Application Layer: Client-Server and the Web

Objectives: basic socket programming: TCP sockets, port numbers and de-multiplexing; client-server model of network applications; web servers and HTTP: HTTP requests and replies, persistent and non-persistent HTTP, web caching & performance impact; introduction of network tool: wireshark packet sniffer and analyzer

NS5: April 2, 2018
Textbook (K&R): Sections 2.2, 2.7

Naming Application Endpoint

- We learned how to name machines (DNS)
  - But *is* that enough?
- Consider visiting a web page by URL
  - http://www.cs.purdue.edu/people/yau

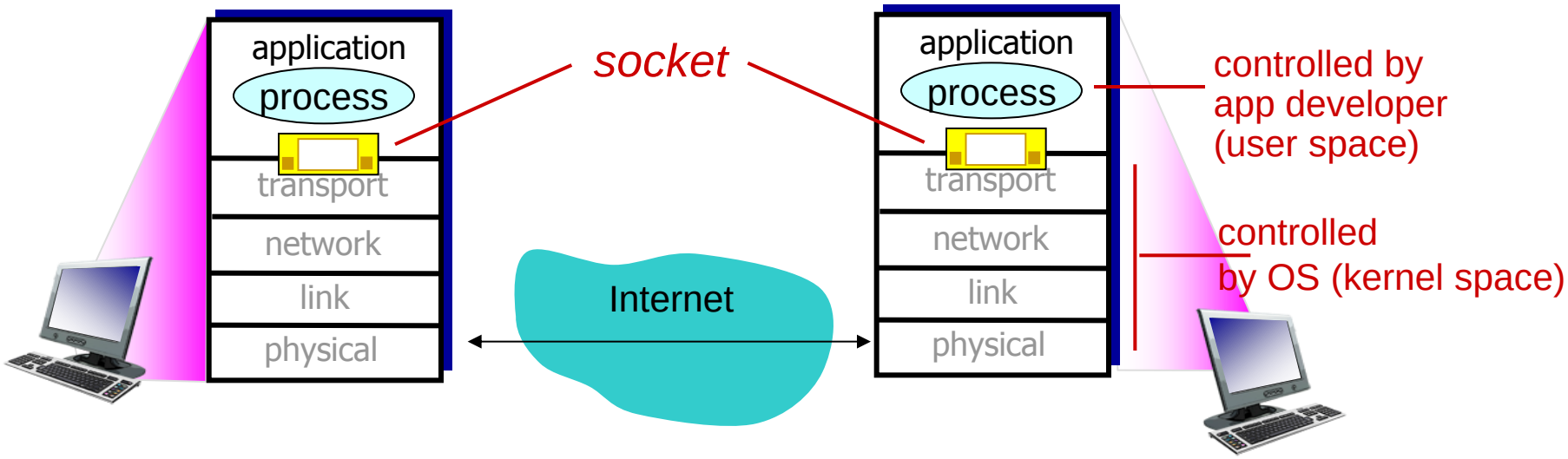
Machine Resource (path name)

- Resource name encodes machine id
- But in fact talking to some *application* (web server) running on that machine (recall: running app = process)
  - *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
    - And, using process id (pid) to further identify process won't work well! (pid is volatile local id not meant for external consumption)

Socket to the rescue!

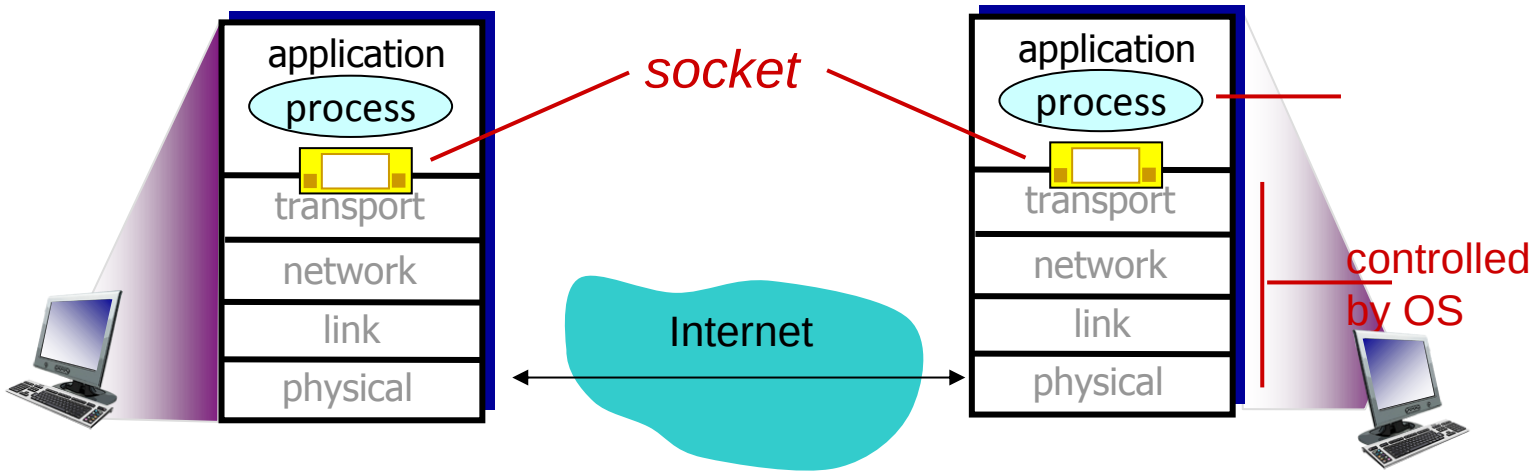
goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



What is a socket?

