COMPUTER NETWORKS

TEAM NETWORKS

Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Department of Computer Science and Engineering

Unit – 1 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

Application Layer: Overview

Our goals:

- Conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm

- Learn about protocols by examining popular application-layer protocols
  - HTTP

Some Network Apps

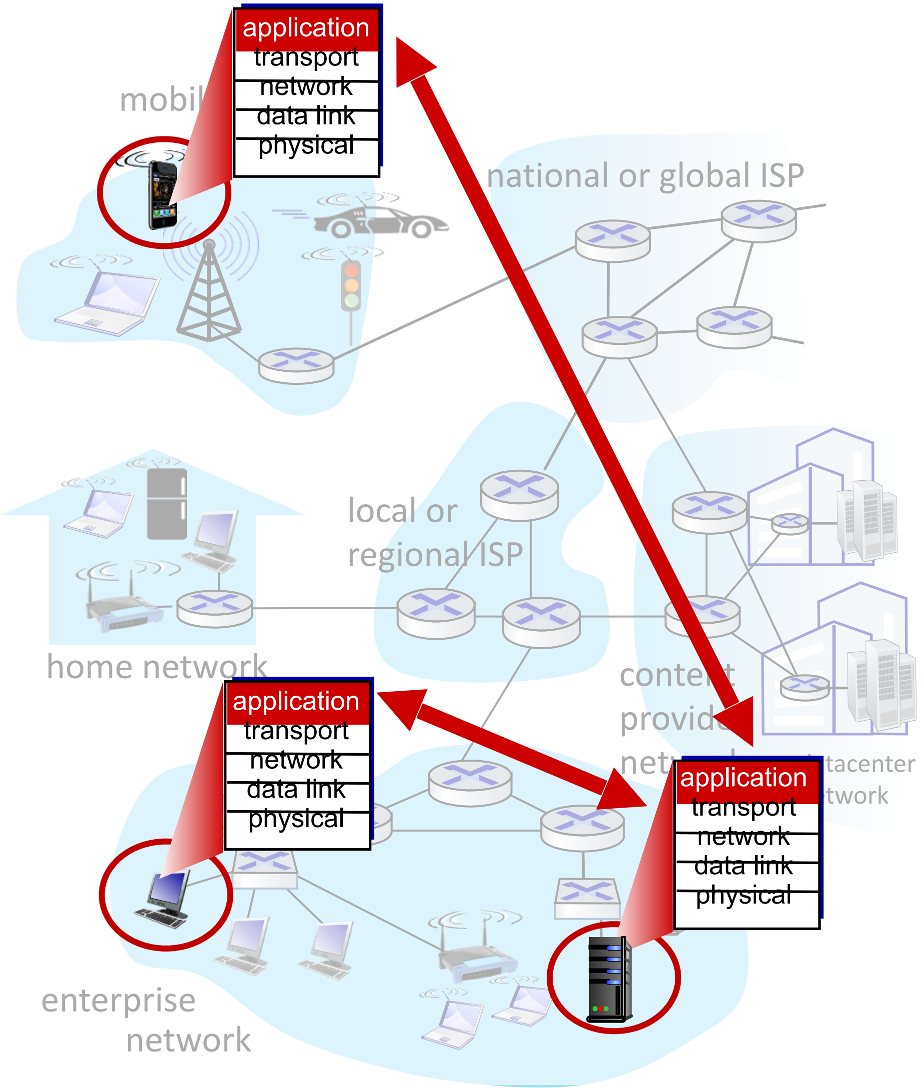
- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- P2P file sharing
- voice over IP
- real-time video conferencing (e.g., Skype, Hangouts)
- Internet search
- remote login
- …

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



COMPUTER NETWORKS

Client-Server Paradigm

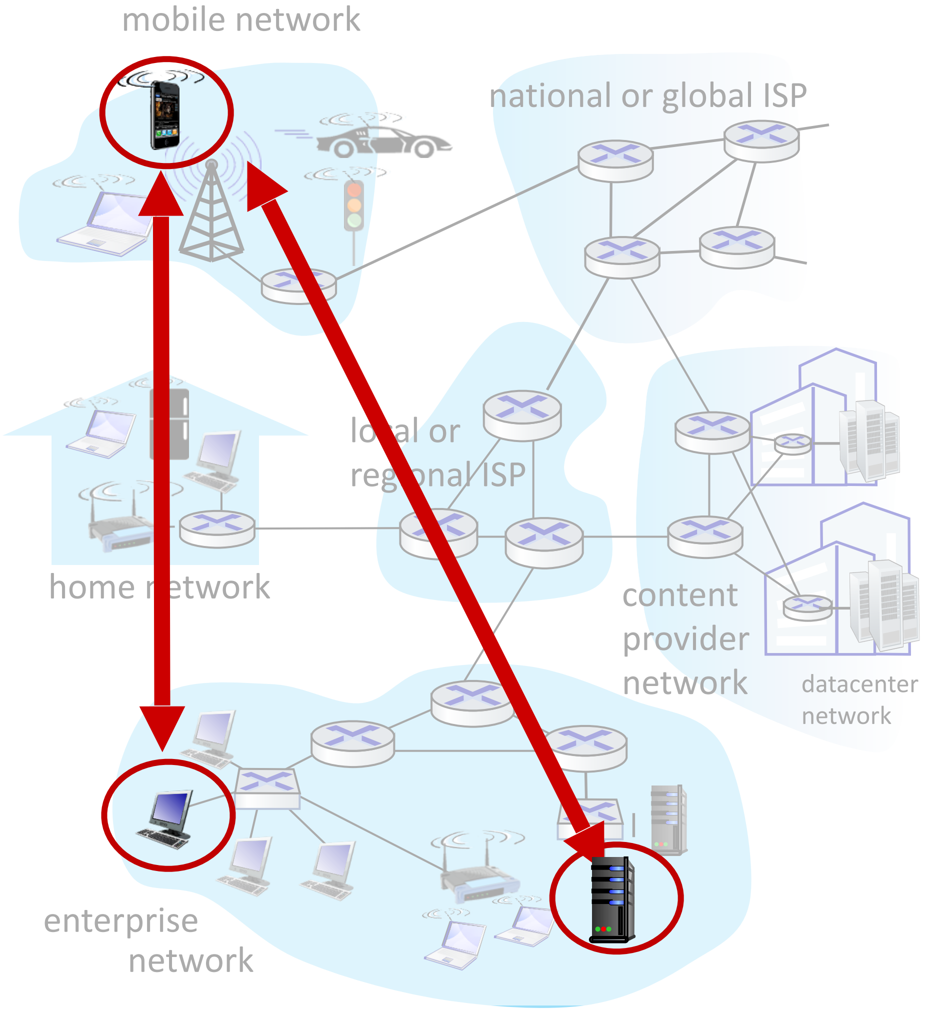
server:

- always-on host
- permanent IP address
- often in data centers, for scaling

clients:

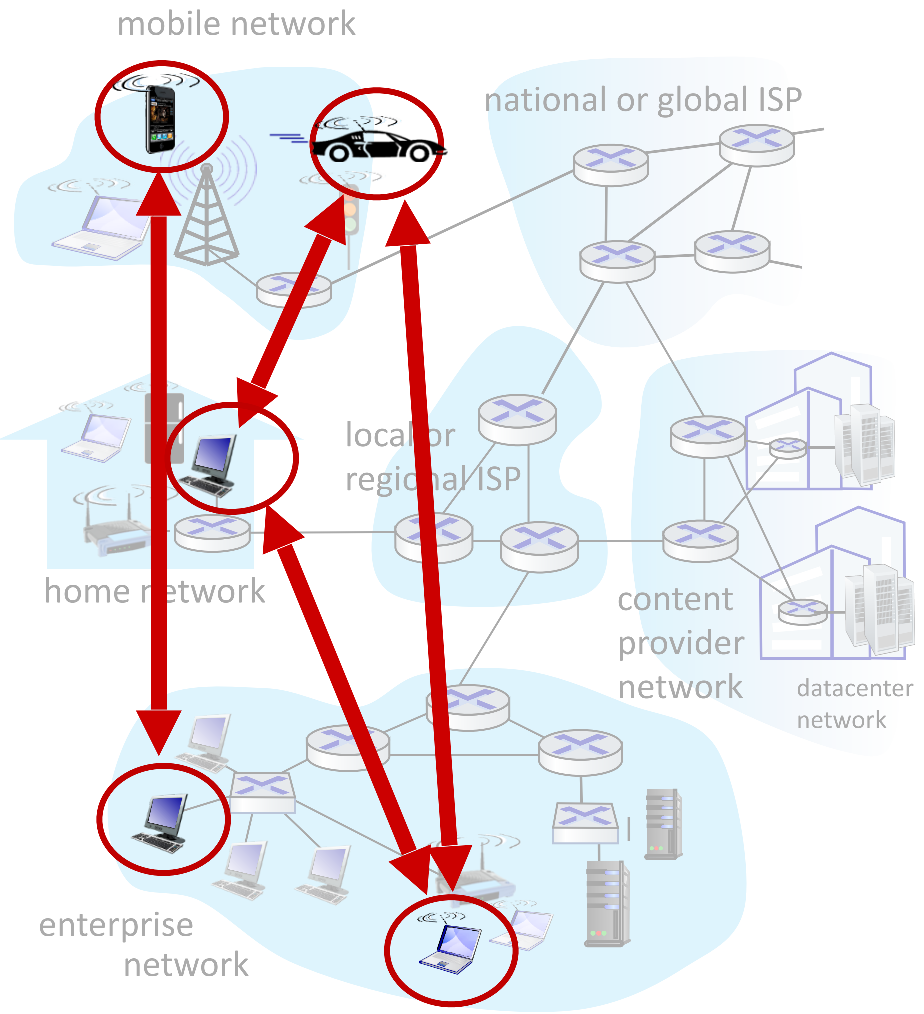
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP



Peer-to-Peer 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
- example: P2P file sharing



