COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

Unit – 2 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

2.3 The Domain Name System

2.4 P2P Applications

2.5 Socket Programming with TCP & UDP

2.6 Other Application Layer Protocols

Unit – 2 Application Layer

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
  - SMTP, IMAP
  - DNS
- Programming network applications
  - socket API

Some Network Apps

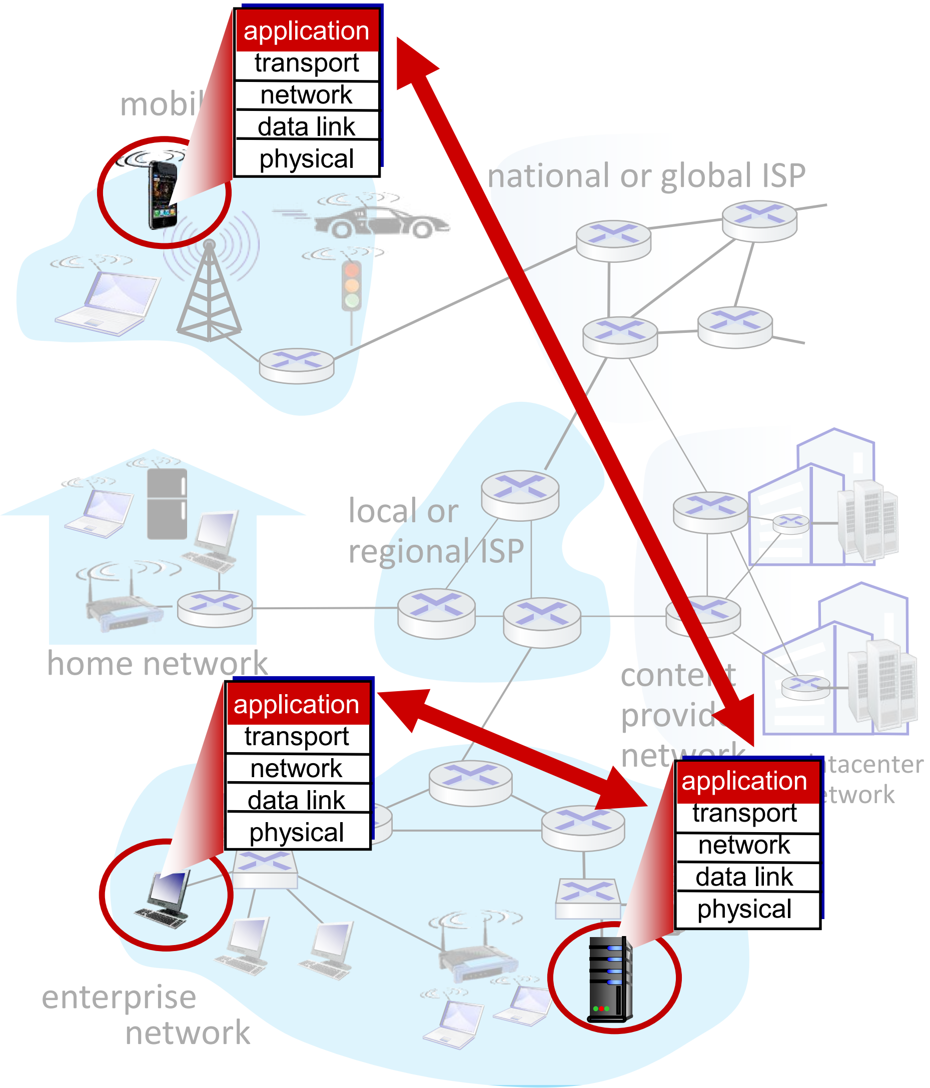
- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- P2P file sharing
- voice over IP
- real-time video conferencing (e.g., Skype, Hangouts)
- Internet search
- remote login
- …

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

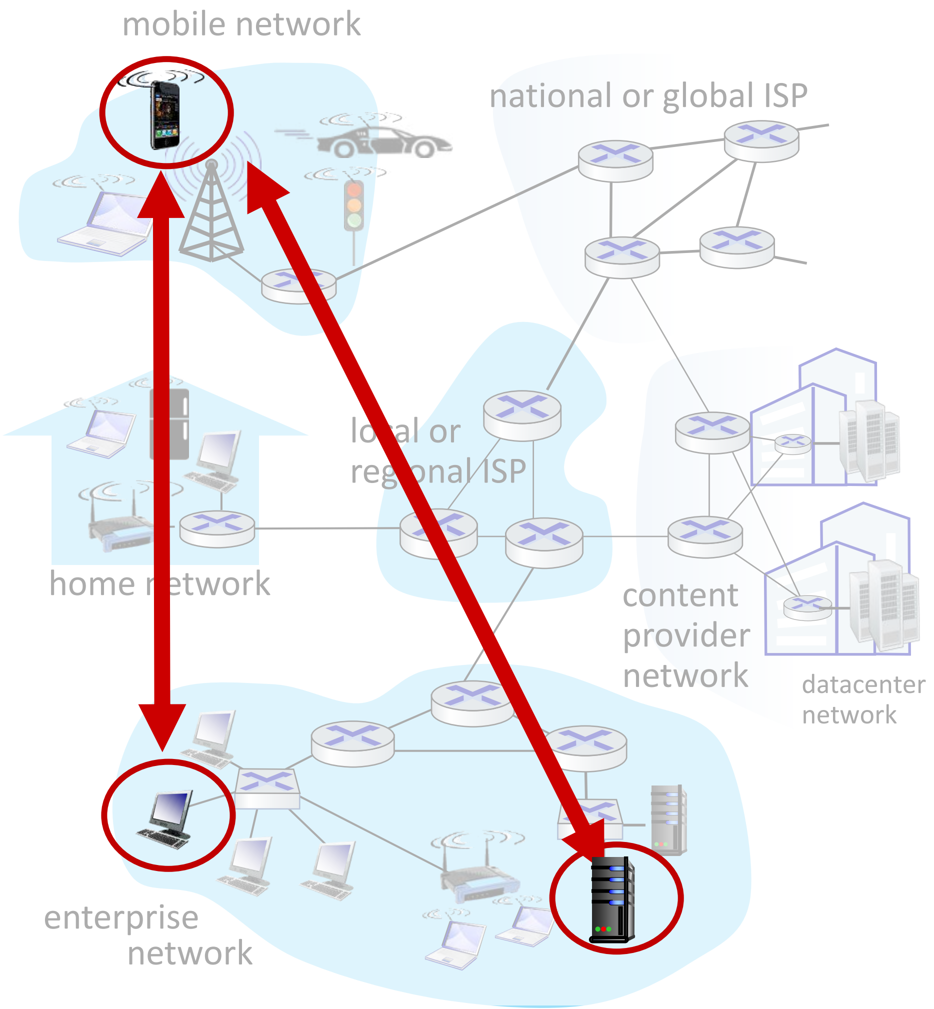
- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



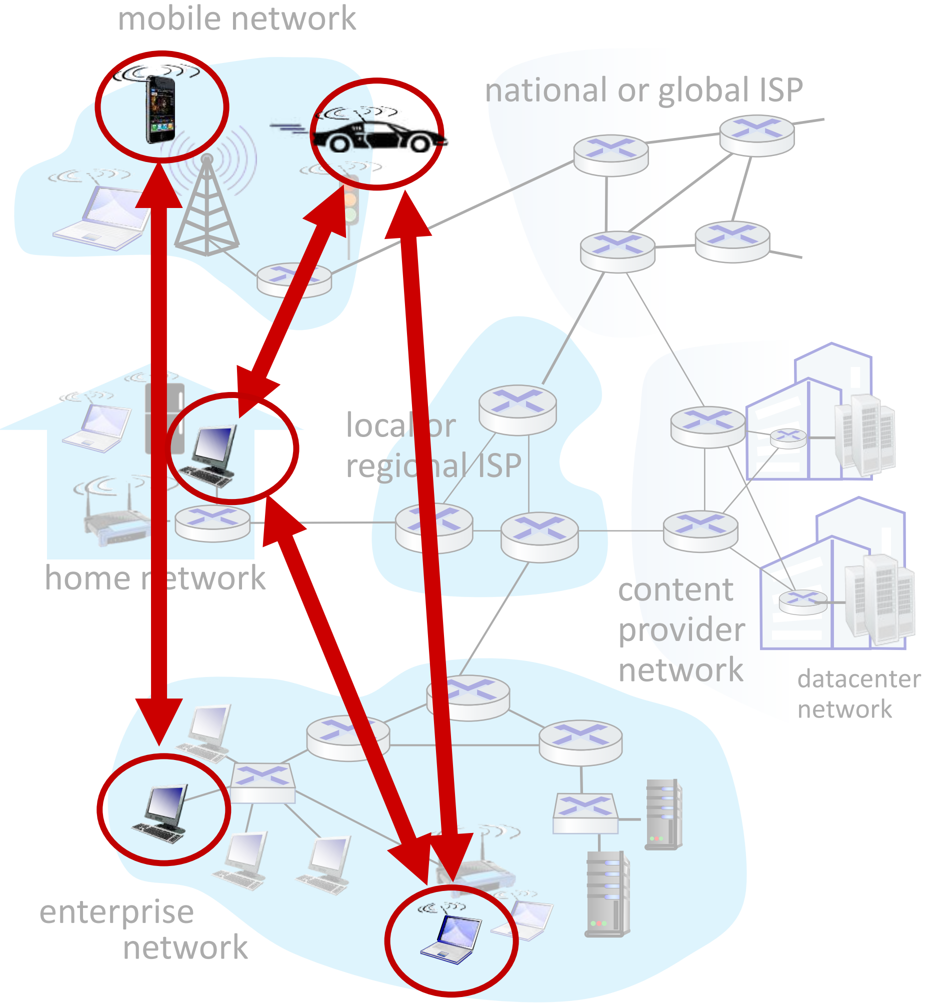
server:
- always-on host
- permanent IP address
- often in data centers, for scaling

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



- *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



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

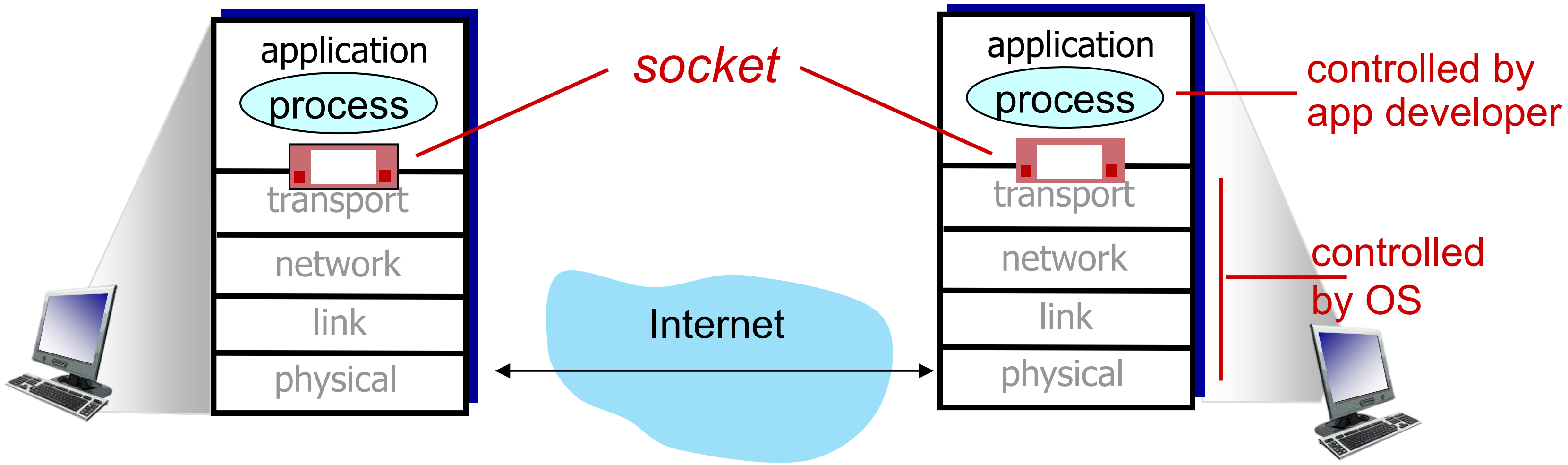
client process: process that initiates communication

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

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

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

Unit – 2 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

2.3 The Domain Name System

2.4 P2P Applications

2.5 Socket Programming with TCP & UDP

2.6 Other Application Layer Protocols

Unit – 2 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

2.3 The Domain Name System

2.4 P2P Applications

2.5 Socket Programming with TCP & UDP

2.6 Other Application Layer Protocols

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
  - SMTP, IMAP
  - DNS
- Programming network applications
  - socket API

Some Network Apps

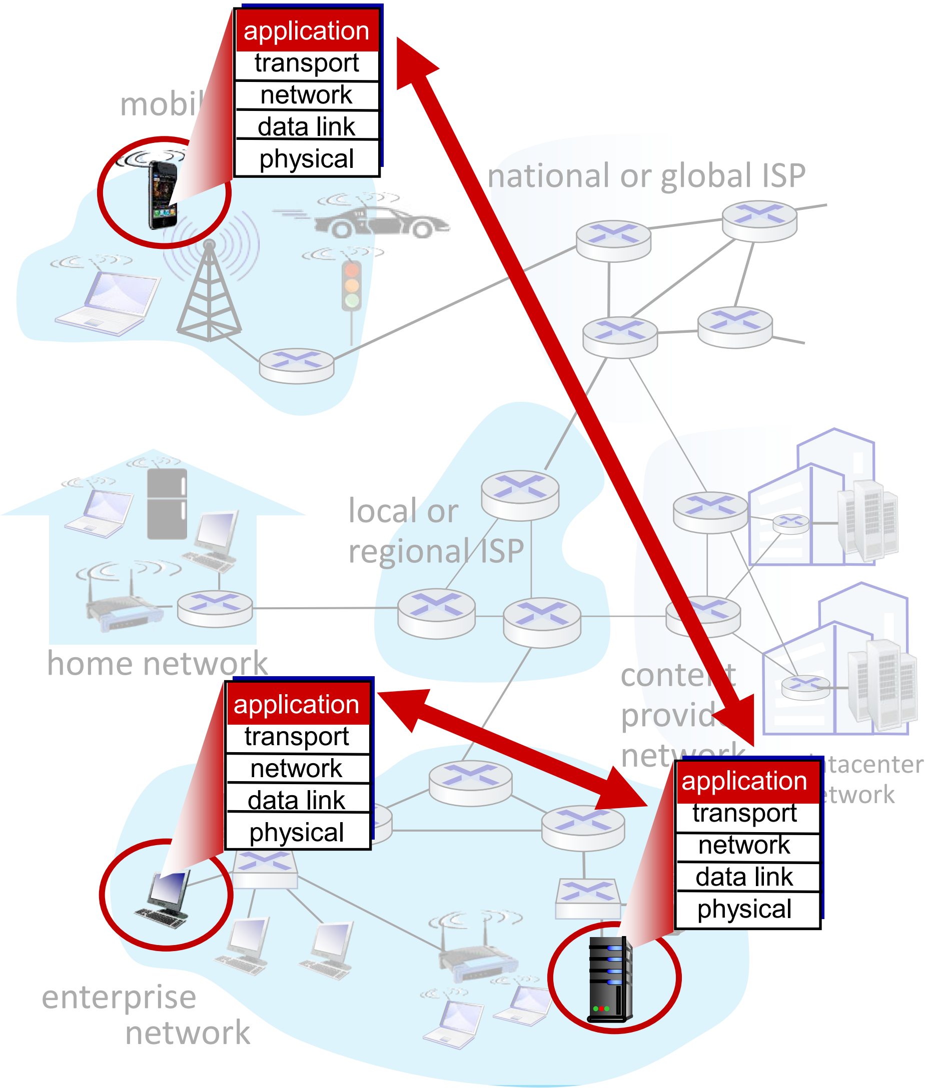
- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- P2P file sharing
- voice over IP
- real-time video conferencing (e.g., Skype, Hangouts)
- Internet search
- remote login
- …

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

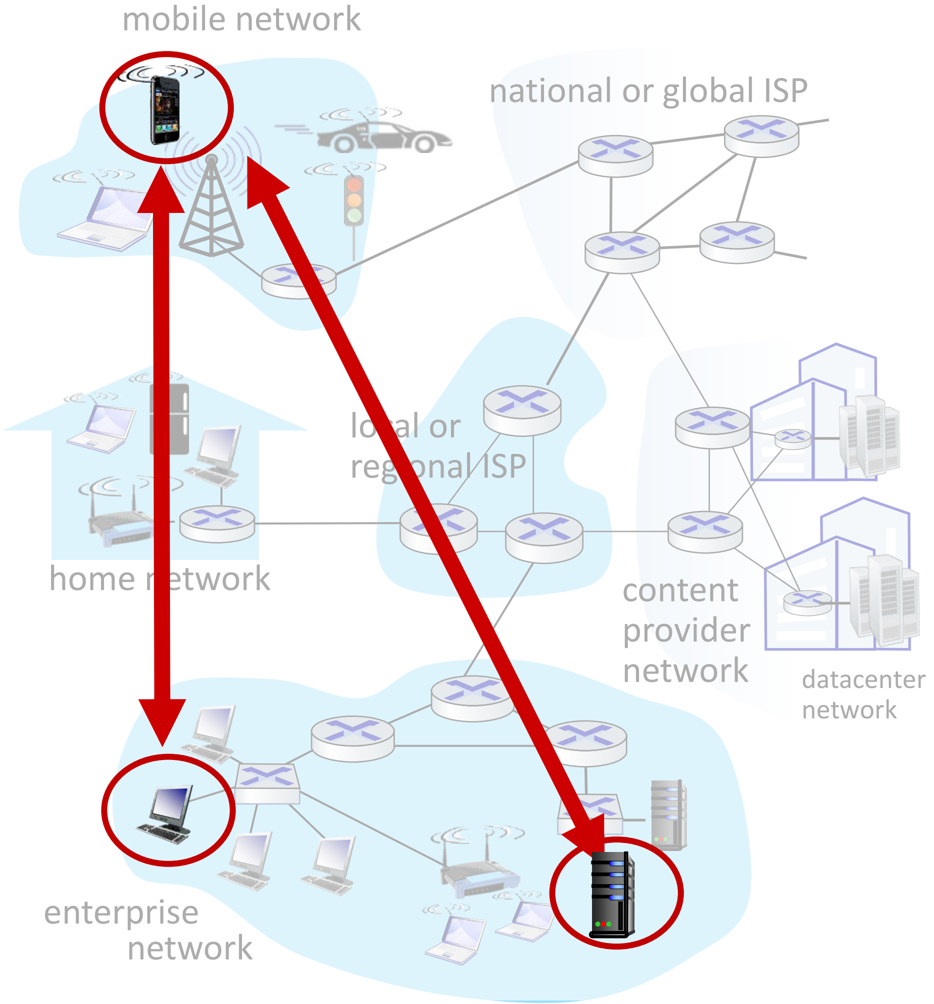
- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



