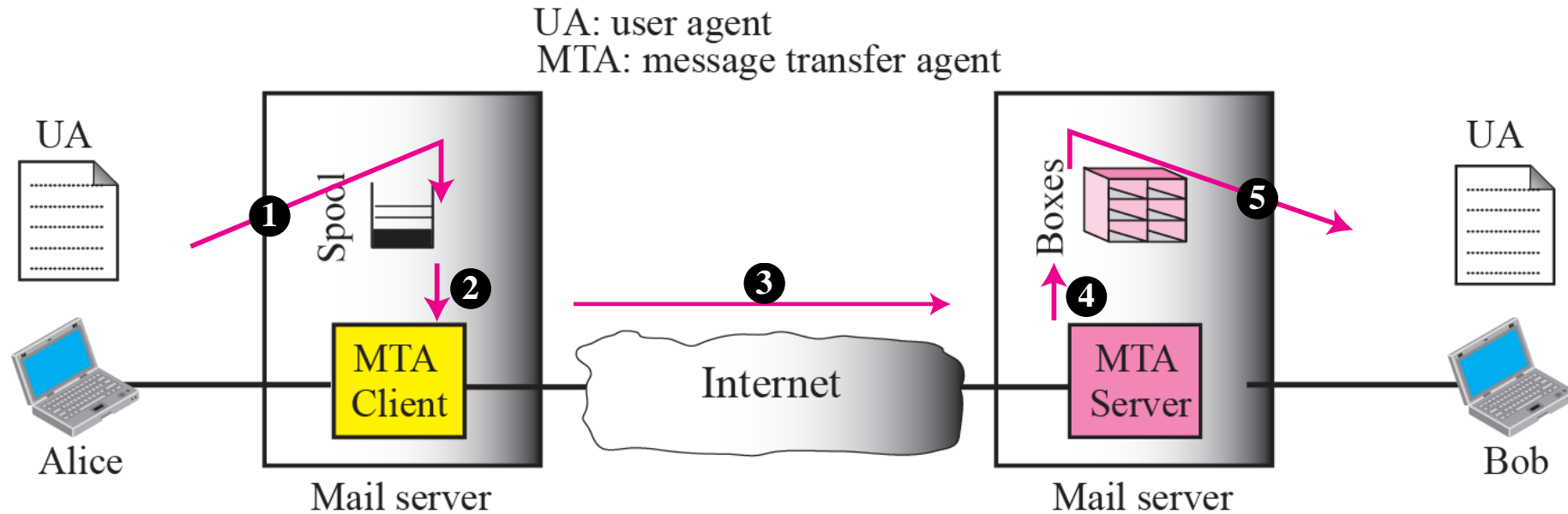
- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP, FTP)
  - **commands**: ASCII text
  - **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 "to" `bob@some school.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



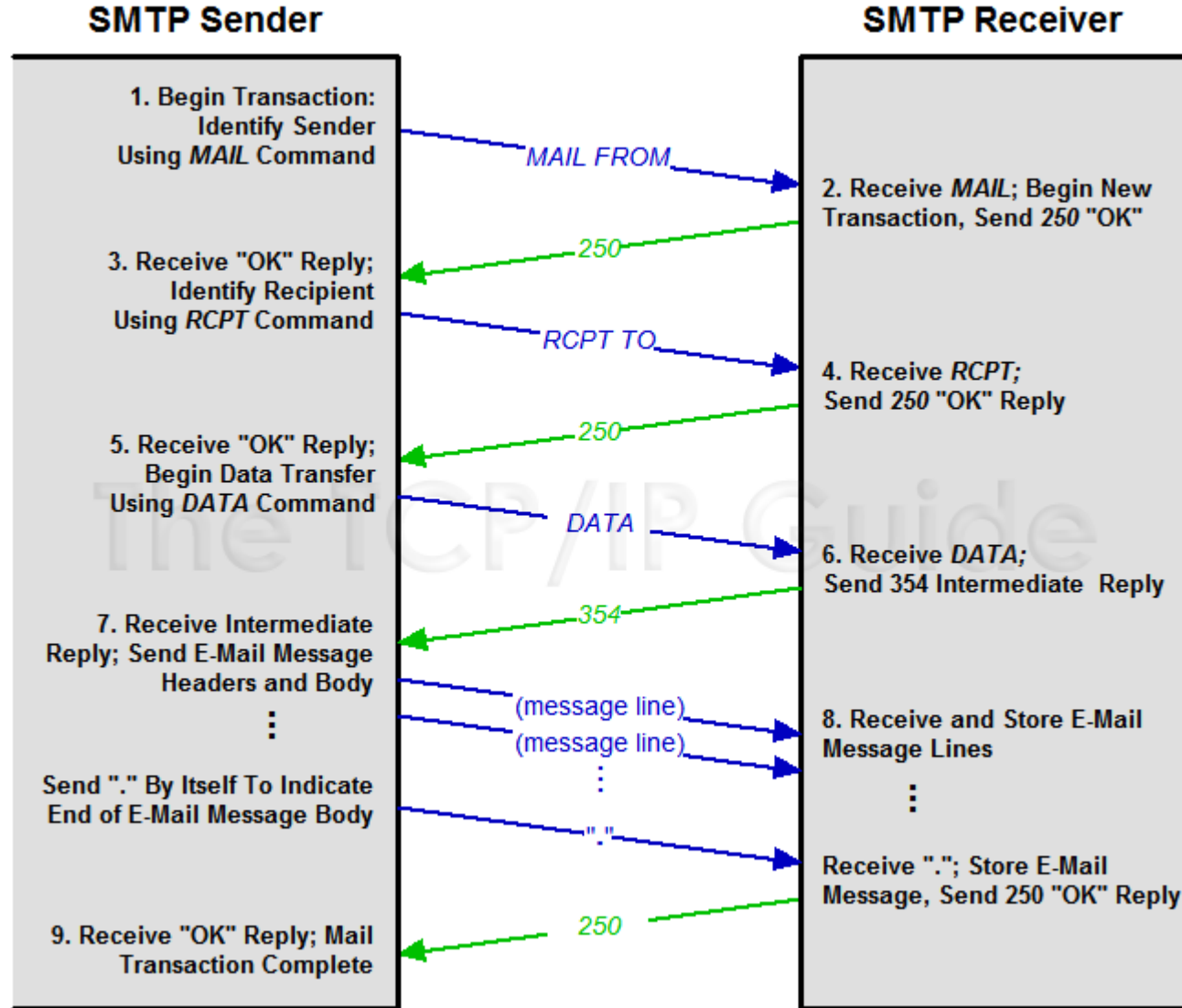
# Scenario: Alice sends message to Bob





# Sample SMTP interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```



# Try SMTP interaction for yourself:

- `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)

# SMTP: final words

---

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII

## *comparison with HTTP:*

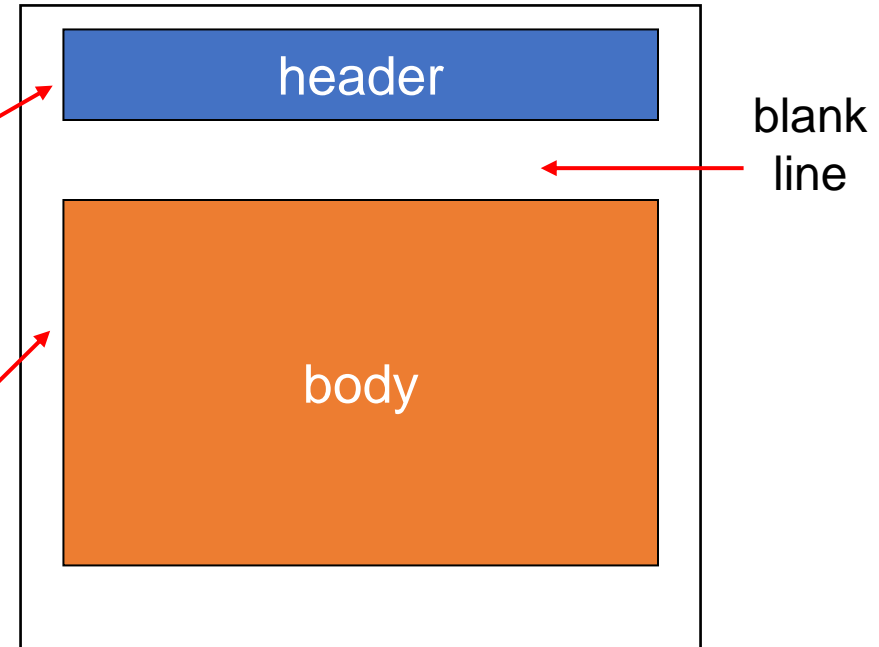
- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in multipart msg

# Mail message format

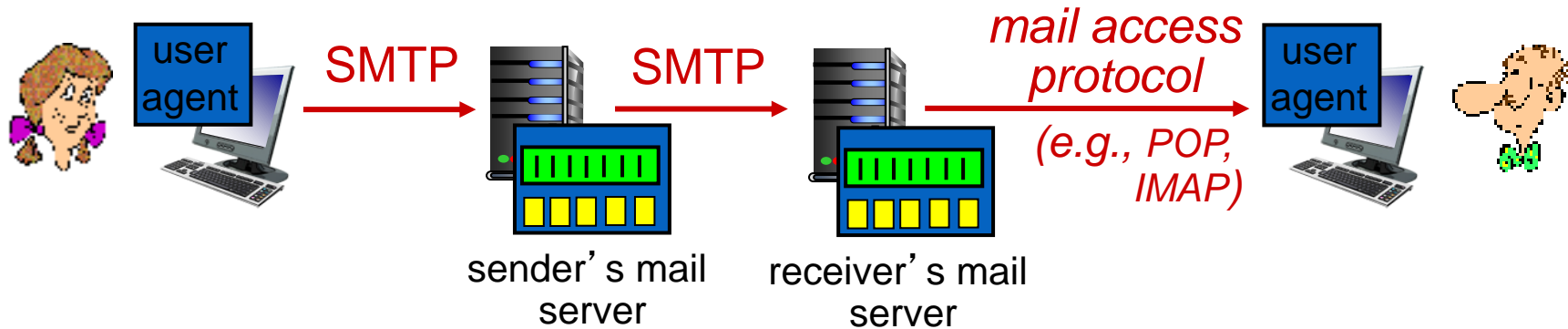
SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format:

- header lines, e.g.,
  - To:
  - From:
  - Subject:
- Body: the “message”
  - ASCII characters only



# 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**: gmail, Hotmail, Yahoo! Mail, etc.

# POP3 protocol

## *authorization phase*

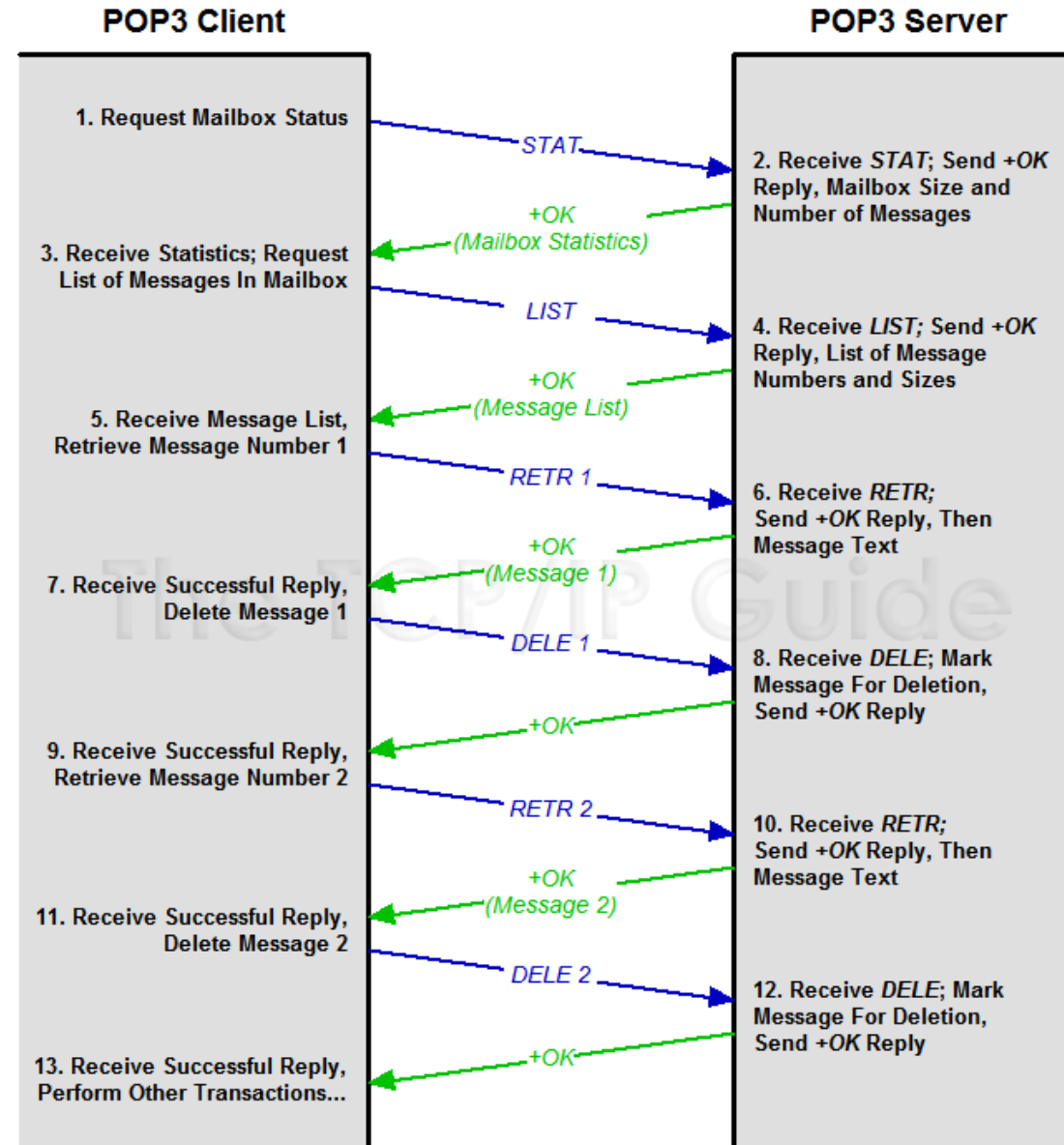
- client commands:
  - **user**: declare username
  - **pass**: password
- server responses
  - **+OK**
  - **-ERR**

## *transaction phase, client:*

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

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

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





# POP3 (more) and IMAP

## *more about POP3*

- previous example uses 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

# Questions