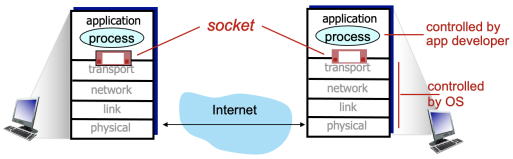
12 - Application Protocols

Client-Server vs. P2P

- Client-server - server always on with permanent IP, client connects with dynamic IP
  - e.g., HTTP, IMAP, FTP
- P2P - no such server, all inter client connects, scalable
  - e.g., torrent file sharing - bittorrent

Process Communication

- processes on the same host comms via Inter-Process Comms (IPC) via OS
- client-server - processes are either clients or servers and are always listening or trying to connect
- p2p - procs have both client and server ports



- comms via sockets (abstraction)
- procs identified by both IP and Port
  - port e.g., HTTP:80, SMTP:25 (mail)

Application Level Protocols

- protocols contain/define
  - 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, Zoom

- apps require transport service

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

  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

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

  security
  - encryption, data integrity, ...

- e.g. proc examples

application	data loss	throughput	time sensitive?
file transfer/download	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, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

TCP vs. UDP

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *connection-oriented:* setup required between client and server processes
- *does not provide:* timing, minimum throughput guarantee, security

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.

Q: why bother? *Why* is there a UDP?

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP [RFC 7230, 9110]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7230], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

- e.g., app transport protocols

Vanilla TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

TLS implemented in application layer

- apps use TLS libraries, that use TCP in turn
- cleartext sent into "socket" traverse Internet *encrypted*
- See our earlier lecture or Chapter 8 in Kurose-Ross

- TCP security (TLS)

Web and HTTP

- web page consists of objects stored in a DOM structure
- object an be HTML file/element, images, applets, auio, etc.
- web-page consists of base HTML file (e.g. index.html) with several referenced objets

addressable via URL e.g. `www.someschool.edu/someDept/pic.gif`

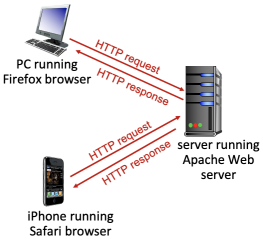
host name | path name

HTTP

- HTTP - hypertext transfer protocol

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 is persistent or non-persistent because storing state is complex

Non-persistent HTTP

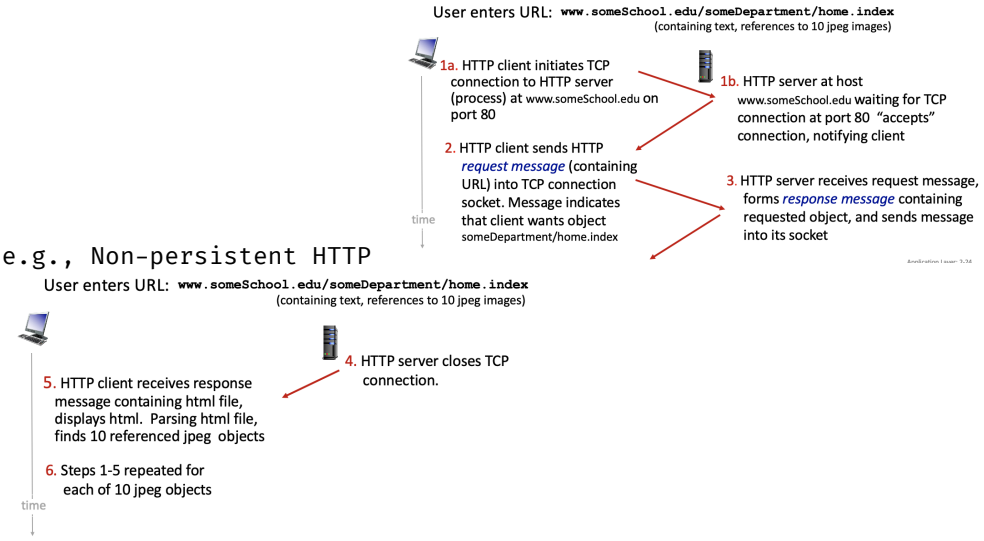
1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