Processes Communicating

PES UNIVERSITY
CELEBRATING 50 YEARS

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

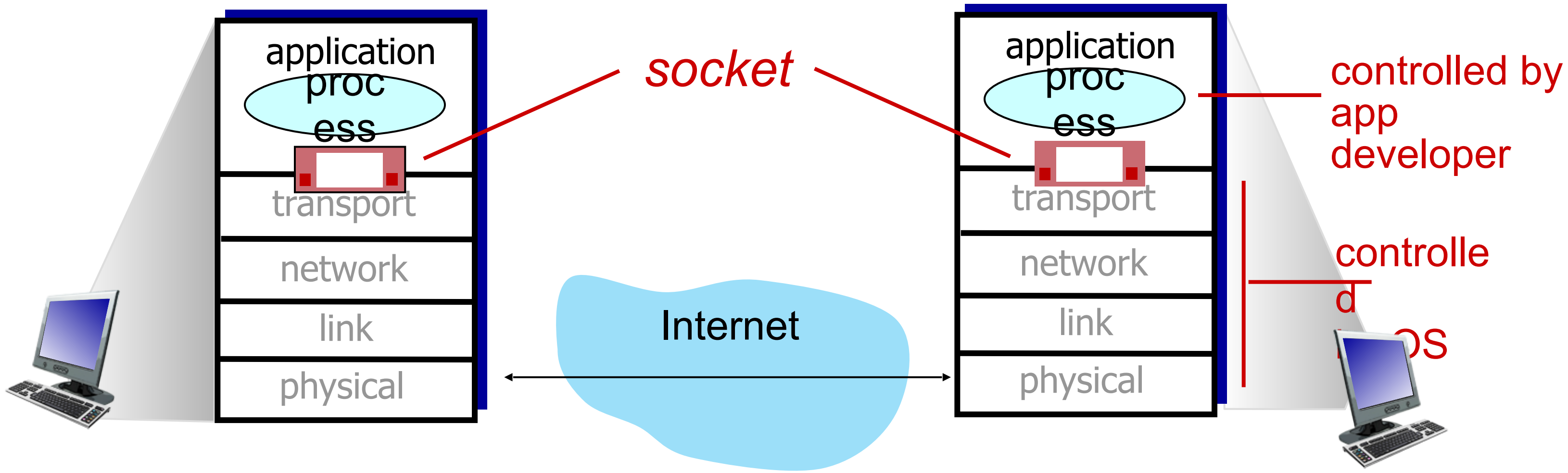
client process: process that initiates communication

server process: process that waits to be contacted

- note: 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



Addressing Processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *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…

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

What transport service does an App needs?

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

security

- encryption, data integrity, …

Transport service requirements: common apps

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

Internet Transport Protocol 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.

Internet Transport Protocol Services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (Skype)	TCP or UDP
streaming audio/video	DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Unit – 1 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

First, a quick review...

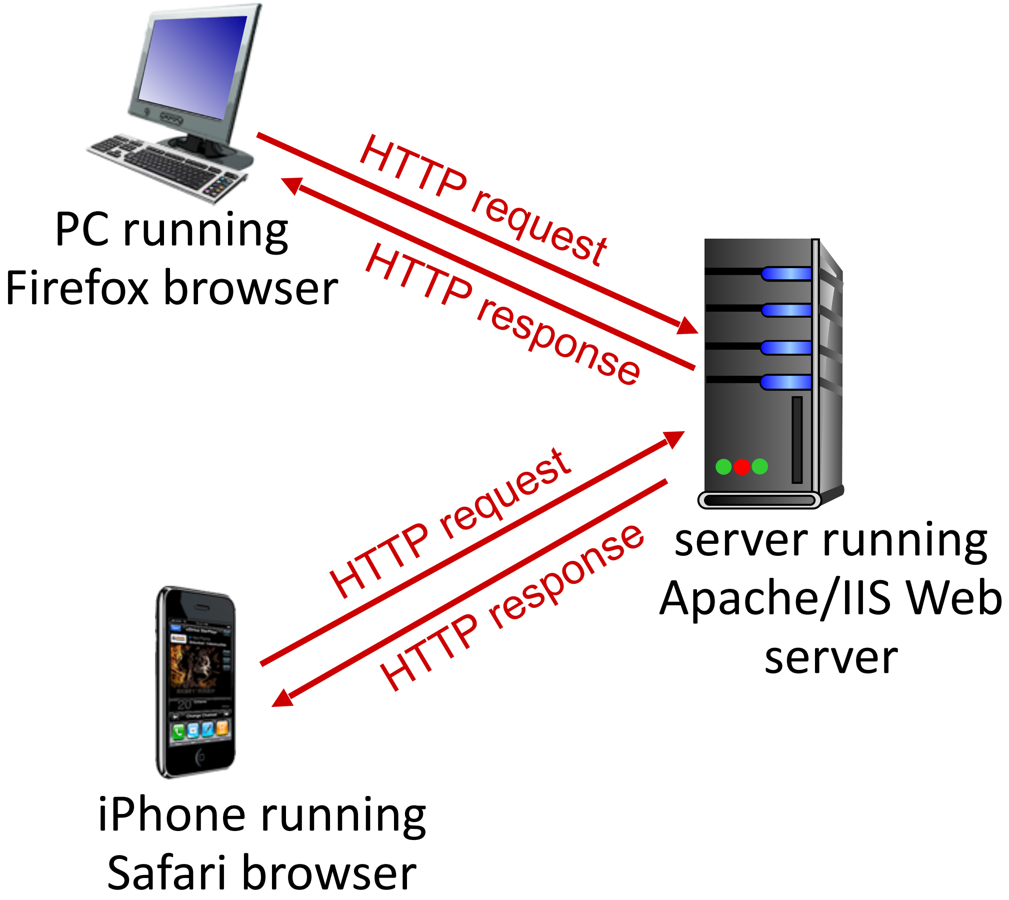
- web page consists of *objects,* each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,...
- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL,* e.g.,
- If a Web page contains HTML text and 5 JPEG images, then the Web page has 6 objects: the base HTML file plus the 5 images.

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

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



Defined in RFC 1945; RFC 2616

HTTP Overview (more)

HTTP 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

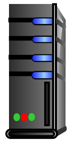
Persistent HTTP

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

Non-persistent HTTP: example

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(base HTML file containing 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

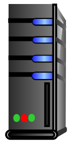
2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index

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

time

Non-persistent HTTP: example (more)

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing text, references to 10 jpeg images)



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

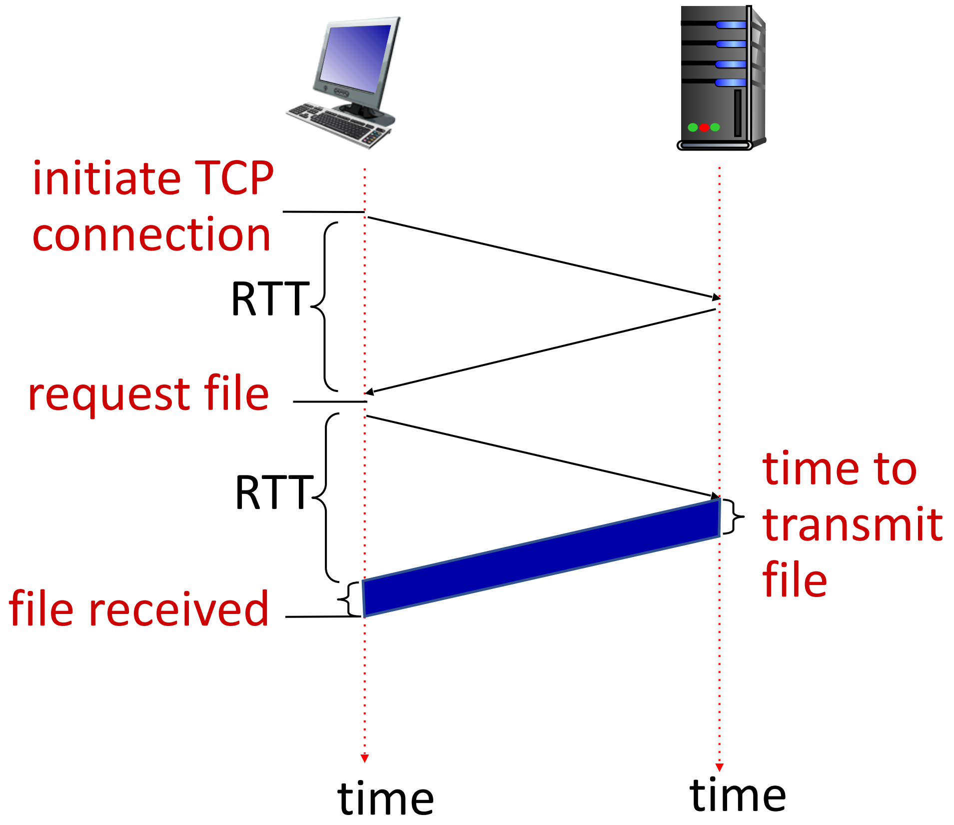
time

Non-persistent HTTP: response time

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
- obect/file transmission time