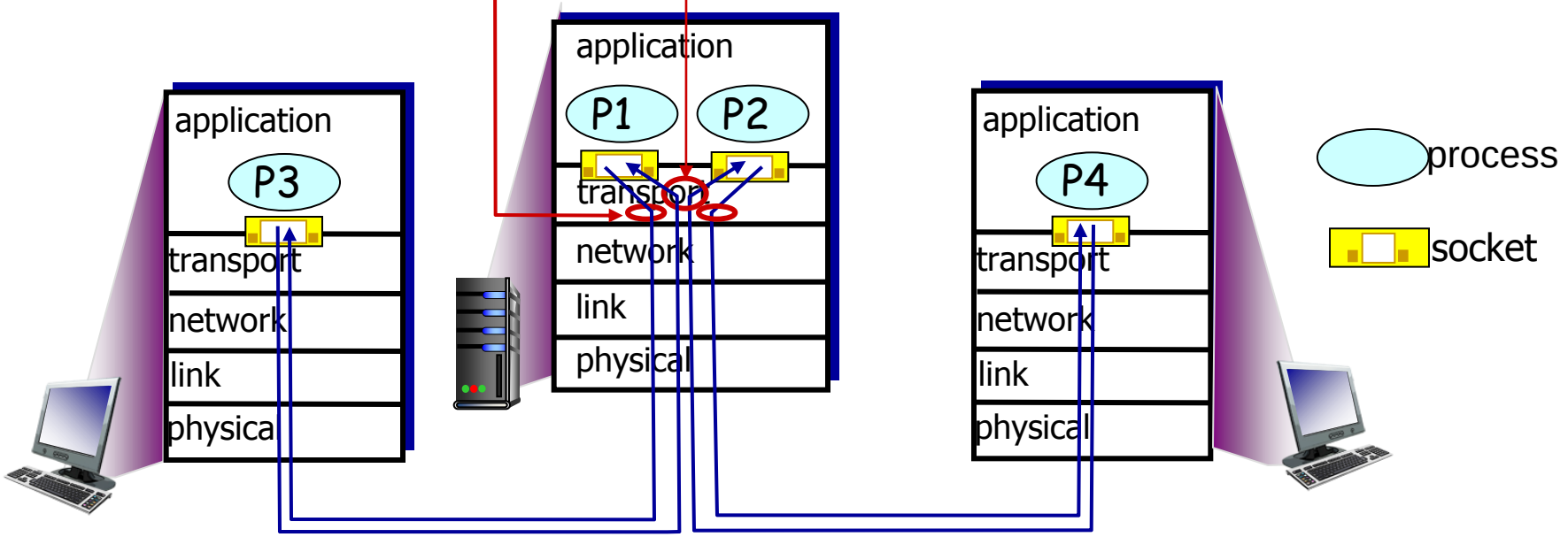
- Network access is a kind of I/O (like files)
  - Let processes open "file descriptors" to do the I/O
  - Yes, but use a *special* kind of fd's called *sockets*
- *identifier* w/ socket: both IP address and port number associated with the application
- Example (TCP) port numbers:
  - HTTP server: 80
  - mail server: 25
- To send HTTP message to gaia.cs.umass.edu web server, put following in packet header:
  - IP address: 128.119.245.12 (network layer)
  - port number: 80 (transport layer – port number is specific to transport protocol, mainly, TCP vs. UDP)



Multiplexing/demultiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



P1 – mail server; P2 – web server
P3 – Outlook mailer; P4 – Firefox browser

Connection-oriented (TCP) demux

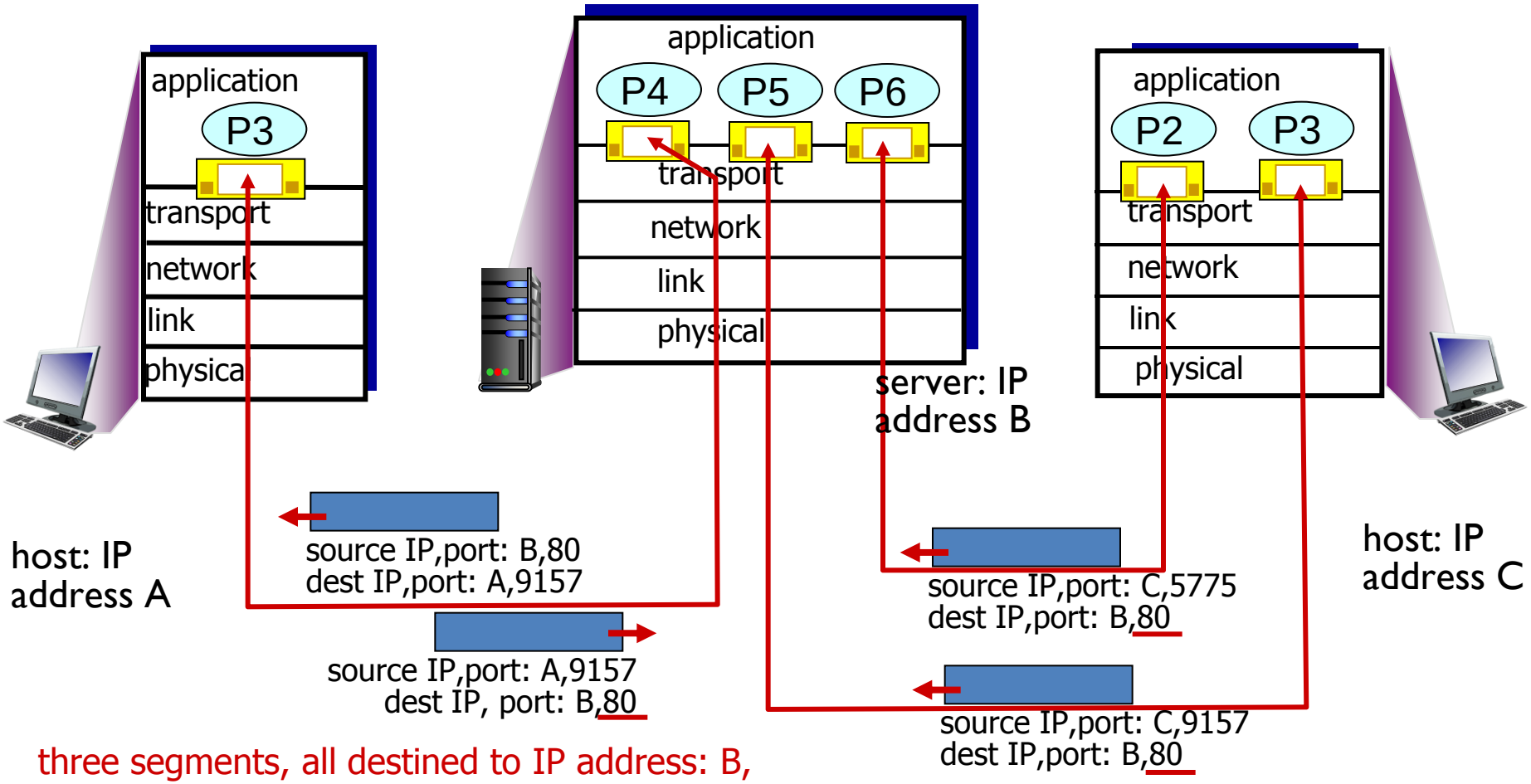
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four* values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple

Connection Oriented (TCP) vs. Connection Less (UDP):

- connection setup
- reliability
- flow control
- congestion control

Connection-oriented demux: example



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to different sockets

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

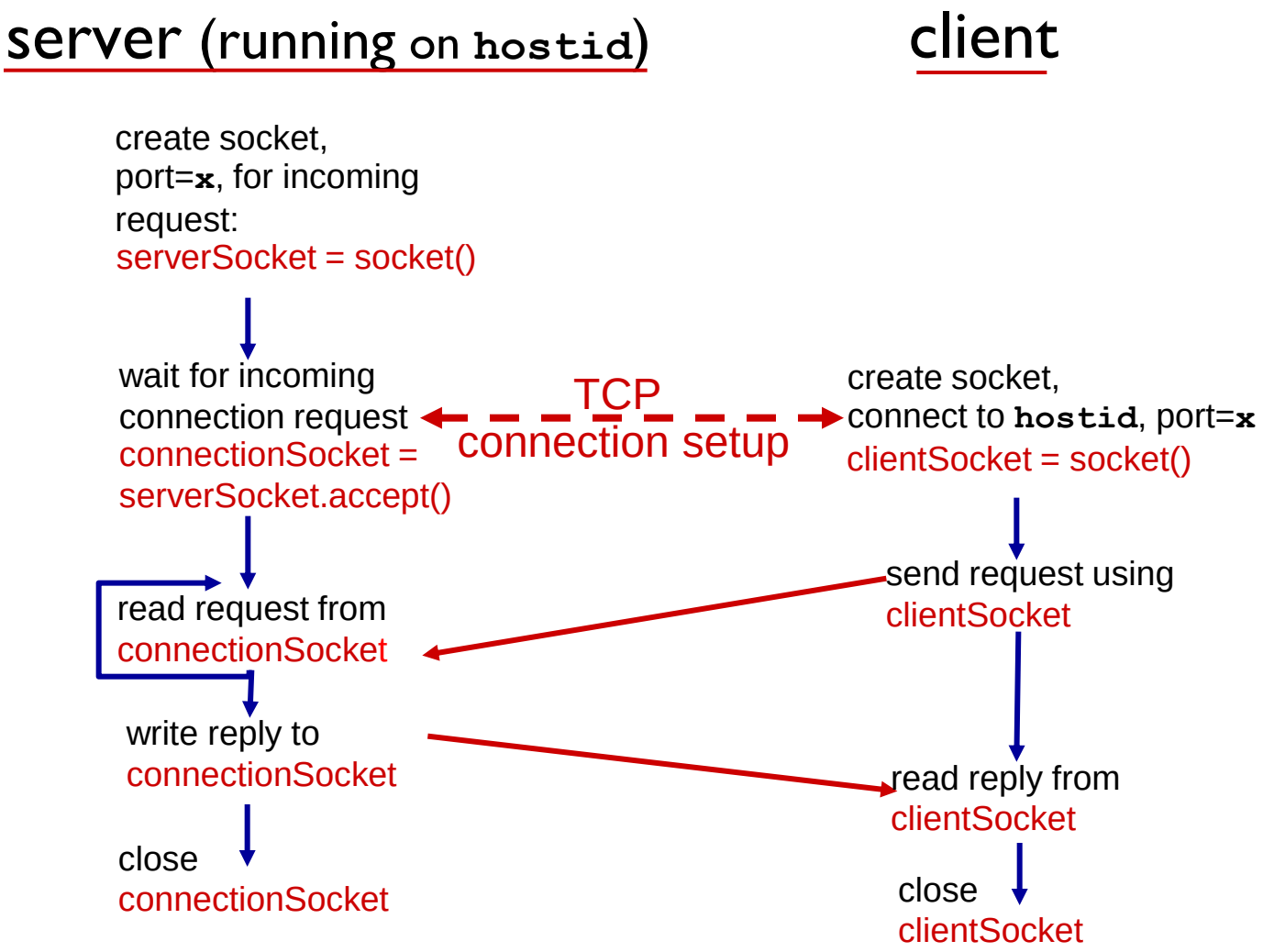
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket

close connection to this client (but *not* welcoming socket, i.e., serverSocket)

Example app: TCP client

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

UDP socket programming

UDP: no "connection" between client & server

- no handshaking (to open connection) before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

"Iconic" TCP server: 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,* e.g.,

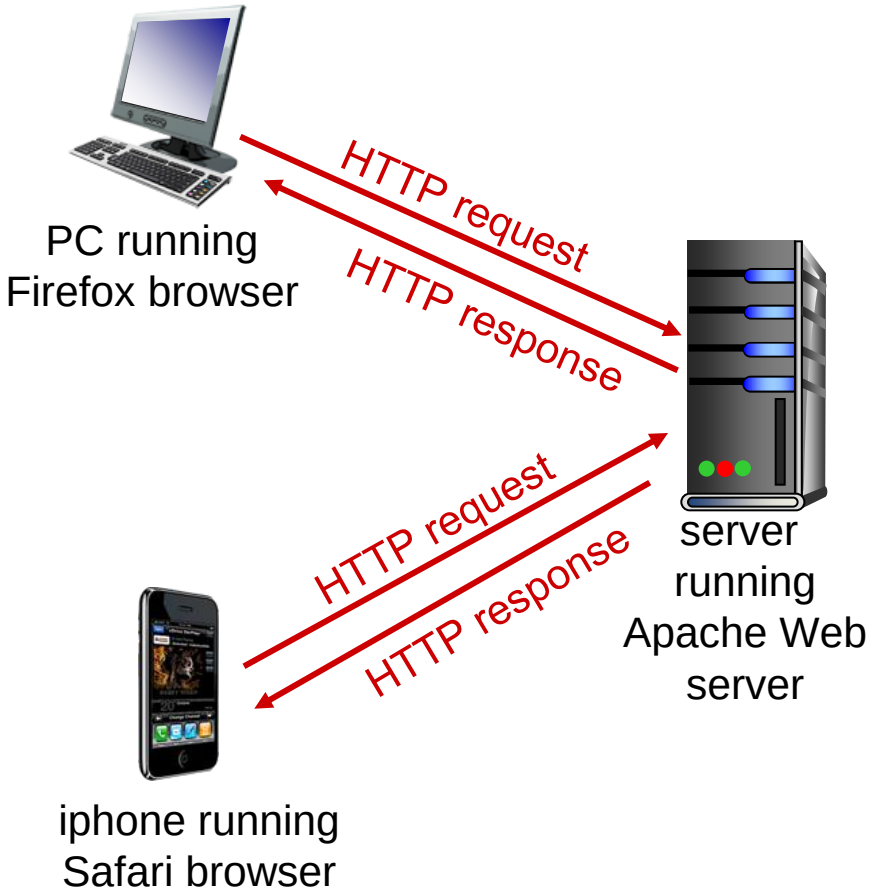
`www.someschool.edu/someDept/pic.gif`

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



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

Basic HTTP is "stateless"

- server maintains no information about past client requests
- Simple: no need to maintain history and no need to clean up state if server/client crashes
- But *cookies* can be added to keep state

HTTP connections

non-persistent HTTP

(HTTP/1.0)

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

persistent HTTP

(HTTP/1.1)

- server leaves connection open after sending response
- multiple objects can be sent over single TCP connection between client, server

Example:

- http://www.nyu.edu/projects/keithwross/

1. See this web page displayed in your browser.
2. Look at the HTML source of the web page.
3. Identify where the HTML source references Keith's profile picture.
4. Think about the *order* in which the browser retrieves different elements of the web page. E.g., any *dependency* between them?