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

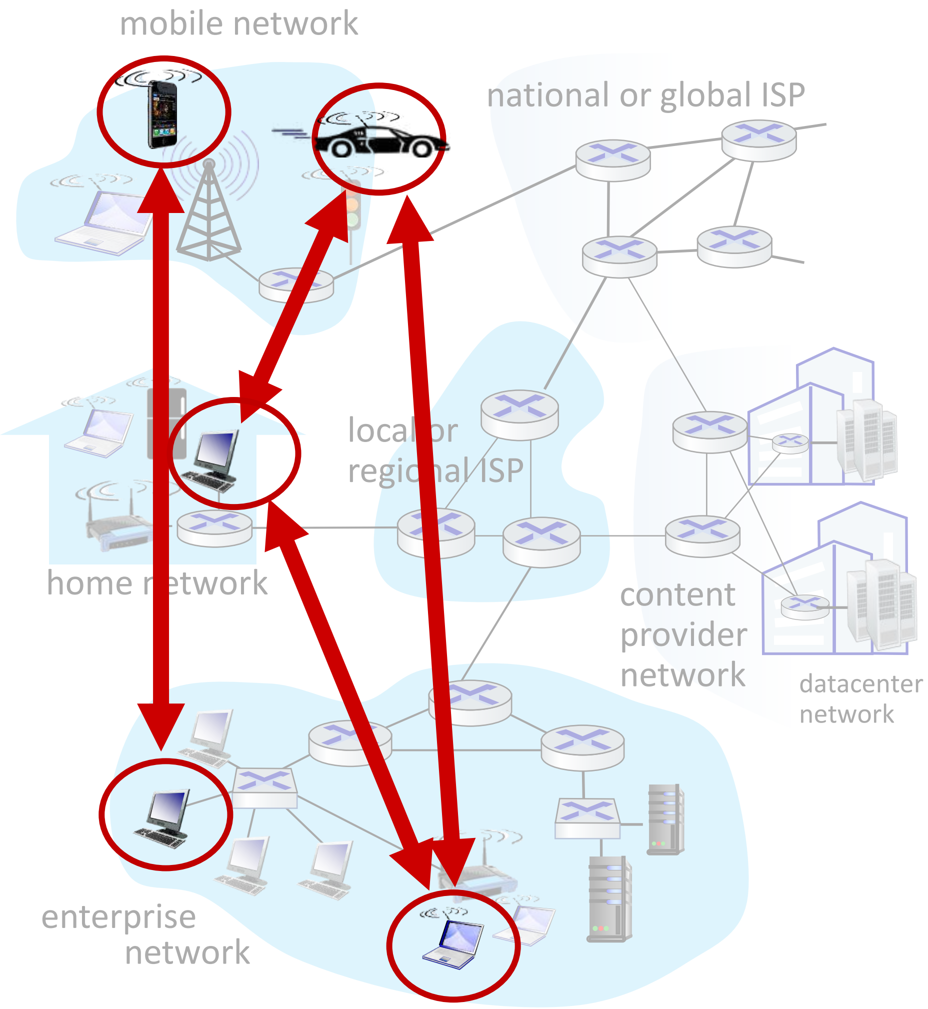
clients:
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- 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

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

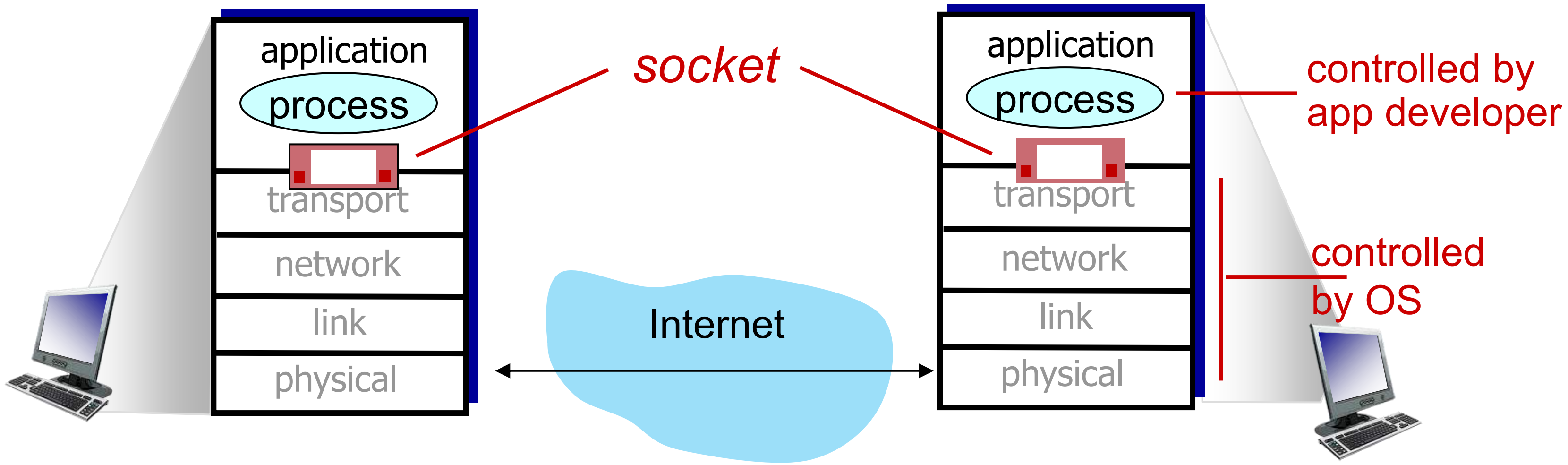
client process: process that initiates communication

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

PES UNIVERSITY ONLINE

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

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

Unit – 2 Application Layer

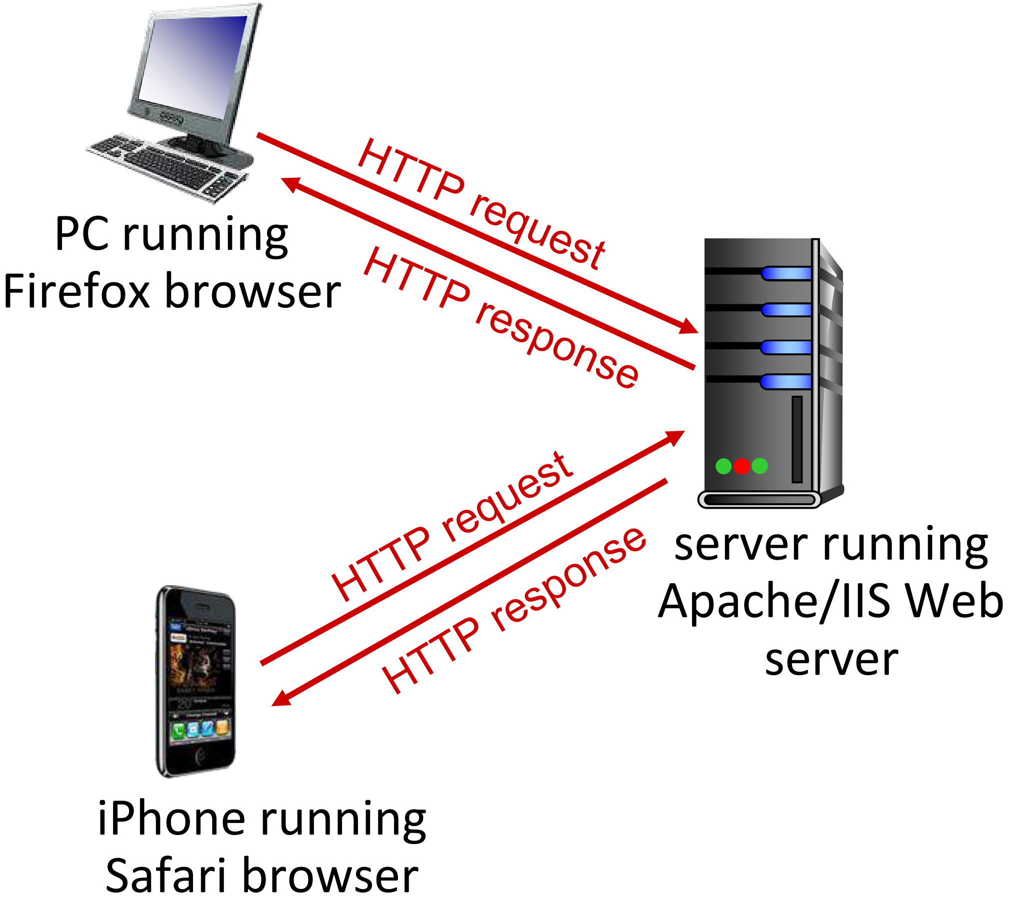
First, a quick review…

- 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: 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

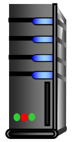
- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

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

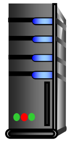
3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

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

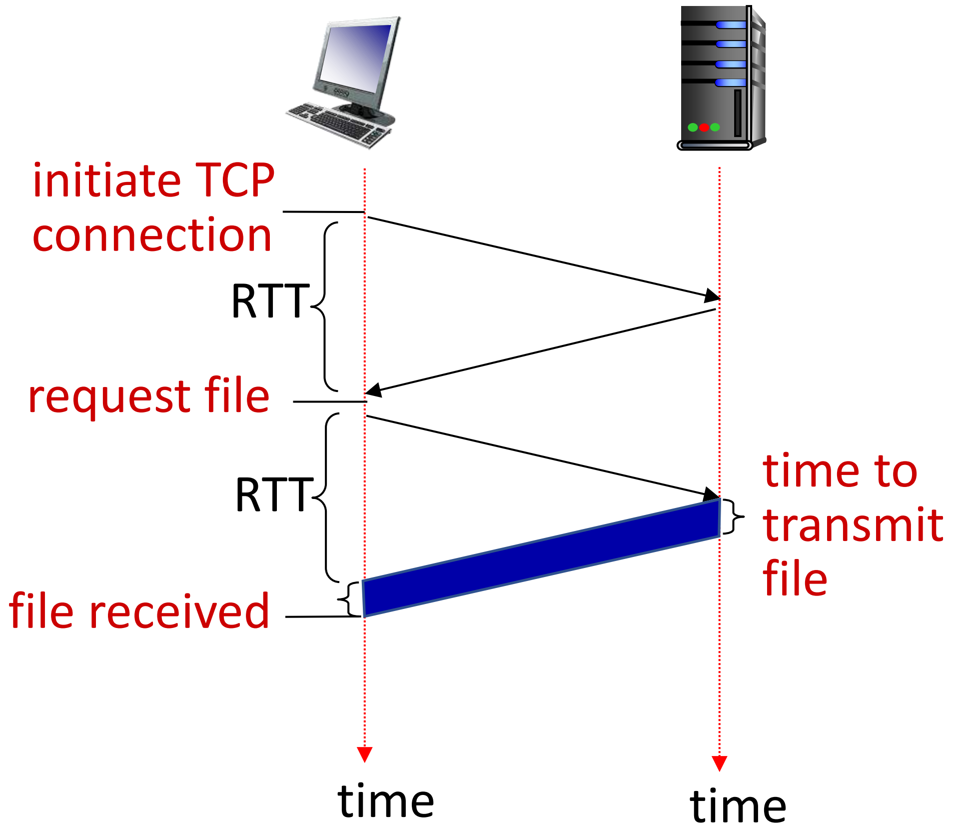
time

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

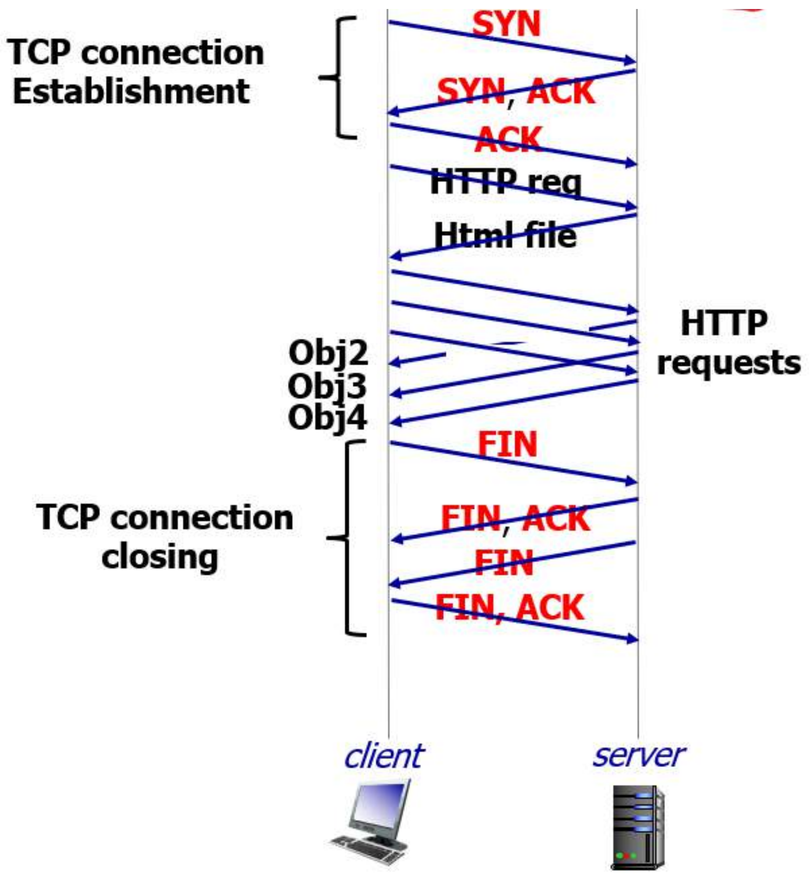
Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:

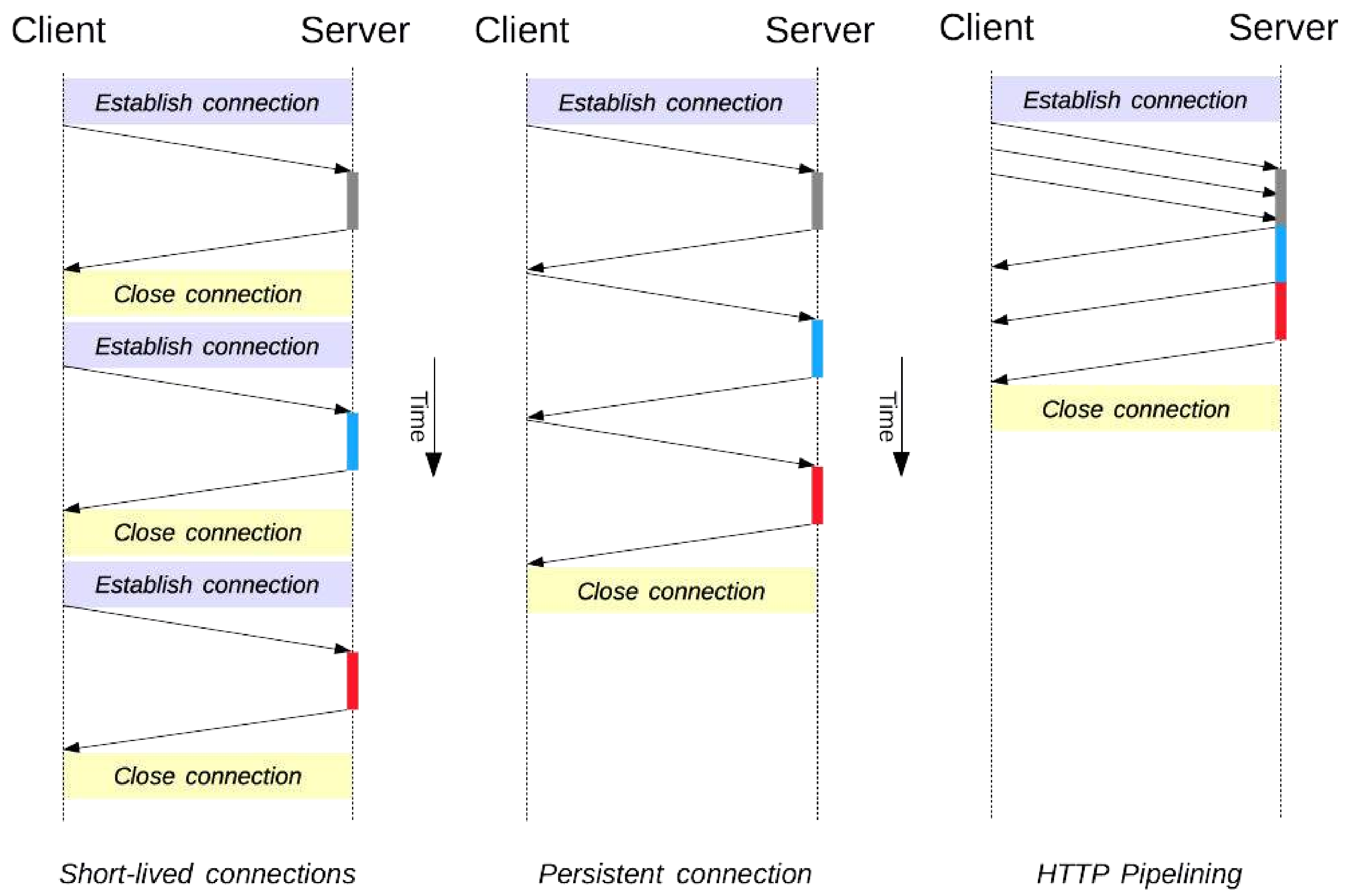
- requires 2 RTTs per object
- OS overhead for *each* TCP connection (TCP buffer and variables)
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

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



THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

Unit – 2 Application Layer

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)