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

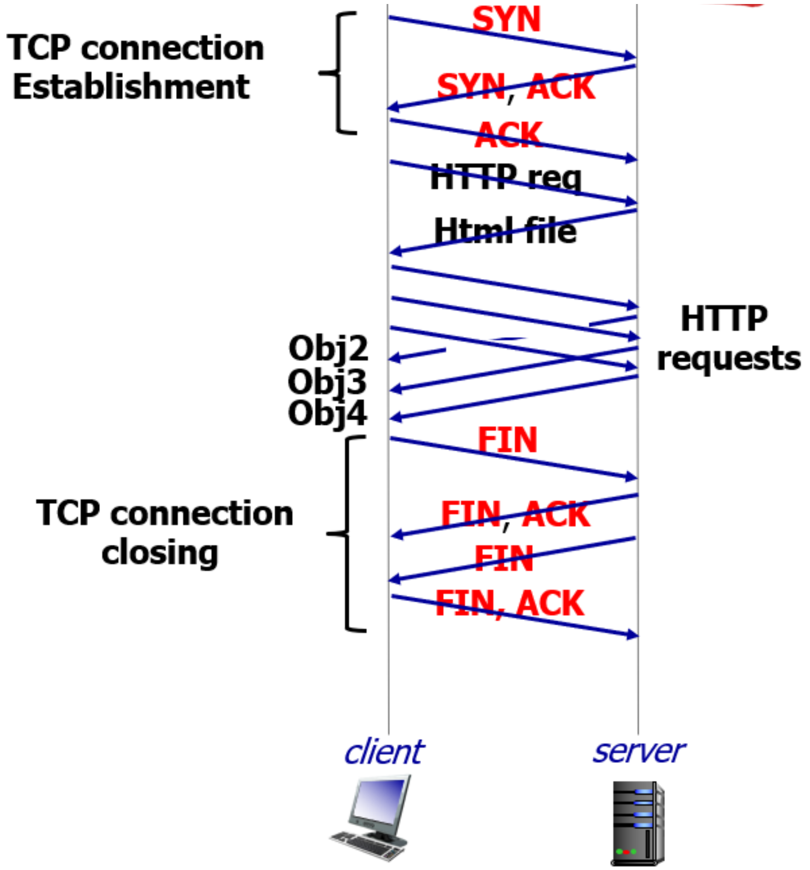
Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:

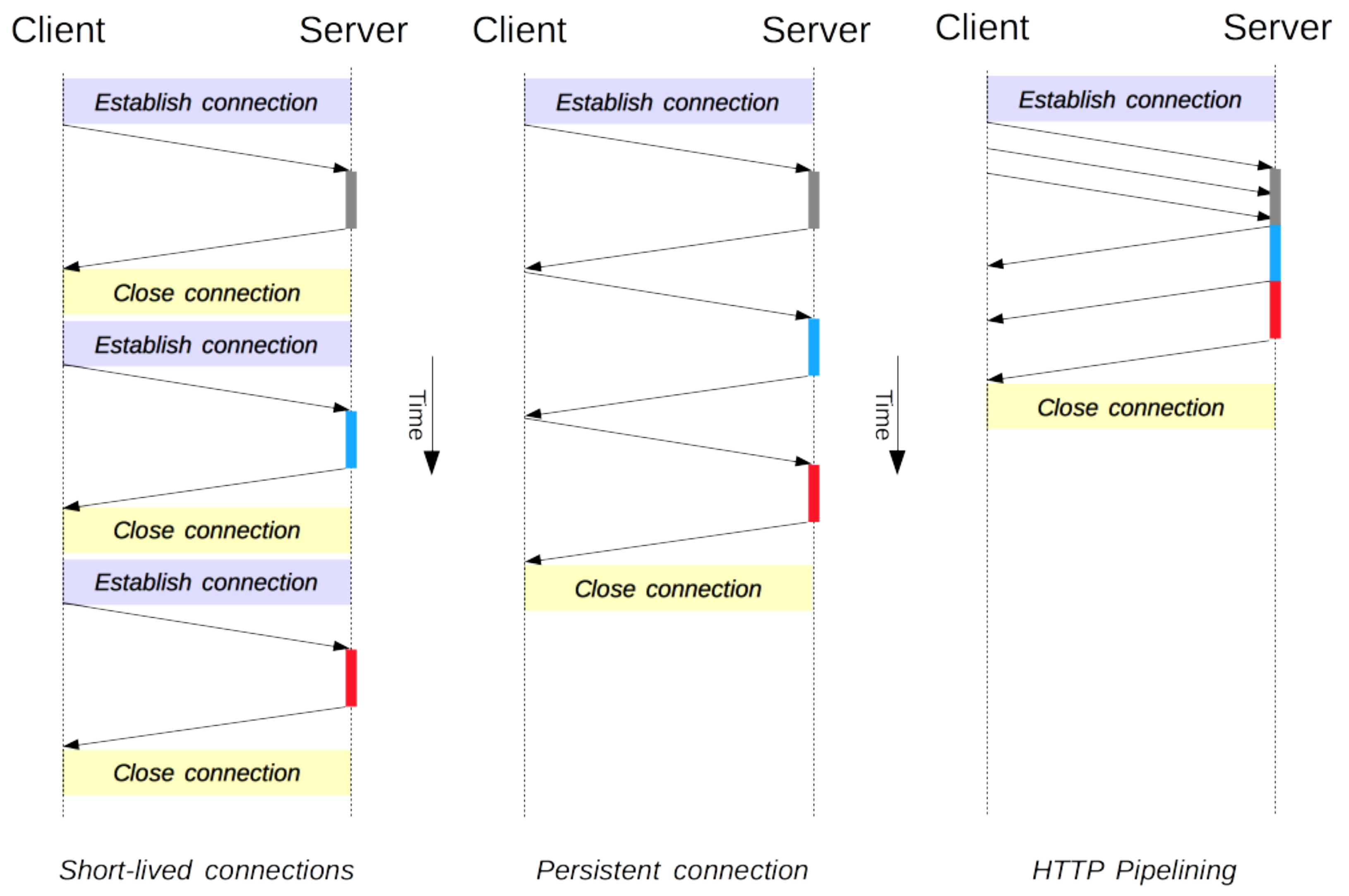
- requires 2 RTTs per object
- OS overhead for *each* TCP connection (TCP buffer and variables)
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

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)



COMPUTER NETWORKS
Connection Management in HTTP/1.x



Unit – 1 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

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)

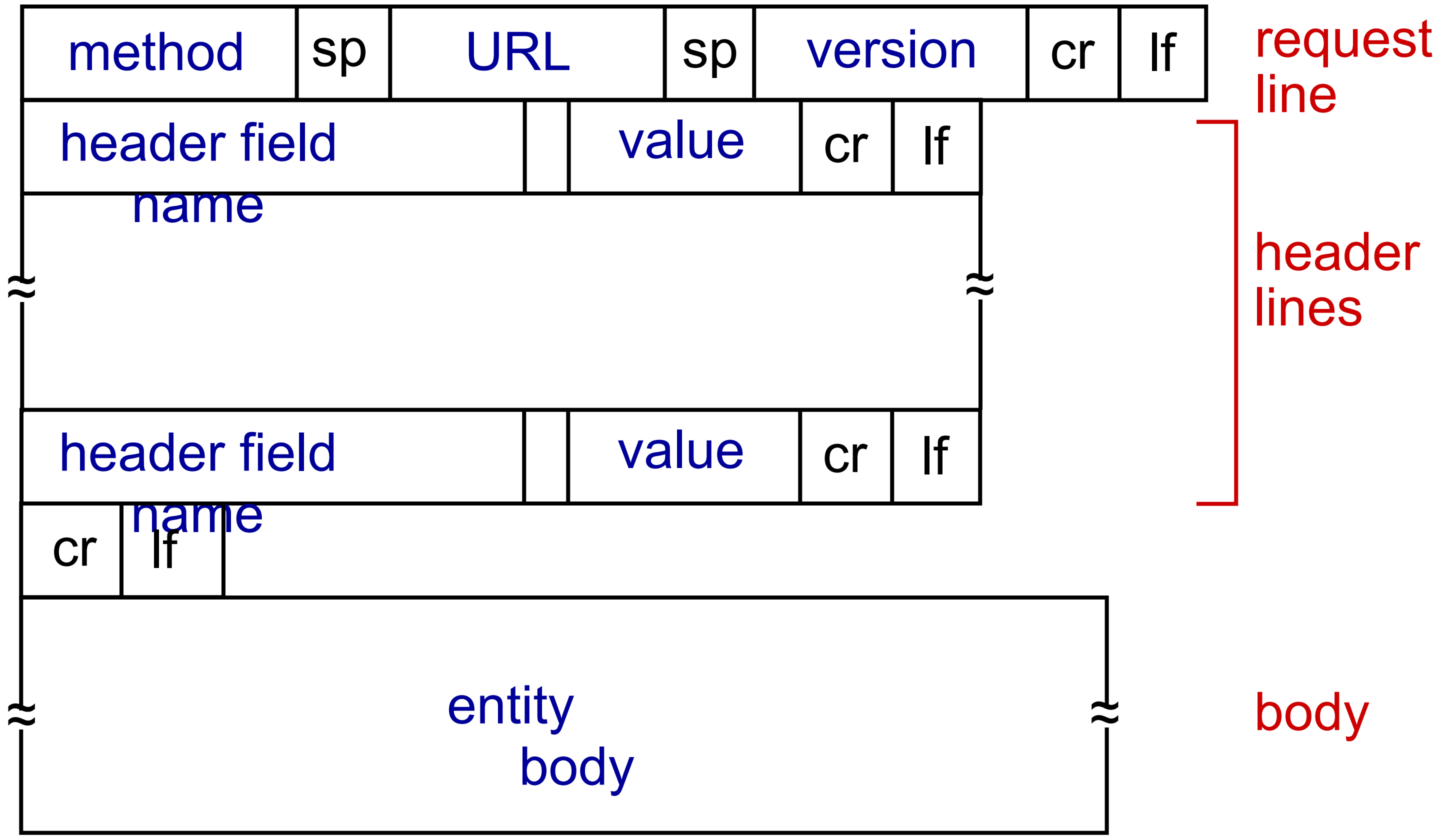
header lines

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

```
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
```

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

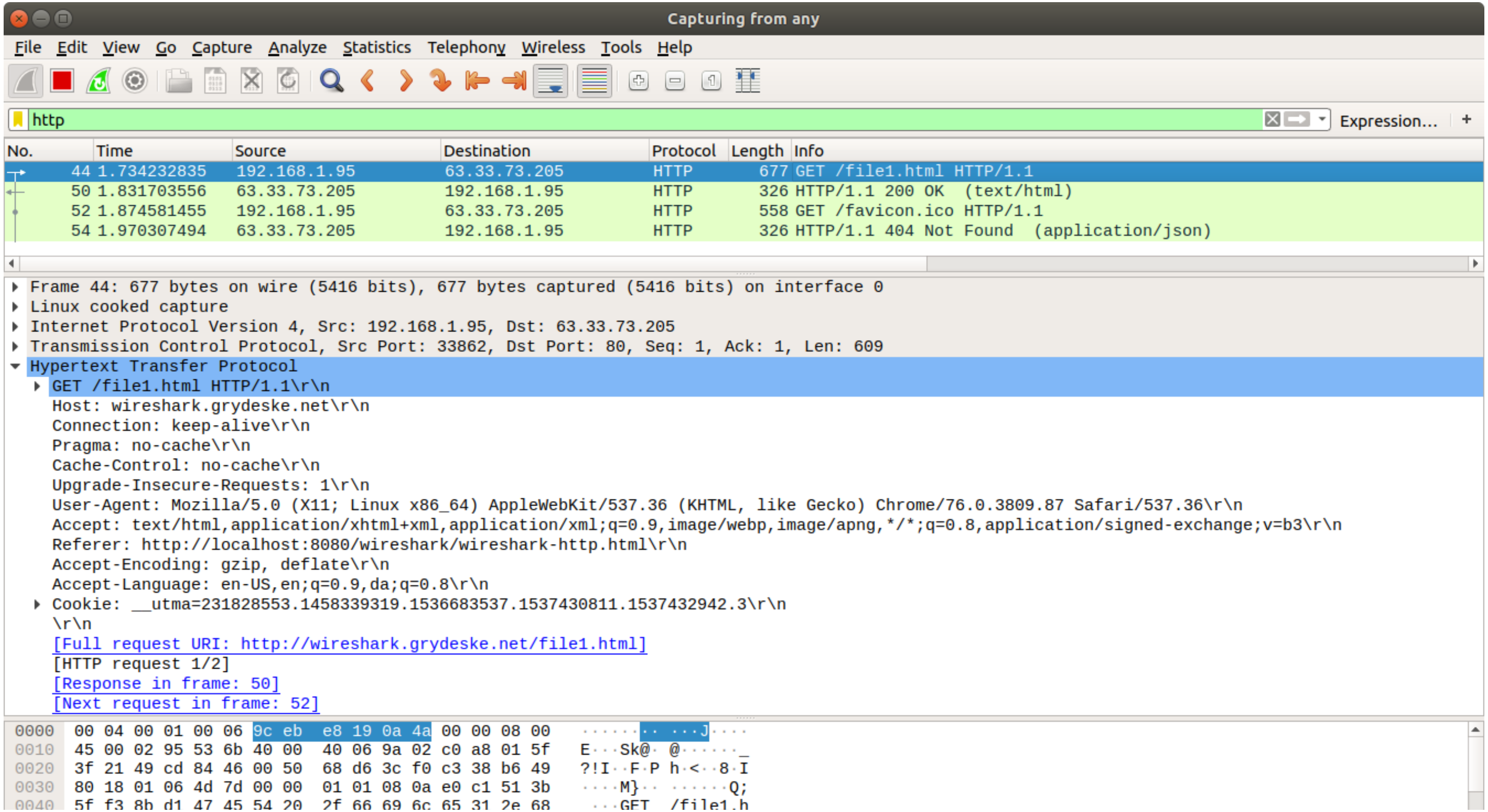
HTTP Request Message: General Format



HTTP specifications [RFC 1945; RFC 2616; RFC 7540]

COMPUTER NETWORKS

HTTP Request Message – Wireshark Capture



Other HTTP Request Messages

POST method:

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

HEAD method:

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

GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

www.somesite.com/animalsearch?monkeys&banana

HTTP Response Message

status line (protocol status code status phrase)

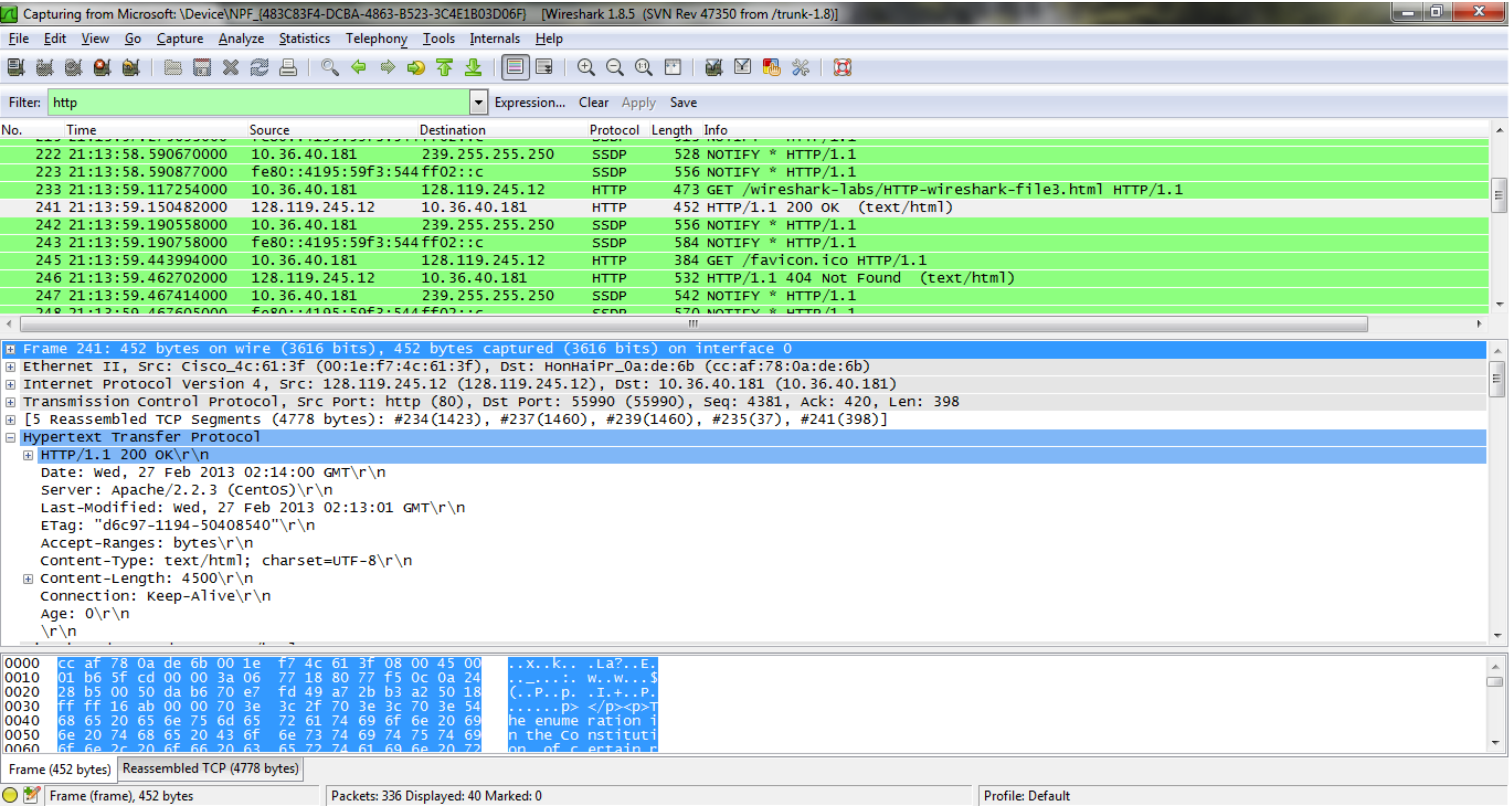
header lines

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 ...
```

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP Response Message – Wireshark Capture



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 message

301 Moved Permanently

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

400 Bad Request

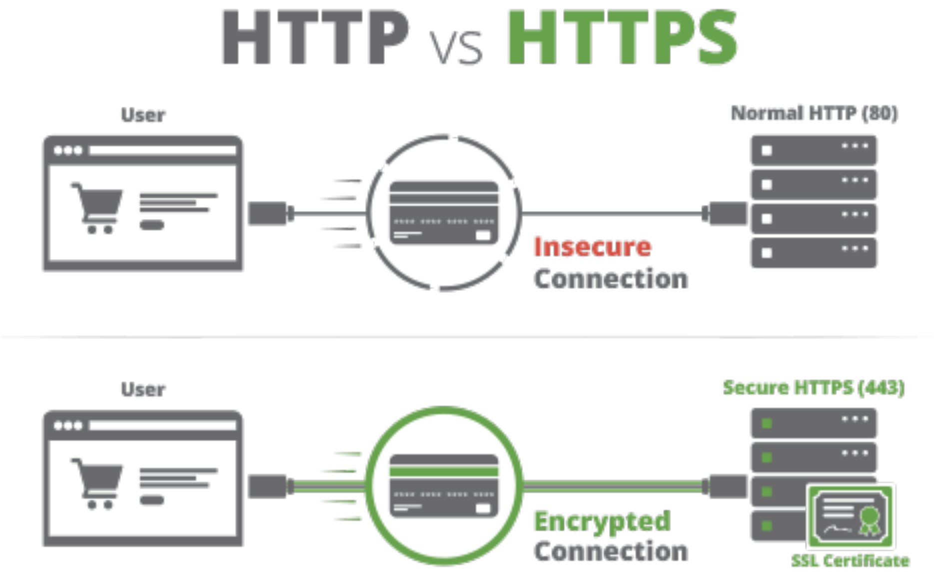
- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

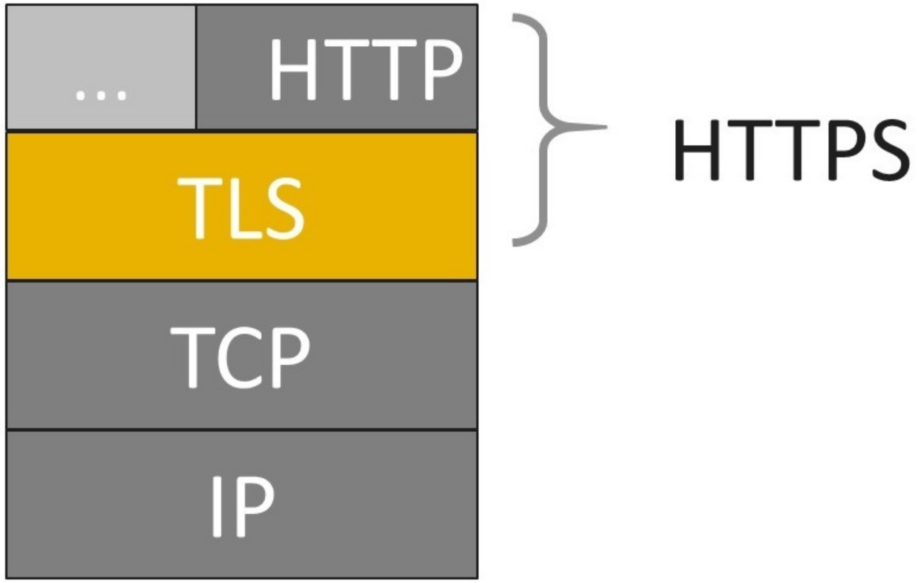
HTTP vs HTTPS



- HTTPS is HTTP with encryption – All communications between browser and server are encrypted (bi-directional).
- ‘S’ refers ‘Secure’ or HTTP over Secure Socket Layer.
- Uses TLS (SSL) to encrypt normal HTTP requests and responses.
- Attackers can't read the data crossing the wire and you know you are talking to the server you think you are talking too.

HTTP vs HTTPS (more)

- HTTP + TLS -> Encrypted
- Uses port no. 443 for data communication.
- HTTPS is based on public/private-key cryptography.
  - The public key is used for encryption
  - The secret private key is required for decryption.
- SSL certificate is a web server's digital certificate issued by a third party CA.
  - Create an encrypted connection and establish trust.
- Is my certificate SSL or TLS?



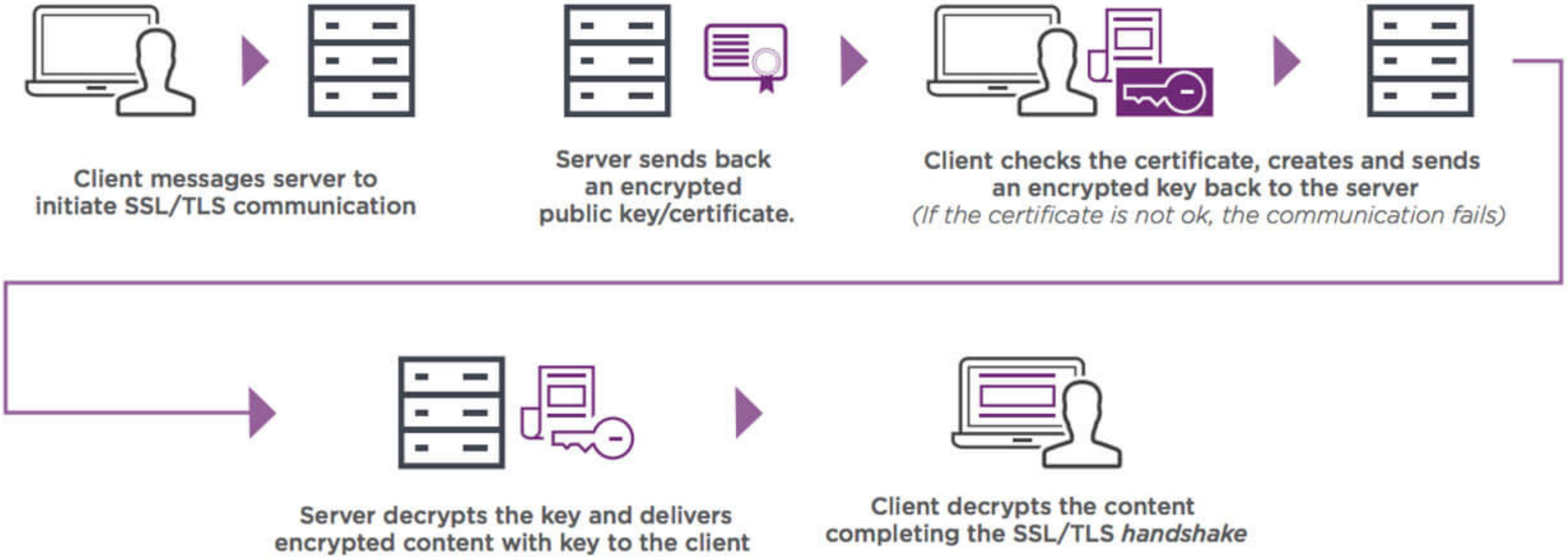
Any message encrypted with Bob's public key can be only decrypted with Bob's private key.

How does SSL works?

- Step 1: Browser requests secure pages (HTTPS) from a server.
- Step 2: Server sends its public key with its SSL certificate (digitally signed by a third party – CA).
- Step 3: On receipt of certificate, browser verifies issuer's digital signature. (green padlock key)
- Step 4: Browser creates a symmetric key (shared key), keeps one and gives a copy to server. Encrypts it using server's public key.
- Step 5: On receipt of encrypted secret key, decrypts it using its private key and gets browser's secret key.

- Asymmetric and Symmetric key algorithms work together.
- Asymmetric key algorithm – verify identity of the owner & its public key -> Establish trust.
- Once connection is established, Symmetric key algorithm is used to encrypt and decrypt the traffic.

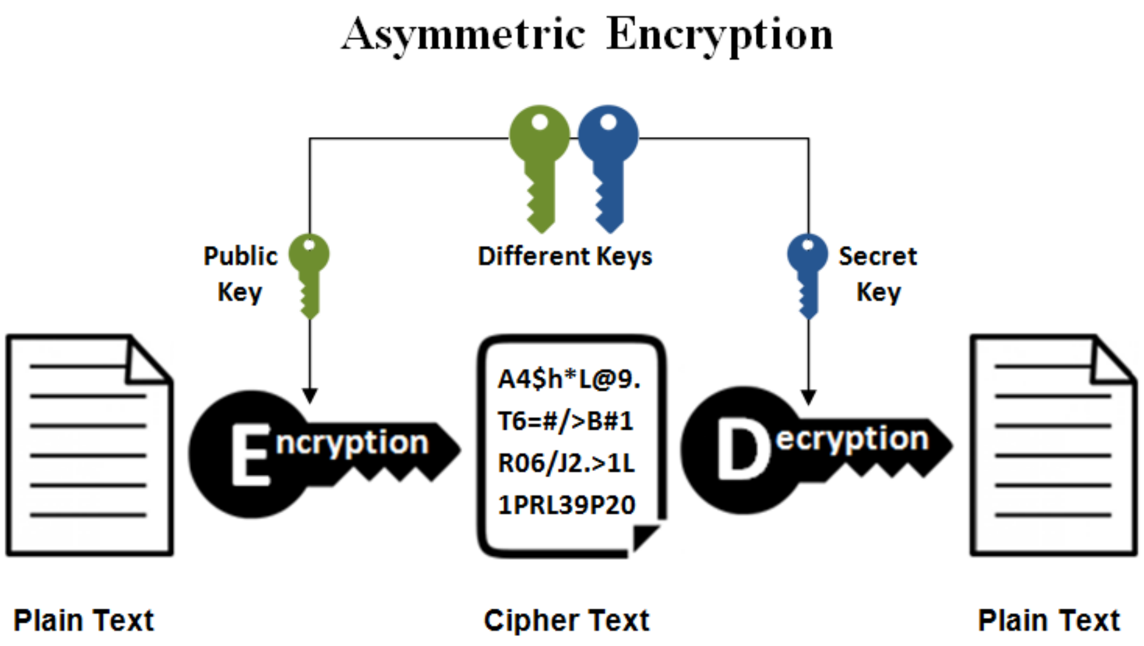
How does SSL works?

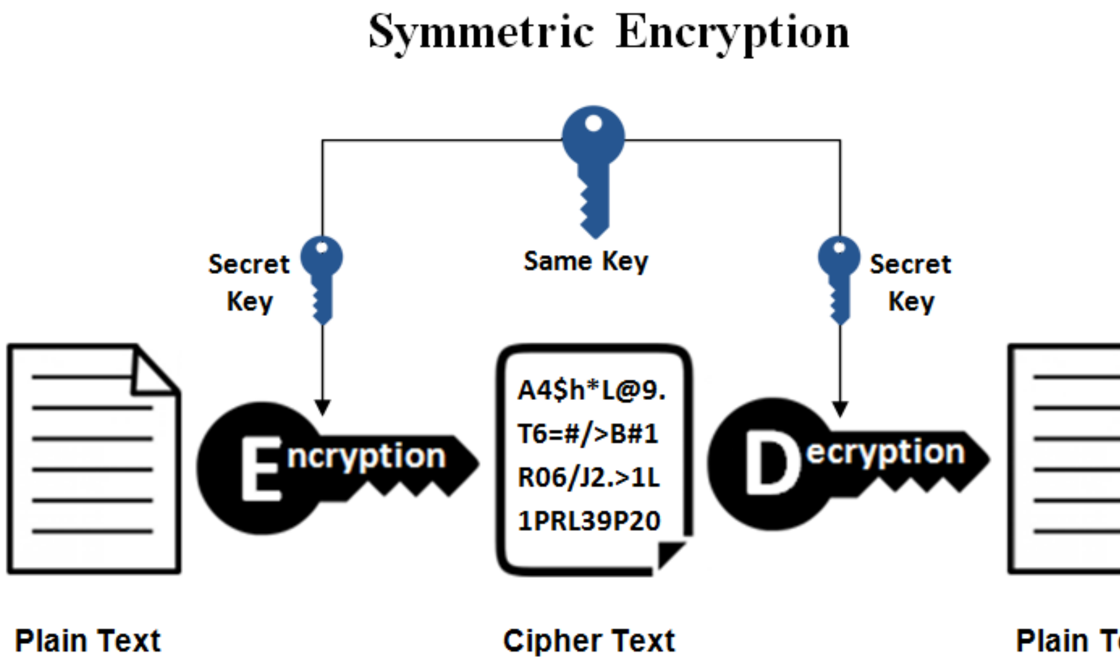


Benefits of HTTPS over HTTP using SSL Certificates

- Stronger Google ranking.
- Updated browser labels.
- Improved security.
- Increased customer confidence / safer experience.
- Build customer trust and improve conversions.

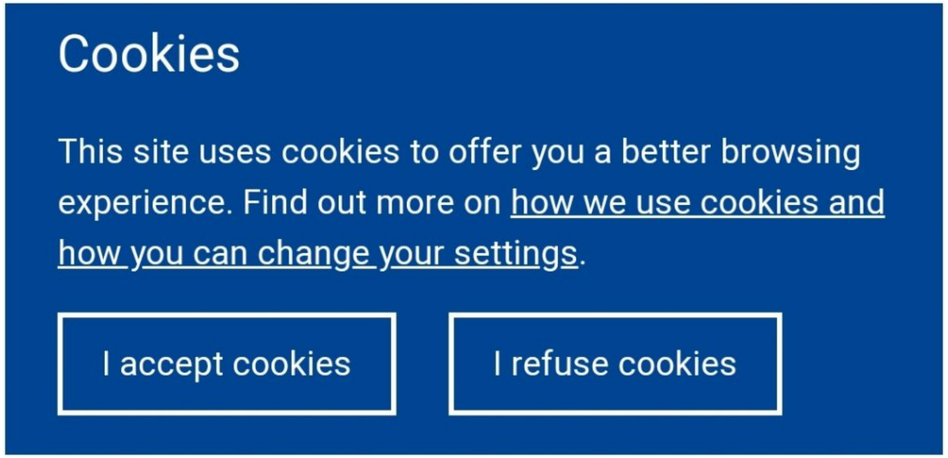
Benefits of HTTPS over HTTP using SSL Certificates

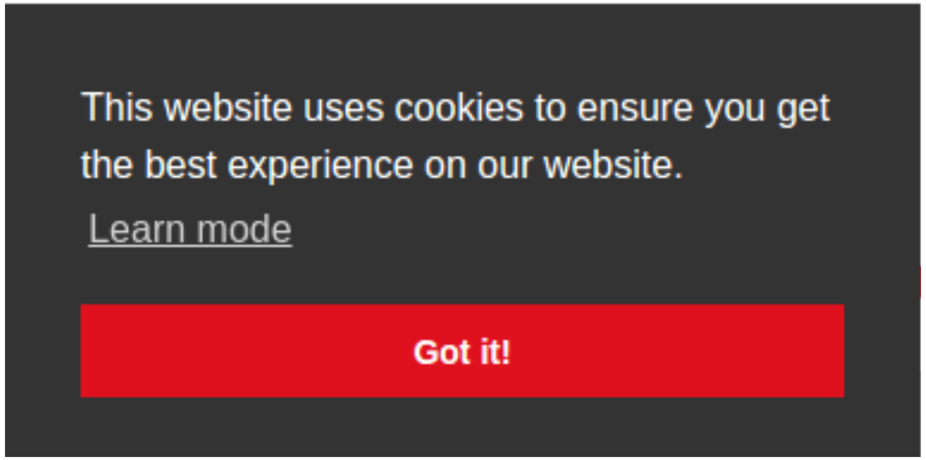




Cookies

- Website/HTTP/Internet cookies
- Piece of data from a specific website
- Stored on a user’s computer
- Allows sites to keep track of users
- Eg: language selection



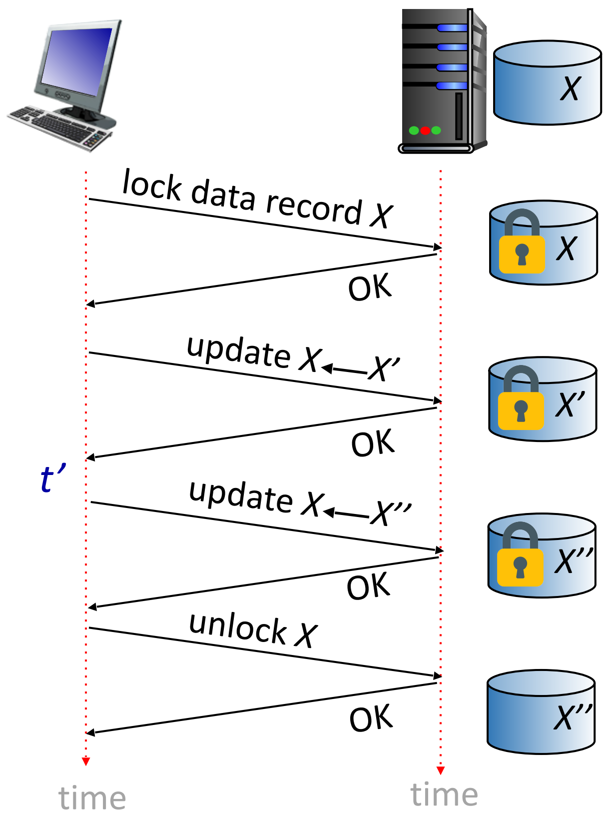


Maintaining user/server state: cookies

Recall: HTTP GET/response interaction is *stateless*

- no notion of multi-step exchanges of HTTP messages to complete a Web "transaction"
  - no need for client/server to track "state" of multi-step exchange
  - all HTTP requests are independent of each other
  - no need for client/server to "recover" from a partially-completed-but-never-completely-completed transaction

a stateful protocol: client makes two changes to X, or none at all



Q: what happens if network connection or client crashes at *t'* ?

Maintaining user/server state: cookies (more)

Web sites and client browser use *cookies* to maintain some state between transactions

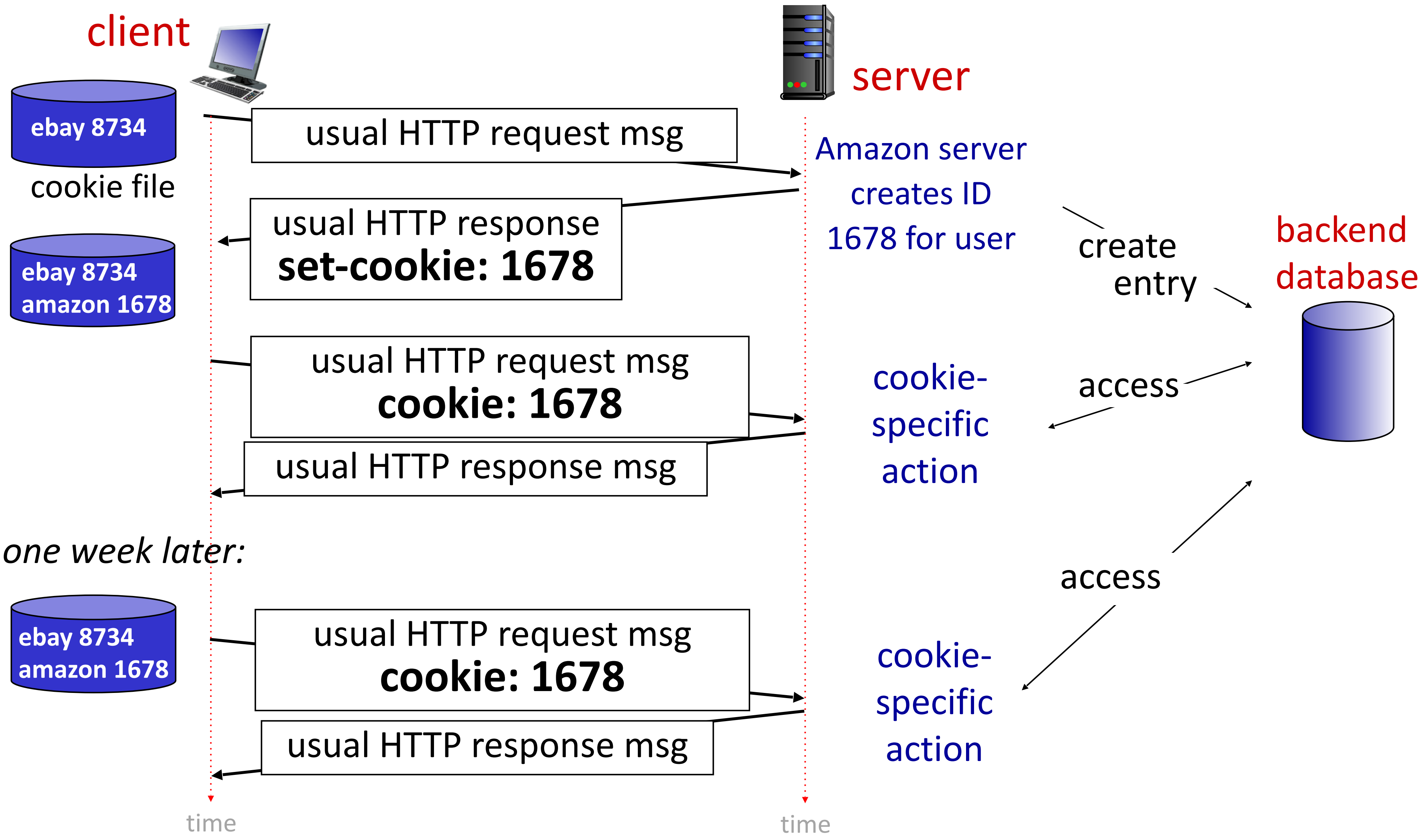
four components:

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 uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID (aka "cookie")
  - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

Maintaining user/server state: cookies (more)



HTTP Cookies: Comments

What cookies can be used for:

- track user's browsing history
- remembering login details
- track visitor count
- shopping carts
- recommendations
- save coupon codes for you

Challenge: How to keep state:

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

aside

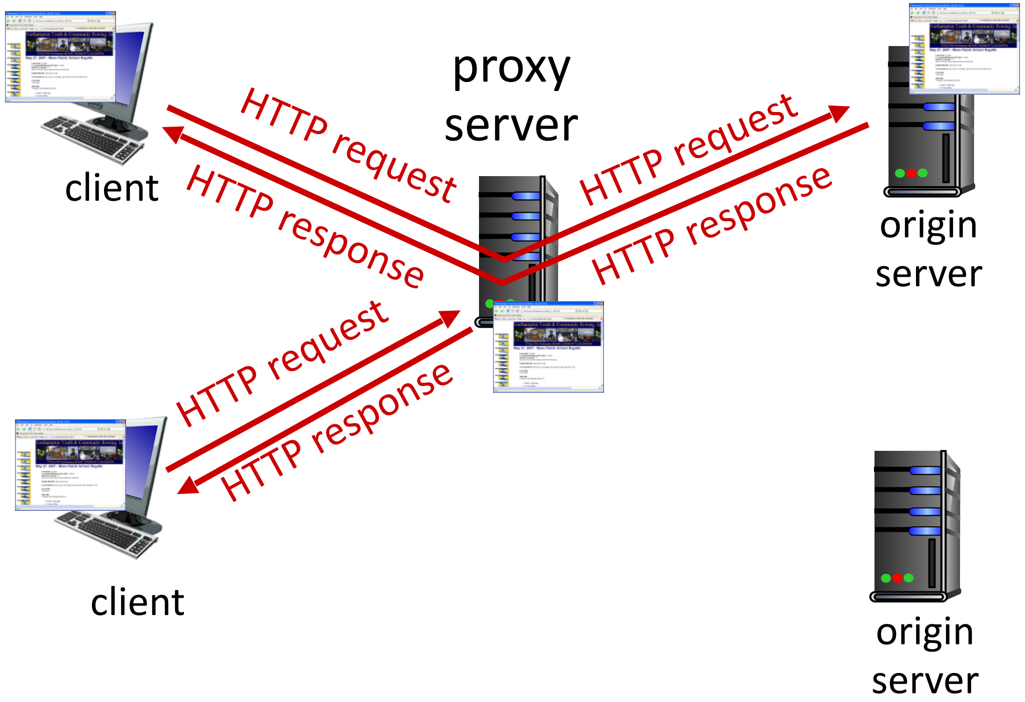
cookies and privacy:

- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

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



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 (speed)
  - cache is closer to client
- reduce traffic on an institution's access link (saves bandwidth)
- internet is dense with caches
  - enables "poor" content providers to more effectively deliver content
- privacy – surf the internet anonymously
- activity logging

Caching example

(15 req/sec) * (100 Kbits/req)/(1.54 Mbps) = 0.974

(15 req/sec) * (100 Kbits/req)/(1 Gbps) = 0.0015

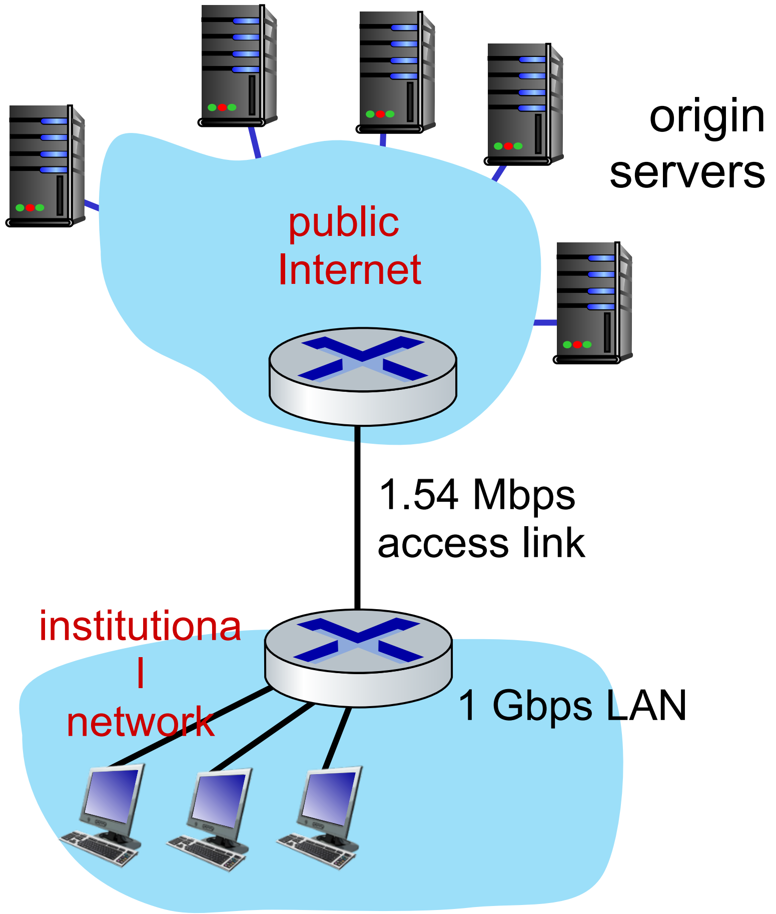
Scenario:

- 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!



Caching example: buy a faster access link

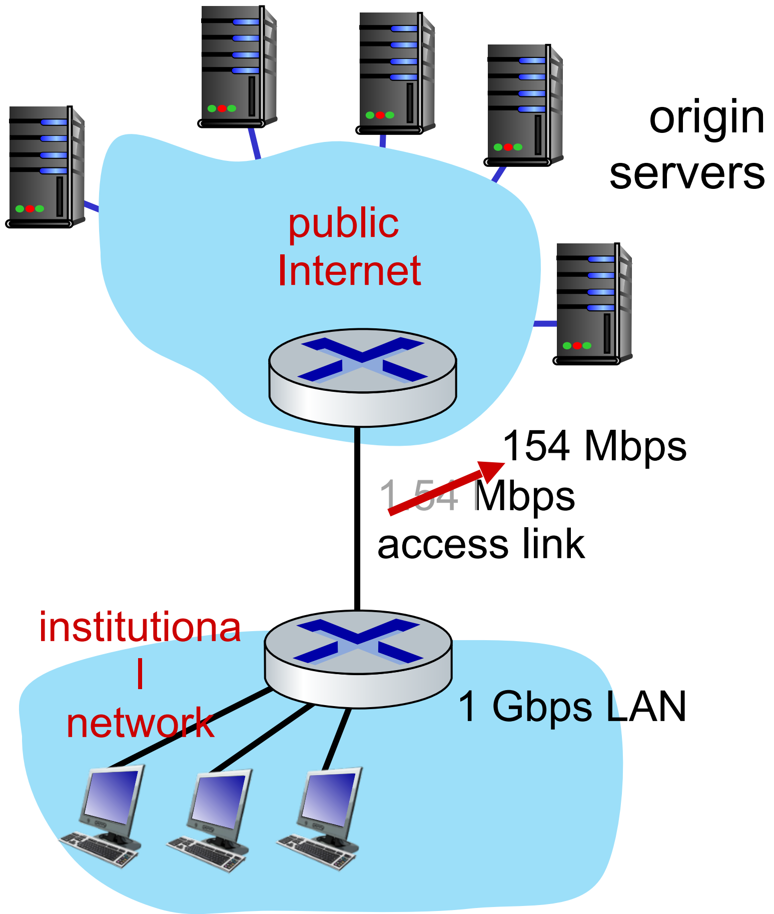
Scenario:

- access link rate: ~~1.54~~ Mbps → 154 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: .0015
- access link utilization = ~~.97~~ → .0097
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + ~~minutes~~ + usecs → msecs

Cost: faster access link (expensive!)



Caching example: install a web cache

Scenario:

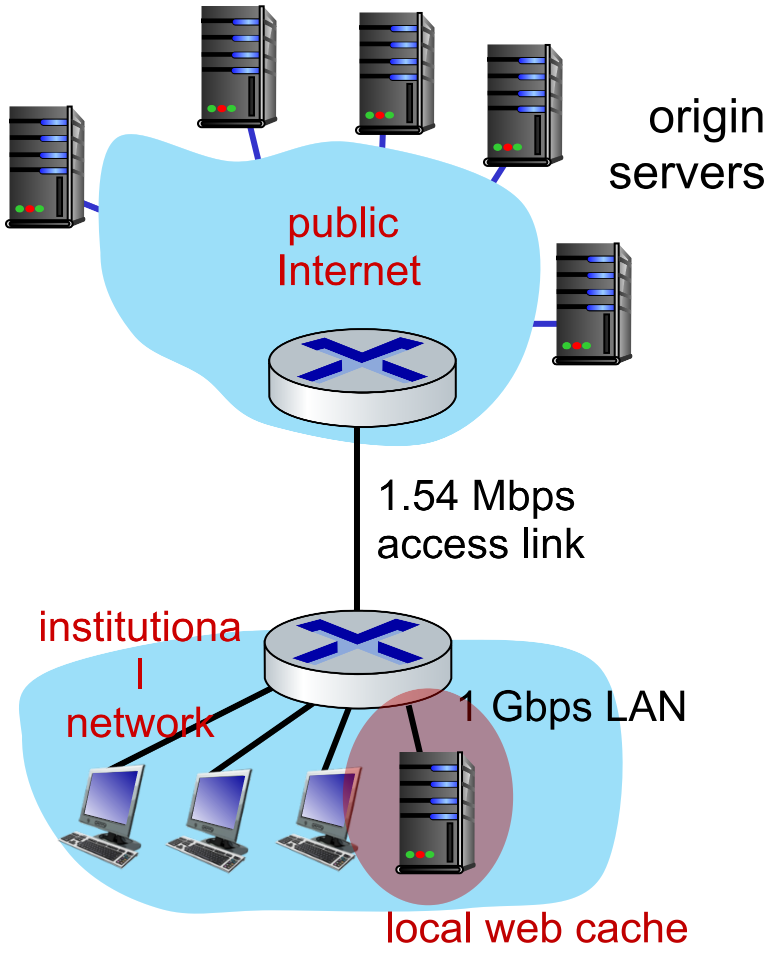
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

Performance:

- LAN utilization: .?
- access link utilization = ?
- average end-end delay = ?

How to compute link utilization, delay?

Cost: web cache (cheap!)



Caching example: install a web cache

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

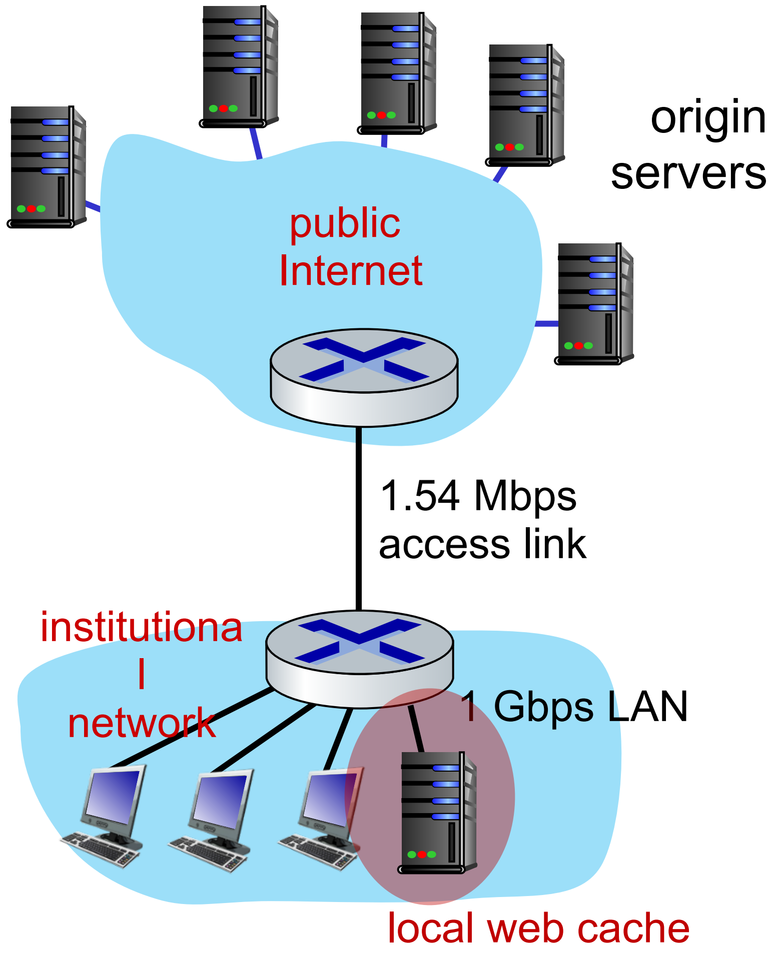
- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link

  = 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!)

Conditional Get

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

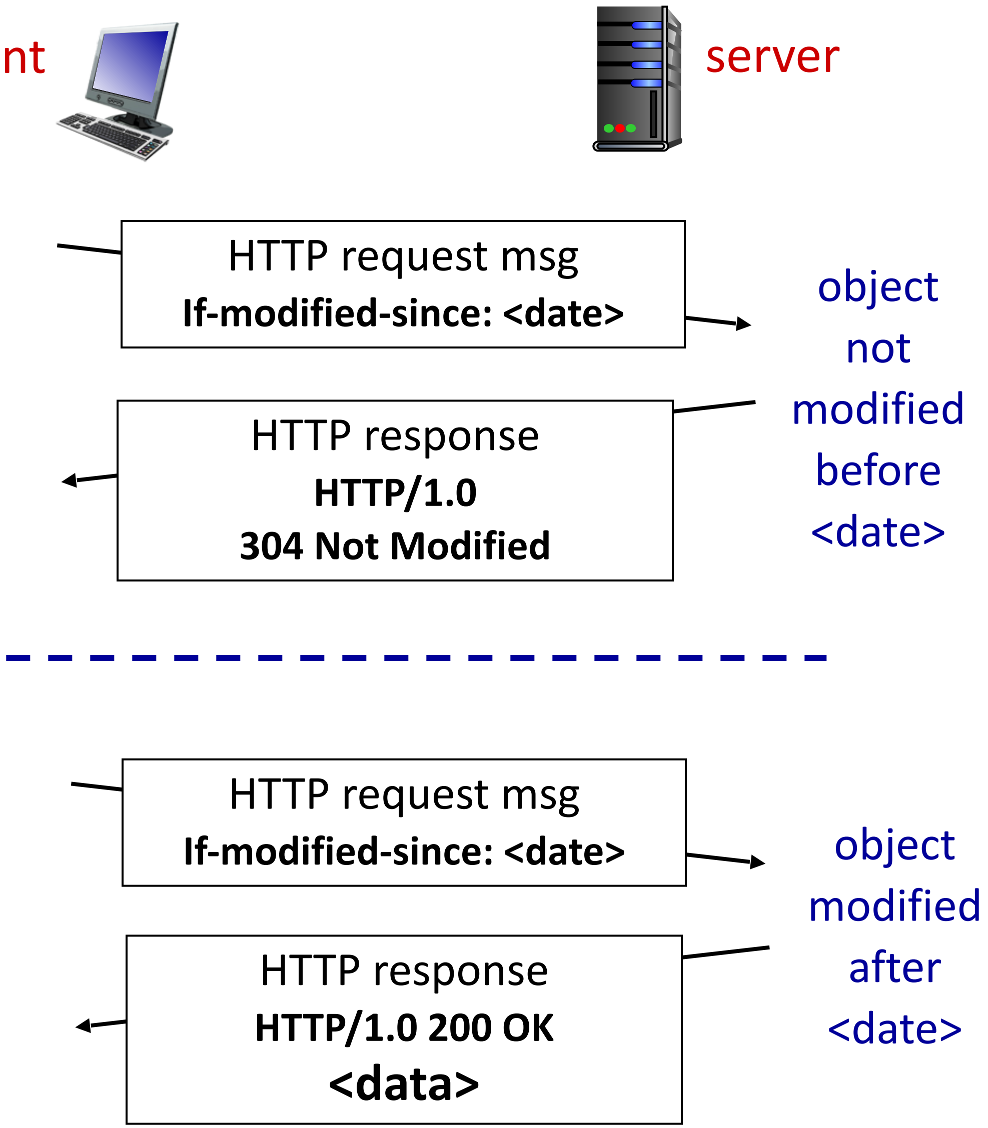
- no object transmission delay
- lower link utilization

- *cache:* specify date of cached copy in HTTP request

  If-modified-since: <date>

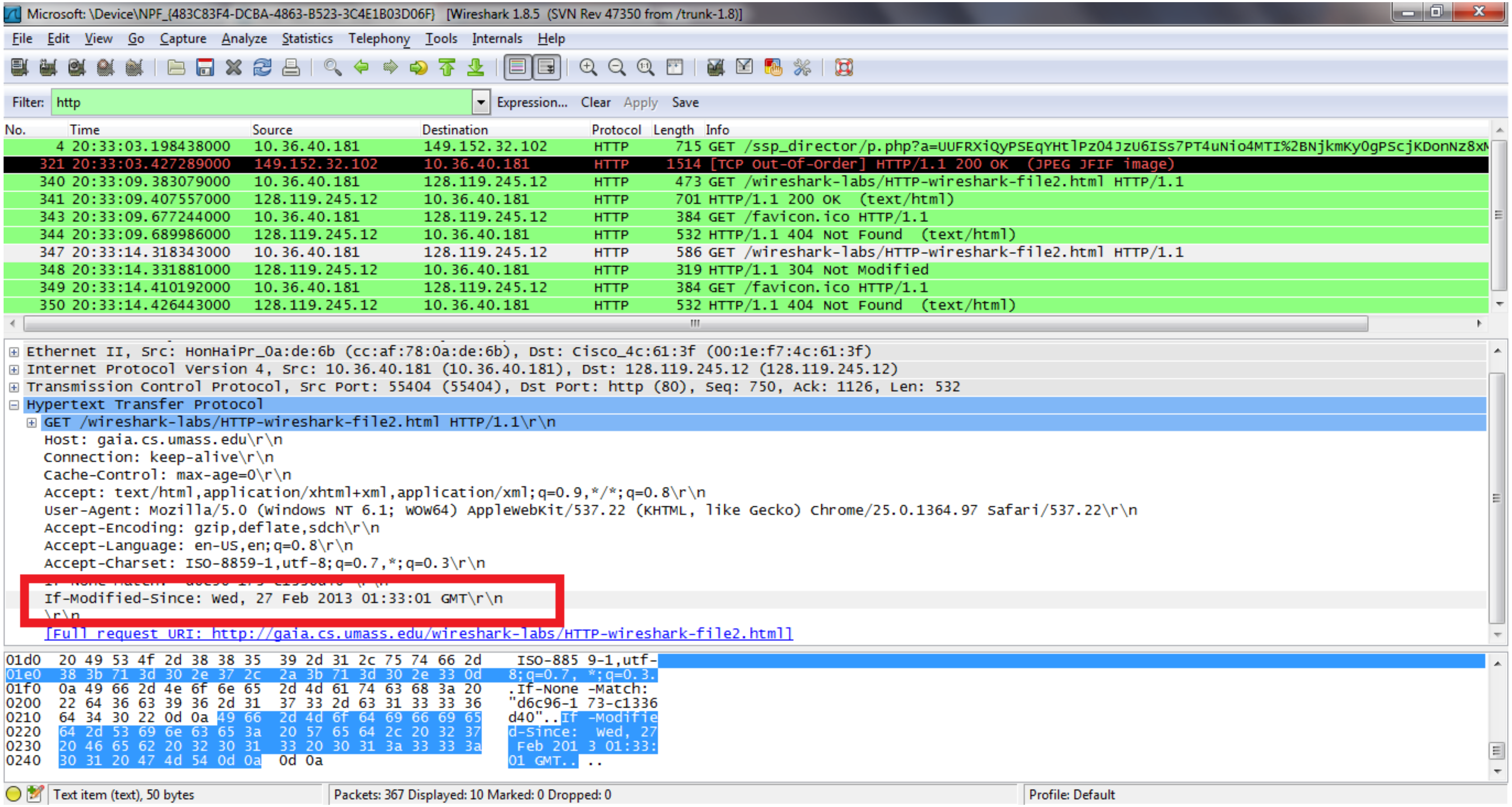
- *server:* response contains no object if cached copy is up-to-date:

  HTTP/1.0 304 Not Modified



COMPUTER NETWORKS

Conditional Get (more)



THANK YOU

TEAM NETWORKS
Department of Computer Science and Engineering