Non-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

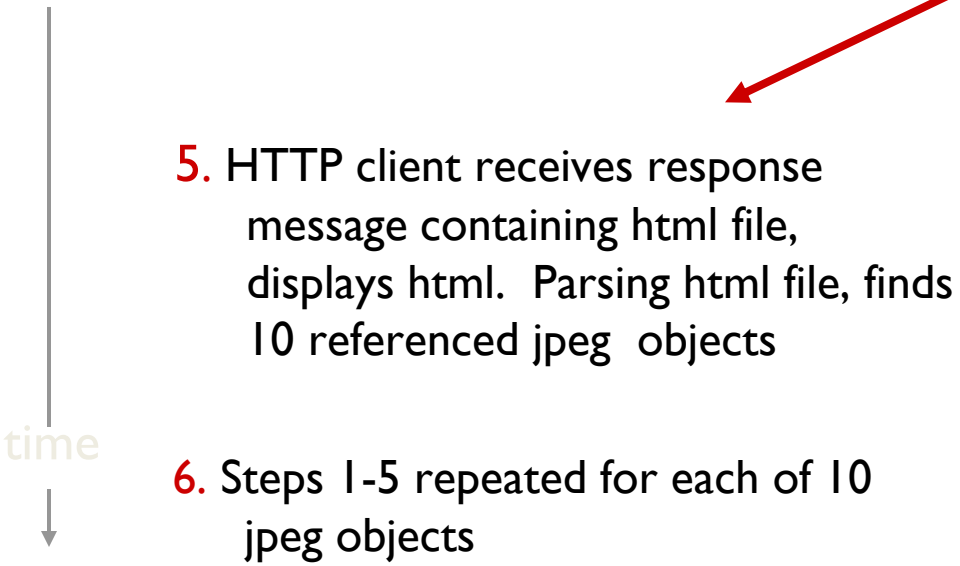
1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

2. HTTP client sends HTTP *request message* (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

Non-persistent 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

NB. TCP uses *3-way handshake* to open a connection. The 3 messages are: SYN, then SYN + ACK, then ACK. Application (e.g., HTTP) data can be "piggy-backed" onto the last ACK. (See next slide for illustration.)

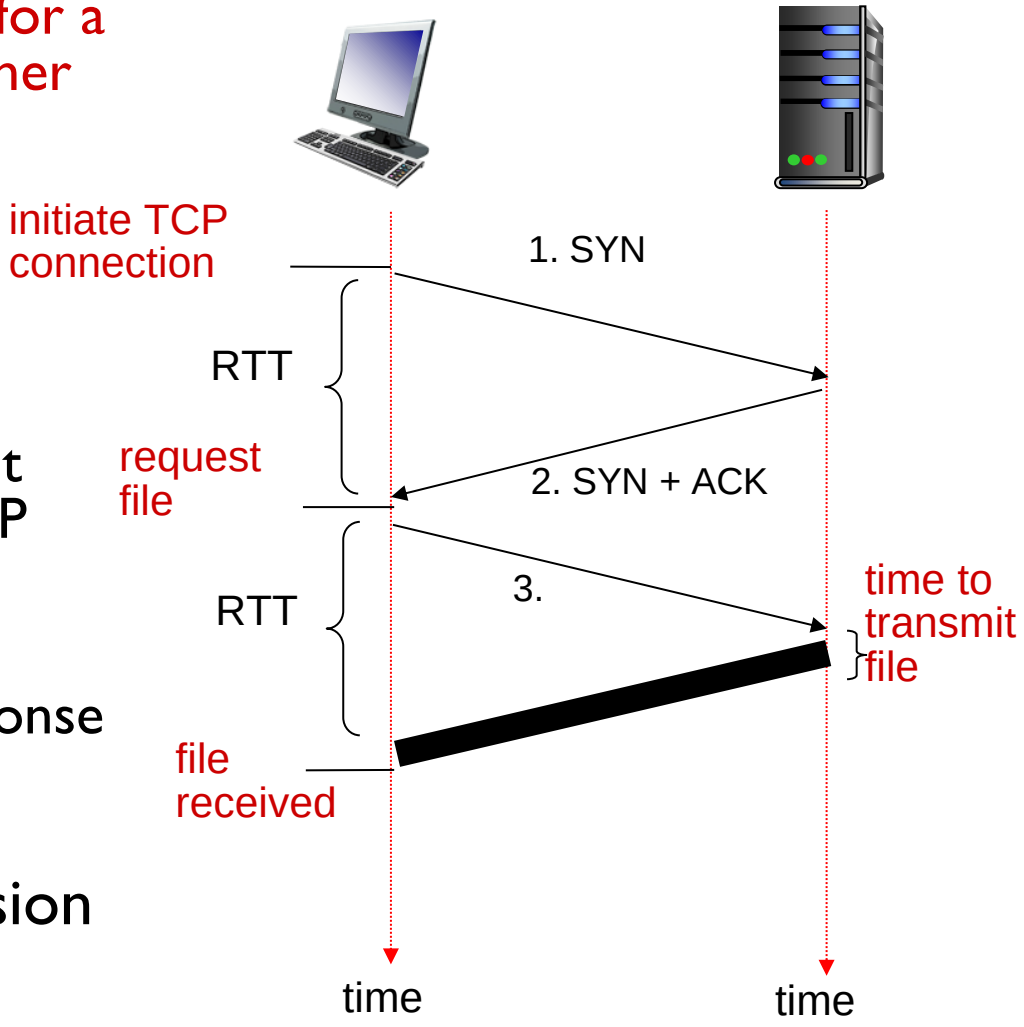
Non-persistent HTTP: response time

RTT = round trip time (time for a small packet to go to another machine & come 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 (for *one* URL object retrieved as a file) =

2RTT+ file transmission time



Message 3 = ACK (TCP), with piggy-backed HTTP request for URL

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 (this is called *pipelining*)
  - i.e., start retrieving another object before completing retrieval of a previous object
  - Usually faster than serial retrievals (by increasing utilization of the network)

persistent HTTP:

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

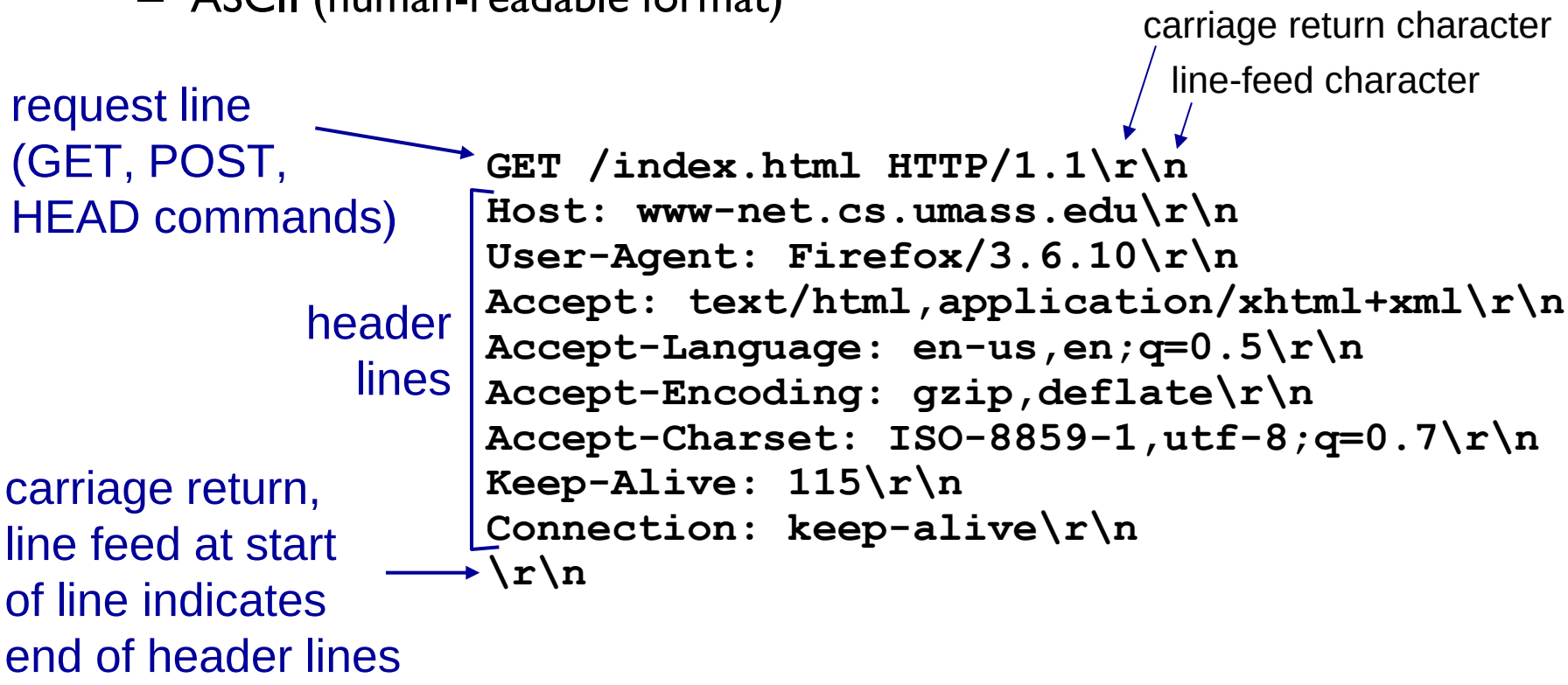
NS Activity 5.1

Draw space-time diagram for the previous HTTP request scenario assuming persistent connections and pipelining (assume full parallelism for all 10 JPEG images)

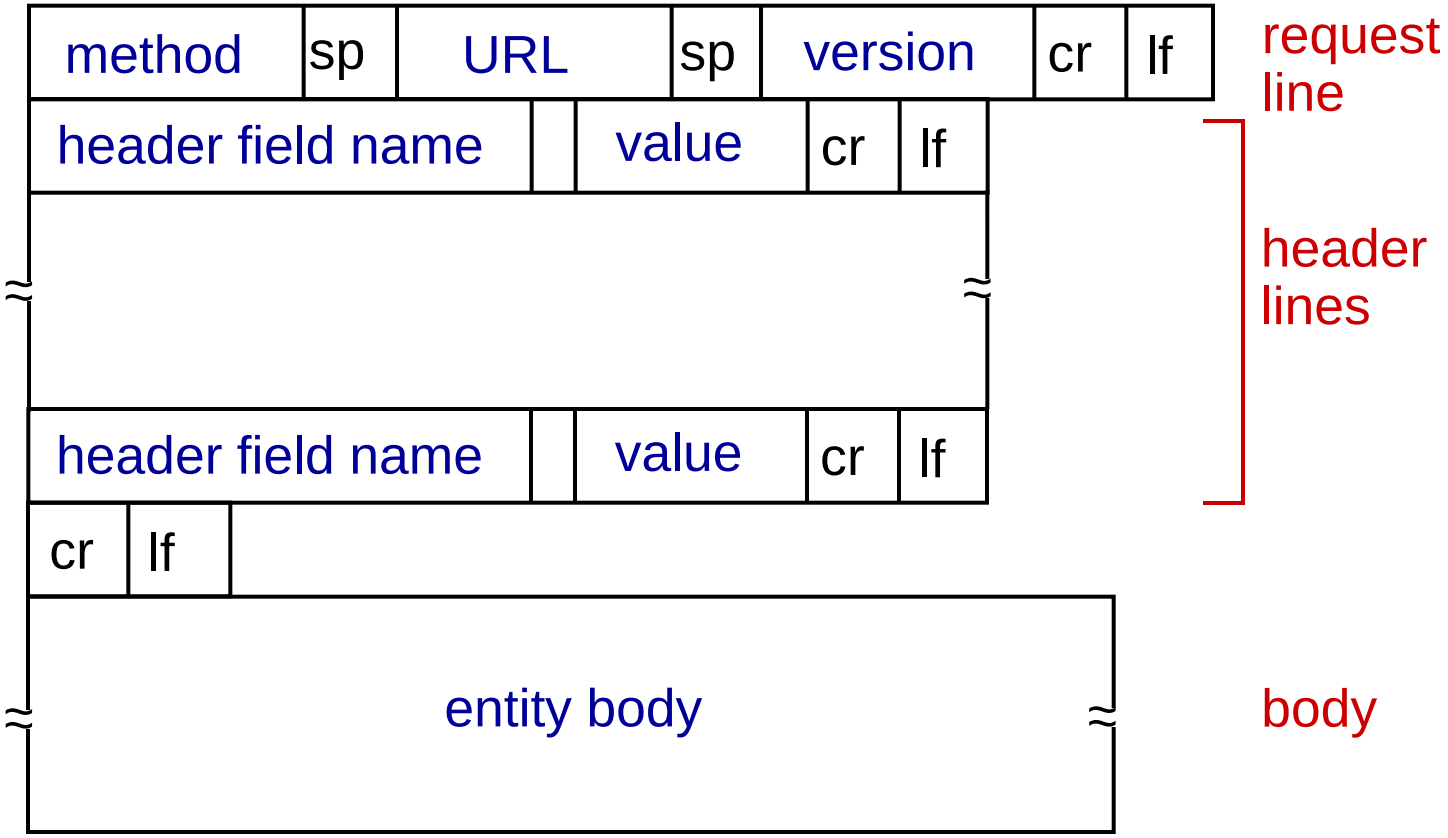
Hint. You can ignore TCP close.

HTTP request message

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



HTTP request message: general format



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

NS Activity 5.2: Pretend you're the browser!

1. Launch www.sutd.edu.sg/education from (say) firefox
2. Now telnet to the same Web server (port 80 is HTTP):

```
telnet sutd.edu.sg 80
```

opens TCP connection to port 80 (default HTTP server port) at sutd.edu.sg
anything typed in sent to port 80 at sutd.edu.sg

3. Type in a (minimal) GET HTTP request (request line + 1 header line):

```
GET /education/ HTTP/1.1
Host: sutd.edu.sg
```

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

4. Look at response message sent by HTTP server! Why is it all text? What does the text say?

Connect to HTTPS

- HTTPS
  - Runs at port 443
  - Uses secure socket layer (SSL): encrypted communications
- You still can connect to it though
  - % openssl s_client -connect sutd.edu.sg:443
- Now, what text do you get when you enter the GET request from the previous page? (Verify with Firefox it's the web page loaded on the browser.)