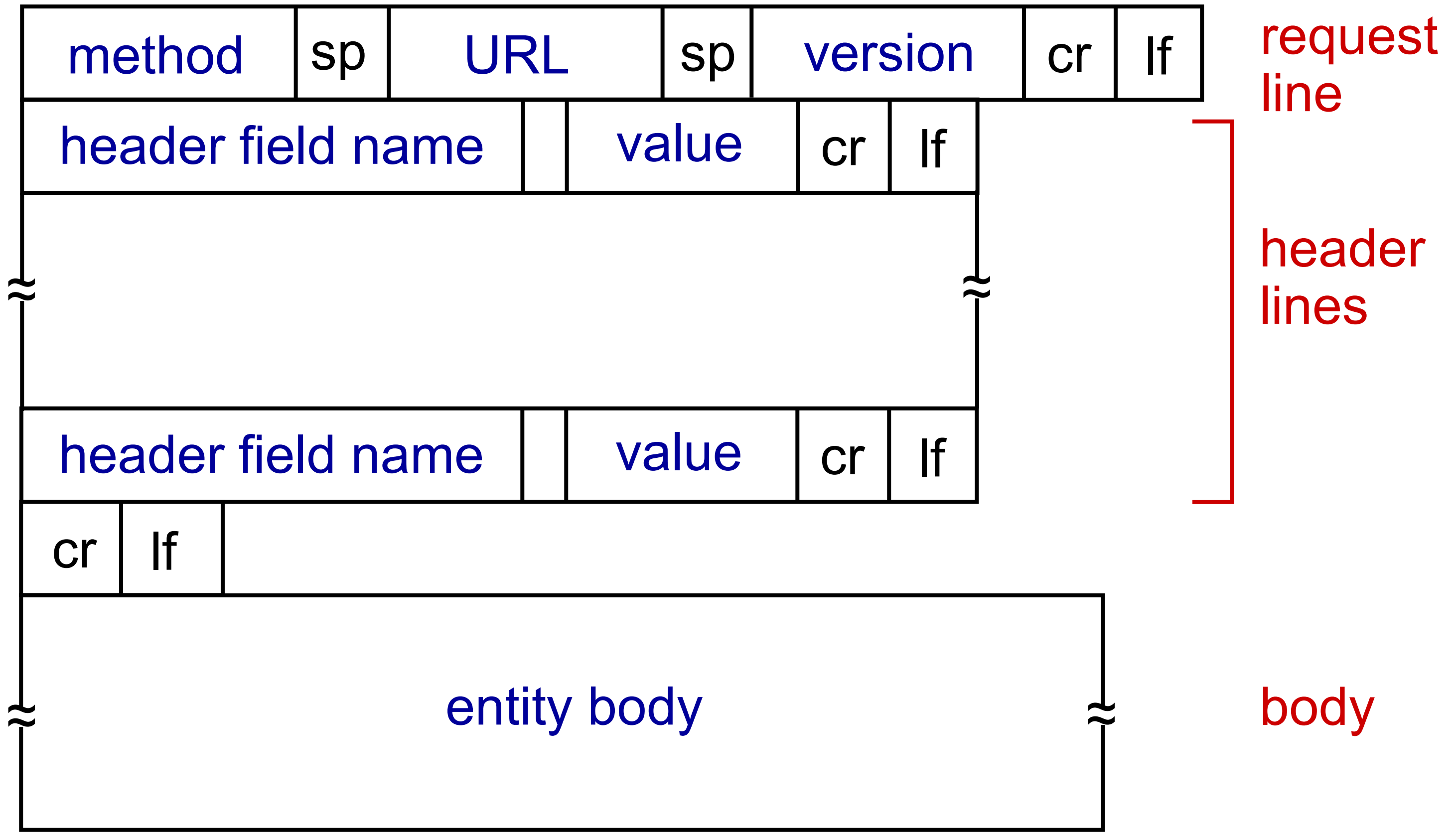
```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

header lines

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

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

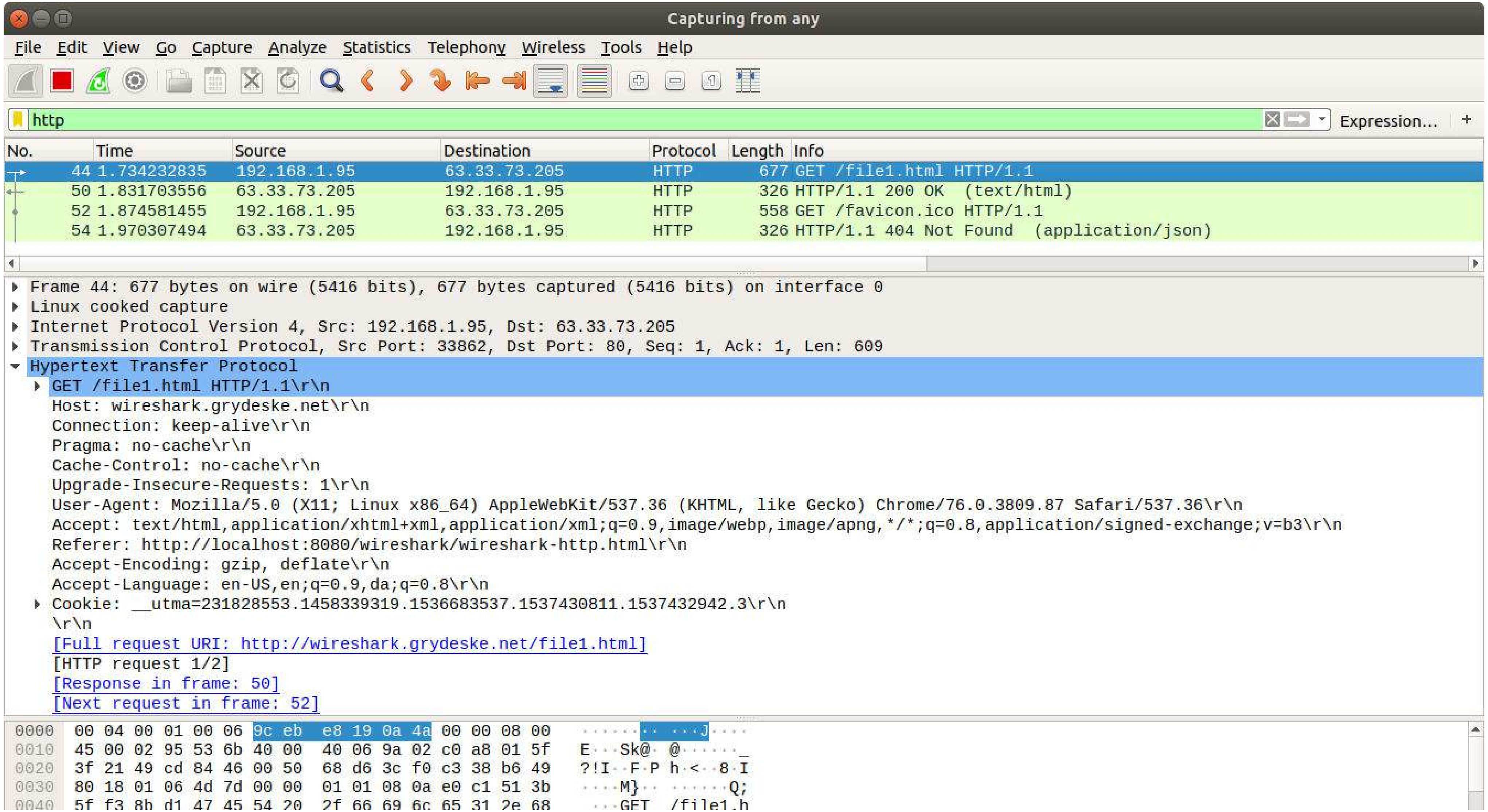
HTTP Request Message: General Format



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)

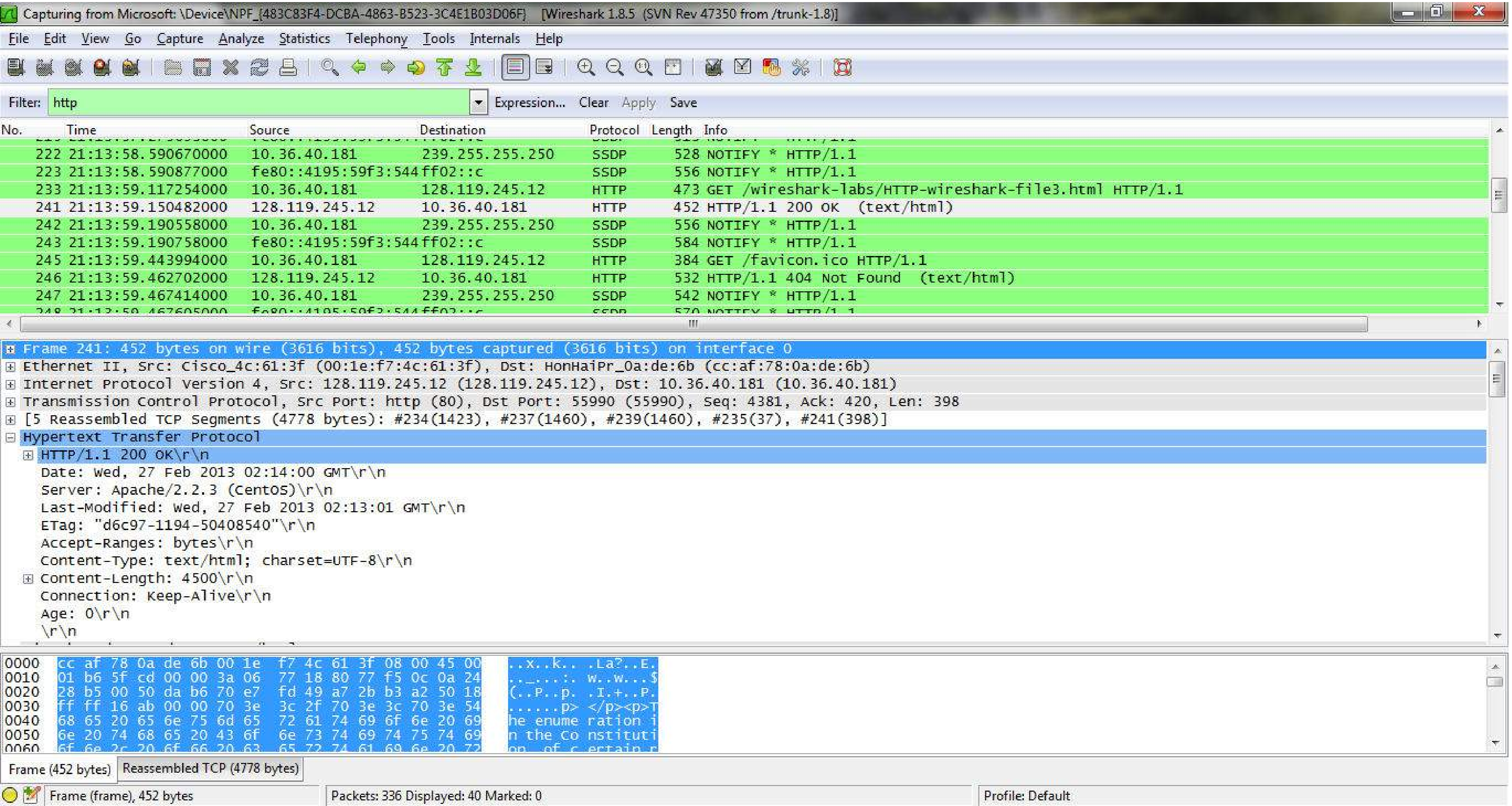
header lines

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

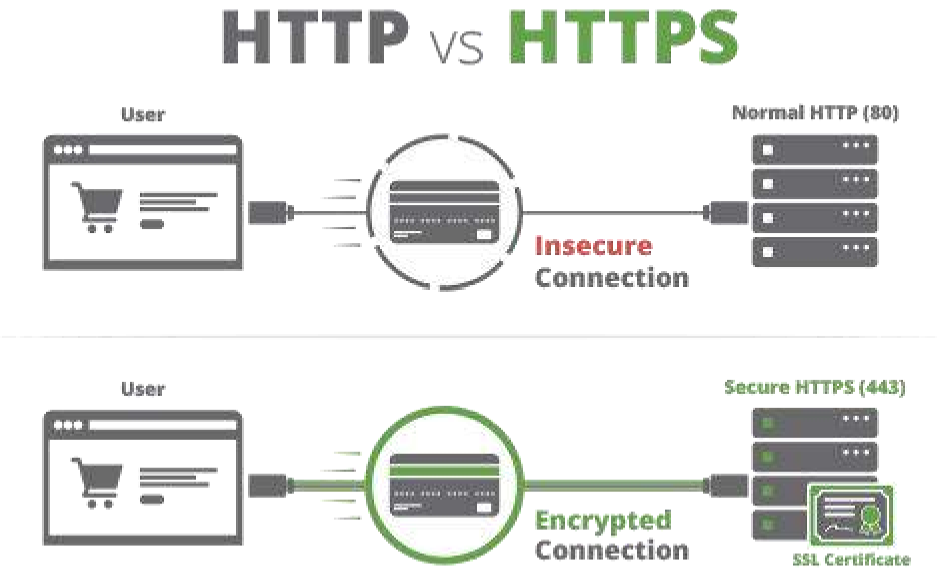
505 HTTP Version Not Supported

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

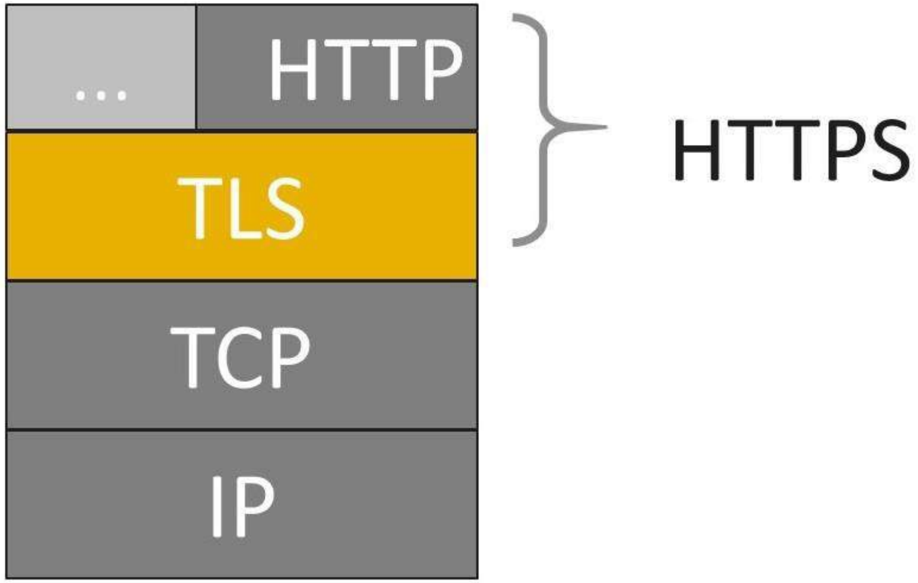
HTTP vs HTTPS



- 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?



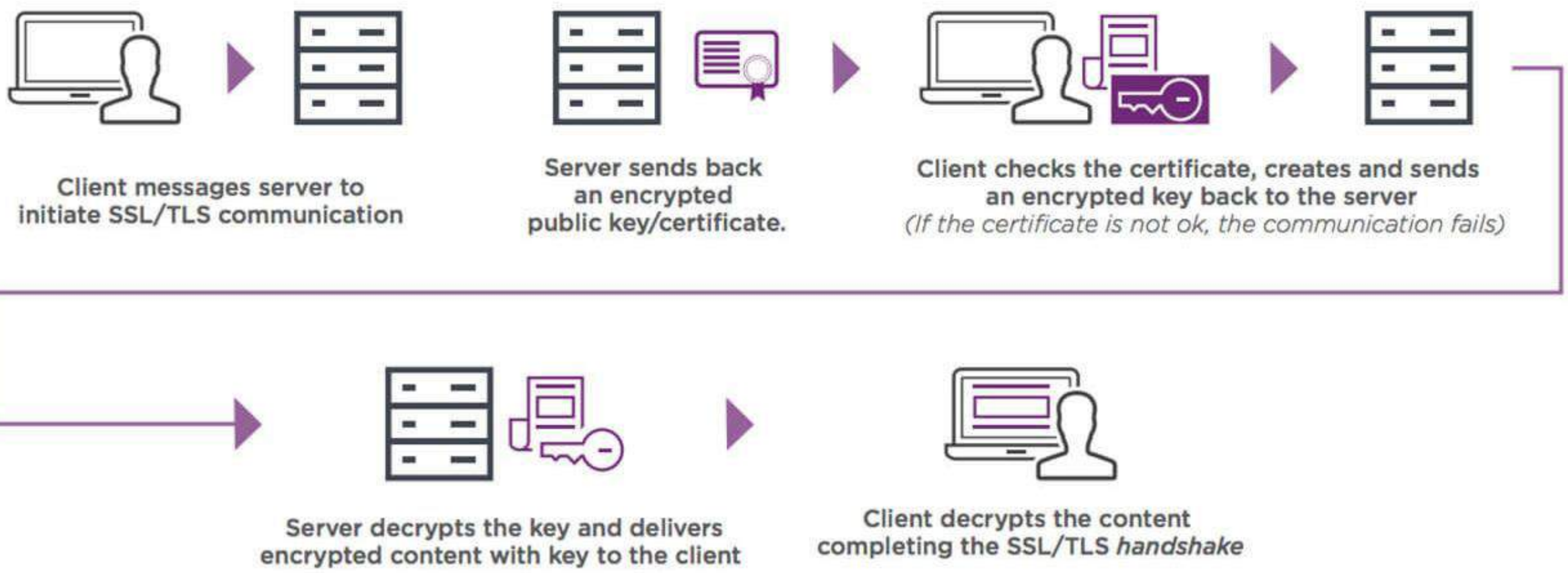
Any message encrypted with Bob’s public key can be only decrypted with Bob’s private key.

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.

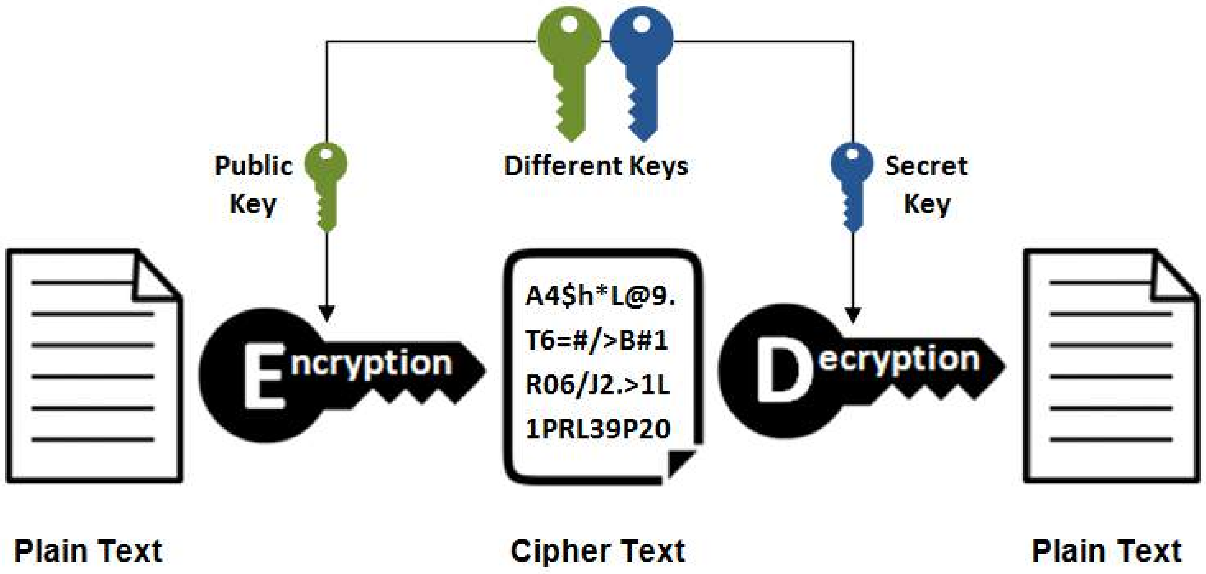
How does SSL works?



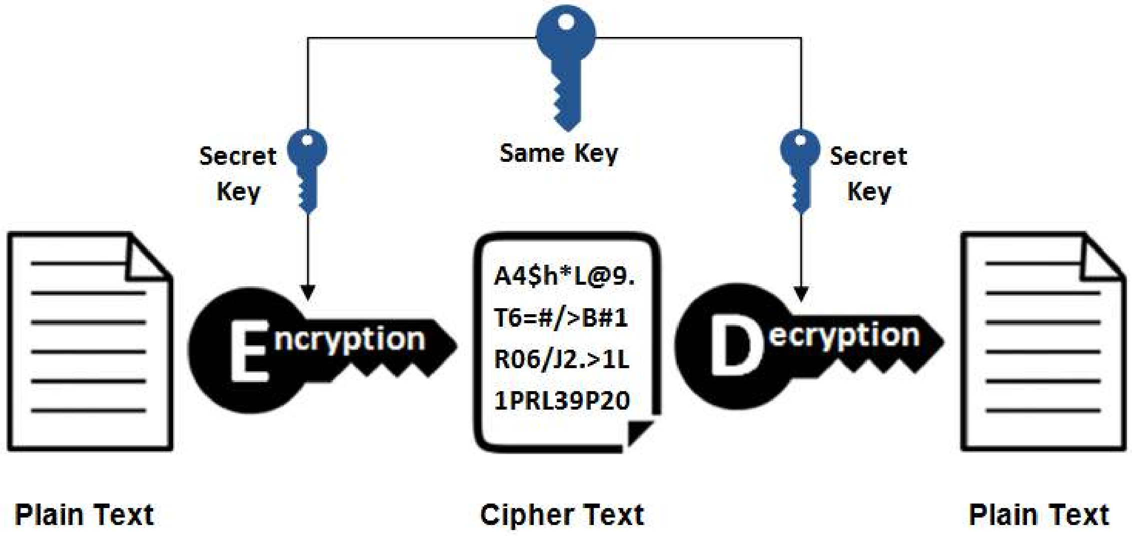
Benefits of HTTPS over HTTP using SSL Certificates

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

Asymmetric Encryption



Symmetric Encryption



THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

Unit – 2 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

2.3 The Domain Name System

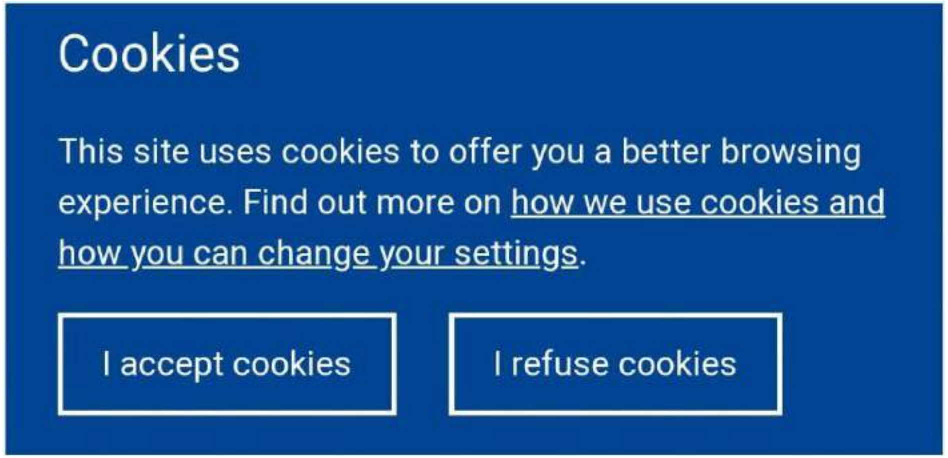
2.4 P2P Applications

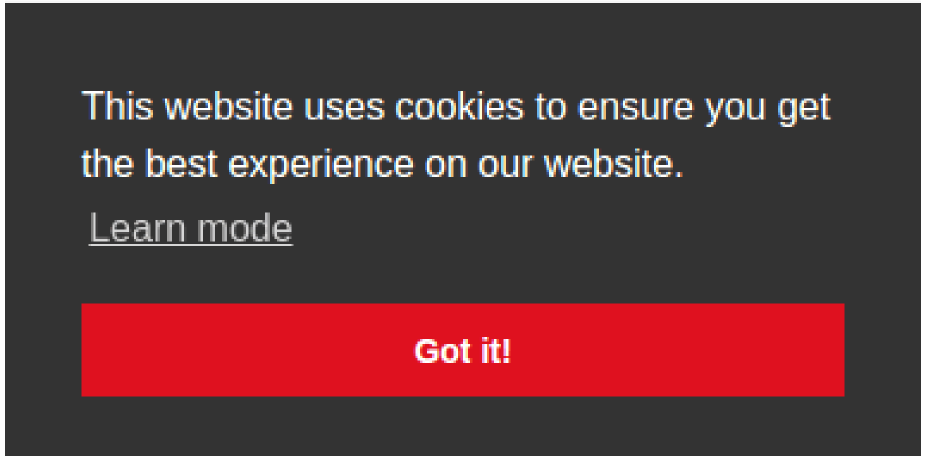
2.5 Socket Programming with TCP & UDP

2.6 Other Application Layer Protocols

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



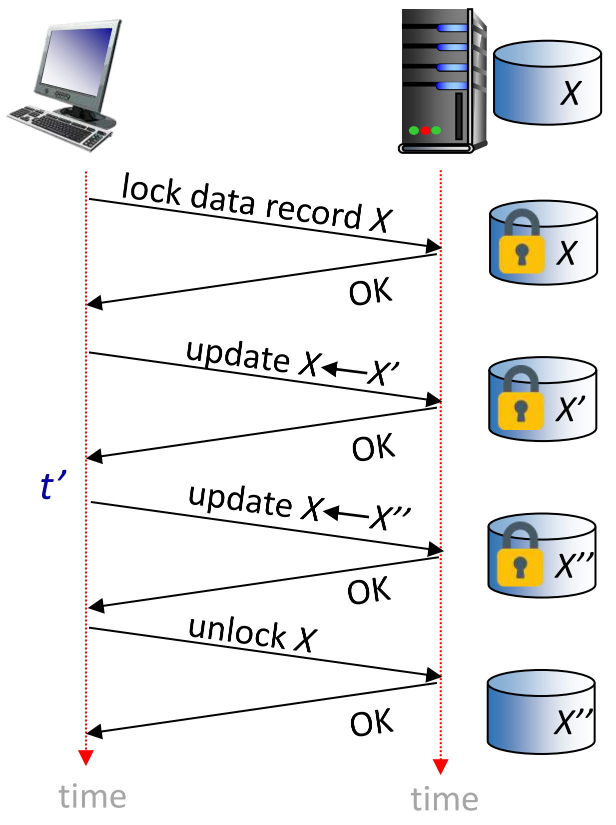


Maintaining user/server state: cookies

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

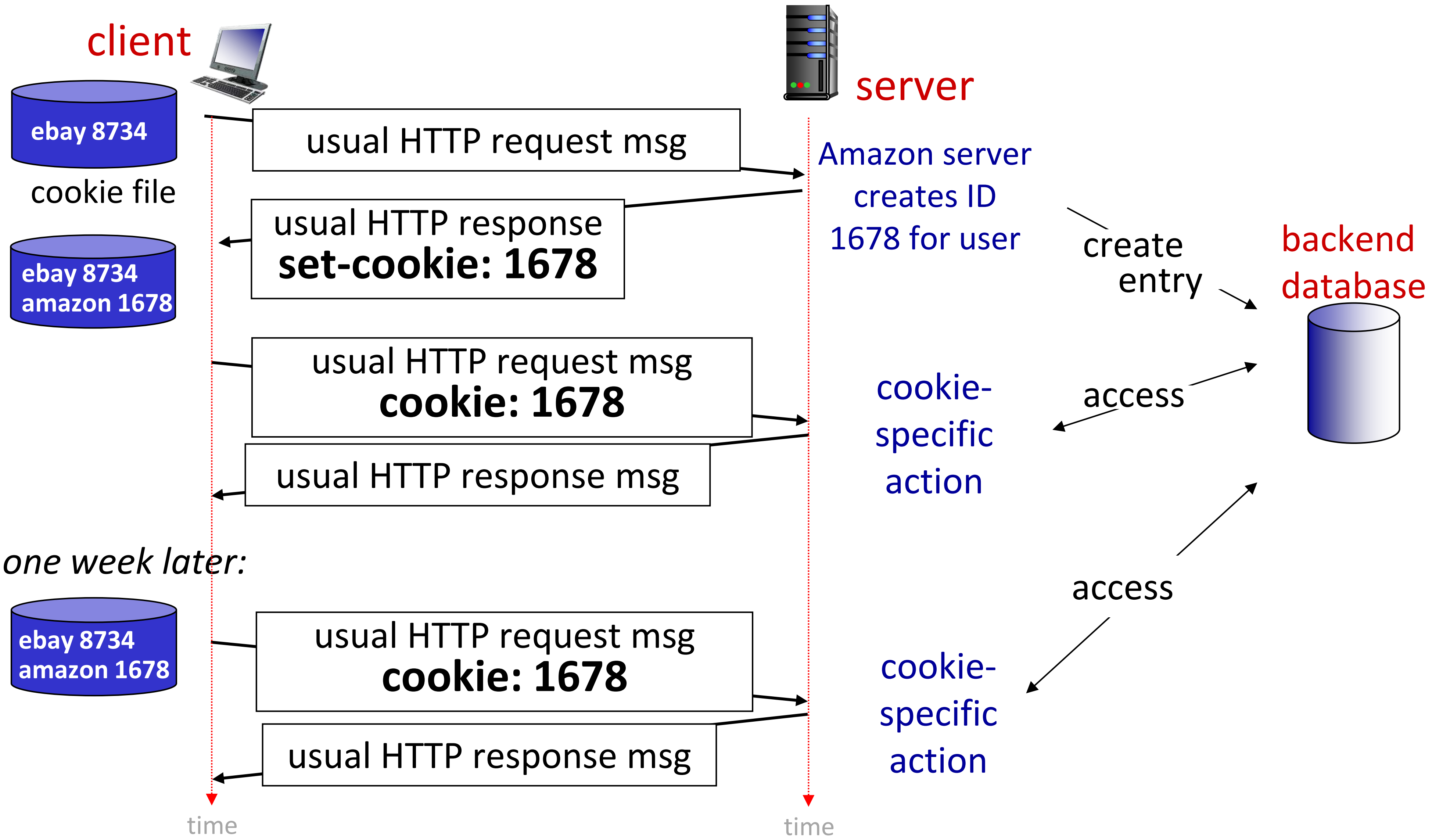
four components:

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

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

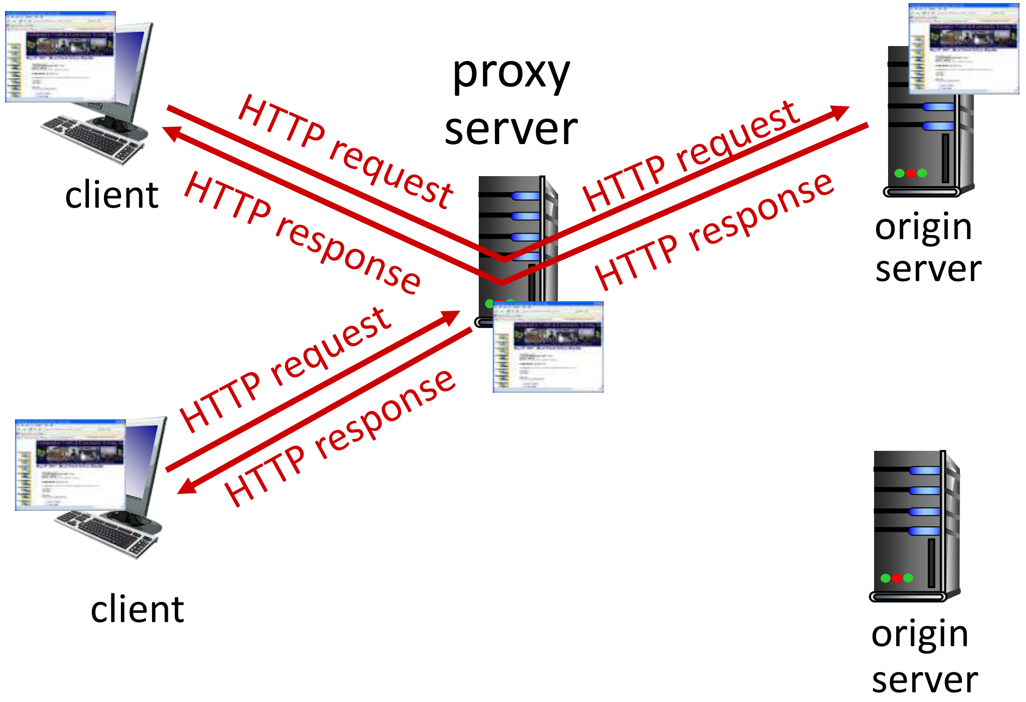
Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

Unit – 2 Application Layer

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

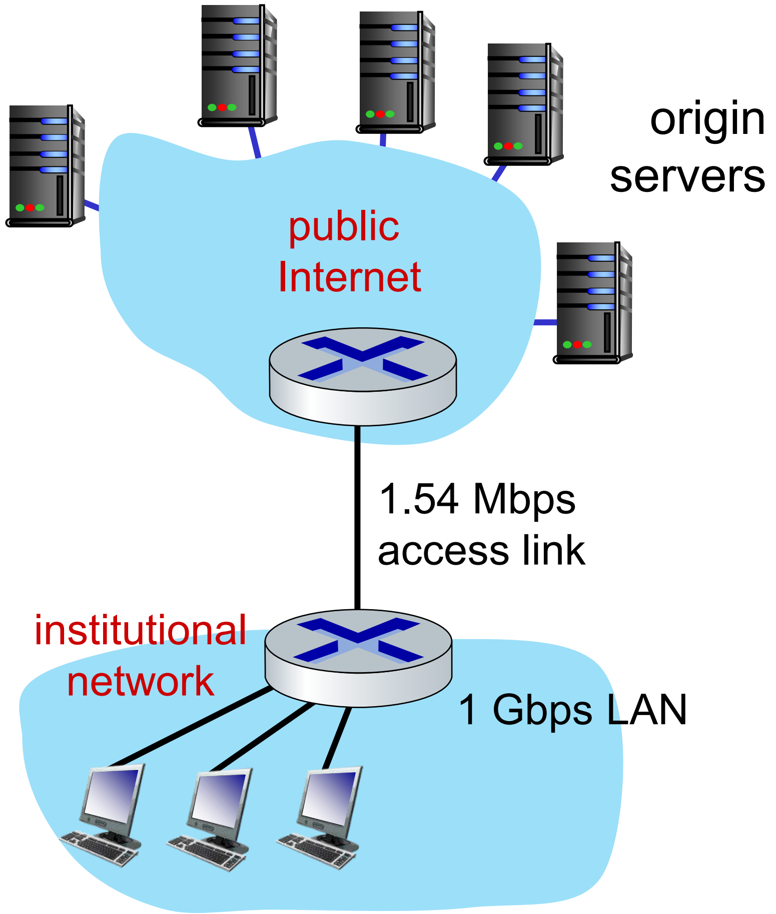
Scenario:

- 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

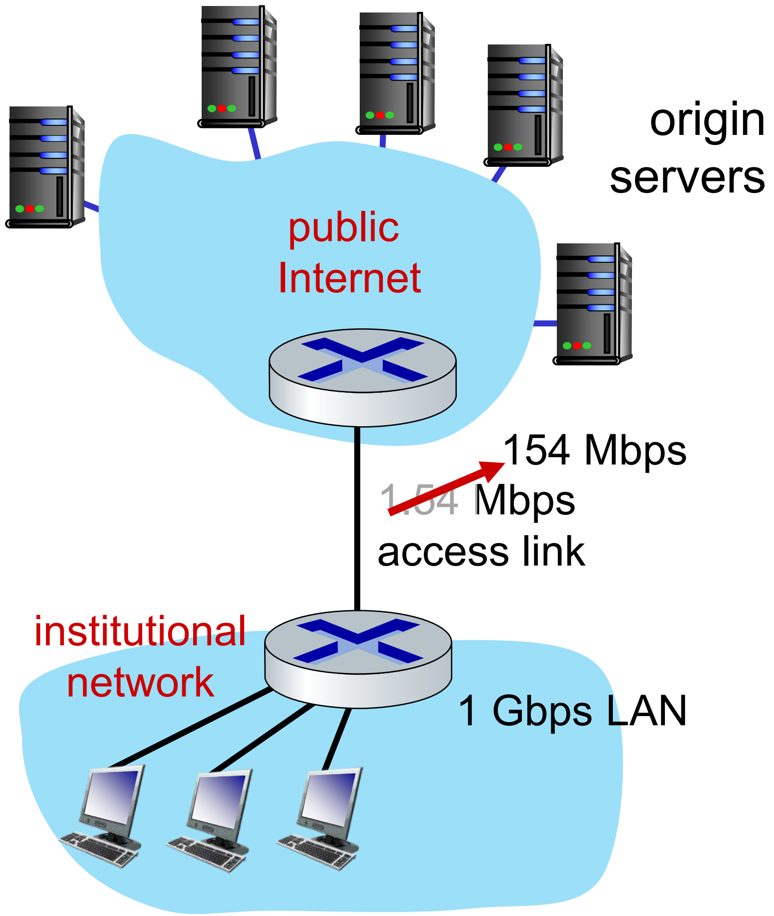
Scenario:

- access link rate: ~~1.54~~ 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~~ msecs + usecs

Cost: faster access link (expensive!)



Caching example: install a web cache

Scenario:

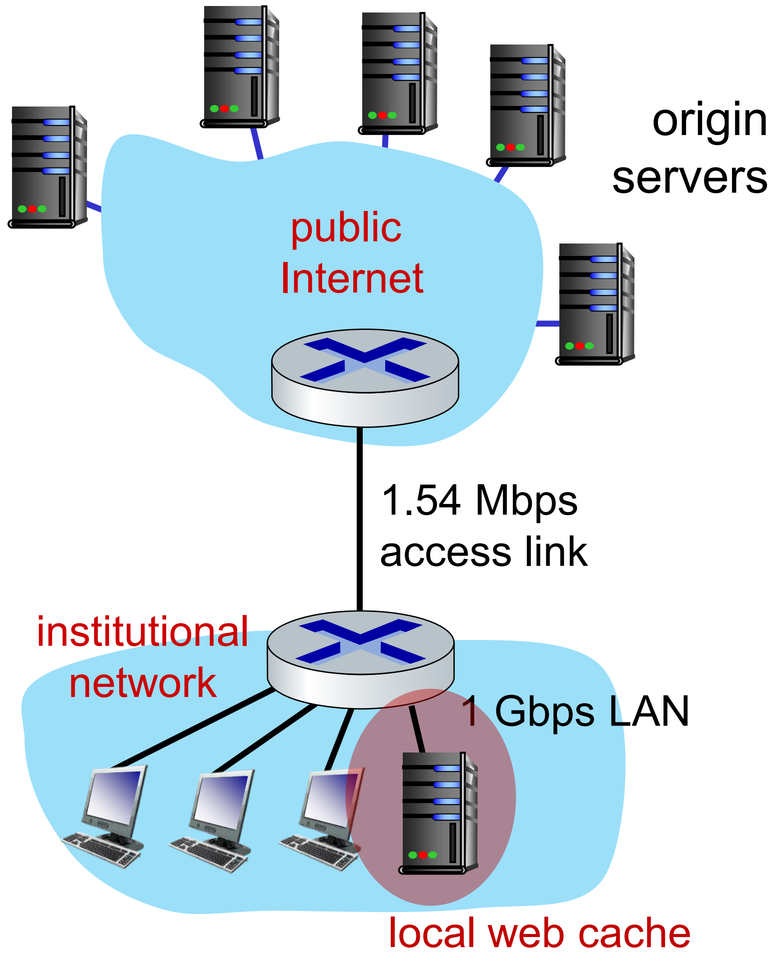
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

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:

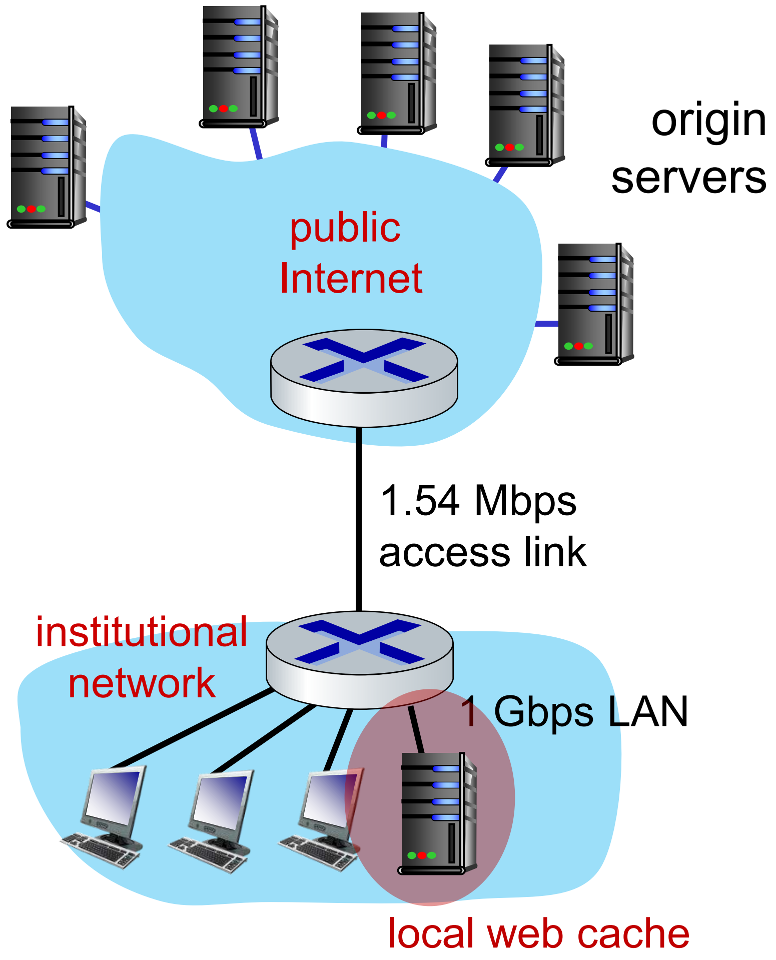
- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link

  = 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

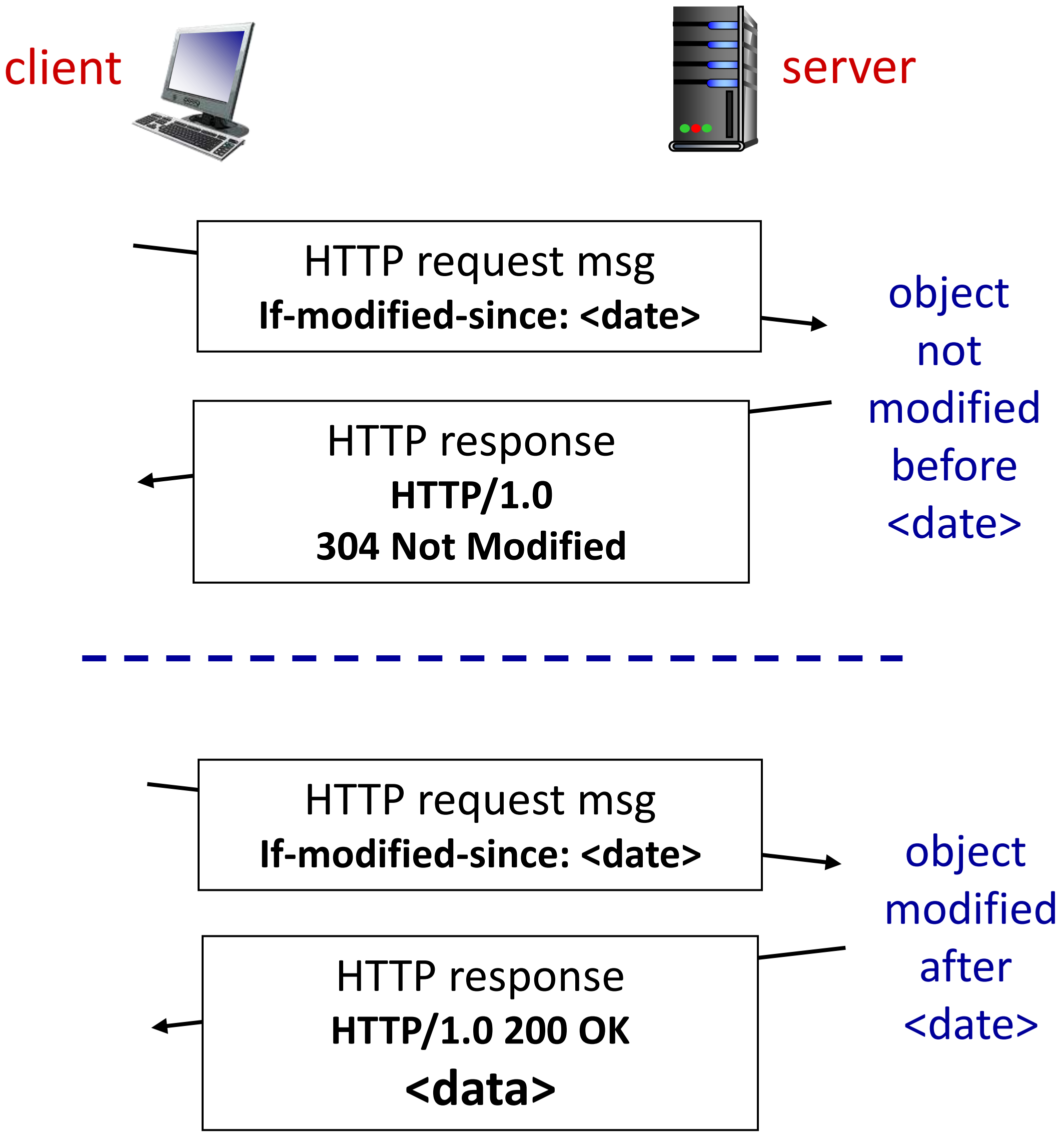
- no object transmission delay
- lower link utilization

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

  If-modified-since: <date>

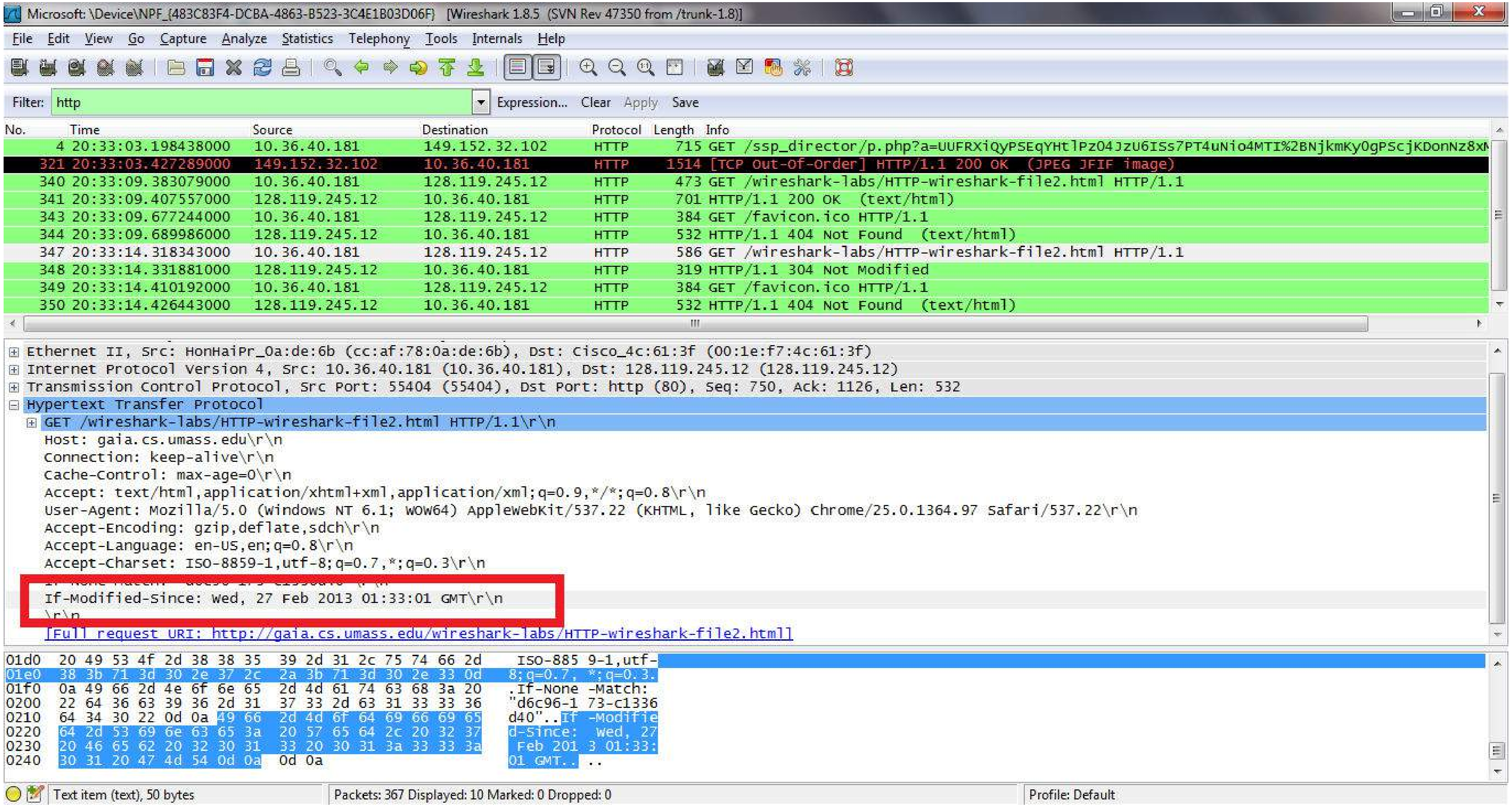
- *server:* response contains no object if cached copy is up-to-date:

  HTTP/1.0 304 Not Modified



COMPUTER NETWORKS

Conditional Get (more)



THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

Unit – 2 Application Layer

DNS: Domain Name System

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., cs.umass.edu - used by humans

Q: how to map between IP address and name, and vice versa ?

Domain Name System:

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, *implemented as application-layer protocol*
  - complexity at network’s “edge”

www.abc.example.com -> **Canonical Host Name**
www.example.com -> **Alias Name**

DNS services

- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

Q: Why not centralize DNS?

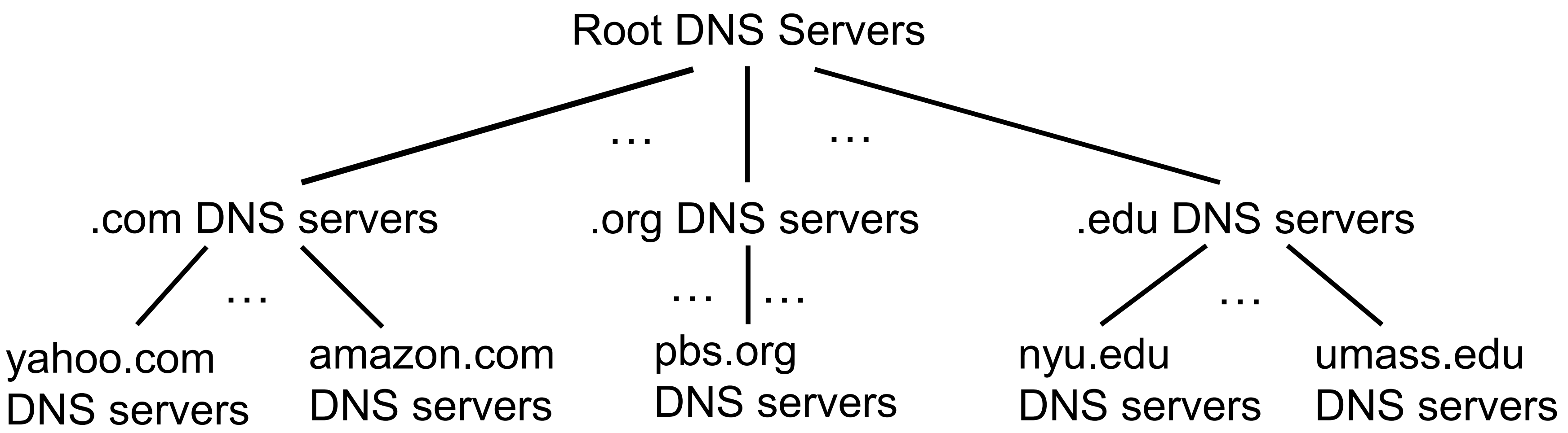
- single point of failure
- traffic volume
- distant centralized database
- maintenance

A: doesn't scale!

- Comcast DNS servers alone: 600B DNS queries per day

www.abc.example.com -> **Canonical Host Name**
bob@example.com -> **Alias Name**

DNS: a distributed, hierarchical database



Root

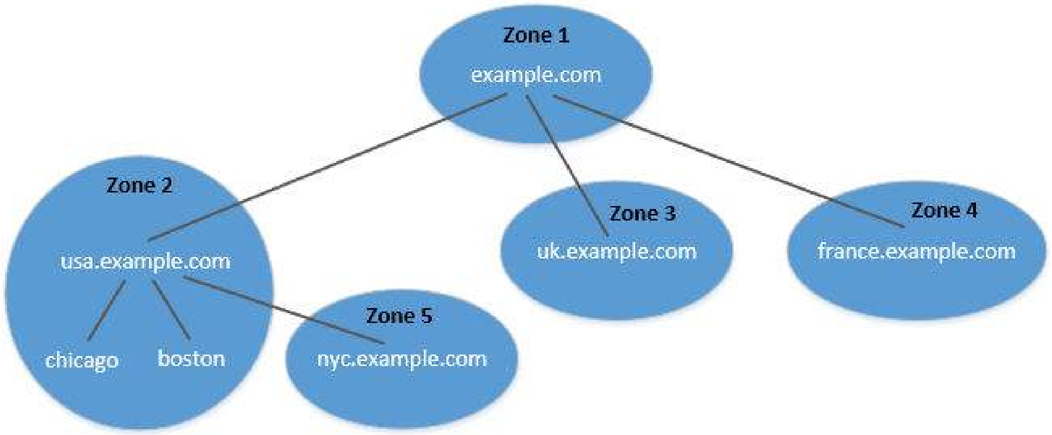
Top Level Domain

Authoritative

Client wants IP address for www.amazon.com; 1st approximation:

- client queries root server to find .com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS Zone vs Domain



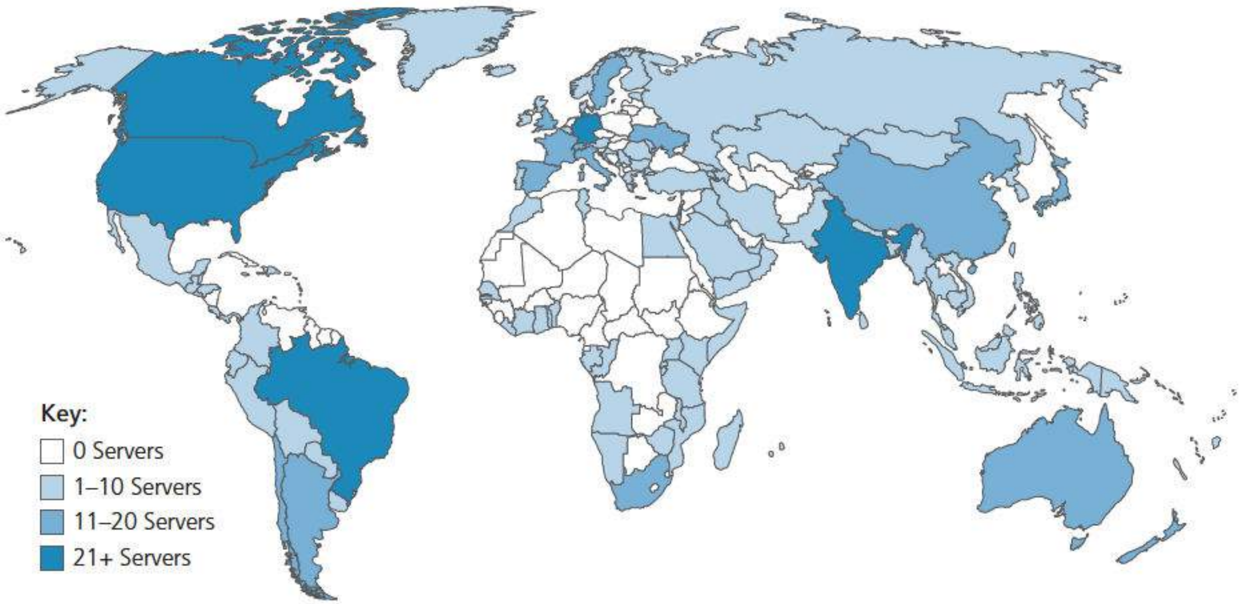
- DNS is organized according to zones.
- A zone groups contiguous domains and subdomains on the domain tree.
- Assign management authority to an entity.

- The tree structure depicts subdomains within example.com domain.
- Multiple DNS zones one for each country. The zone keeps records of who the authority is for each of its subdomains.
- The zone for example.com contains only the DNS records for the hostnames that do not belong to any subdomain like mail.example.com

DNS: root name servers

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name "servers" worldwide each "server" replicated many times (~200 servers in US)



TLD: authoritative servers

Top-Level Domain (TLD) servers:

- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .cn, .uk, .fr, .ca, .jp
- Network Solutions: authoritative registry for .com, .net TLD
- Educause: .edu TLD

Authoritative DNS servers:

- organization’s own DNS server(s), providing authoritative hostname to IP mappings for organization’s named hosts
- can be maintained by organization or service provider

Local DNS Name Servers

- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

Unit – 2 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

2.3 The Domain Name System

2.4 P2P Applications

2.5 Socket Programming with TCP & UDP

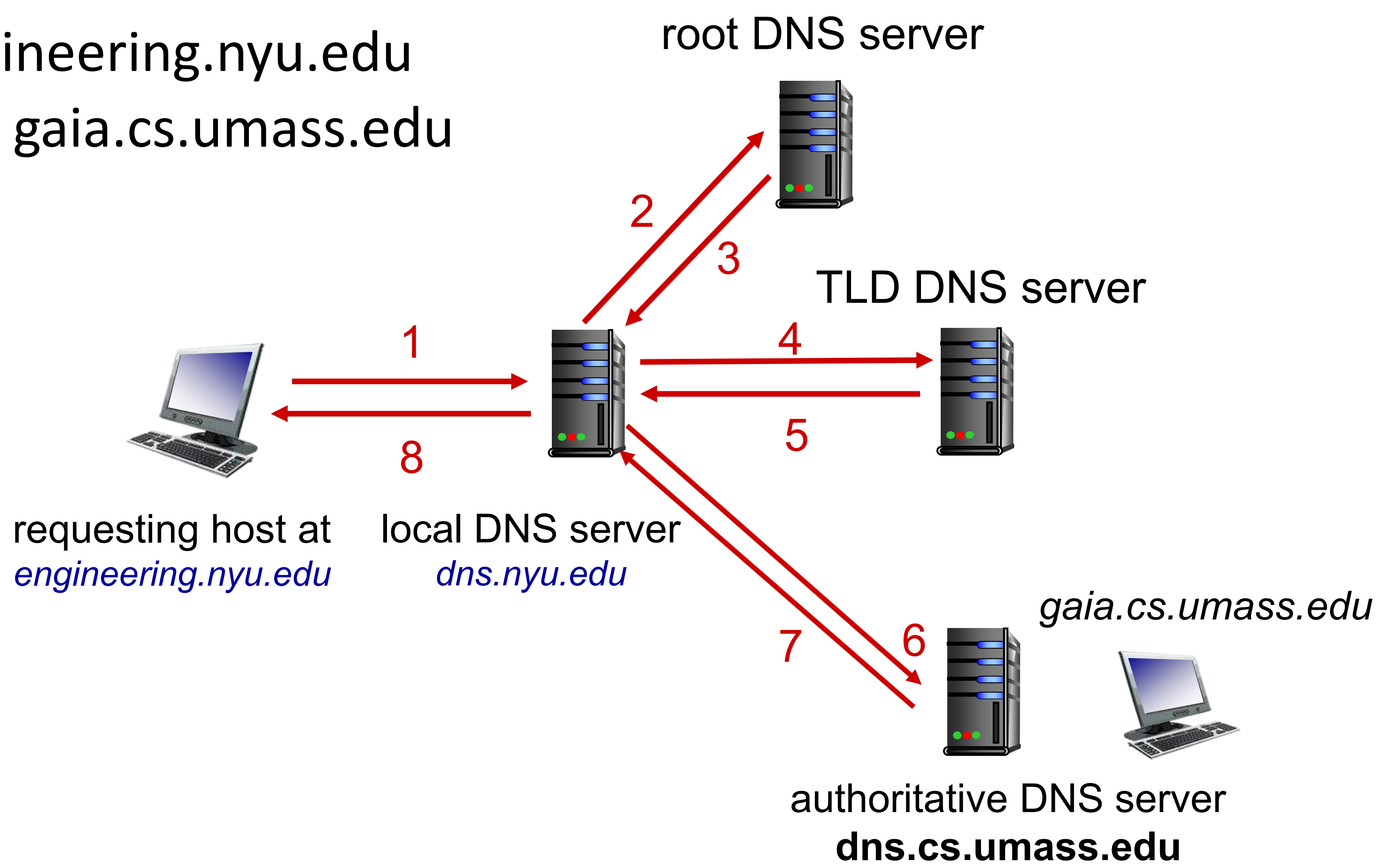
2.6 Other Application Layer Protocols

DNS name resolution: iterated query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

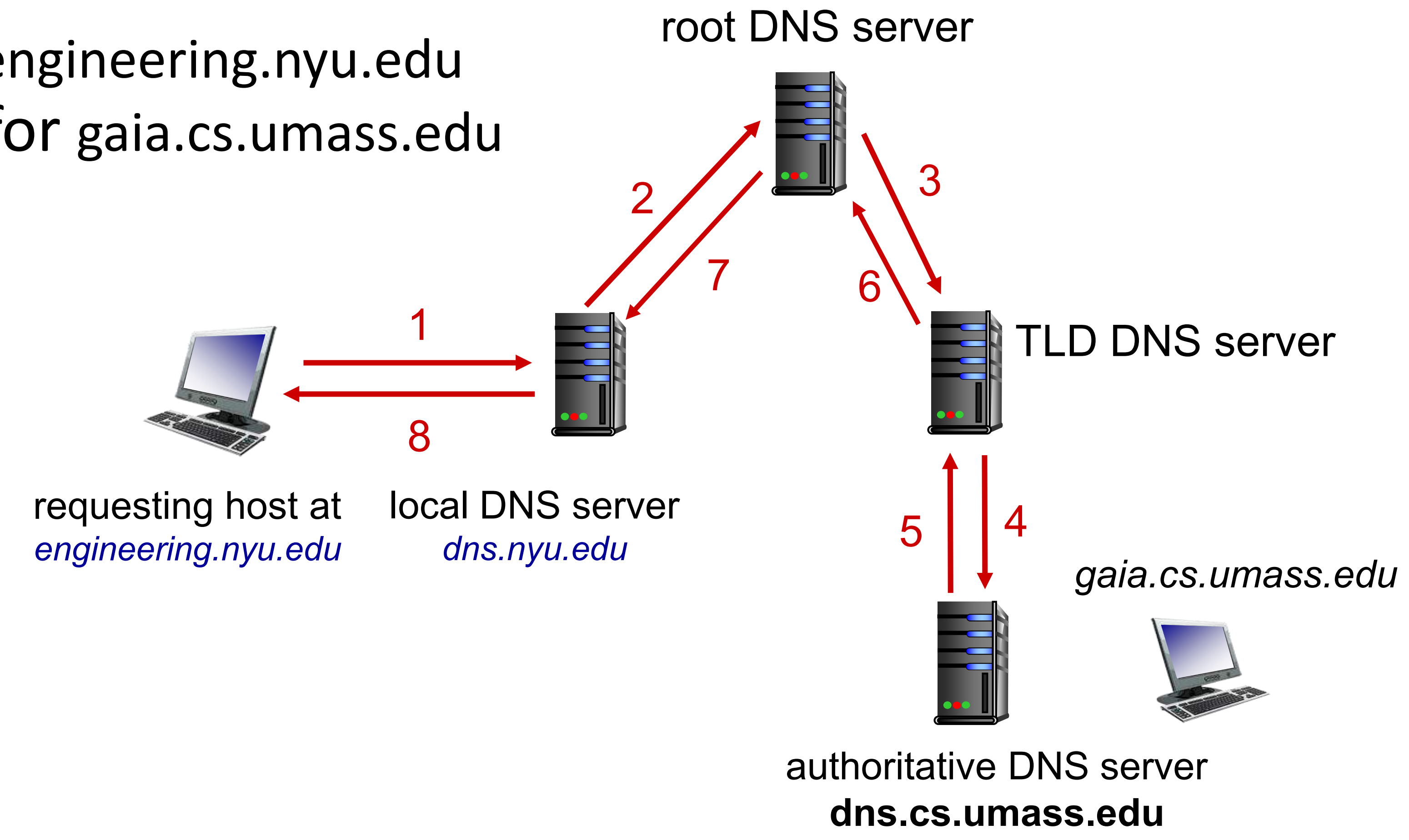
Iterated query:

- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution: recursive query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu



Recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?

Caching and Updating DNS Records

- Suppose that a host **apricot.nyu.edu** queries **dns.nyu.edu** for the IP address for the hostname **cnn.com**. After an hour later, another NYU host, say, **kiwi.nyu.edu**, also queries **dns.nyu.edu.**
- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best-effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire!
- update/notify mechanisms proposed IETF standard
  - RFC 2136

DNS: distributed database storing resource records (RR)

RR format: (name, value, type, ttl)

type=A

- name is hostname
- value is IP address

relayl.bar.foo.com, 145.37.93.126, A

type=NS

- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

foo.com, dns.foo.com, NS

type=CNAME

- name is alias name for some “canonical” (the real) name
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical name

ibm.com, servereast.backup2.ibm.com, CNAME

type=MX

- value is canonical name of a mailserver associated with alias hostname name

example.com, mail.example.com, MX

DNS Protocol Messages

DNS *query* and *reply* messages, both have same *format:*

message header:

- identification: 16 bit # for query, reply to query uses same #
- flags:
  - query or reply (1-bit)
  - recursion desired
  - recursion available
  - reply is authoritative

2 bytes	2 bytes	
identification	flags	**12 bytes**
# questions	# answer RRs	
# authority RRs	# additional RRs	
questions (variable # of questions)		Name, type fields for a query
answers (variable # of RRs)		RRs in response to query
authority (variable # of RRs)		Records for authoritative servers
additional info (variable # of RRs)		Additional “helpful” info that may be used

DNS Protocol Messages

DNS *query* and *reply* messages, both have same *format:*

2 bytes	2 bytes
identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

- questions (variable # of questions): name, type fields for a query
- answers (variable # of RRs): RRs in response to query
- authority (variable # of RRs): records for authoritative servers
- additional info (variable # of RRs): additional " helpful" info that may be used

Type (for example, A, NS, CNAME, and MX), the Value, and the TTL.

Emulating Local DNS Server (Step 1: Ask Root)

Directly send the query to this server.

```
seed@ubuntu:~$ dig @a.root-servers.net www.example.net

(Only a portion of the reply is shown here)
;; QUESTION SECTION:
;www.example.net.                IN      A

;; AUTHORITY SECTION:
net.                    172800   IN      NS      m.gtld-servers.net.
net.                    172800   IN      NS      l.gtld-servers.net.
net.                    172800   IN      NS      k.gtld-servers.net.

;; ADDITIONAL SECTION:
m.gtld-servers.net.     172800   IN      A       192.55.83.30
l.gtld-servers.net.     172800   IN      A       192.41.162.30
k.gtld-servers.net.     172800   IN      A       192.52.178.30
```

No answer (the root does not know the answer)

Go ask them!

Steps 2-3: Ask .net & example.net servers

```
seed@ubuntu:~$ dig @m.gtld-servers.net www.example.net

;; QUESTION SECTION:
;www.example.net.                IN      A

;; AUTHORITY SECTION:
example.net.            172800  IN      NS      a.iana-servers.net.
example.net.            172800  IN      NS      b.iana-servers.net.

;; ADDITIONAL SECTION:
a.iana-servers.net.     172800  IN      A       199.43.132.53
b.iana-servers.net.     172800  IN      A       199.43.133.53
```

Ask a .net nameservers.

Go ask them!

```
seed@ubuntu:$ dig @a.iana-servers.net www.example.net

;; QUESTION SECTION:
;www.example.net.               IN      A

;; ANSWER SECTION:
www.example.net.        86400   IN      A       93.184.216.34
```

Ask an example.net nameservers.

Finally got the answer

Inserting records into DNS

Example: new startup “Network Utopia”

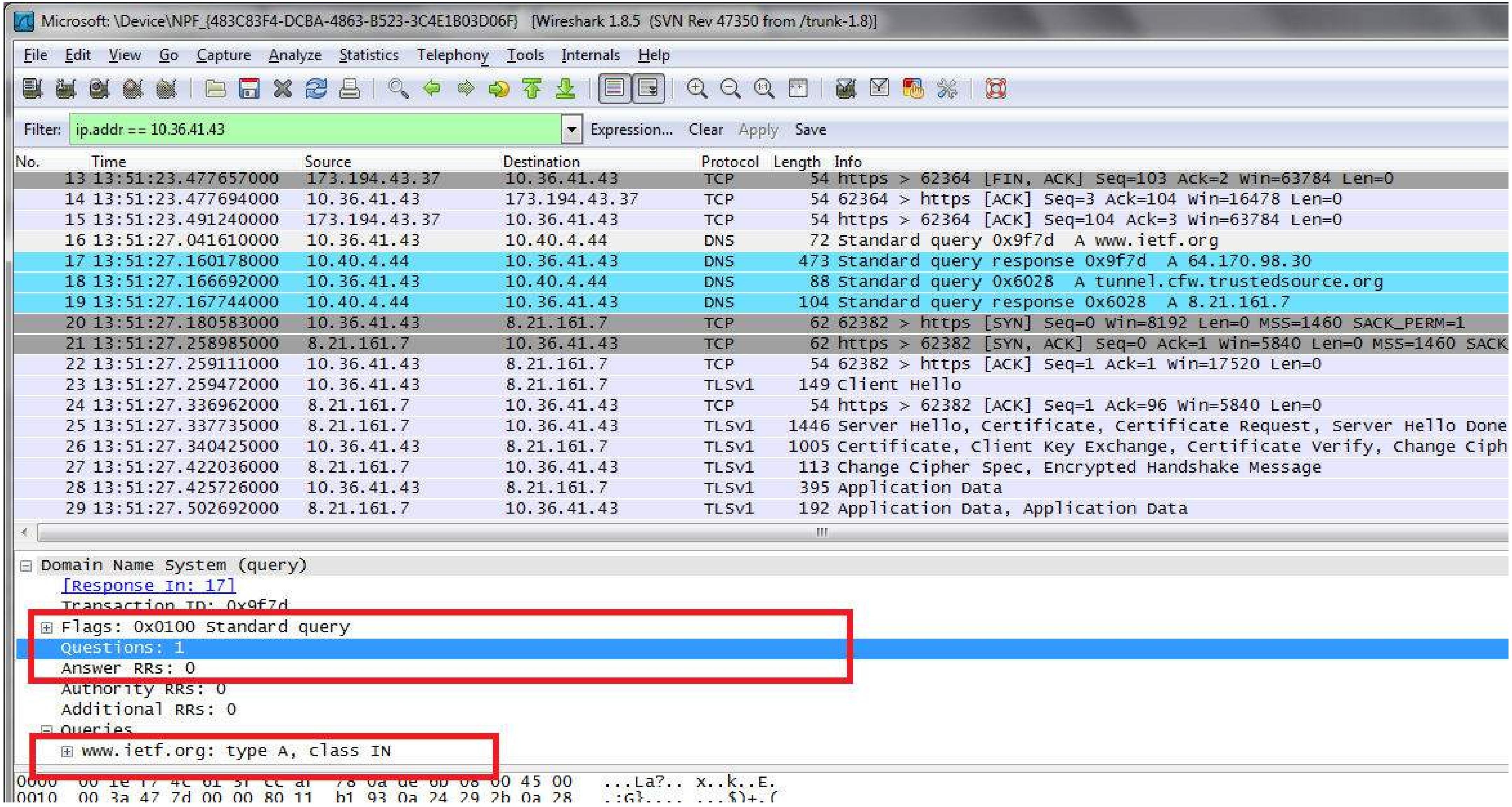
- register name **networkuptopia.com** at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts NS, A RRs into .com TLD server:

    `(networkutopia.com, dns1.networkutopia.com, NS)`

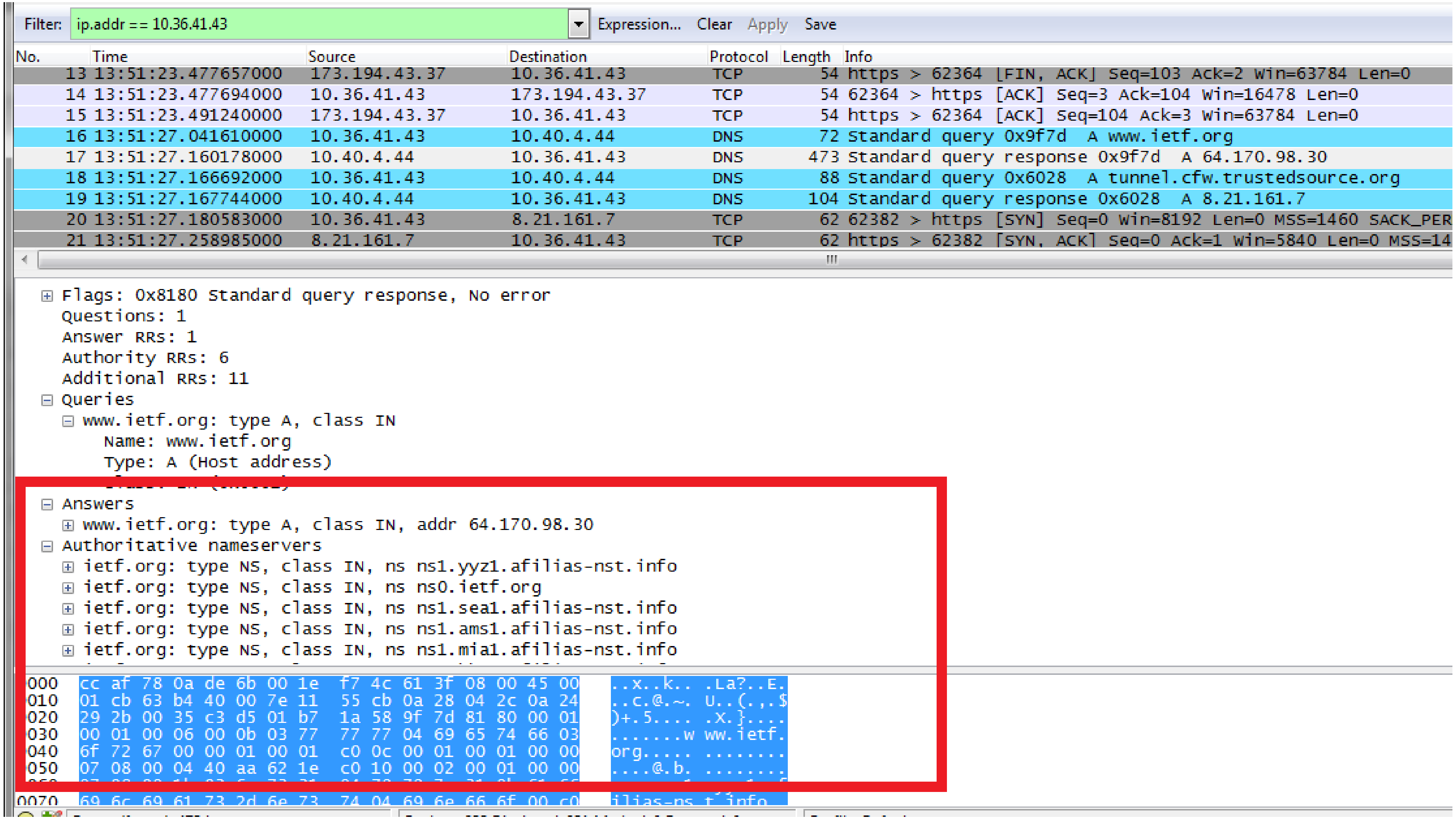
    `(dns1.networkutopia.com, 212.212.212.1, A)`
- create authoritative server locally with IP address `212.212.212.1`
  - type A record for www.networkuptopia.com
  - type MX record for networkutopia.com

COMPUTER NETWORKS

DNS Request - Wireshark Packet Capture



DNS Response - Wireshark Packet Capture



Suggested Readings

- DNS (Domain Name System) – Explained – https://youtu.be/JkEYOt08-rU
- How a DNS Server (Domain Name System) works – https://youtu.be/rdVPflECed8
- Wireshark Lab: DNS v7.0 – http://www-net.cs.umass.edu/wireshark-labs/Wireshark_DNS_v7.0.pdf

Q&A

Thank You
For Your Attention

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

Unit – 2 Application Layer

2.1 Principles of Network Applications

2.2 Web, HTTP and HTTPS

2.3 The Domain Name System

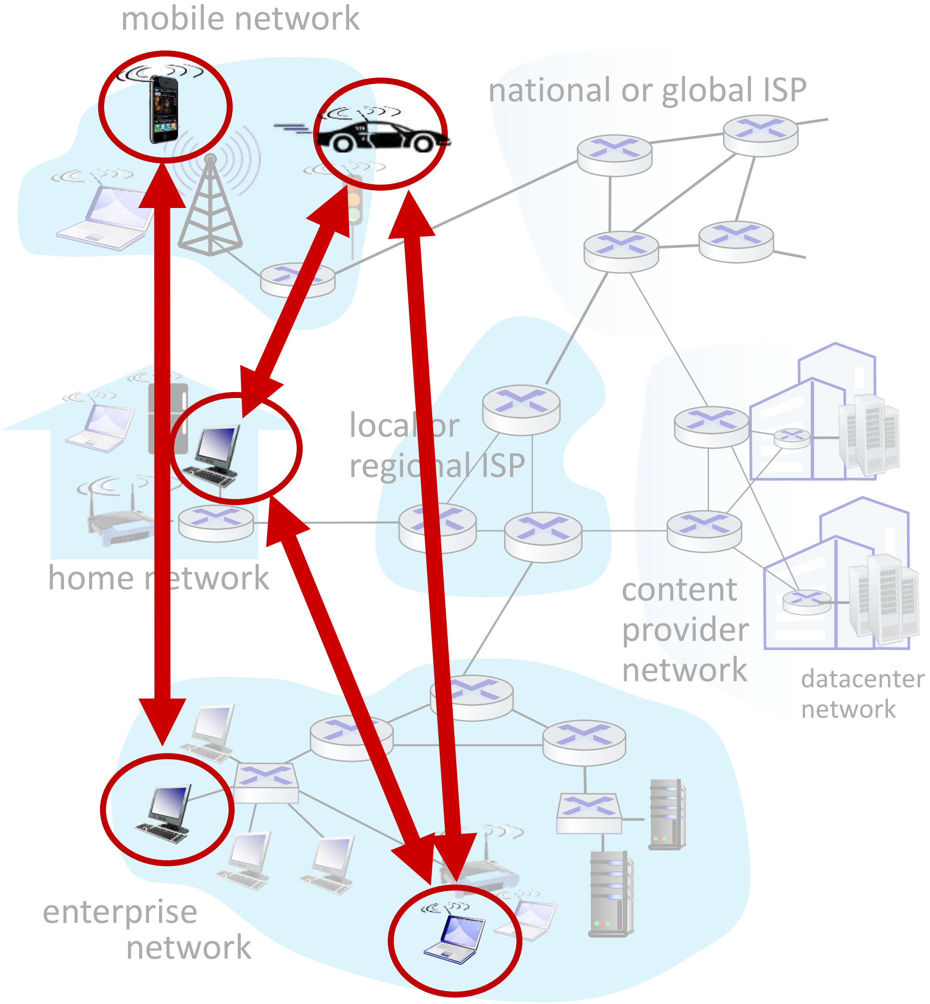
2.4 P2P Applications

2.5 Socket Programming with TCP & UDP

2.6 Other Application Layer Protocols

Peer-to-peer (P2P) architecture

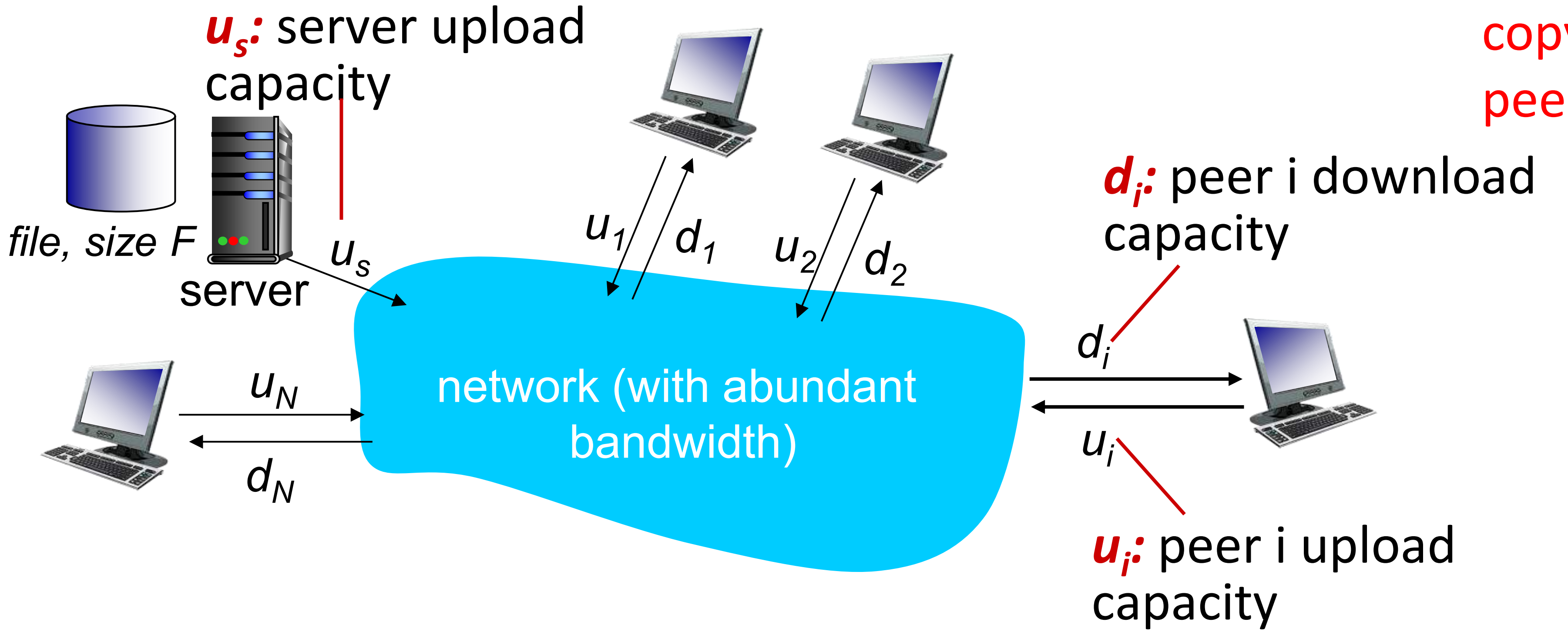
- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- examples: P2P file sharing (BitTorrent), media streaming (Spotify), VoIP (Skype)



File distribution: client-server vs P2P

Q: how much time to distribute file (size *F*) from one server to *N* *peers*?

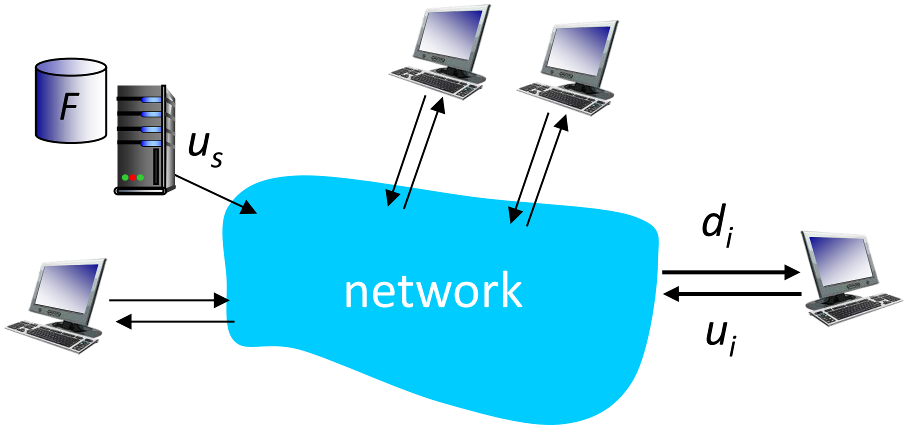
- peer upload/download capacity is limited resource



The **distribution time** is the time it takes to get a copy of the file to all *N* peers.

File distribution time: client-server

- *server transmission:* must sequentially send (upload) N file copies:
  - time to send one copy: F/u_s
  - time to send N copies: NF/u_s



- *client:* each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}

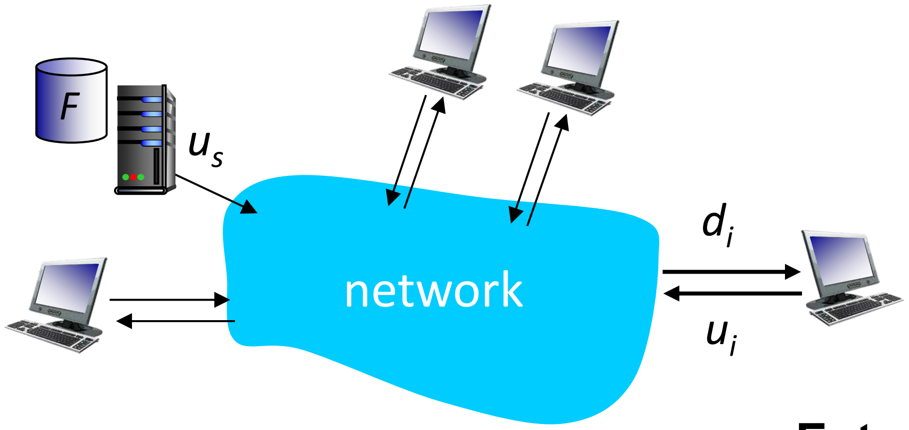
time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} > max\{NF/u_s, F/d_{min}\}$$

increases linearly in N

File distribution time: P2P

- *server transmission:* must upload at least one copy:
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *clients:* as aggregate must download *NF* bits
  - max upload rate (limiting max download rate) is $u_s + \Sigma u_i$



Total upload capacity of the system as a whole

Eqtn - provides a lower bound for the minimum distribution time for the P2P architecture.

time to distribute F to N clients using P2P approach

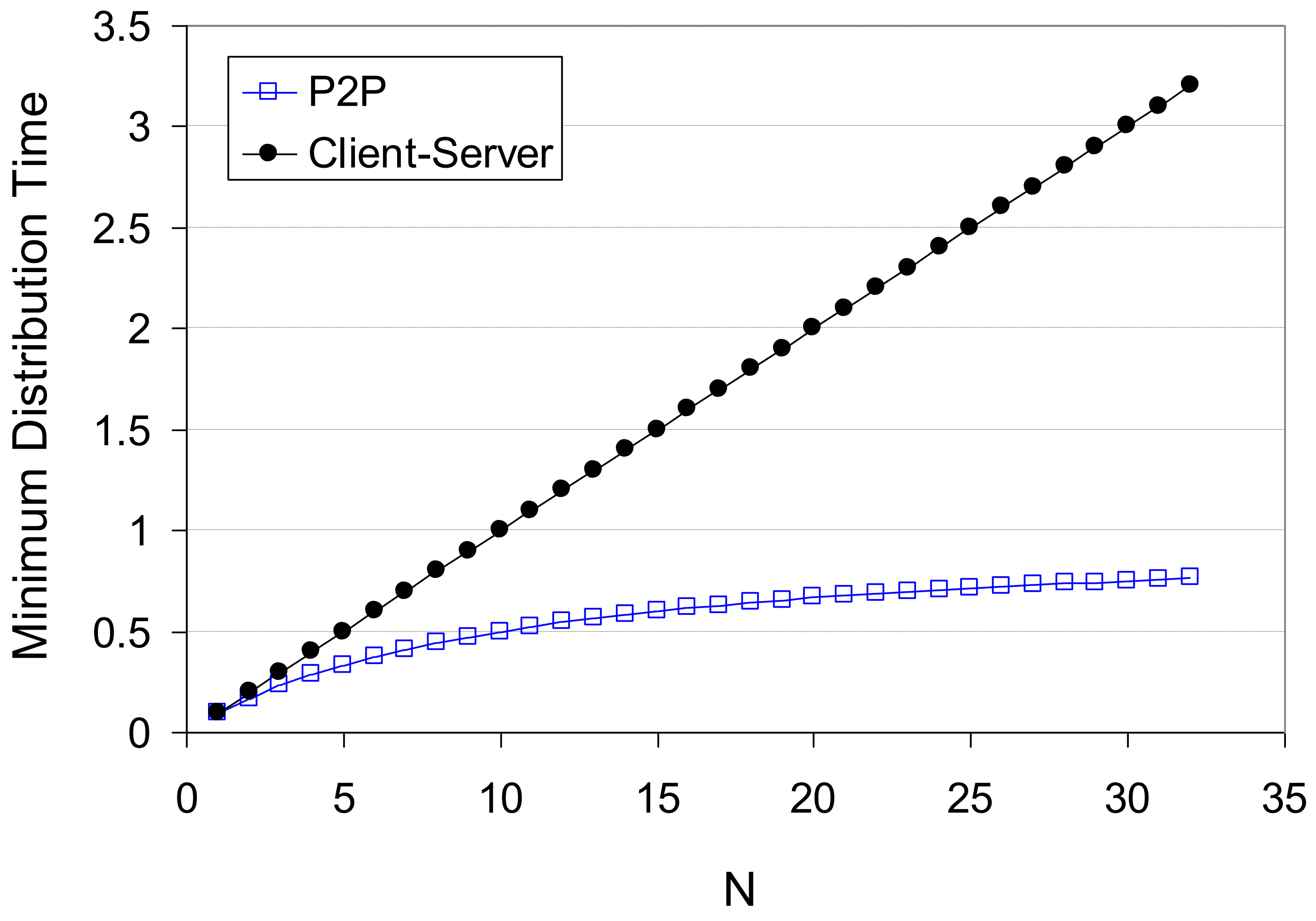
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

… but so does this, as each peer brings service capacity

Client-server vs. P2P: example

Client (all peers) upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$



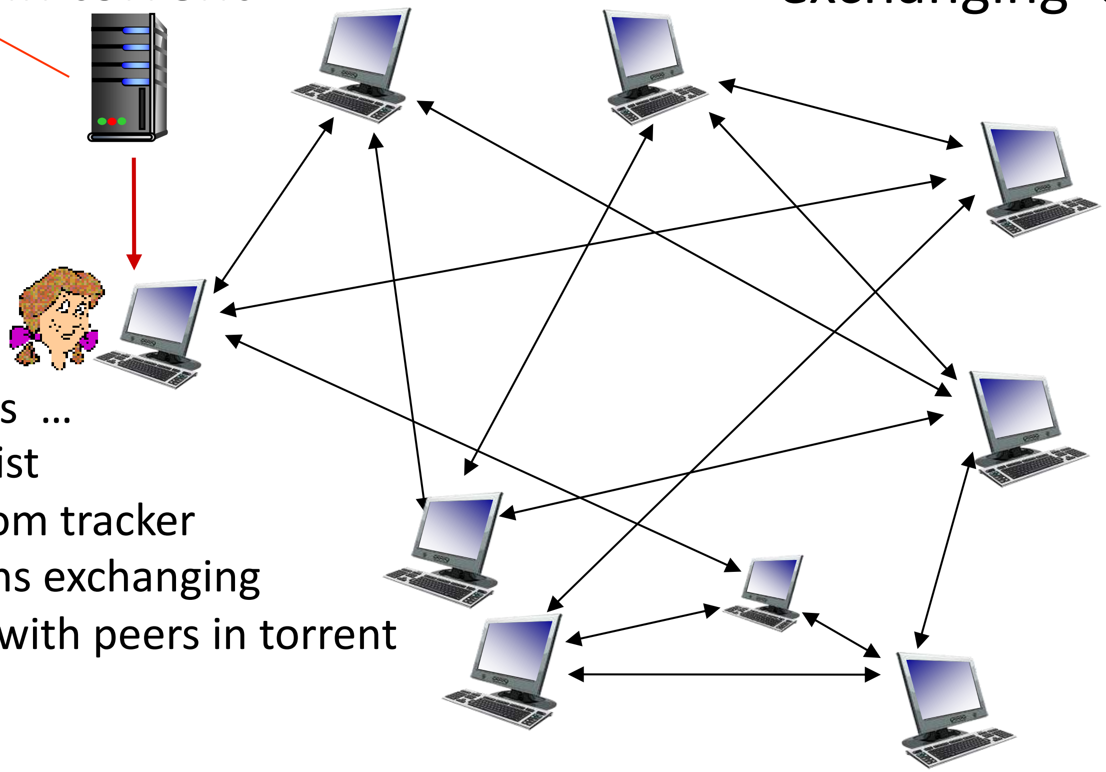
- A peer can transmit the entire file in one hour.
- The server transmission rate is 10 times the peer upload rate.
- Peer download rates are set large enough so as not to have an effect.

P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

tracker: tracks peers participating in torrent

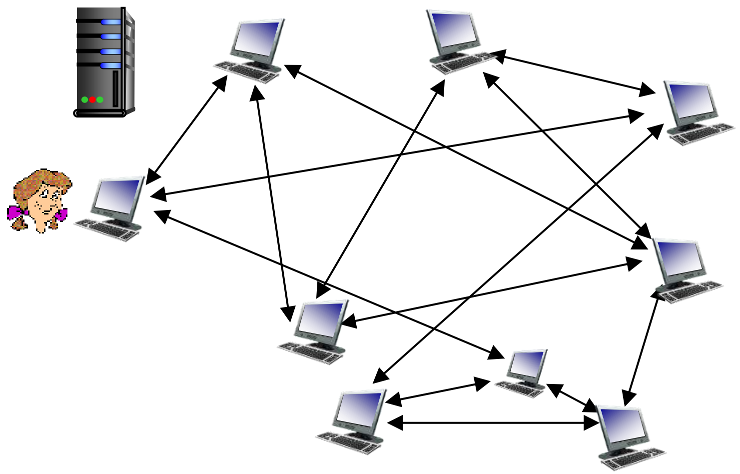
torrent: group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")



- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

Requesting chunks:

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, **rarest first**

Sending chunks: tit-for-tat

- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "**optimistically unchoke**" this peer
  - newly chosen peer may join top 4

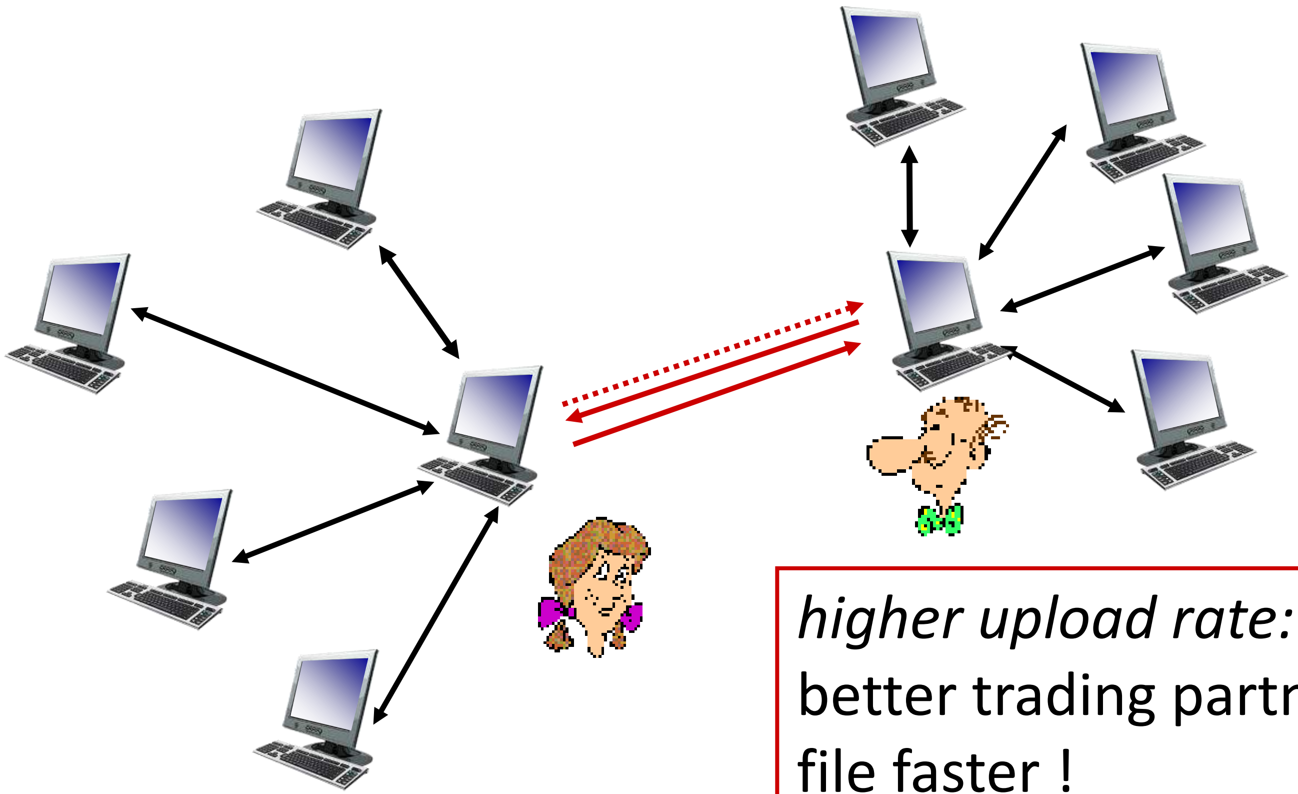
BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates

(3) Bob becomes one of Alice's top-four providers

Pieces (mini-chunks), pipelining, random first selection, endgame mode, and anti-snubbing



higher upload rate: find better trading partners, get file faster !

Suggested Readings

- BitTorrent (BTT) White Paper – https://www.bittorrent.com/btt/btt-docs/BitTorrent_(BTT)_White_Paper_v0.8.7_Feb_2019.pdf
- Peer-to-peer networking with BitTorrent – http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf
- Torrents Explained: How BitTorrent Works – https://youtu.be/urzQeD7ftbI

Thank You
For Your Attention

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

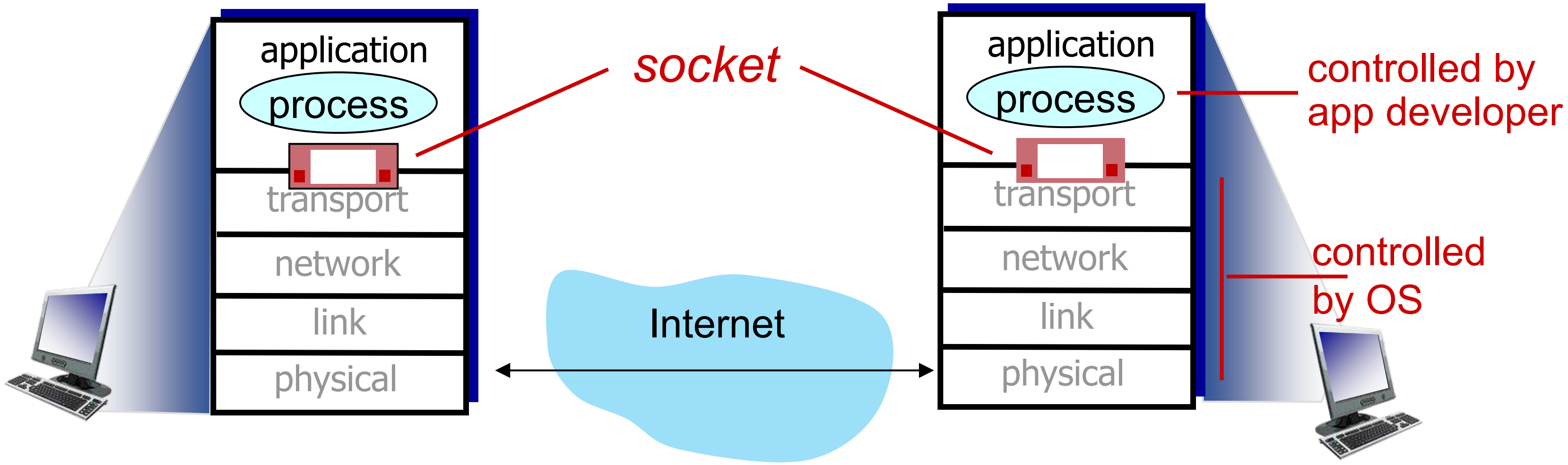
Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

Unit – 2 Application Layer

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

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



Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

UDP: no "connection" between client & server

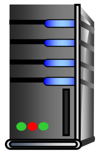
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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

Client/Server socket interaction: UDP



server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example app: UDP client

Python UDPClient

include Python's socket library →
```
from socket import *
serverName = 'hostname'
serverPort = 12000
```
create UDP socket for server →
```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```
get user keyboard input →
```
message = raw_input('Input lowercase sentence:')
```
attach server name, port to message; send into socket →
```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
```
read reply characters from socket into string →
```
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```
print out received string and close socket →
```
print modifiedMessage.decode()
clientSocket.close()
```

Example app: UDP server

Python UDPServer

	from socket import *
	serverPort = 12000
create UDP socket →	serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 →	serverSocket.bind(('', serverPort))
	print ("*The server is ready to receive*")
loop forever →	while True:
Read from UDP socket into message, getting client's address (client IP and port) →	message, clientAddress = serverSocket.recvfrom(2048)
	modifiedMessage = message.decode().upper()
send upper case string back to this client →	serverSocket.sendto(modifiedMessage.encode(), clientAddress)

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

Application Layer

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

Unit – 2 Application Layer

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:

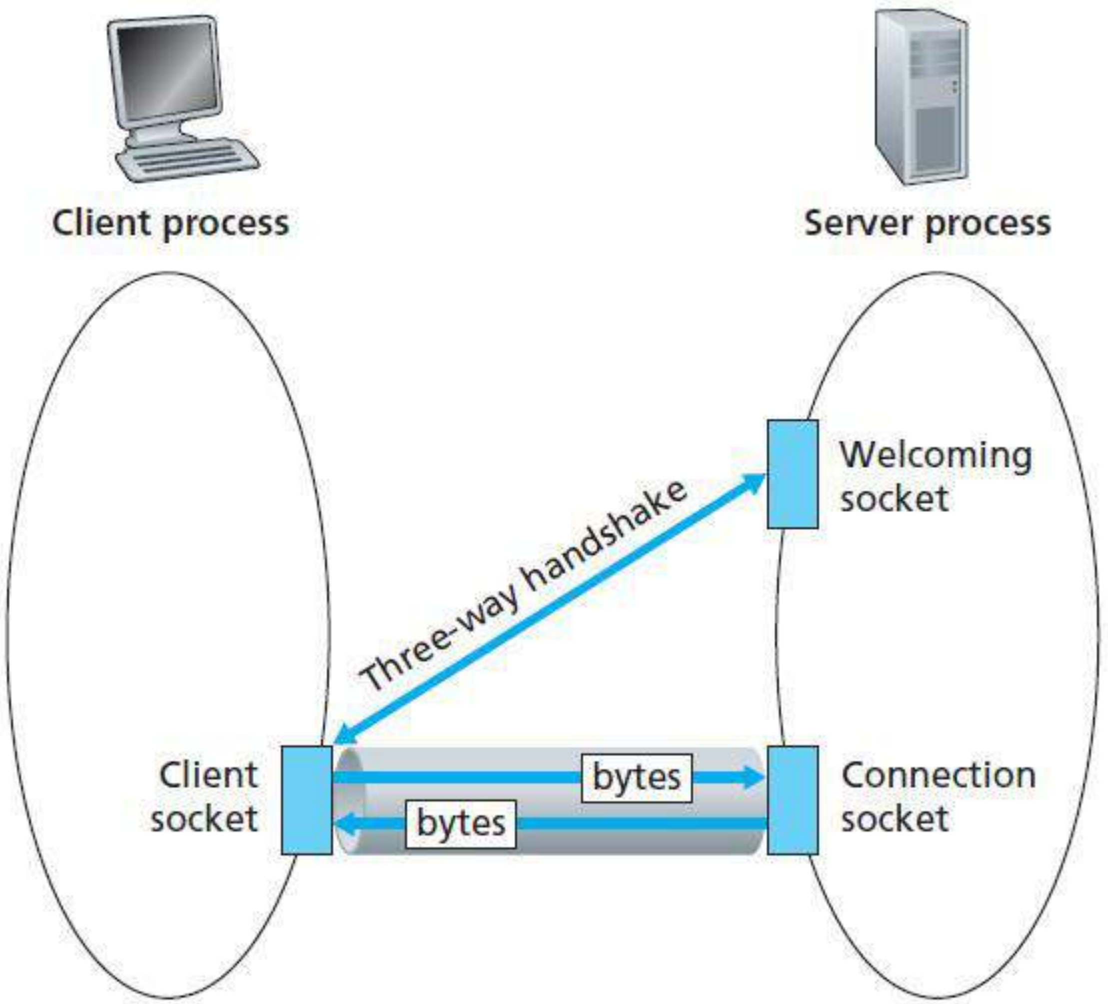
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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 (more in Chap 3)

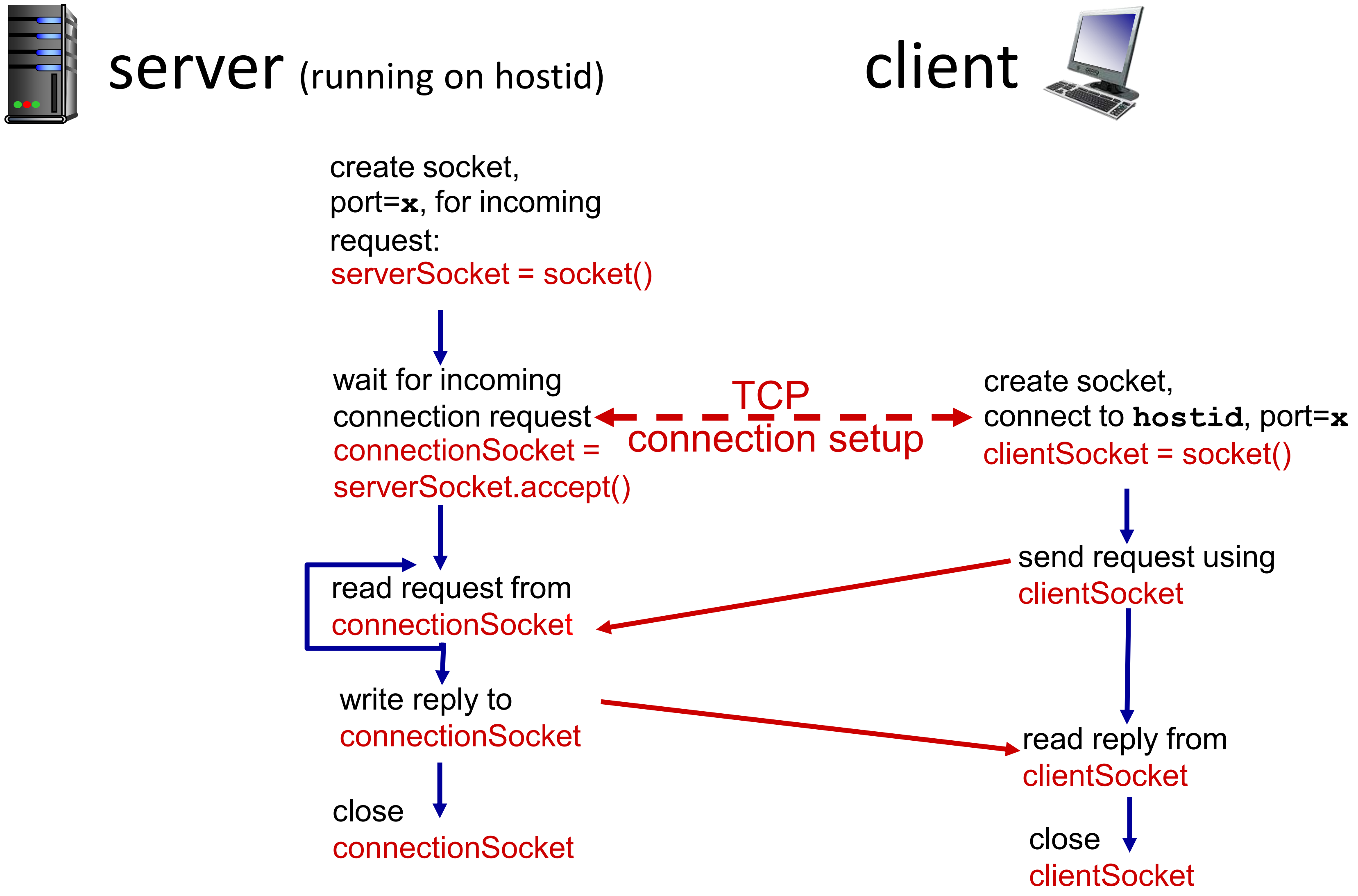
Application viewpoint

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

The TCPServer Process has Two Sockets



Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

create TCP socket for server, remote port 12000

No need to attach server name, port

```
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.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)        # create TCP welcoming socket
serverSocket.bind(('',serverPort))
serverSocket.listen(1)                            # server begins listening for incoming TCP requests
print 'The server is ready to receive'
while True:                                       # loop forever
    connectionSocket, addr = serverSocket.accept()   # server waits on accept() for incoming requests, new socket created on return

    sentence = connectionSocket.recv(1024).decode()  # read bytes from socket (but not address as in UDP)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                       encode())
    connectionSocket.close()                     # close connection to this client (but not welcoming socket)
```

- create TCP welcoming socket → `serverSocket = socket(AF_INET,SOCK_STREAM)`
- server begins listening for incoming TCP requests → `serverSocket.listen(1)`
- loop forever → `while True:`
- server waits on accept() for incoming requests, new socket created on return → `connectionSocket, addr = serverSocket.accept()`
- read bytes from socket (but not address as in UDP) → `sentence = connectionSocket.recv(1024).decode()`
- close connection to this client (but *not* welcoming socket) → `connectionSocket.close()`

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834

COMPUTER NETWORKS

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

COMPUTER NETWORKS

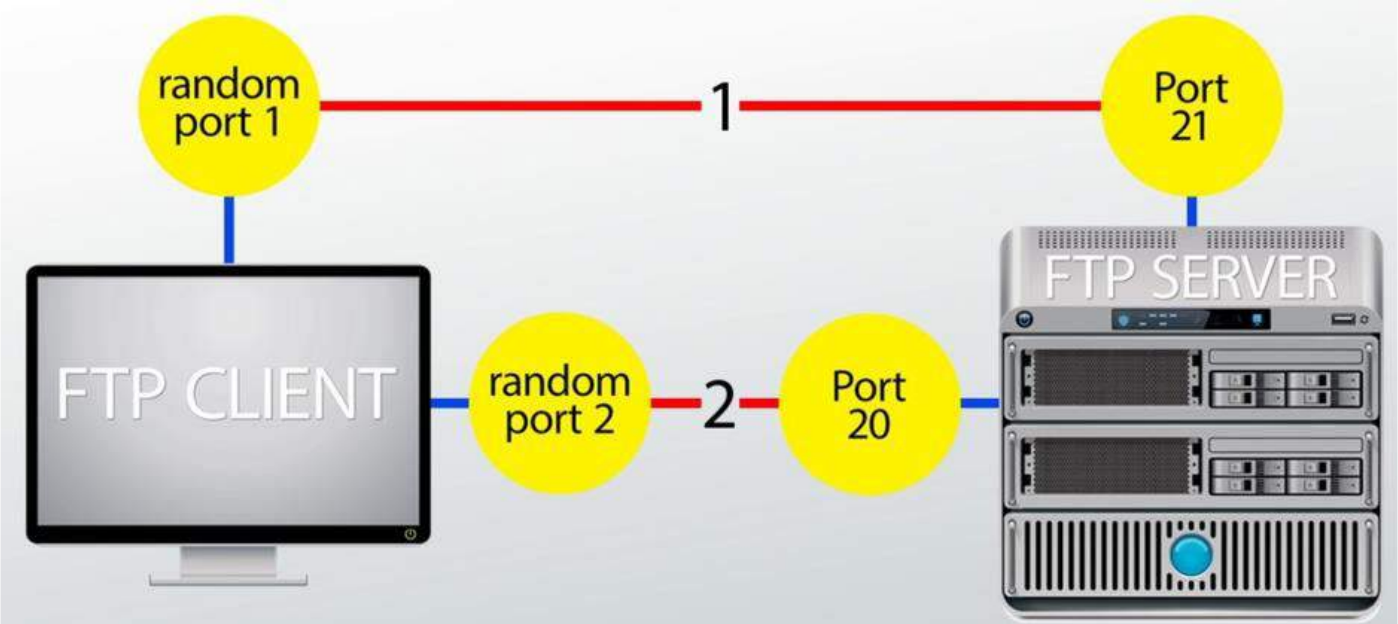
Application Layer

Sivaraman Eswaran Ph.D.

Department of Computer Science and Engineering

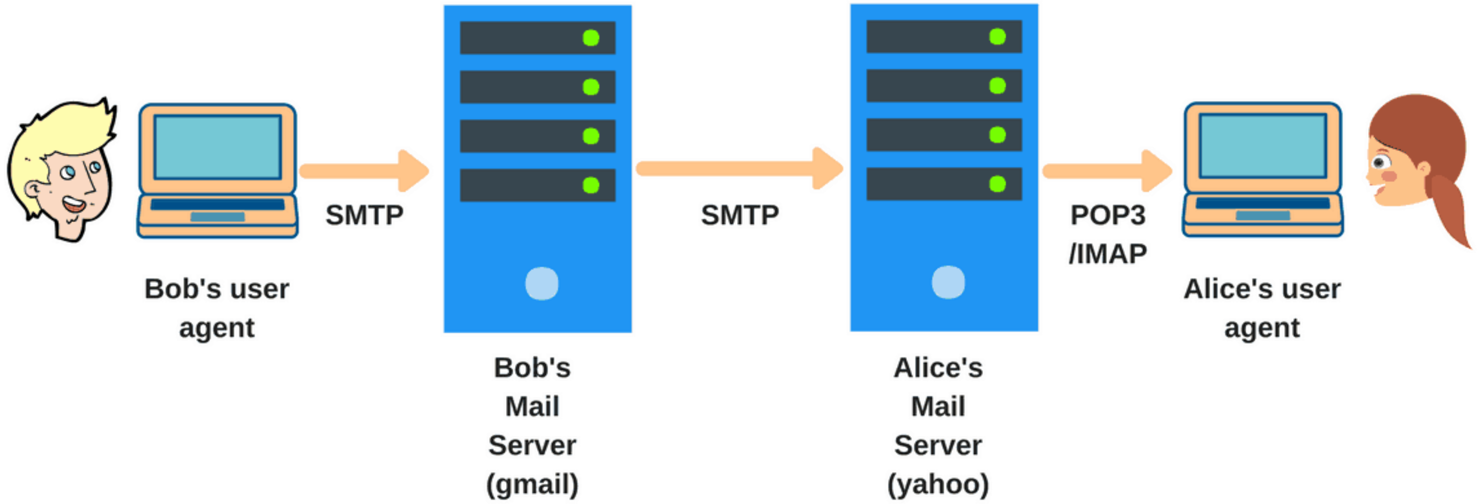
Unit – 2 Application Layer

- File Transfer Protocol (FTP) - used to exchange large files on the internet TCP
- Invoked from the command prompt or some GUI.
- Allows to update (delete, rename, move, and copy) files at a server.
- Data connection (Port No. 20) & Control connection (Port No. 21)



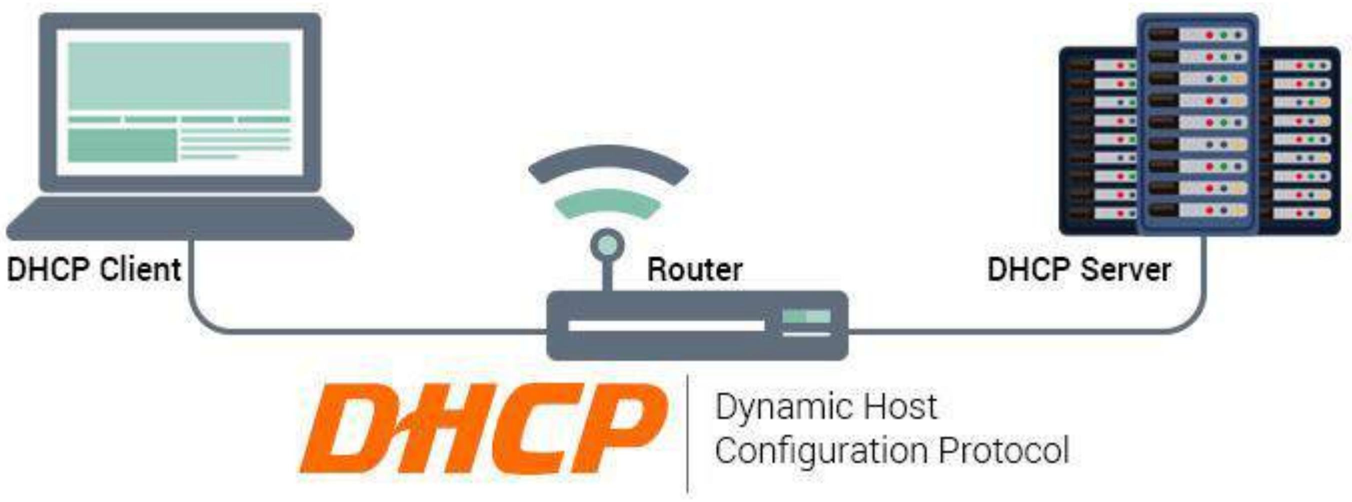
Other Application Layer Protocols - SMTP

- **Simple Mail Transfer Protocol** - an internet standard for e-mail Transmission.
- Connections are secured with SSL (Secure Socket Layer).
- Messages are stored and then forwarded to the destination (relay).
- SMTP uses a port number 25 of TCP.



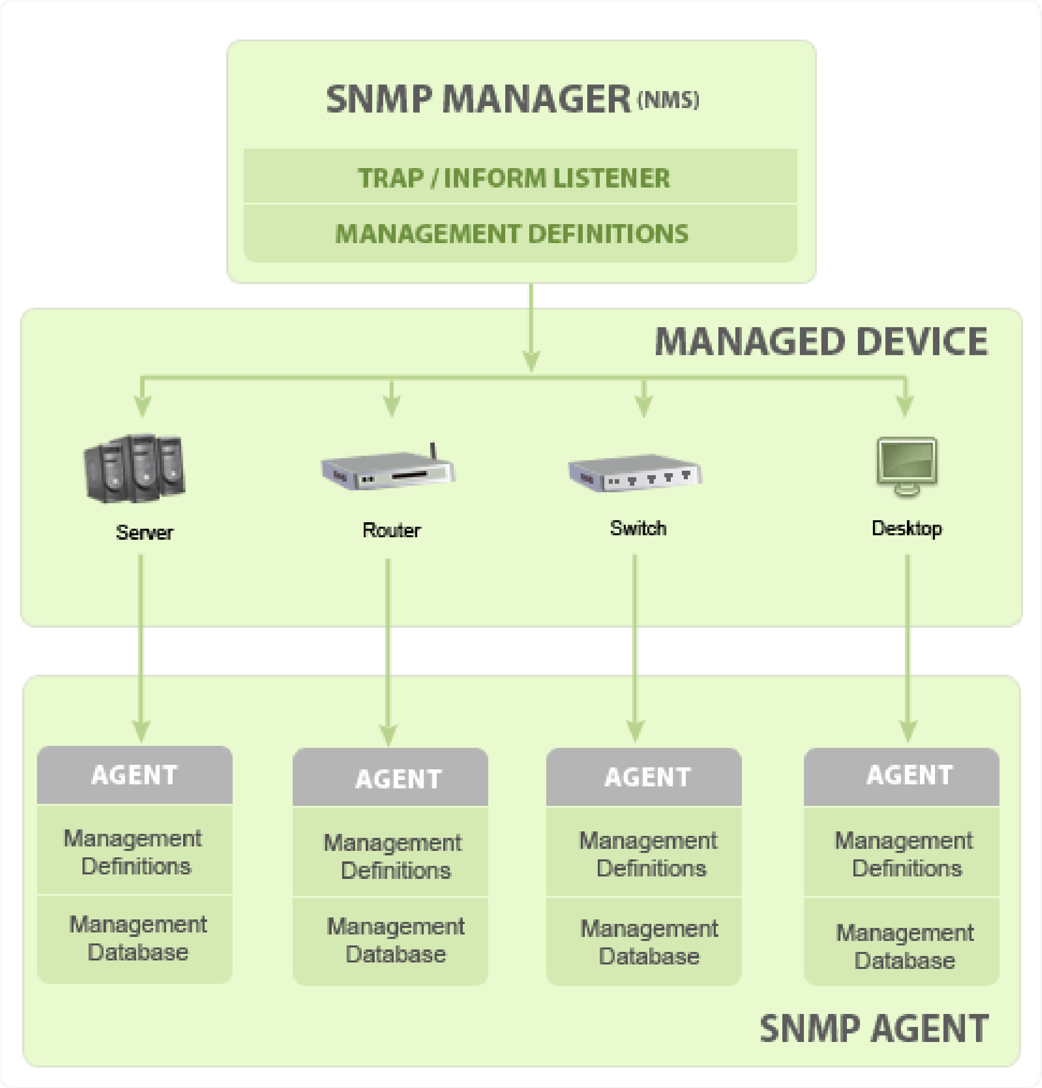
Other Application Layer Protocols - DHCP

- **Dynamic Host Configuration Protocol** - assign IP addresses to computers in a network dynamically.
- IP addresses may change even when computer is in network (DHCP leases).
- DHCP port number for server is 67 and for the client is 68.
- A client-server model & based on **discovery, offer, request, and ACK**.
- Includes subnet mask, DNS server address, default gateway



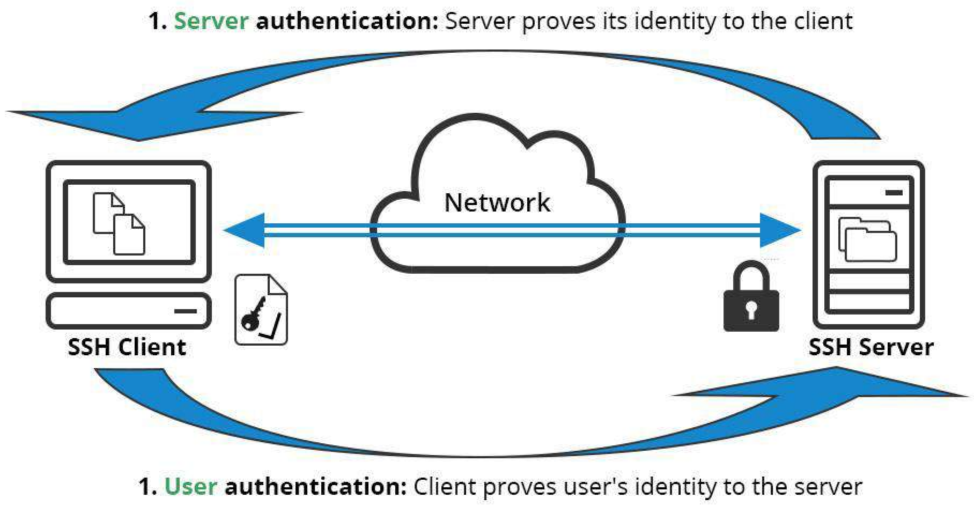
Other Application Layer Protocols - SNMP

- **Simple Network Management Protocol** - exchange management information between network devices.
- Basic components & functionalities
  - SNMP Manager
  - Managed Devices
  - SNMP Agents
  - MIB (Management Information Base)



Other Application Layer Protocols – Telnet & SSH

- Allows a user to communicate with a remote device.
- Used mostly by network admin to remotely access and manage devices.
- Telnet client and server installed – uses TCP port no. 23
- SSH – uses public key **encryption** & TCP port 22 by default.



Summary of Application Layer Protocols

Port #	Application Layer Protocol	Type	Description
20	FTP	TCP	File Transfer Protocol - data
21	FTP	TCP	File Transfer Protocol - control
22	SSH	TCP/UDP	Secure Shell for secure login
23	Telnet	TCP	Unencrypted login
25	SMTP	TCP	Simple Mail Transfer Protocol
53	DNS	TCP/UDP	Domain Name Server
67/68	DHCP	UDP	Dynamic Host
80	HTTP	TCP	HyperText Transfer Protocol
123	NTP	UDP	Network Time Protocol
161,162	SNMP	TCP/UDP	Simple Network Management Protocol
389	LDAP	TCP/UDP	Lightweight Directory Authentication Protocol
443	HTTPS	TCP/UDP	HTTP with Secure Socket Layer

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - DNS
  - P2P: BitTorrent
- socket programming:
  TCP, UDP sockets

Summary (more)

Most importantly: learned about *protocols*!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data:* info(payload) being communicated

important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- “complexity at network edge”

THANK YOU

Sivaraman Eswaran Ph.D.
Department of Computer Science and Engineering

sivaramane@pes.edu

+91 80 6666 3333 Extn 834