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

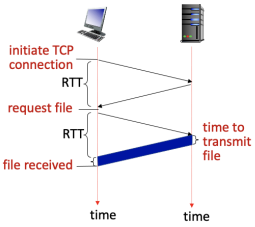
- e.g., Non-persistent HTTP



- non-persistent performance

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

HTTP response time (per object):
- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



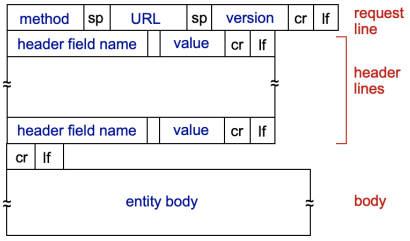
Non-persistent HTTP response time = 2RTT+ file transmission time

- persistent (HTTP 1.1) v. non-persistent pro cons

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

Persistent HTTP (HTTP1.1):
- 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 (cutting response time in half)

- messages are request (GET/POST/PUT …) or response (status ode, e.g. 404)



POST method:
- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET method (for sending data to server):
- include user data in URL field of HTTP GET request message (following a '?'):

`www.somesite.com/animalsearch?monkeys&banana`

HEAD method:
- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

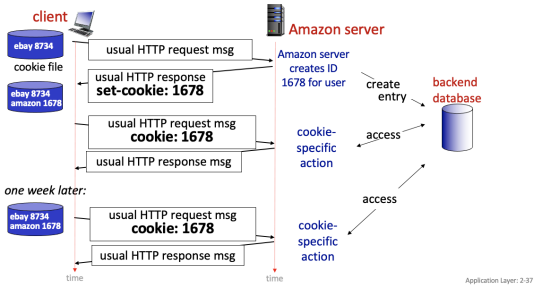
PUT method:
- uploads new file (object) to server
- completely replaces file that exists at specified URL with content

- request messages

- request messages
  - 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 (in Location: field)
    - 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 - State Management

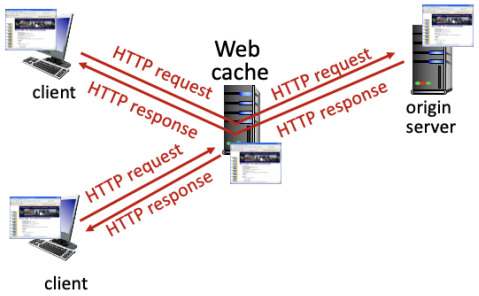
- HTTP requests are stateless so for multi-step exchange or repeated connections, store cookies to maintain state (stored in the browser)



- components
  - cookie header line of HTTP **response** message
  - cookie header line in next HTTP **request** message
  - cookie kept on user's host and managed by user's browser
  - back-end DB at server website containing cookie id val to map state
- **first party cookies** - track user behavior for the given website
- **third party cookies** - tracks user behavior across multiple websites without visiting third party site
- GDPR (EU general data protection regulation) requires sites to inform users about 3rd party cookies

Web Cache

- to inc performance and decrease load on server, initial http get may include data and web cached data
- may be done via proxy server which acts both as client and server intermediary

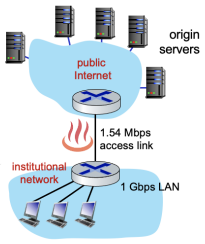


- example

- og bottleneck

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

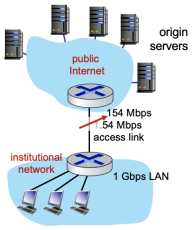


- sol1: get faster link
- sol2: web cache, much cheaper than more expensive link

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

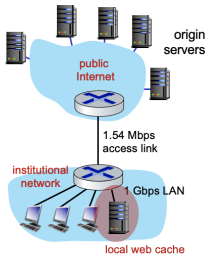
Performance:
- access link utilization = ~~.97~~ .0097
- LAN utilization: .0015
- end-end delay = Internet delay + access link delay + LAN delay = 2 sec + ~~minutes~~ + usecs

Cost: faster access link (expensive!) → msecs



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 Mbps = .9 Mbps
  - access link utilization = 0.9/1.54 = .58 means low (msec) queueing delay at access link
- 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!)