HTTP response message

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
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 ...
```

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

User-server state: cookies

many Web sites use cookies

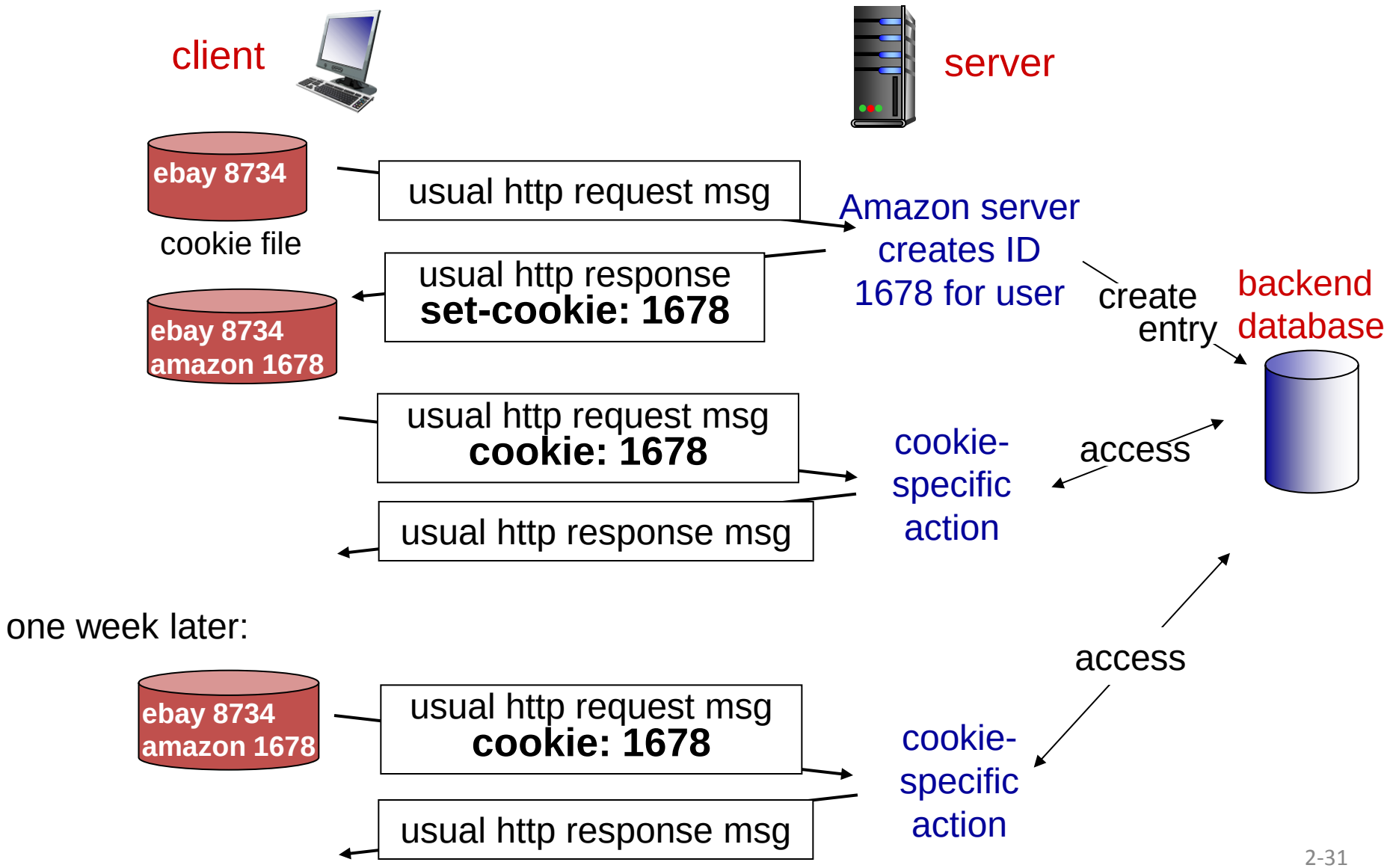
four components:

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

example:

- Susan always access 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: keeping "state" (cont.)



Cookies (continued)

what cookies can be used for:

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

aside

cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

how to keep "state":

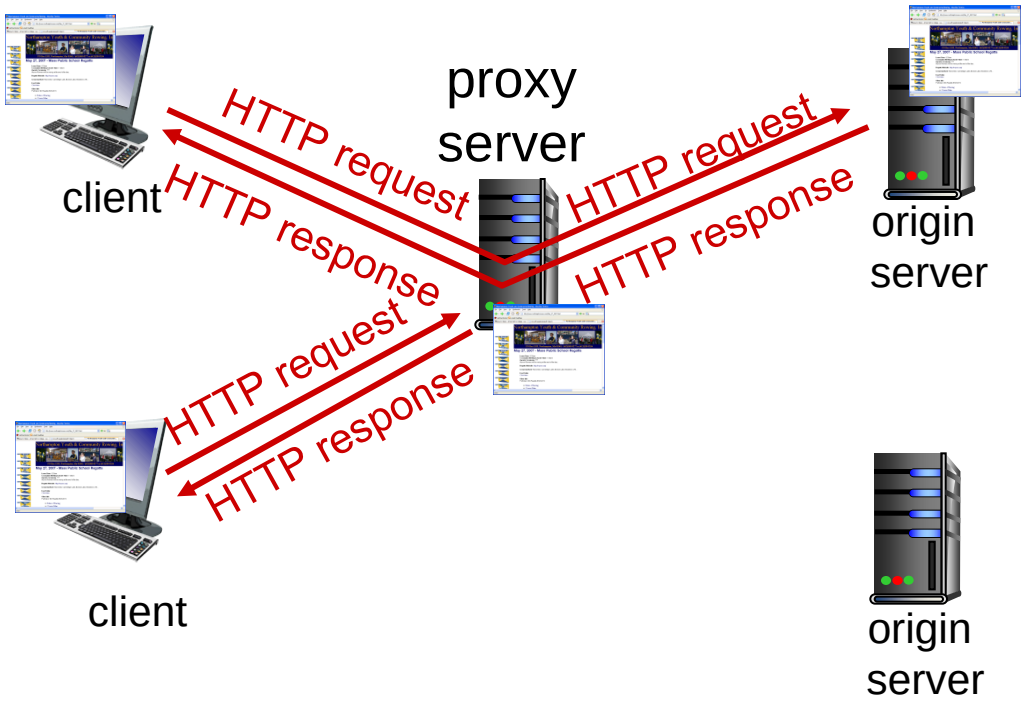
- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Advertising, insights and measurement	Things like Cookies and similar technologies (such as information about your device or a pixel on a website) are used to understand and deliver ads, make them more relevant to you, and analyze products and services and the use of those products and services. For example, we use cookies so we, or our affiliates and partners, can serve you ads that may be interesting to you on Facebook Services or oth websites and mobile applications. We may also use a cookie to learn whether someone who was served an ad on Facebook Services later makes a purchase on the advertiser's site or installs the advertised app. Similarly, our partners may use a cookie or another similar technology to determine whether we've served an ad and how it performed or provide us with information about how you interact with them. We also may work with an advertiser or its marketing partners to serve you an ad on or off Facebook Services, such as after you've visited the advertiser's site or app, or show you an ad based on the websites you visit or the apps you use – all across the Internet and mobile ecosystem. We also may use Cookies to provide advertisers with insights about the people who see and interact with their ads, visit their websites, and use their apps. Learn more about the information we receive, how we decide which ads to show you on and off Facebook Services, and the controls available to you.

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
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client



More about Web caching

why Web caching?

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

- typically cache is installed by ISP (university, company, residential ISP)
- cache acts as both client and server
  - server for original requesting client
  - client to origin server

Caching example:

assumptions:

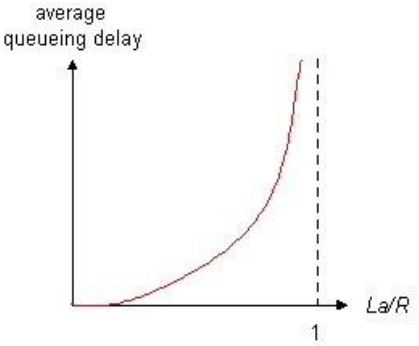
- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

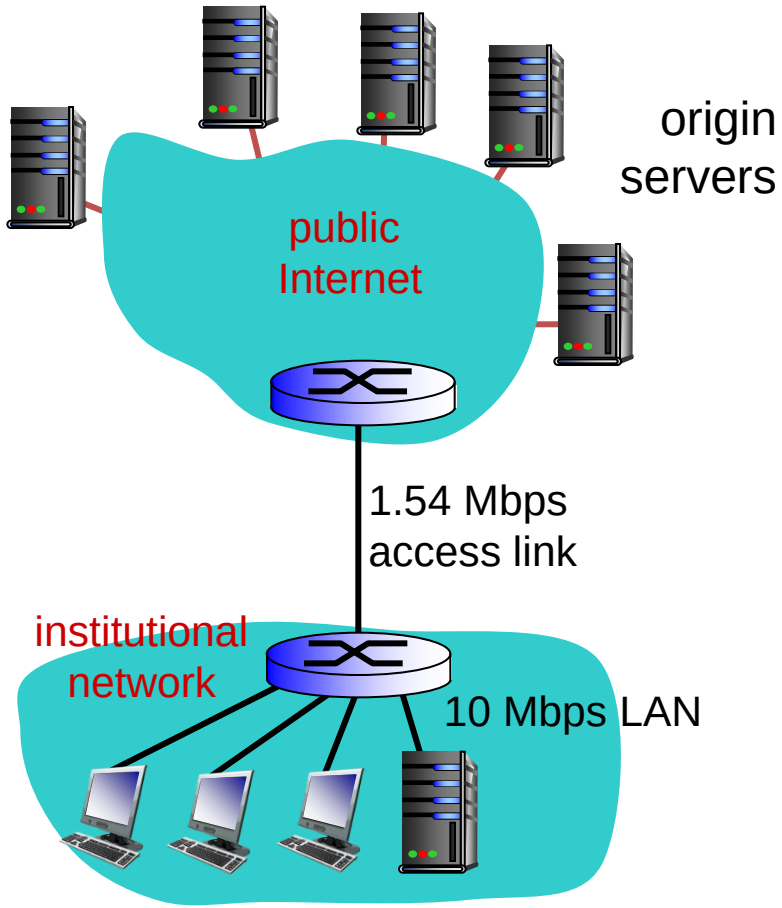
consequences:

- LAN utilization: 15%
- access link utilization = 99%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + usecs

problem!

Recall:





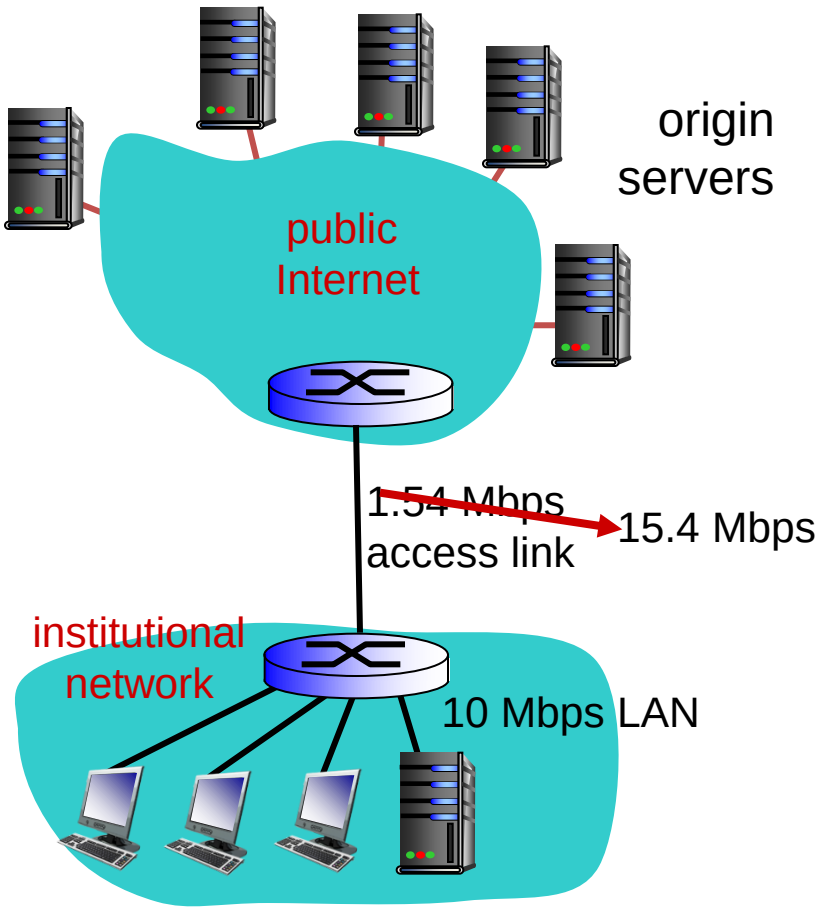
Caching example: fatter access link

assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: ~~1.54 Mbps~~ 15.4 Mbps

consequences:

- LAN utilization: 15%
- access link utilization = ~~99%~~ 9.9%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + ~~minutes~~ + usecs
  msecs



Cost: increased access link speed (not cheap!)

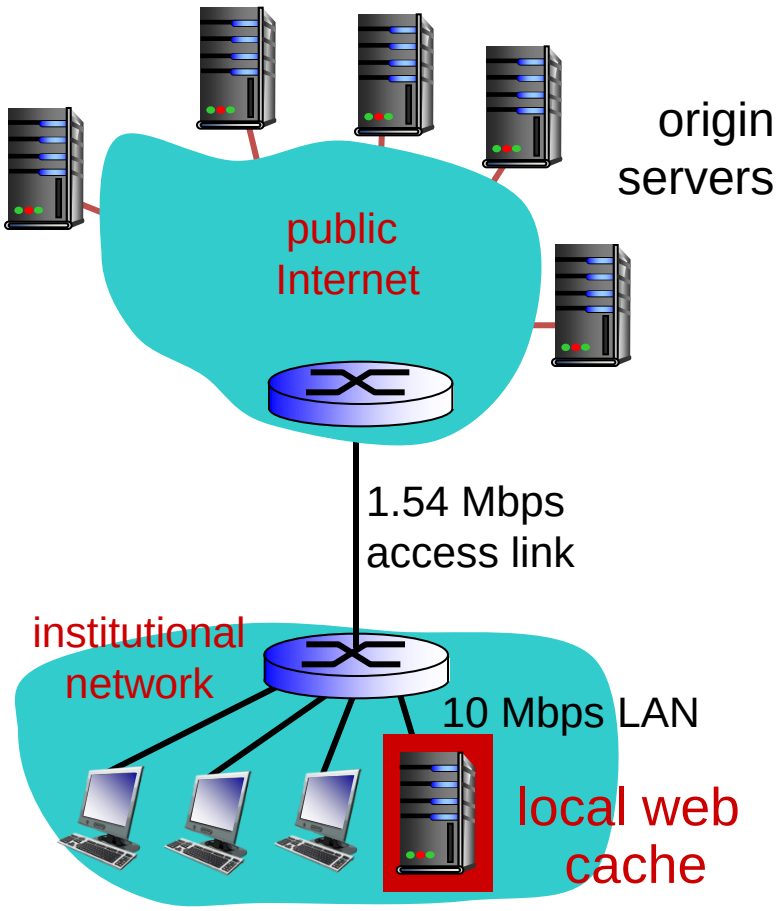
NS Activity 5.3: Impact of using local web cache

assumptions:

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

consequences:

- Assume
  - cache hit rate is 0.4
  - Delay when request satisfied from local cache is ~msecs
  - Access delay @ <70% util. = ~ms
- What is
  - Access link utilization?
  - Total delay?



Caching example: install local cache

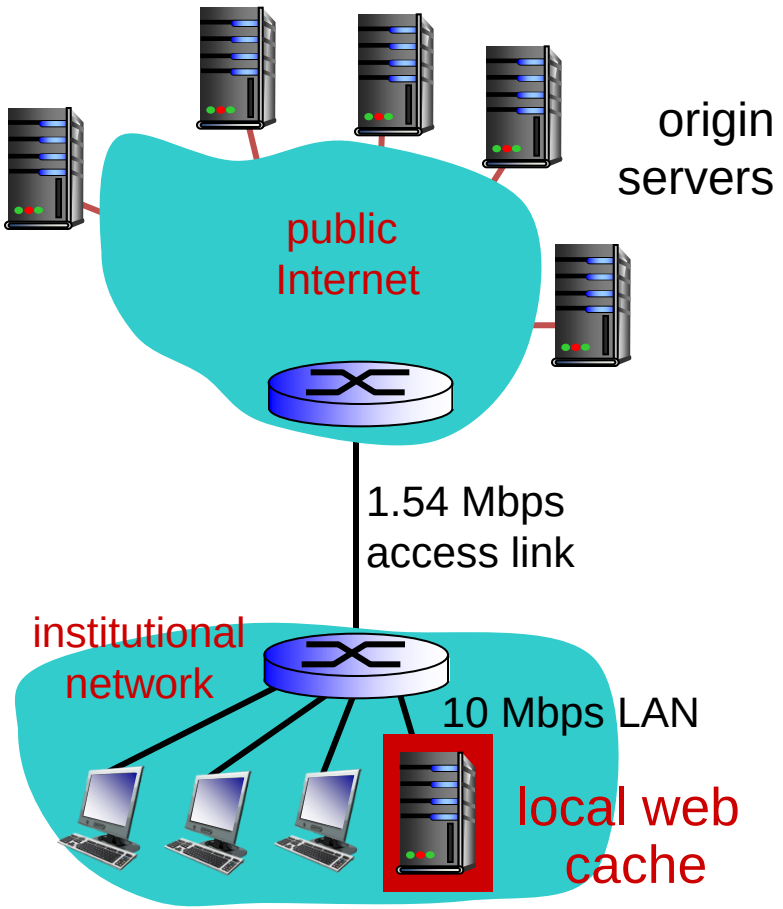
Calculating access link utilization, delay with cache:

- suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin

- access link utilization:

- data rate to browsers over access link

- total delay



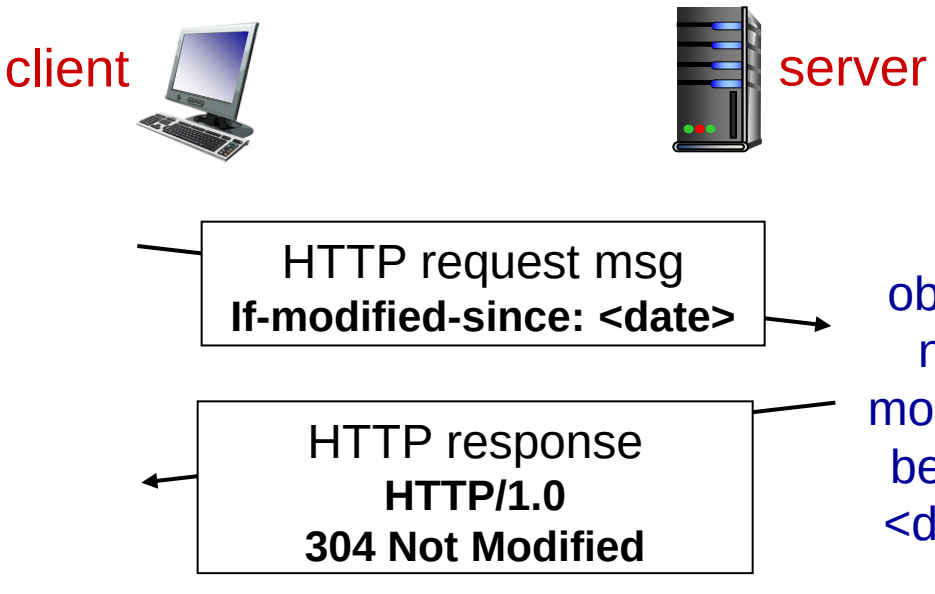
Cost: web cache (cheap!)

Conditional GET

- *Goal:* don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- *cache:* 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`



NS Activity 5.4: Wireshark and HTTP

- Wireshark: packet sniffer and analyzer
  - Used in NS Lab 3 already for DNS traces
  - Capture (or "sniff") packets from physical link
  - Sufficient to sniff your own packets
  - Has "promiscuous" mode: use w/ permission!
- From wireshark, open recorded trace:
  - http-ethereal-trace-3
- See displayed list of captured packets
  - Some fields: time of capture, source, destination, protocol, packet length, synopsis of payload
  - Click header field label to sort in order of that field
  - From "View", can turn on "packet details", can further explode each layer's view

Activity 5.4 (cont'd)
http-ethereal-trace-3

- Select earliest HTTP packet
  - Was it request or response? What type? What's HTTP version?
  - Turn on "packet details", then expand HTTP view
    - Identify the requesting browser and some of its settings
- How long did it take for the HTTP request to complete?
- Was the request successful? If so, what was the type of the web object returned? How big was the returned object? If not, what was the error condition?

Activity 5.4 (cont'd)
http-ehtereal-trace-4

- Now open http-ehtereal-trace-4
- What's the URL of the HTML web page being retrieved? What version of HTTP was used?
- How many embedded objects did the HTML page contain? What kinds of objects were they respectively?
- Did the web client use new TCP connections for retrieving the embedded objects? [*Hint*: Look for any TCP SYN message that precedes an HTTP GET to the web server]
  - Why were new connections used or not? [*Hint*: Examine the HTTP GET requests to identify the web server for each object]
- Were parallel connections or pipelining used?