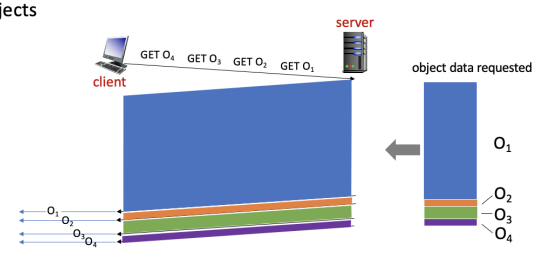
HTTP/2

- dec delay iin multi obj HTTP requets
- HTTP1.1 introdued multiple pipelined gets over single TCP connection
- server responds in order (FCFS) - this may cause head of line (HOL) blocks for small objs behind large objects → loss recovery stalling transmission

HTTP/2: [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:
- methods, status codes, most header fields unchanged from HTTP 1.1
- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking

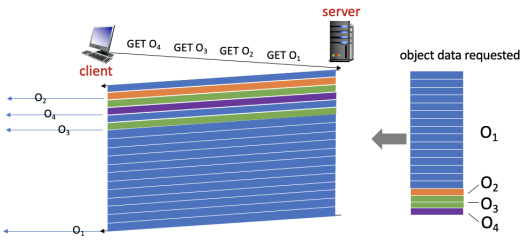
- e.g, HTTP/1.1 HOL issue

HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



objects delivered in order requested: O_2, O_3, O_4 wait behind O_1

HTTP/2: objects divided into frames, frame transmission interleaved



- e.g., HTTP/2 frame sol

HTTP/3

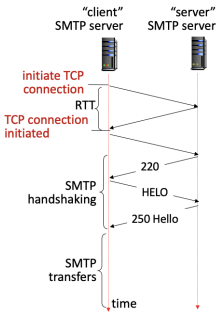
- HTTP/2 disadvantages
  - single connection means recovery from packet loss still stalls object transmissions
  - no security over vanilla TCP
- HTTP/3 adds security and per object error and congestion control via pipelining over UDP

Email, SMTP, IMAP

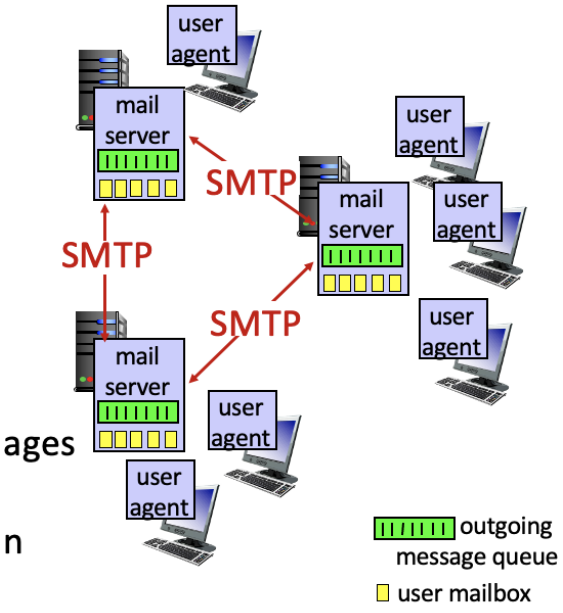
- composed of: user agents, mail servers, and SMTP
- SMTP - simple mail transfer protocol
  - between mail servers on client (sender) - server (receiver) protocol

SMTP RFC (5321)

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
  - direct transfer: sending server (acting like client) to receiving server
- three phases of transfer
  - SMTP handshaking (greeting)
  - SMTP transfer of messages
  - SMTP closure
- command/response interaction (like HTTP)
  - commands: ASCII text
  - response: status code and phrase



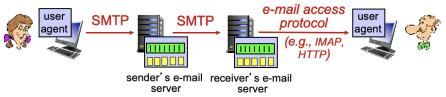
- user agent - mail reader, creating editing reading sending mail on various clients - outlook, gmail, iphone, etc,
- mail server - composed of mailbox (contains incoming messages) and message queue (outgoing mail queue)



comparison with HTTP:

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

- request and response messages
- **IMAP** - internet mail access protocol - provides retrieval and deletion



- SMTP: delivery/storage of e-mail messages to receiver's server
- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
- HTTP: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages