Computer Networks

Chapter 2 – Application Layer

Edition8-2

Topics

- Application-layer concepts:
  - network services required by applications
  - clients and servers and peer to peer architectures
  - inter-processes communication
  - transport layer interfaces
- Network applications in detail
  - Web,
  - e-mail,
  - DNS,
  - peer-to-peer (P2P) file distribution, and video streaming.
- Network application development, over both TCP and UDP (socket interface) in Python

Applications and Computer Networks

- No useful applications: No need for networking infrastructure and protocols
- Applications are driving force behind Internet's success, motivating people in homes, schools, governments, and businesses to make Internet an integral part of their daily activities
- Internet applications: e-mail, remote access to computers, file transfers, World Wide Web, Voice over IP and video conferencing such as Skype, user generated video such as YouTube, movies on demand such as Netflix, multiplayer online games, social networking applications such as Facebook, Instagram, and Twitter (which have created human networks on top of Internet's network), cloud computing (on-demand availability of computer system resources, especially data storage and computing power), e-banking, e-government, e-business, metaverse, …

Applications and Computer Networks

- mobile apps
  - check-in
  - road-traffic
  - mobile payment apps
  - messaging apps (e.g. WhatsApp)
  - video call
  - online games
  - …

Some network applications

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

Some application layer protocols

- BGP (routing)
- DHCP (ad configuration)
- DNS (Direc
- FTP (file transfer)
- HTTP, HTTPS (web)
- IMAP (email)
- LDAP (active directory)
- MGCP (voice over IP)
- MQTT (telemetry transport, IOT)
- NNTP (news articles)
- NTP (clock synchronization)
- POP (email)
- ONC/RPC (Remote Procedure Call)
- RTP (multimedia streams)
- RTMFP (Real Time Media Flow Protocol)
- RTSP (streaming streams)
- RIP (routing)
- SIP (signaling for real-time multimedia)
- ail)
- SNM t)
- SSH (security)
- Telnet (remote terminal)
- TLS/SSL (security)
- XMPP (instant messaging)
- ...

Why so many application layer protocols?

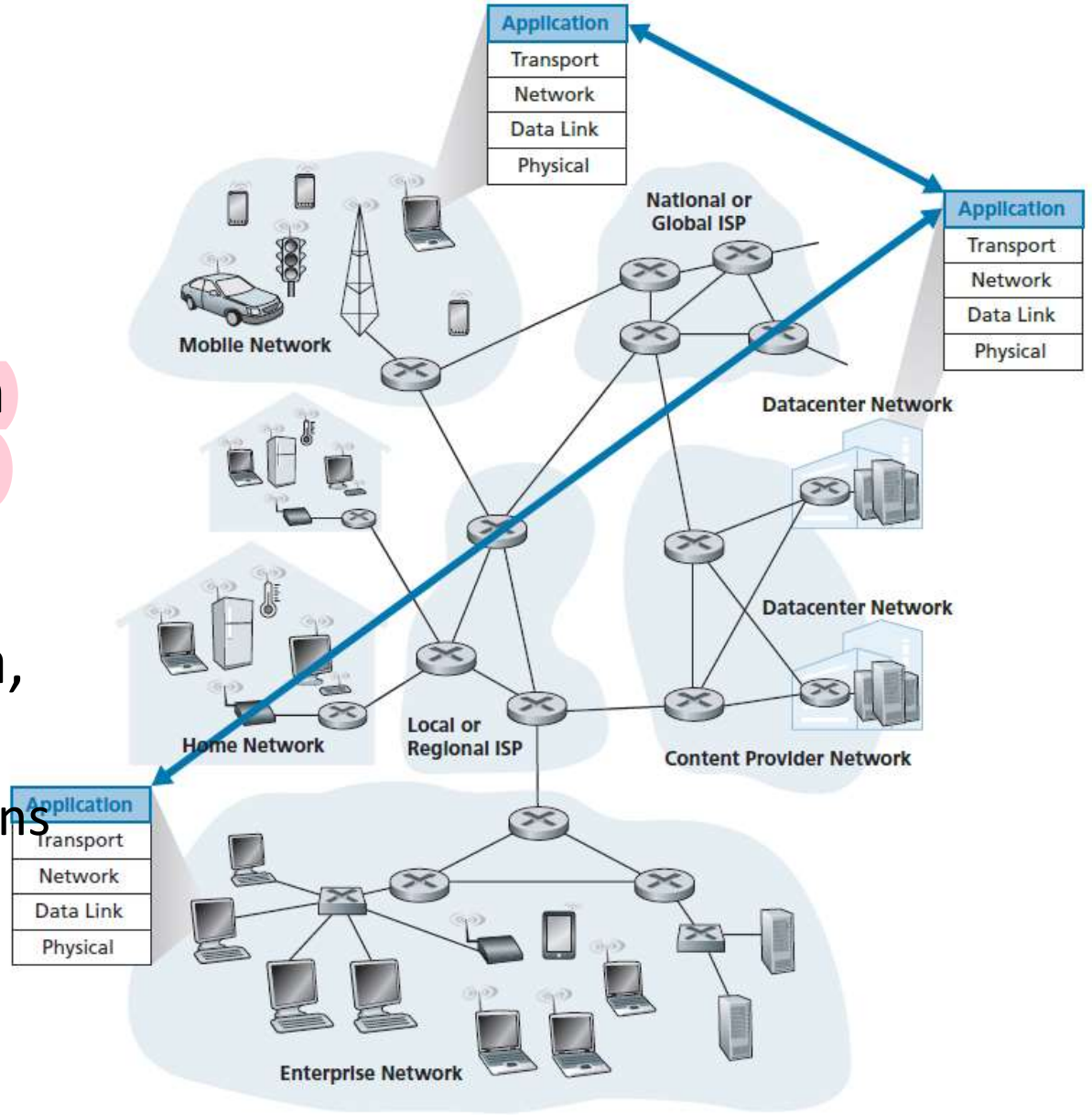
Contents

2.1 Principles of Network Applications

- Network application development: writing programs that run on different end systems and communicate with each other over network
- Web application: two distinct programs that communicate with each other,
  - Browser program running in user's host (desktop, laptop, tablet, smartphone, …)
  - Web server program running in Web server host
- Video on Demand application (Netflix, Section 2.6):
  - Netflix-provided program running on user's smartphone, tablet, or computer
  - Netflix server program running on Netflix server host

Figure 2.1

- Communication for a network application takes place between end systems at application layer
- Any APP runs on multiple host
- APPs could be written, in C, Java, Python, ….
  - no need to write software that runs on routers or link-layer switches
  - so, Net APP development and deployment can be fast



2.1.1 Network Application Architectures

- Network architecture is fixed and provides a specific set of services to applications
- Application architecture is designed by application developer and dictates how application is structured over various end systems

Two predominant architectural paradigms:

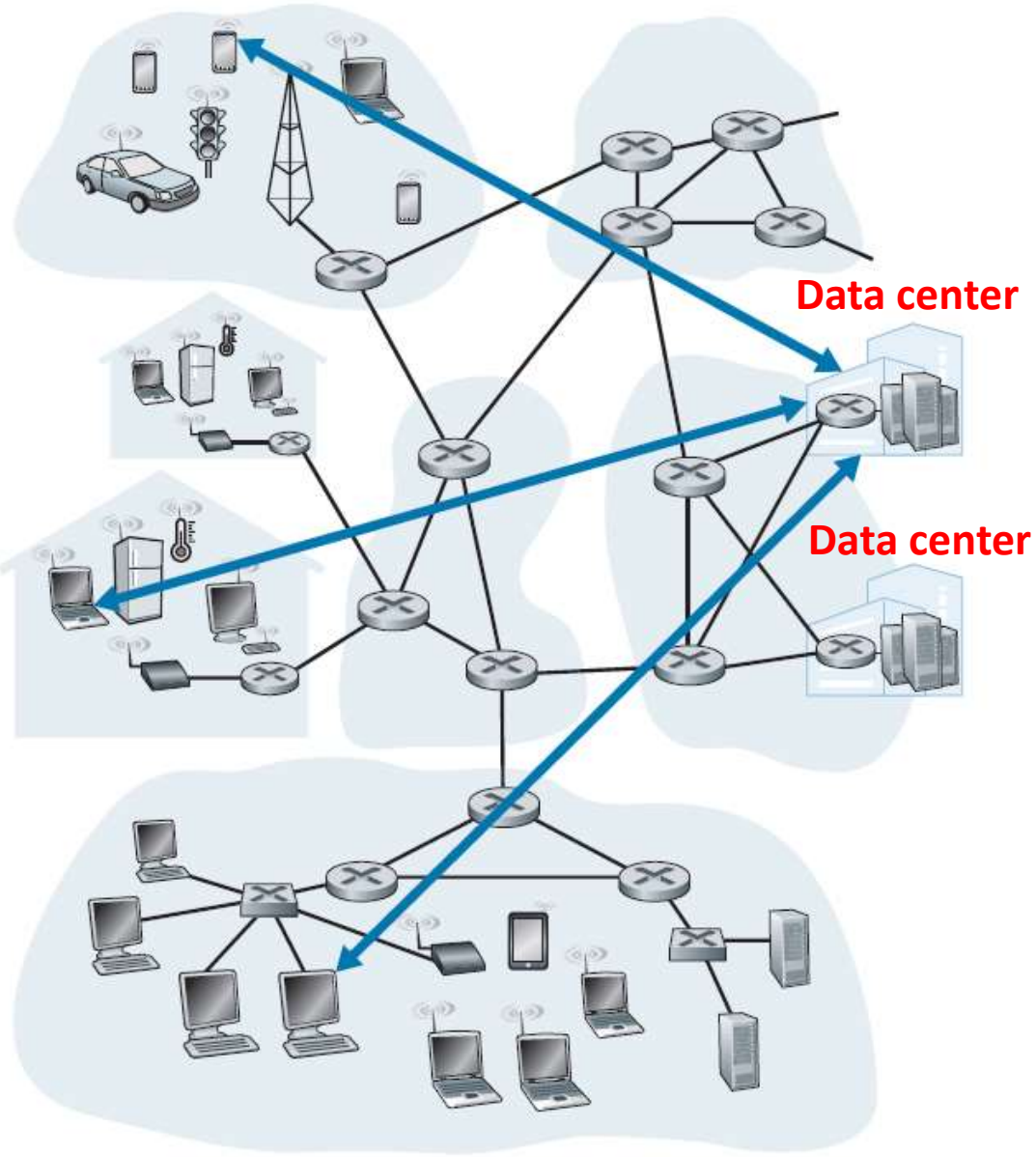
- client-server architecture
- peer-to-peer (P2P) architecture

Client-server architecture

- There is an always-on host, *server*, services requests from many other hosts, *clients*
- Example: Web application services requests from browsers. Web server receives a request for an object, sends requested object to client host
- Clients do not directly communicate with each other
- Server has a fixed IP address
- Client contacts server by sending a packet to server's IP address

Figure 2.2-Client-server

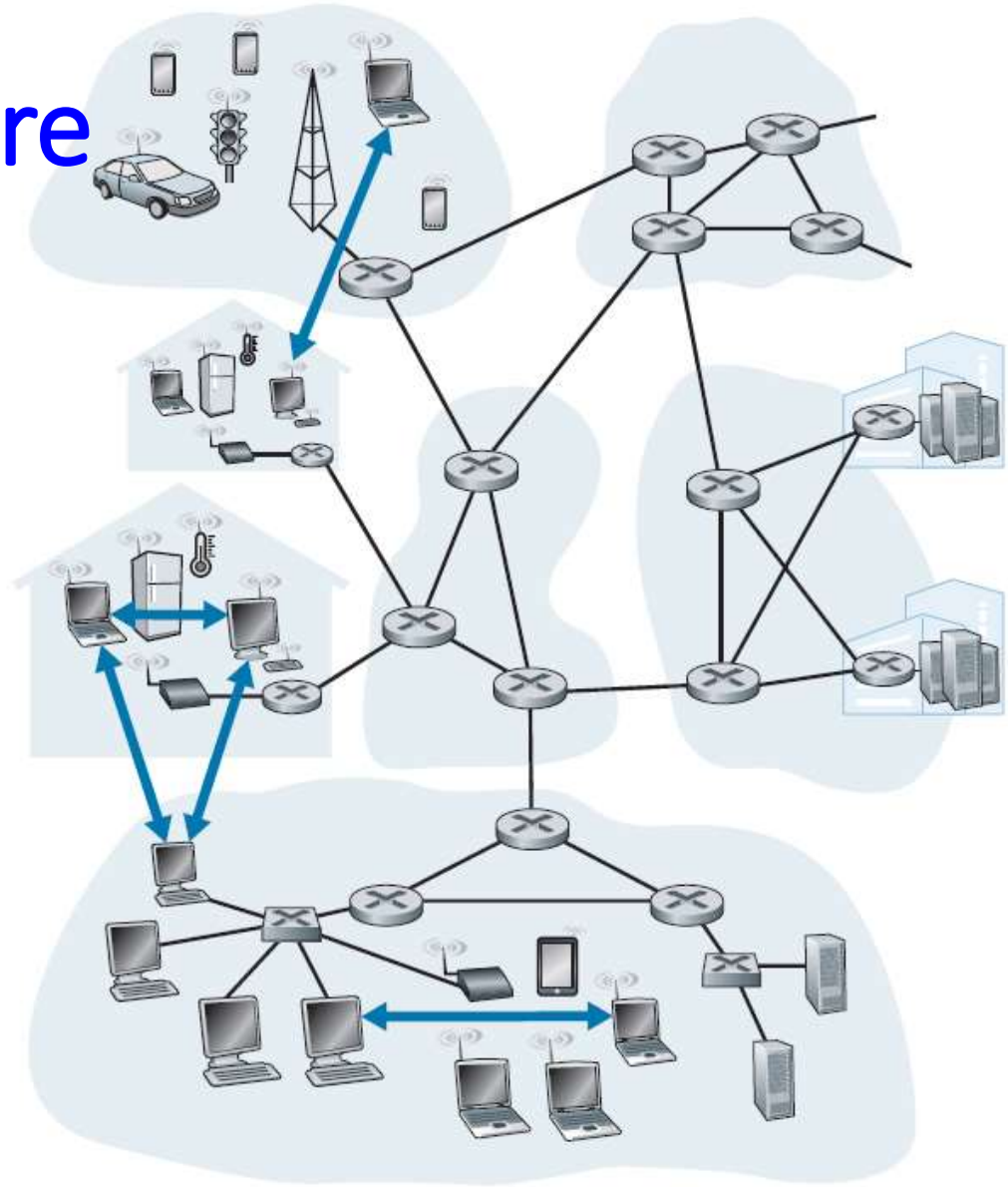
- Often a single-server host is incapable of keeping up with all requests from clients
- A popular social-networking site can overwhelmed if it has only one server
- Data centers!



Servers for popular Apps

- Data center, housing a large number of servers, to create a powerful virtual server
- Most popular Internet services (e.g., Google, Amazon, eBay, Gmail and Yahoo Mail, Facebook, Instagram, …) run in one or more data centers
- Service providers use their own data centers or pay for interconnection and bandwidth costs, sending data from hosting data centers

Figure 2.2-P2P architecture



- Direct communication between App pairs running on connected hosts (peers)
- Peers are desktops and laptops controlled by users (not owned by service provider), most of peers residing in homes, universities, offices
- Peer-to-Peer: Peers communicate without passing through a dedicated server
- Example: BitTorrent (file-sharing application)

P2P architecture

- Self scalability in P2P file-sharing: each peer generates request (demand), and also, each peer also adds service capacity to system by distributing files to other peers
- Cost effective: they normally don't require significant server infrastructure and server bandwidth
- Challenges: they are highly decentralized, and therefor faces:
  - security challenges
  - performance challenges
  - reliability challenges

2.1.2 Processes Communicating

- A process: a program that is running within an end system

- Processes are running on same host: communicate with each other with interprocess communication, **using rules and details that are governed by host's operating system**

2.1.2 Processes Communicating

- Processes running on *different* hosts (with potentially different operating systems): communicate by exchanging messages across network, **using rules and details that are governed by application layer protocol**
  - Sending process creates and sends messages into network; receiving process receives messages and possibly responds by sending messages back
  - In Internet, processes reside in application layer

Client and Server Processes

- Pair of processes send messages to each other over a network
- For each pair of communicating processes, we typically label one of two processes as client and other process as server
- Example of client-server Apps: Web client browser process exchanges messages with a Web server process
- Example of P2P Apps (file sharing): peer that is downloading file is labeled as client, and peer that is uploading file is labeled as server

Client and Server Processes

- P2P App: a process can be both a client and a server
  - A process in a P2P file-sharing can both upload and download files
- For any given communication session between a pair of processes, we can still label one process as client and other process as server.

- Client and server processes definition:

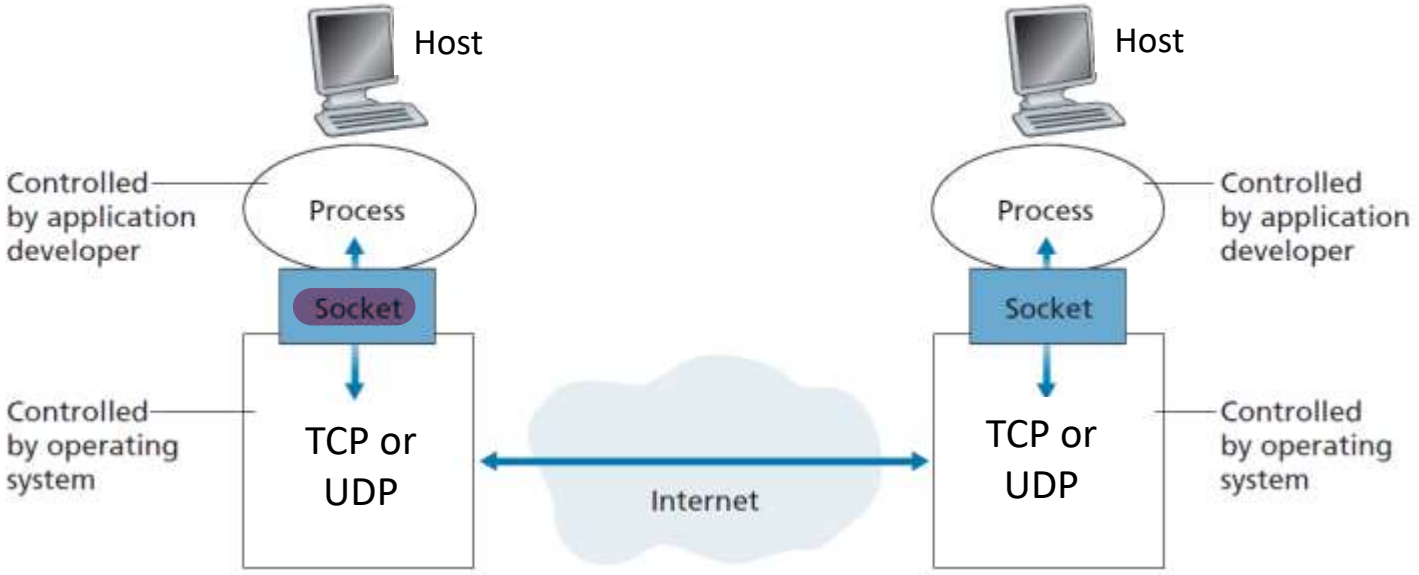
- In a communication session between a pair of processes, process that initiates communication is labeled as the client. Process that waits to be contacted to begin session is server.

Interface Between Process and Network

- Any message sent from one process to another must go through network
- A process sends messages into, and receives messages from, network through a software interface called a socket
- Sending process assumes there is a transportation infrastructure on other side of interface that will transport message to interface of destination process
- Once message arrives at destination host, message passes through receiving process's socket, and receiving process then gets message

Figure 2.3 Application processes, sockets, transport protocol

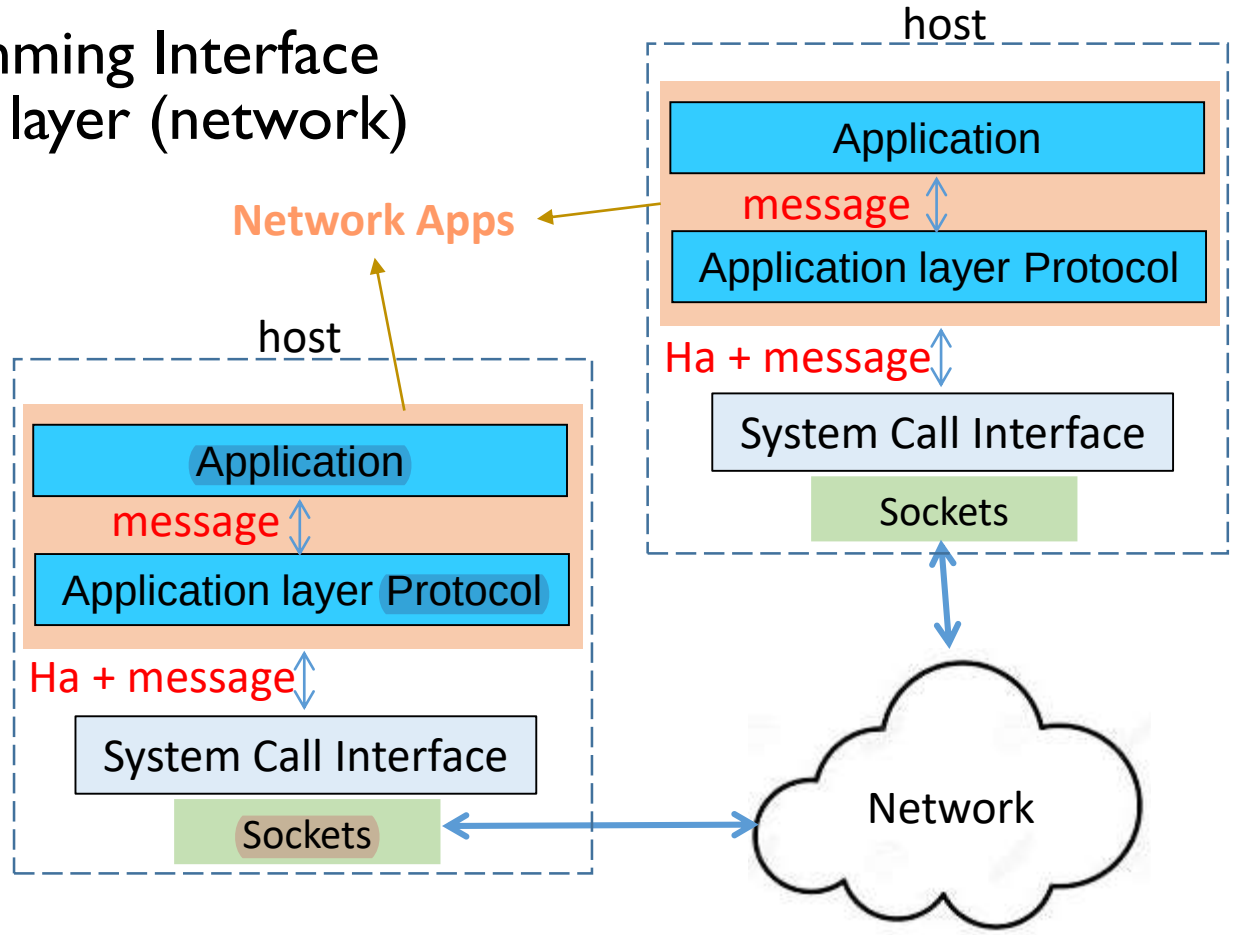
- Socket (Application Programming Interface): interface between application and network (between application layer and transport layer within a host)
- APP developer has control on application-layer side of socket but little control of transport-layer side of socket
- Only control that APP developer has on transport layer side is
  - Choice of transport protocol
  - Perhaps the ability to fix a few parameters



Network System Calls = SOCKET APIs

SOCKET APIs: Application Programming Interface between applications and transport layer (network)

- Socket( ): Create an endpoint for communication
- Bind( ): Bind an IP and a port to a socket
- Listen( ): Listen for connections on a socket
- Accept( ): Accept a connection on a socket
- Connect( ): Initiate a connection on a socket
- Recvfrom( ): Receive a message from a socket
- Sendto( ): Send a message on a socket
- Close( ): Shut down part of a full-duplex connection
- …



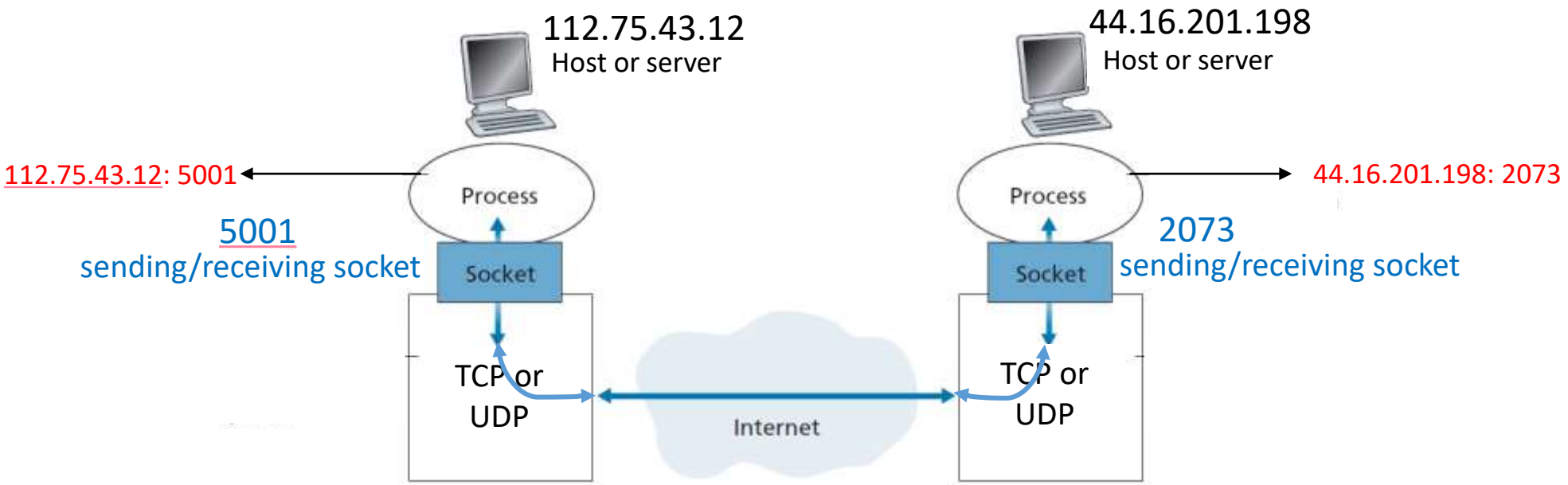
Ha = Header of application layer protocol

Addressing Processes

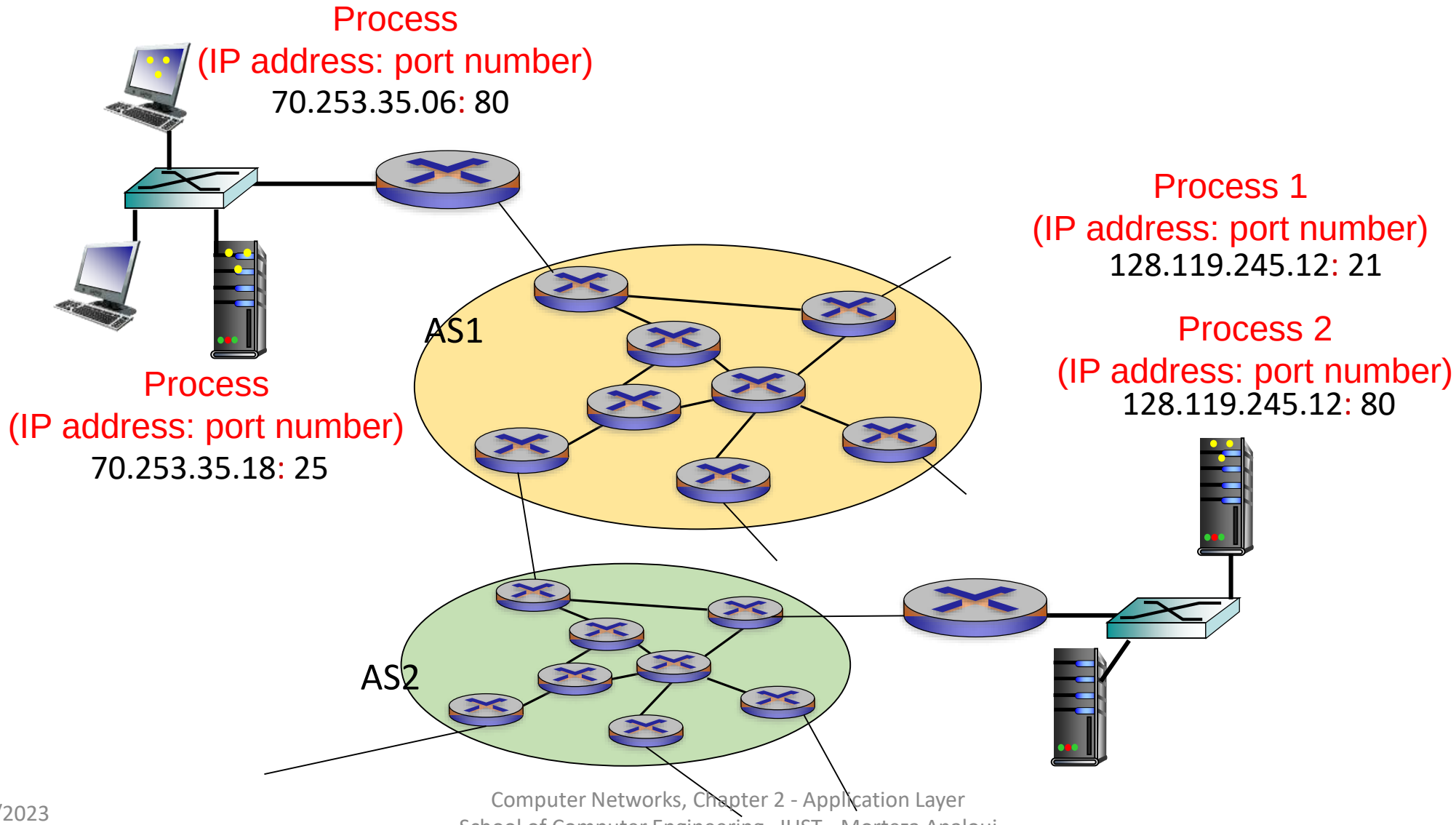
- To send postal mail to a destination, it needs to have an address
- A process sends packets to a process running on another host, receiving process needs to have an address
- Receiving process address (identifier):
  - address of host (IP address)
  - identifier of receiving process in destination host, that is receiving socket (port number)
    - Popular applications have specific port numbers. Example, Web server Process: 80, Mail server process: 25.
    - A list of well-known port numbers: www.iana.org

Process's Address

- Process address= (host address + socket address in the host)

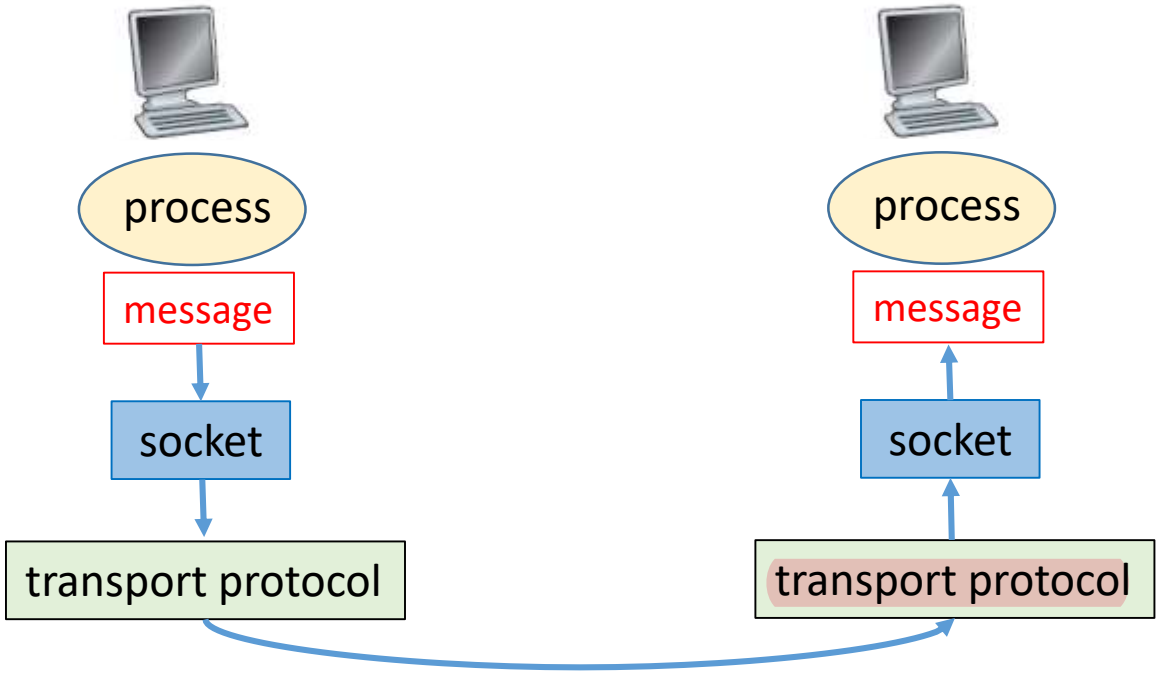


Process's Address



2.1.3 Transport Services Available to Applications

- App at sending side pushes messages through socket
- At other side of socket, transport-layer protocol has responsibility of getting messages to socket of receiving process
- There are more than one transport-layer protocol
- App developer must choose one of available transport-layer protocols



Transport layer choices

- **Q**: How do you make this choice?
- **A**: I study services provided by each transport-layer protocols, and then pick protocol that best match my application's needs
- Situation is similar to choosing either train or bus transport for travel between two cities
- **Q**: What transport-layer services offered to applications?
- **A**: reliable data transfer, throughput, timing, security, …

Reliable Data Transfer

- APPs such as email, file transfer, remote host access, Web document transfers, and financial application: data loss can have devastating consequences.
- Reliable Data Transfer (rdt): To guarantee, data sent by one APP is delivered correctly and completely to other APP

- A transport-layer protocol service can be process-to-process reliable data transfer

Reliable Data Transfer

- When a transport protocol provides **rdt**, sending process just pass its data into socket and is confidence data will arrive without errors at receiving process

- When a transport-layer protocol doesn't provide reliable data transfer, some of data sent by sending process may never arrive at receiving process. This may be acceptable for loss-tolerant applications, such as audio/video that can tolerate some amount of data loss

Throughput

- In Chapter 1, we defined available throughput: Rc, Rs, bottleneck bandwidth, …
- Several sessions share bandwidth along network path, and they are coming and going: available throughput can fluctuate with time
- A transport-layer protocol service can be to guarantee available throughput at some specified rate
- APP requests a guaranteed throughput of r bits/sec, and transport protocol ensure that available throughput is always at least r bits/sec.
- Such a guarantee would appeal to many applications. For example, if an Internet telephony application encodes voice at 32 kbps, it needs to send data into network and have data delivered to receiving application at this rate

Throughput

- APPs have throughput requirements are said bandwidth-sensitive applications
- Many current multimedia applications are bandwidth sensitive
- Elastic applications can make use of as much, or as little, throughput as happens to be available. Electronic mail, file transfer, and Web transfers are all elastic applications.
- More throughput is always better

Timing

- A transport-layer protocol can also provide timing guarantees
- Example: to guarantee every bit that sender pumps into socket arrives at receiver's socket no more than 100msec later
- Such a service needed by interactive real-time applications, such as Internet telephony, virtual environments, teleconferencing, and multiplayer games
- For non-real-time applications, lower delay is always preferable to higher delay, but no tight constraint is placed on end-to-end delays

Security

- A transport protocol can provide an application with one or more security services
- In sending host, a transport protocol can encrypt all data transmitted by sending process, and in receiving host, transport-layer protocol can decrypt data before delivering data to receiving process
- Such a service provides confidentiality between two processes
- A transport protocol can also provide other security services, including data integrity and end-point authentication

2.1.4 Transport Services Provided by Internet

- Internet (TCP/IP networks) makes two transport protocols available to applications, UDP and TCP
- App developer creates a new network application for Internet
  - one of first decisions developer to make is whether to use UDP or TCP
- Each of UDP and TCP offers a different set of services to Apps
- TCP service model: connection-oriented service and reliable data transfer service
- TCP and UDP provides no security services
  - No confidentiality: sending process passes data into its socket is same data that travels over network to destination process (cleartext = unencrypted))
  - No data integrity and no end-point authentication

TCP Services - Connection-oriented service

- Connection-oriented service: TCP has client and server **transport layer control information**, *before* application-level messages begin to flow
  - This is called handshaking procedure, allowing two hosts **get ready for packets exchange**
- After handshaking phase, a TCP connection is said to exist between sockets of two processes
- A TCP connection is full-duplex: Two processes can send messages to each other over connection at same time
- When APPs finish sending messages, they close down connection
  - Discard connection control information as well as buffered messages (packets)

TCP Services - Reliable data transfer service

- Reliable data transfer service: Communicating processes can rely on TCP to deliver all data sent without error and in proper order.
- When one side passes a stream of bytes into a socket, it can count on TCP to deliver same stream of bytes to receiving socket, with no missing or duplicate bytes or reordering
- TCP's resolution of rdt service is Byte

More services of TCP

- TCP includes a congestion-control mechanism, a service for general welfare of Internet rather than for direct benefit of communicating processes
- TCP congestion-control mechanism throttles a sending process (client or server) when network is congested between sender and receiver

- Congestion: one or more router in path between communicating hosts experiencing high buffer queue or buffer over flow

UDP Services

- UDP is a lightweight transport protocol, providing minimal services
- UDP is connectionless, so there is no handshaking before two processes start to communicate
- UDP provides an unreliable data transfer, *no* guarantee that message will ever reach receiving socket
- Messages that do arrive at receiving process may arrive out of order
- UDP does not include a congestion-control mechanism, sending side of UDP can pump data into network at any rate it pleases
  - Note: actual end-to-end throughput may be less than this rate due to limited transmission capacity of intervening links (bottleneck) or due to congestion

SECURING TCP

- Transport Layer Security (TLS) is developed to enhance TCP
- TLS services: encryption, data integrity, and end-point authentication
- TLS is implemented in application layer. App needs to include TLS code (existing, highly optimized libraries and classes) in both client and server sides
- TLS has its own socket API that is similar to TCP socket API
  - Sending process passes cleartext data to TLS socket; TLS encrypts data and passes encrypted data to TCP socket. Encrypted data travels over Internet to TCP socket in receiving process. Receiving socket passes encrypted data to TLS, which decrypts data. Finally, TLS passes cleartext data through its TLS socket to receiving process

Services Not Provided by Internet Transport Protocols

- **Q**: Which of services are provided by TCP and UDP?
- **A**: TCP provides reliable end-to-end data transfer. TCP can be easily enhanced at application layer with TLS to provide security services
- No guarantees of throughput or timing
- **Q**: Sensitive Apps such as Internet telephony cannot run in today's Internet?
- **A**: No. Sensitive Apps often work fairly well because they have been designed to cope, to greatest extent possible, with this lack of guarantee
  - Today's Internet can often provide satisfactory service to time sensitive applications, but it cannot provide any timing or throughput guarantees

Figure 2.4

- Requirements of selected network applications

Application	Data Loss	Throughput	Time-Sensitive
File transfer/download	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic (few kbps)	No
Internet telephony/ Video conferencing	Loss-tolerant	Audio: few kbps–1Mbps Video: 10 kbps–5 Mbps	Yes: 100s of msec
Streaming stored audio/video	Loss-tolerant	Same as above	Yes: few seconds
Interactive games	Loss-tolerant	Few kbps–10 kbps	Yes: 100s of msec
Smartphone messaging	No loss	Elastic	Yes and no

Figure 2.5

- Popular Internet applications, their application-layer protocols, and their underlying transport protocols

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP [RFC 5321]	TCP
Remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP 1.1 [RFC 7230]	TCP
File transfer	FTP [RFC 959]	TCP
Streaming multimedia	HTTP (e.g., YouTube), DASH	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	UDP or TCP

Figure 2.5

- Email, remote terminal access, Web, and file transfer all use TCP
- These applications have chosen TCP primarily because TCP provides reliable data transfer
- Internet telephony applications (such as Skype) can often tolerate some loss but require a minimal rate to be effective
- Developers of Internet telephony applications usually prefer to run their applications over UDP
  - Many firewalls are configured to block (most types of) UDP traffic, Internet telephony applications often are designed to use TCP as a backup if UDP communication fails

2.1.5 Application-Layer Protocols

- Processes communicate with each other by sending messages into sockets
- **Q**: How are these messages structured? What are meanings of various fields in messages? When do processes send messages?
- **A**: Application-layer protocol defines:
  - Types of messages exchanged, for example, request messages and response messages
  - Syntax of various message types, such as fields in message and how fields are delineated
  - Semantics of fields, that is, meaning of information in the fields
  - Rules for determining when and how a process sends messages and responds to messages

Public/Proprietary protocols

- Some application-layer protocols are specified in RFCs and are availabl for everybody. Web's application-layer protocol, HTTP (Hyper Text Transfer Protocol [RFC 7230]), is available as an RFC. If a browser developer follows rules of HTTP RFC, browser will be able to retrieve Web pages from any Web server that has also followed rules of HTTP RFC.
- Many other application-layer protocols are proprietary and intentionally not available in public domain. For example, Skype uses proprietary application-layer protocols

Application-layer protocol in App

- An application-layer protocol is one piece of a network App (a very important piece of App)
- Example:
  - Web is a client-server App that allows users to obtain documents from Web servers on demand. Web App consists of many components, including a standard for document formats (that is, HTML), and so on
  - Web browsers (Chrome, Edge), Web servers (Apache and Microsoft servers IIS) have several pieces including an application-layer protocol
- Web's application-layer protocol, HTTP, defines format and sequence of messages exchanged between browser and Web server
- HTTP is only one piece of Web application

Application-layer protocol in App

- Example:
  - Netflix's video service also has many components, including
  - servers that store and transmit videos,
  - servers that manage billing and other client functions,
  - clients (Netflix App on your smartphone, tablet, or computer),
  - an application-level DASH protocol defines format and sequence of messages exchanged between a Netflix server and client
    - DASH is only one piece (an important piece) of Netflix application (Section 2.6)

2.1.6 Network Applications Covered

- In this chapter, we discuss five important applications: Web, email, directory service (DNS), video streaming, and P2P applications
- Most users do not interact with DNS directly; instead, users invoke DNS indirectly through other applications

- There are a lot that we do not discuss. New applications are being developed every day

Contents

2.2 The Web and HTTP

- Until early 1990s, Internet was used primarily by researchers, academics, and university students to log in to remote hosts, to transfer files from local hosts to remote hosts and vice versa, to receive and send news, and to receive and send electronic mail
- Although these applications were (and continue to be) extremely useful, Internet was essentially unknown outside of academic and research communities
- In early 1990s, a major new application arrived: World Wide Web [Berners-Lee 1994]
- Web was first Internet application that caught general public's eye

Web application

- Web application operates on demand
  - Users receive what they want, when they want it

- Web is easy for any individual to make information available over Web, everyone can become a publisher at extremely low cost

- Web and its protocols serve as a platform for YouTube, Web-based e-mail (such as Gmail), and most mobile Internet applications, including Instagram and Google Maps

2.2.1 Overview of HTTP

- Hyper Text Transfer Protocol (HTTP): Web's application-layer protocol, is defined in [RFC 1945], [RFC 7230] and [RFC 7540]
- HTTP is implemented in two programs: a client program and a server program
- Client program and server program, executing on different hosts, communicate by exchanging messages across network, **using rules and details that are governed** by HTTP
- HTTP defines structure of these messages and how client and server processes exchange messages

Web page

- A Web page (also called a document) consists of objects
- An object is simply a file, such as an HTML file, a JPEG image, a JavaScript file, a CCS[1] style sheet file, or a video clip. Web page is addressable by a single URL (Uniform Resource Locator)
- Most Web pages consist of a base HTML file and several referenced objects. For example, if a Web page contains HTML text and five JPEG images, then Web page has six objects: base HTML file plus five images
- Base HTML file references other objects in page with objects' URLs

1: Cascading **Style Sheets** (**CSS**) is used to format the layout of a webpage

Uniform Resource Locator

Q: How do we address an object stored in a host? **A**: URL

- URL has two components: hostname of server (or IP address of server) that houses object and object's path name
- Example

http://www.someSchool.edu/someDepartment/picture.gif

or

http://126.12.45.80/someDepartment/picture.gif

www.someSchool.edu: hostname of server

126.12.45.80: IP address of host (server)

/someDepartment/picture.gif: path name

General idea of HTTP

- HTTP defines how Web clients request Web pages from Web servers and how servers transfer Web pages to clients
- When a user requests a Web page (for example, clicks on a hyperlink), browser sends a HTTP request message for HTML object of web page to server. Server receives requests and responds with HTTP response message that contain requested object

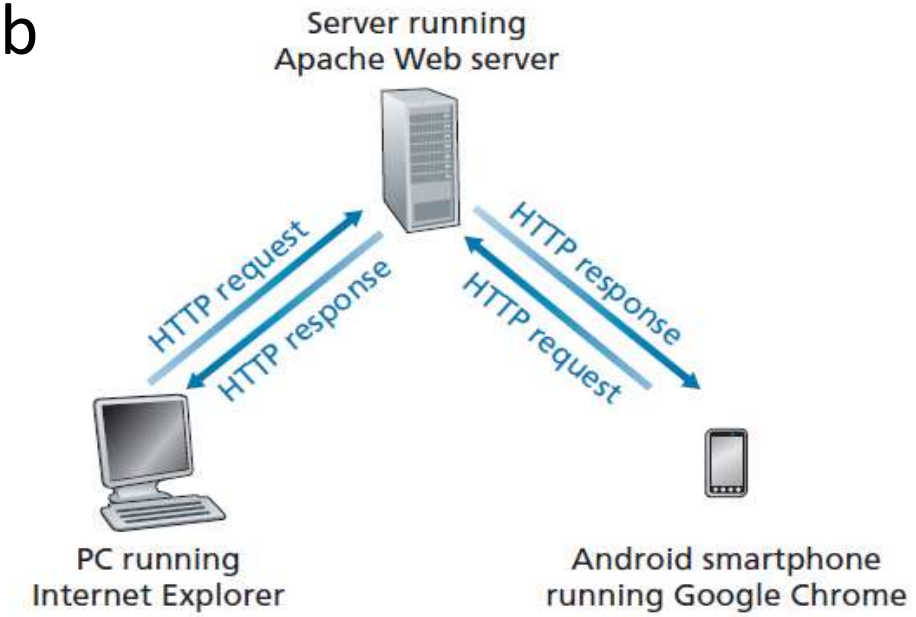


Figure 2.6 HTTP request-response behavior. Web server is always on, with a fixed IP address, and it services requests from potentially millions of different browsers

HTTP uses TCP

- HTTP uses TCP as its underlying transport protocol
- HTTP client first initiates a TCP connection with server
- Once connection is established, browser and server processes access TCP through their socket interfaces
- Client sends HTTP request messages into its socket interface and receives HTTP response messages from its socket interface
- Once client sends a message into its socket interface, message is out of client's hands and is "in hands" of TCP
- Here we see one of great advantages of a layered architecture: HTTP need not worry about lost data or details of how TCP recovers from loss or reordering of data within network. That is job of TCP and protocols in lower layers of protocol stack

HTTP is a stateless protocol

- Server sends requested files to clients without storing any state information about client
- If a particular client asks for same object twice in a period of a few seconds, server does not respond by saying that it just served object to client; instead, server resends object
- HTTP server maintains no information about clients, so, it is a **stateless protocol**

Versions of HTTP

- Original version of HTTP is called HTTP/1.0 (early 1990's).
- As of 2020, majority of HTTP transactions take place over HTTP/1.1

- However, increasingly browsers and Web servers also support a new version of HTTP called HTTP/2
- HTTP/3 !

2.2.2 Non-Persistent and Persistent Connections

- In many Internet applications, client and server communicate for an extended period of time
- Depending on application and on how it is being used, series of requests may be made back-to-back, periodically at regular intervals, or intermittently
- App developer makes an important decision:
  - should each request/response pair be sent over a *separate* TCP connection, (App uses **non-persistent TCP connections**)
  - should all of requests and their corresponding responses be sent over *same* TCP connection (App uses **persistent TCP connections**)?
- HTTP uses **persistent connections** in its default mode
  - HTTP clients and servers can be configured to use non-persistent connections

Example of Non-Persistent Connections

- Example: Transferring a Web page from server to client
- Page consists of a base HTML file and 10 JPEG images, and that all 11 of these objects reside on same server
- Suppose URL for base HTML file is

  http://www.someSchool.edu/someDepartment/home.index

Here is what happens:

1. HTTP client process initiates a TCP connection to server www.someSchool.edu on port number 80, default port number for HTTP. Associated with TCP connection, there will be a socket at client and a socket at server

HTTP with Non-Persistent Connections

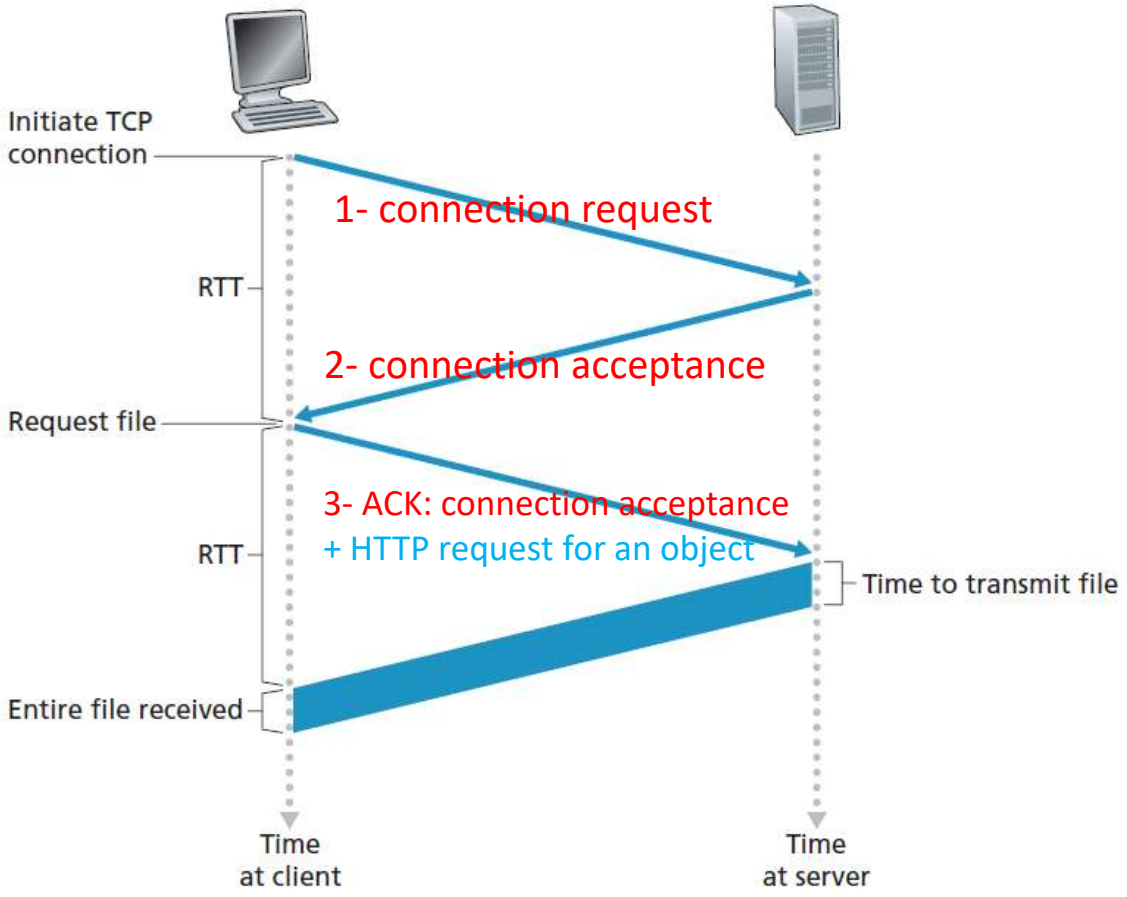
2. HTTP client sends an HTTP request message to server via its socket. Request message includes /someDepartment/home .index
3. HTTP server process receives request message via its socket, retrieves object /someDepartment/home.index from its storage (RAM or disk), encapsulates object in an HTTP response message, and sends response message to client via its socket
4. HTTP server process tells TCP to close TCP connection. (TCP doesn't terminate connection until it knows for sure that client has received response message)
5. HTTP client receives response message. TCP connection terminates. Response message contains an HTML file. Client extracts HTML file from response message, examines HTML file, and finds references to (URL of) 10 JPEG objects
6. Steps 1 to 4 are then repeated for each of referenced JPEG objects

HTTP/1.0 employs non-persistent TCP connections

HTTP has nothing to do with how a Web page is interpreted by a client (Web browser)

Figure 2.7

- a TCP connection between browser and server involves a "three-way handshake
- Total response time roughly is two RTTs plus transmission time at server of object file (HTML)
- **Total time to retrieve web page:** $11\times2RTT+(HTML+O_1+\ldots+O_{10})/Rs$
- **11 TCP connections are generated in sequence** (1 for HTML and 10 serial TCP connections for JPEG)

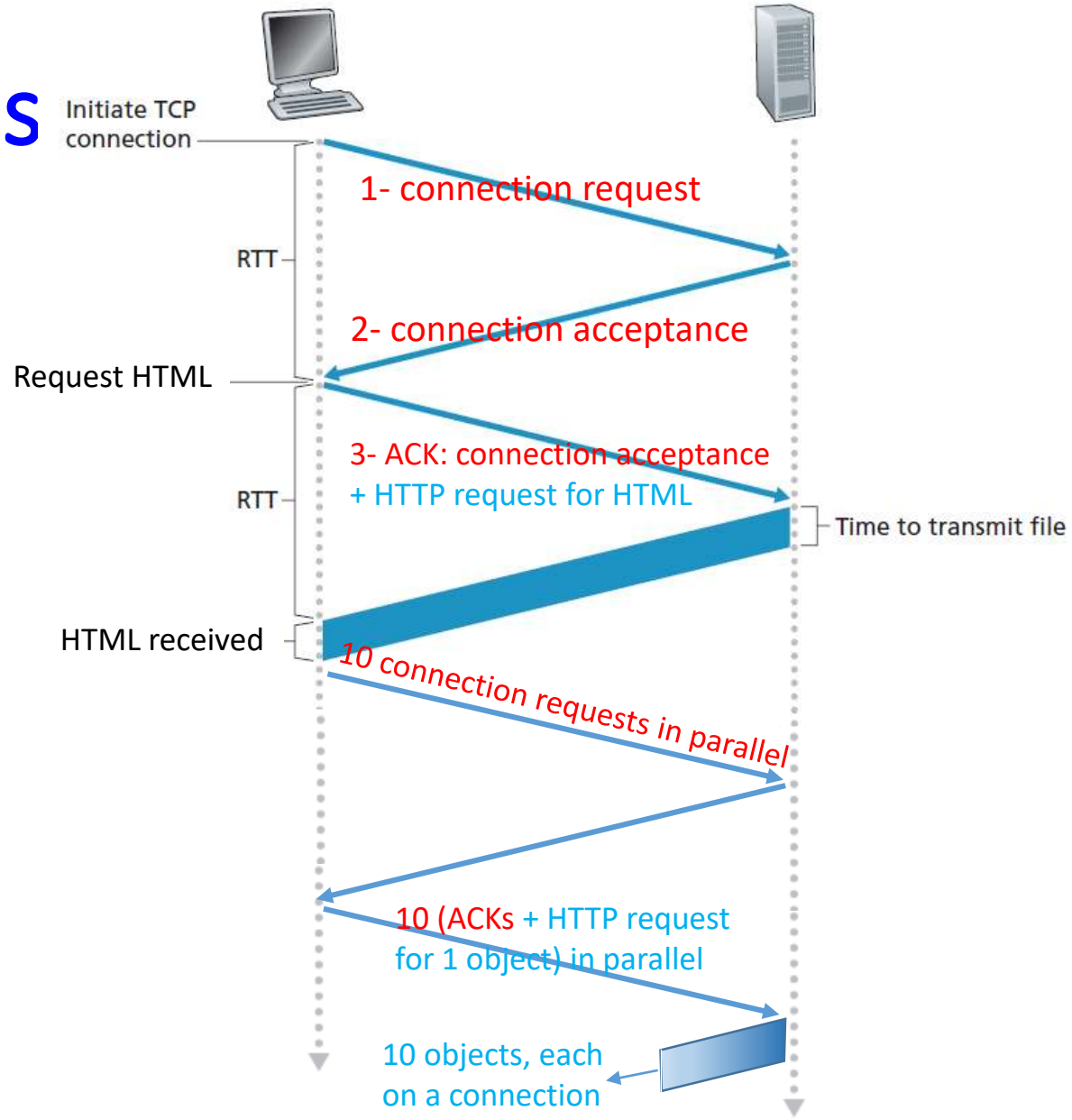


Shortcomings of Non-persistent connections

- A new connection must be established and maintained for *each requested object*
- For each of these connections, TCP buffers must be allocated and TCP variables must be kept in both client and server
- This can place a significant burden on Web server, which may be serving requests from hundreds of different clients simultaneously

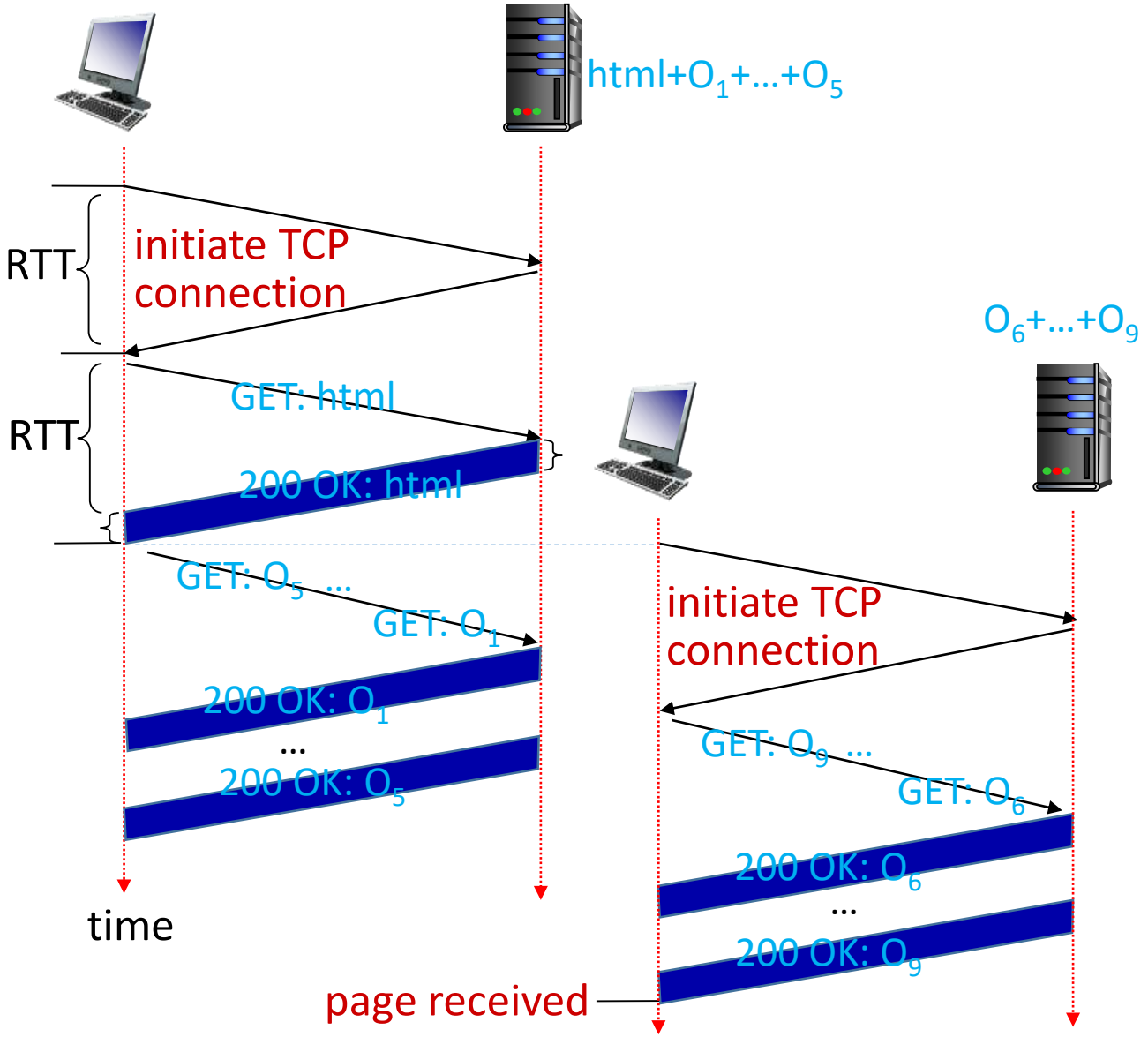
Parallel TCP connections

- Users can configure some browsers to control degree of parallelism
- Browsers may open multiple TCP connections and request different parts of Web page over multiple connections. It shortens response time
- Total number of RTTs = 4RTT
- **Total time to retrieve web page:** $4RTT+(HTML+O_1+...+O_{10})/Rs$



HTTP/1.1 - parallel

- When some objects located at another server: multiple, pipelined GETs over parallel TCP connections

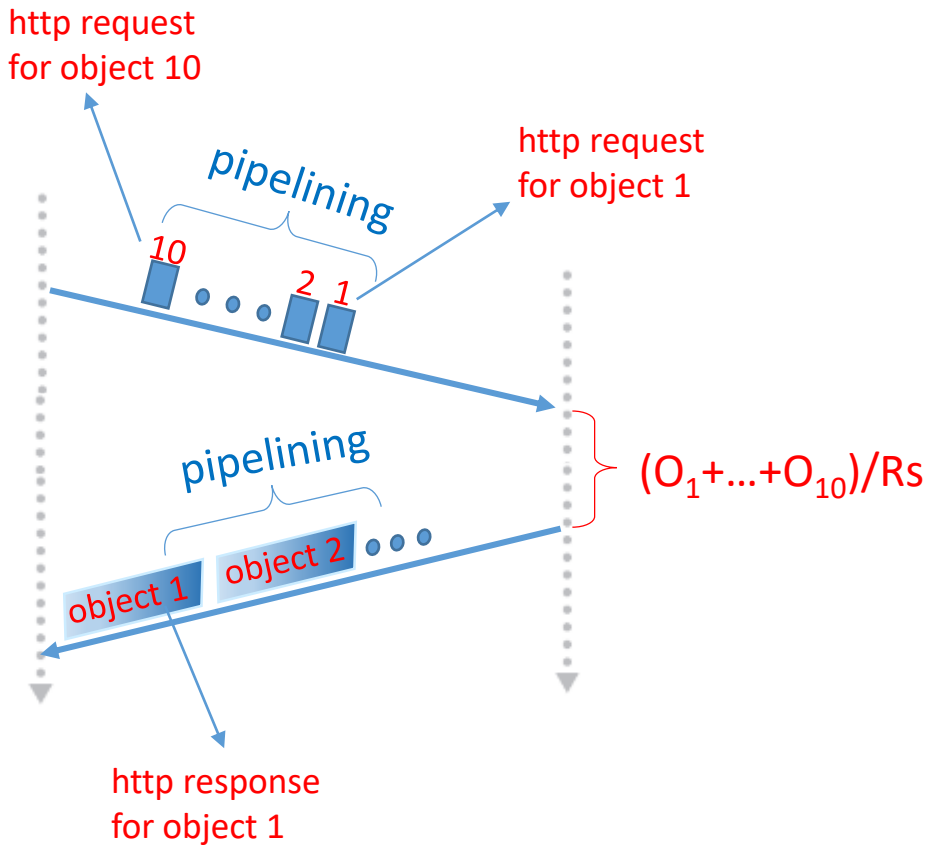


HTTP with Persistent Connections

- HTTP/1.1: Server leaves TCP connection open after sending a response
- Subsequent requests and responses can be sent over same connection
- Entire Web page can be sent over a single persistent TCP connection
- Moreover, multiple Web pages residing on same Web server App can be sent from server to same Web client App over a single persistent TCP connection
- HTTP server closes a connection when it isn't used for a certain time (a configurable timeout interval)
- Default mode of HTTP uses persistent connections with pipelining

HTTP with Pipelining

- Pipelining: requests for 10 objects can be made back-to-back, without waiting for replies
- When server receives back-to-back requests, it sends objects back-to back
- Default mode of HTTP uses persistent connections with pipelining



2.2.3 HTTP Message Format [RFC 1945; RFC 7230; RFC 7540]

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

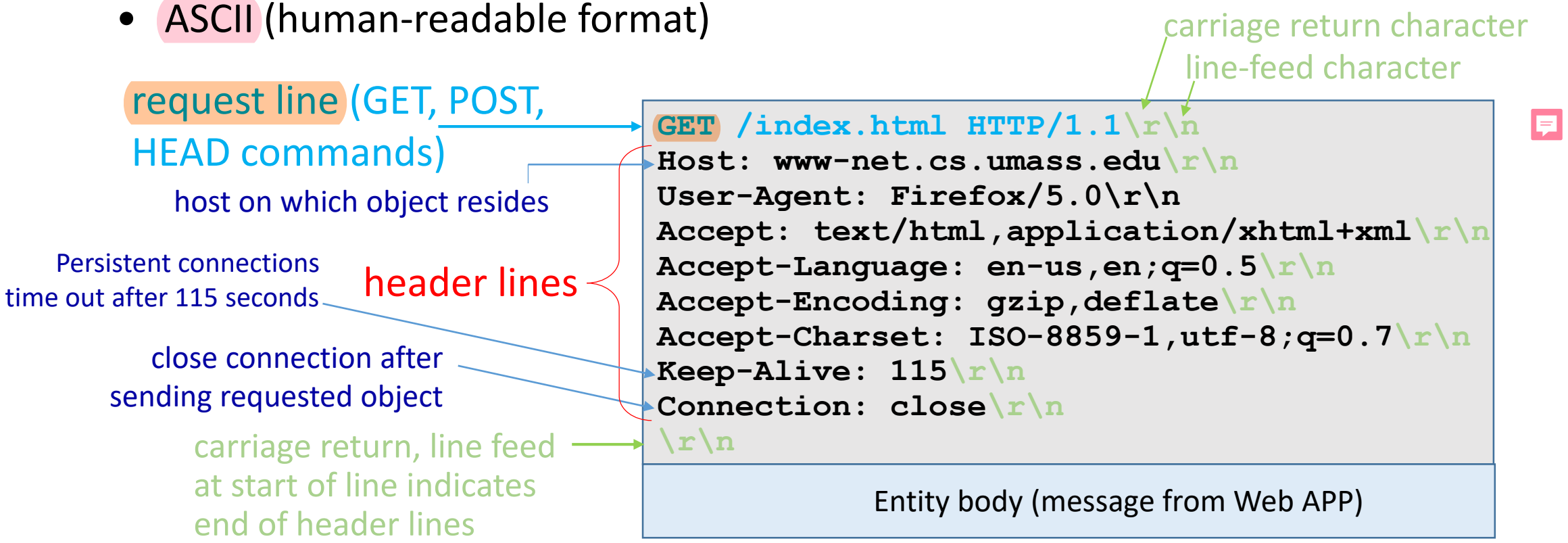
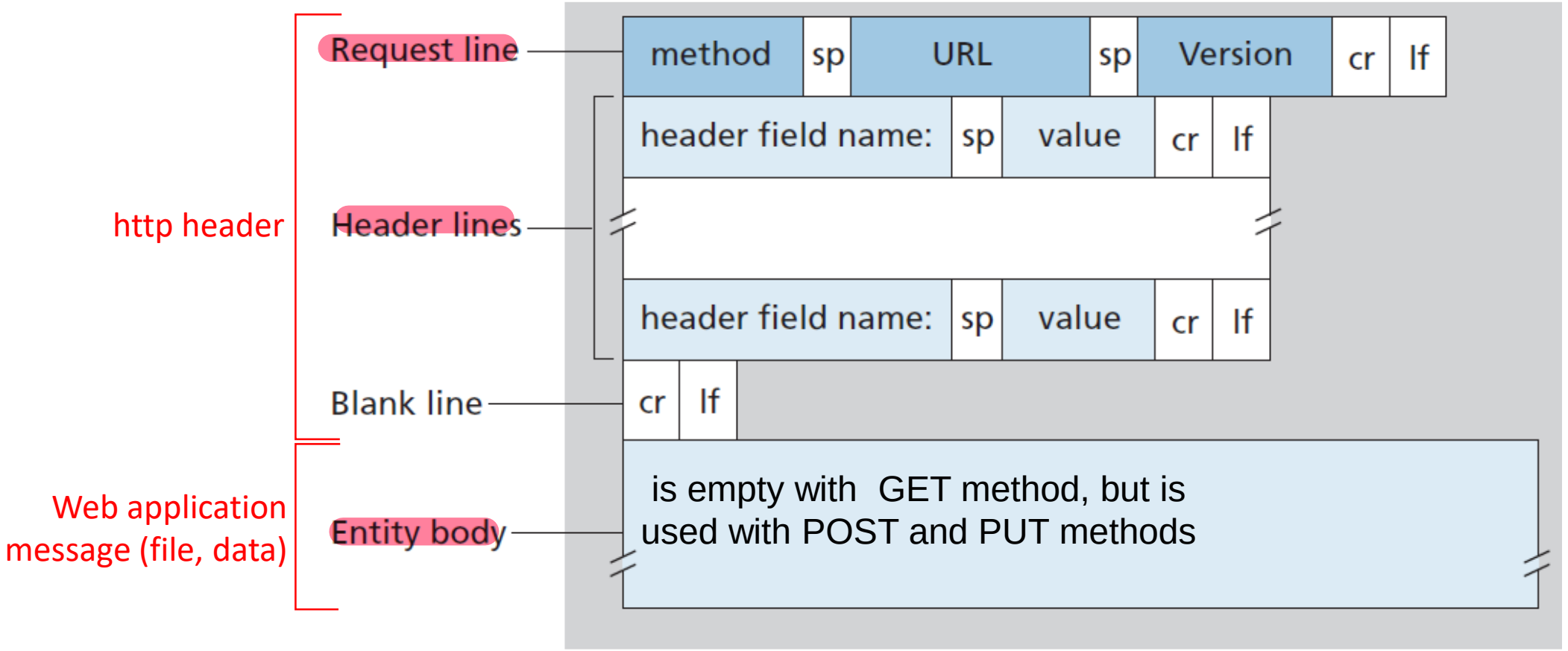


Figure 2.8 HTTP request message: general format



Methods

Get method: (for requesting a file)

- used when browser requests an object
- requested object identified in URL field.
- great majority of HTTP request

GET method (for sending data to server):

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

HOST: www.google.com/search?q=post+office

POST method:

- HTTP client uses POST method when user fills out a form-for example, when a user provides search words to a search engine.
- user input sent from client to server in entity body of HTTP POST request message

Methods

PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of PUT HTTP request message
- used in conjunction with Web publishing tools (creating and uploading websites, updating webpages, posting blogs online)
- used by applications that need to upload objects to Web servers

HEAD method:

- HEAD is similar to GET, when server receives HEAD request, responds with an HTTP message but it leaves out requested object
- Application developers often use HEAD for debugging

DELETE method:

- DELETE allows a user, or an application, to delete an object (file) on a Web server

HTTP response message

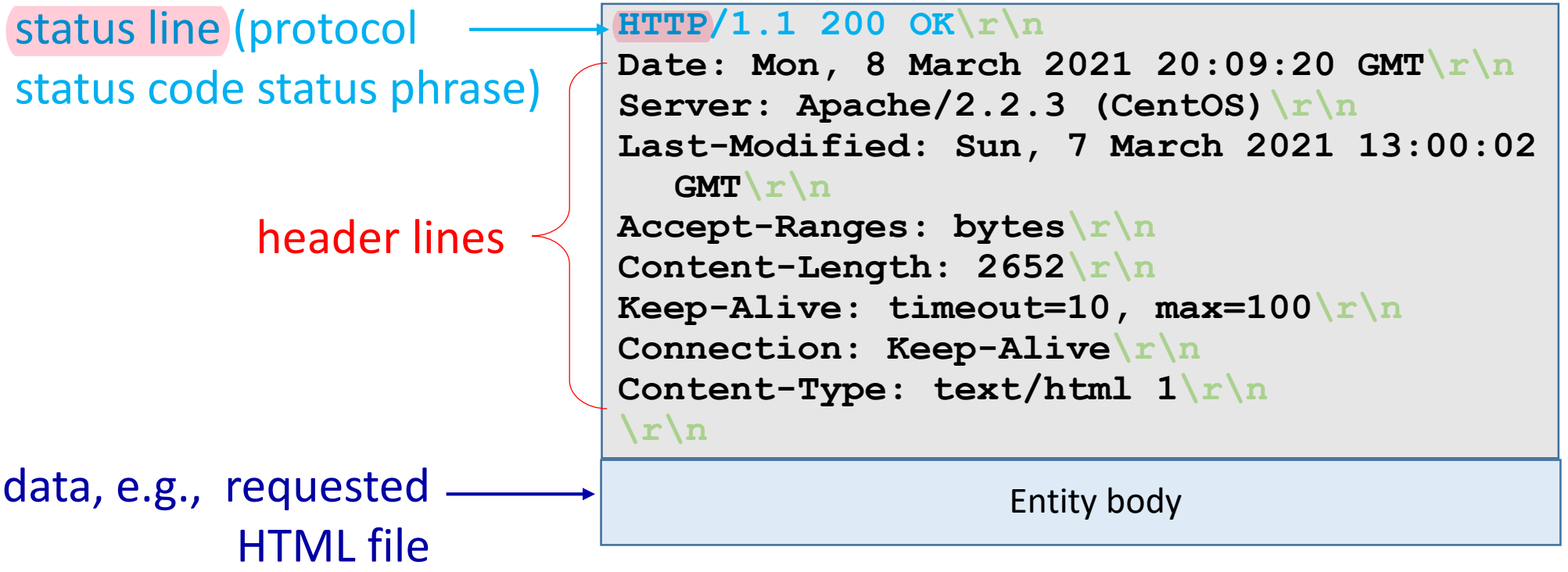
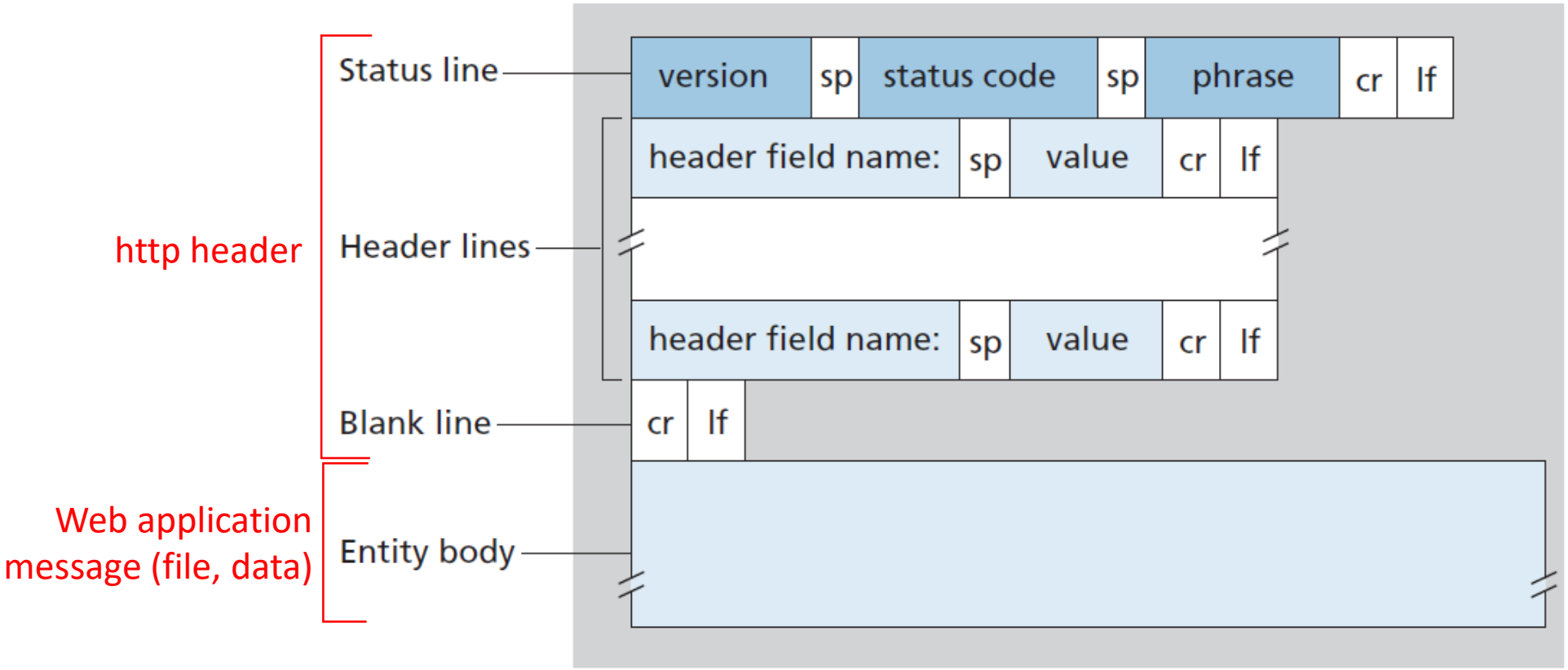


Figure 2.9 General format of an HTTP response message



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

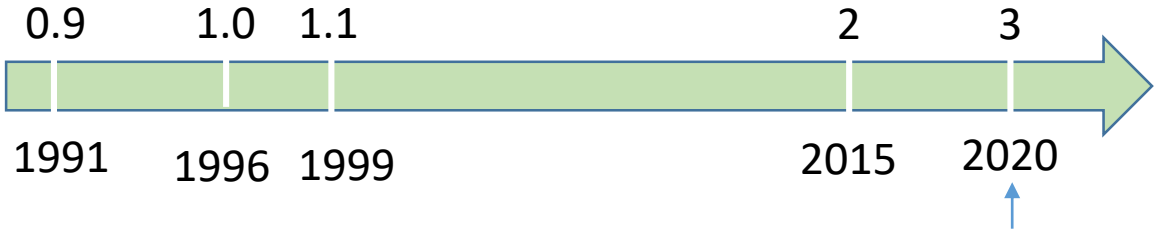
301 Moved Permanently
- requested object moved, new location specified later in this message (in Location: field)

400 Bad Request
- request msg not understood by server

404 Not Found
- requested document not found on this server

505 HTTP Version Not Supported

2.2.6 HTTP/2



- Improve end-user perceived latency
  - decreased delay in multi-object HTTP requests
- Address "head of line blocking"
- Not require multiple connections
- Retain semantics of HTTP/1.1

Requests and Transfer Size
(browsing 490000 sites)

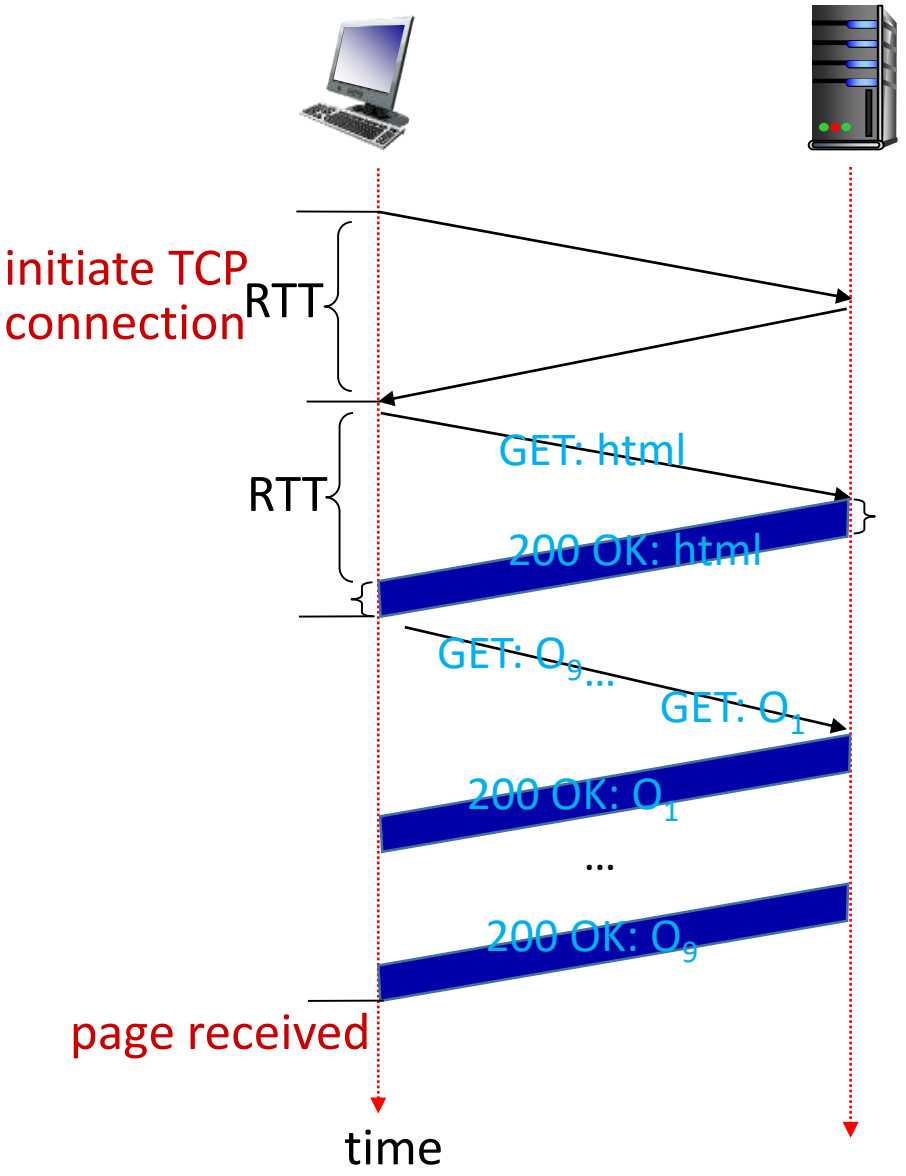
Content Type	Avg # of Requests	Avg size
HTML	9	60 kB
Images	56	1200 kB
Javascript	18	307 kB
CSS	6	64 kB
Webfont	2	80 kB

Cascading Style Sheets (**CSS**) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML.

HTTP/1.1

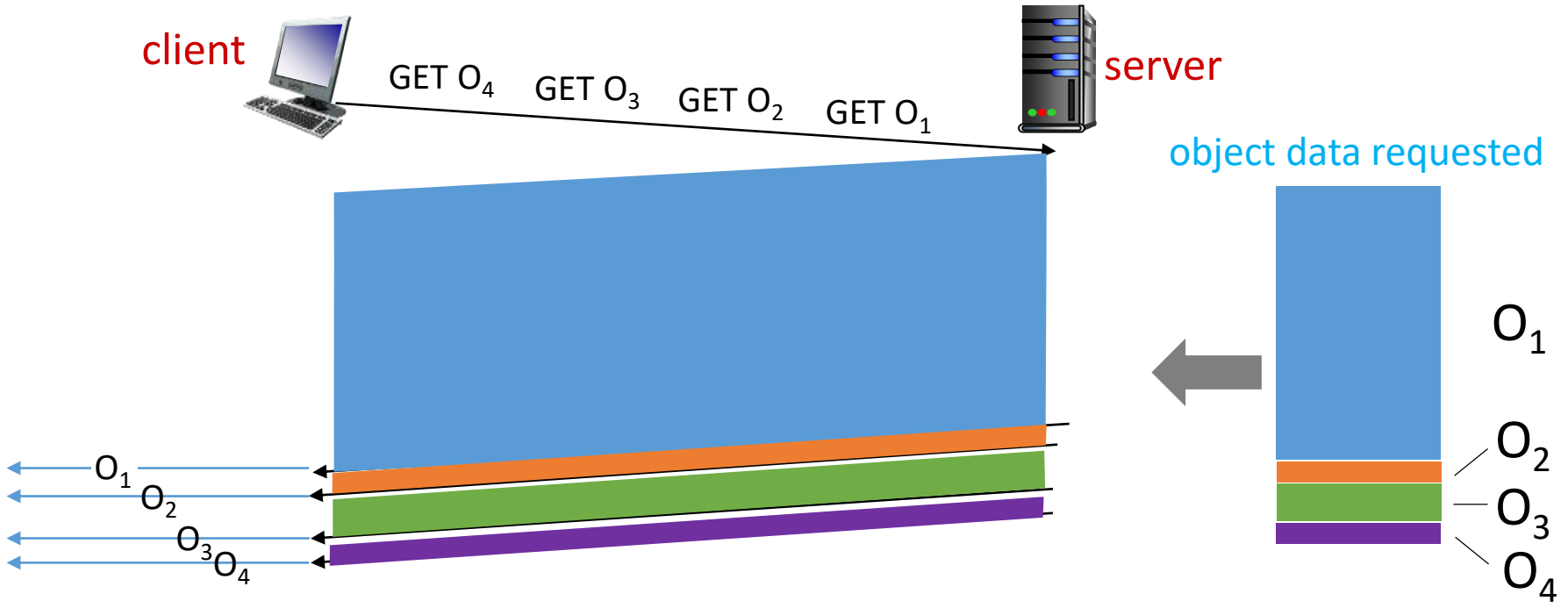
multiple, pipelined GETs over single TCP connection

- server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests
- with FCFS, small object may have to wait for transmission (head-of-line (HOL) blocking) behind large object(s)
- loss recovery (retransmitting lost TCP segments) causes staying on object transmission



HTTP/1: HOL blocking

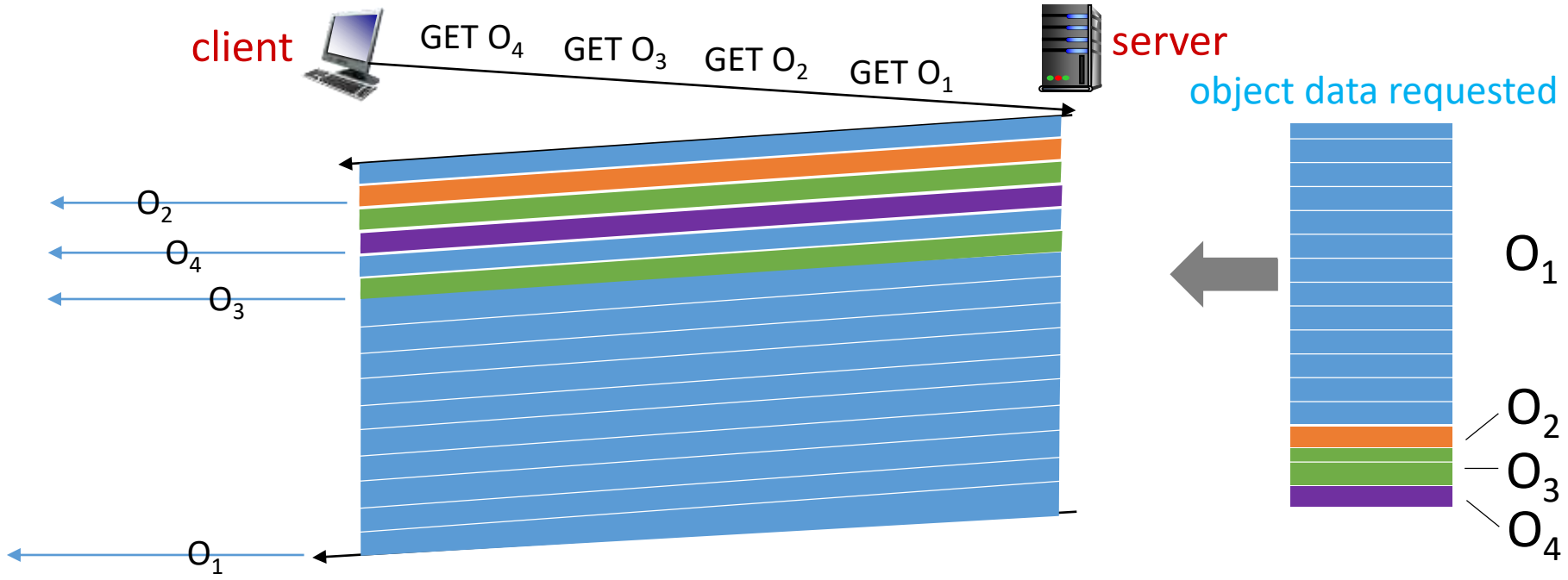
- HTTP 1.1: client requests 1 large object (e.g., video file, and 3 smaller objects)



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

HTTP/2: mitigating HOL blocking

- 4 HTTP/2 Get requests and 4 response (that is 4 streams)
- a stream: a request/response
- objects divided into frames, frame transmission interleaved



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

HTTP/2 [RFC 7540, 2015]

HTTP/2: increased flexibility at server in sending objects to client:

- Methods, status codes, most header fields unchanged from HTTP 1.1
- Transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- Push unrequested objects to client (e.g., pushing critical CSS and JS objects needed for rendering a page first time someone visits it)
- Divide objects into frames, schedule frames to mitigate (reduce) HOL blocking
- Flow control by HTTP/2 at stream level. (TCP's flow control is at connection level)

Server push

- HTTP/2 server CAN send multiple responses for a single client request
- Server can push additional resources to client, without client having to request each one explicitly
- HTTP/2 breaks away from strict request-response semantics
  - enables one-to-many
  - Server can initiate workflows that open up a world of new interaction possibilities both within and outside browser
- This feature have important long-term consequences both for how we think about protocol, and where and how it is used

HTTP/2 has two protocol identifiers: h2, h2c

- "h2" identifies protocol where HTTP/2 uses TLS
- This identifier is used in TLS application-layer protocol negotiation field and in any place where HTTP/2 over TLS is identified
- "h2c" identifies protocol where HTTP/2 is run over cleartext TCP
- This is used in HTTP/1.1 Upgrade header field and in any place where HTTP/2 over TCP is identified. Example:

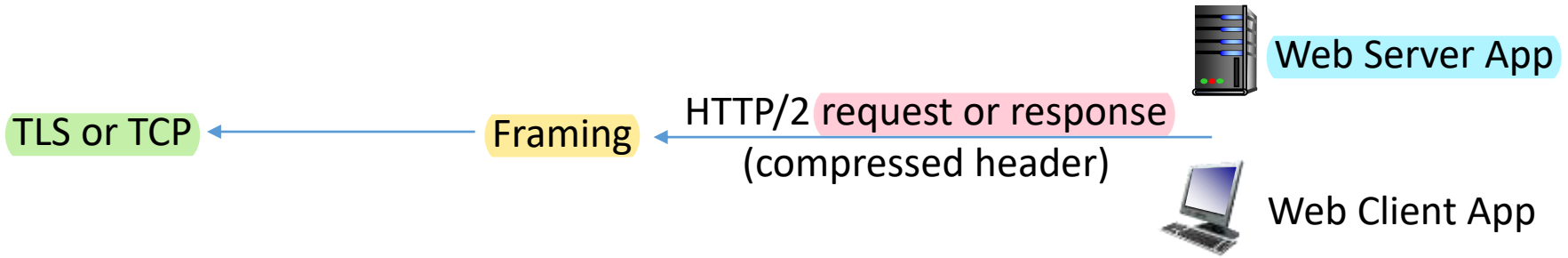
request	a possible response
GET / HTTP/1.1 Host: server.example.com Connection: Upgrade, HTTP2-Settings Upgrade: h2c HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>	HTTP/1.1 101 Switching Protocols Connection: Upgrade Upgrade: HTTP/2.0 [HTTP/2.0 connection ...

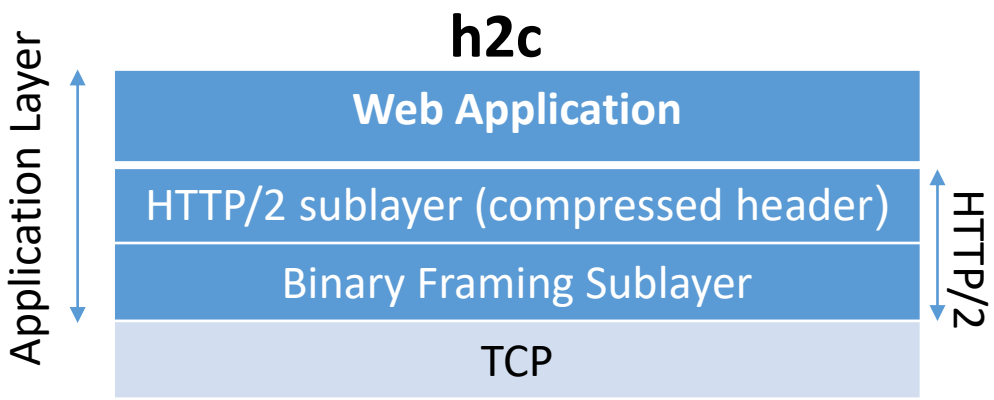
Header compression

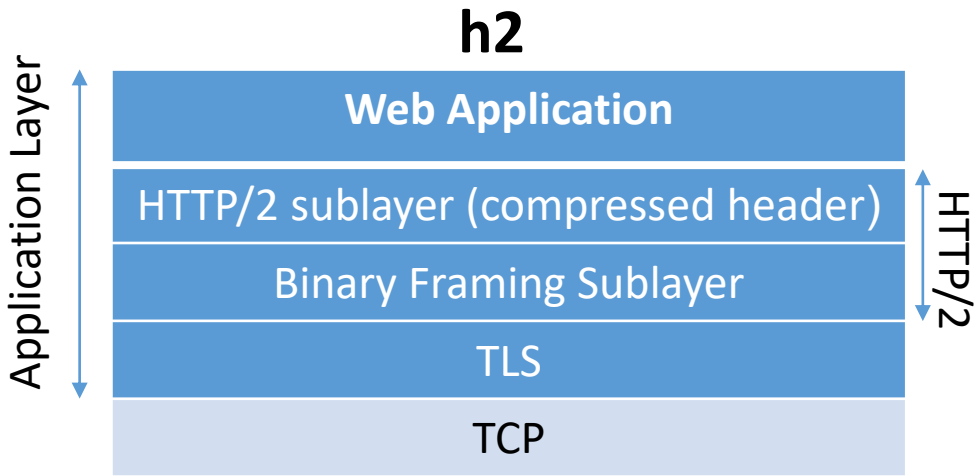
- Each HTTP message (request or response) carries a header
- In HTTP/1, this metadata is always sent as plain text and adds kilobytes more if HTTP cookies are being used
- To reduce this overhead and improve performance,
  - HTTP/2 compresses request and response header metadata using HPACK, rfc7541 (Header Compression for HTTP/2) compression format (Huffman encoding)
  - Average reduction of 30% in header size
- Header compression is statefull. One compression context and one decompression context are used for entire connection

- Transmitted http header is no longer human readable

Framing sublayer of HTTP/2







Framing sublayer of HTTP/2

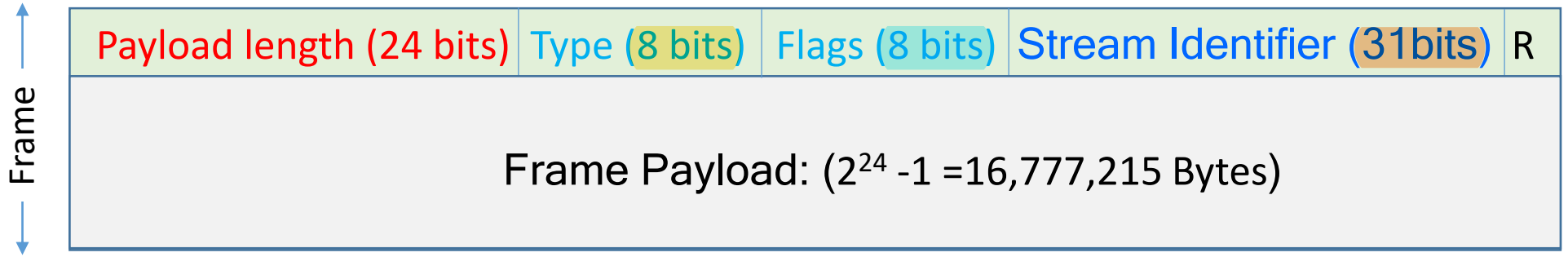
- Framing sublayer breaks down each http request/response into frames
- Each frame has its own header
- The header is **binary encoded** and therefore is not human readable
- Binary protocols are more efficient to parse, lead to slightly smaller frames, and are less error-prone
- Stream: a series of frames making up an individual HTTP request/response pair on a connection

Stream

- Stream: a series of frames making up an individual HTTP request/response pair on a connection
- A HTTP/2 connection can contain multiple concurrently open streams
- When a client makes a request, it initiates a new stream. Server will then reply on that same stream
- Streams are identified by an 31bits integer
- Client sends an HTTP request on a new stream, using a previously unused stream identifier
- As frames arrive at client, they are first reassembled into original response messages and then processed by browser as usual
- HTTP/2 is supported by Edge, Safari, Firefox and Chrome

Frame: A Header on a payload

- Header (9 Bytes)
  - Type determines **format** and **semantics** of frame (structure and content of payload is dependent entirely on frame type)
  - Flags: Boolean flags specific to frame type
  - R: A reserved 1-bit field
- Payload:
  - Default for Max frame size: 2^{14} (16,384)
  - Values greater than 2^{14} MUST NOT be sent unless receiver has set a larger value for SETTINGS_MAX_FRAME_SIZE in **SETTINGS frame**



SETTINGS Frame - Type=0x4

- SETTINGS frame conveys **configuration parameters** that affect how endpoints (client and server) communicate, such as preferences and constraints on behavior
- SETTINGS frame MUST be sent by both client and server at start of a connection and MAY be sent at any other time by either client and server over lifetime of connection
- SETTINGS frames always apply to a connection, never a single stream
- Stream identifier for a SETTINGS frame MUST be zero (0x0)
- Payload of a SETTINGS frame consists of zero or more Parameters
- Each Parameter consists of an 16-bit setting identifier and 32-bit value

SETTINGS frame Parameters

- SETTINGS_MAX_FRAME_SIZE: Indicates size of largest frame payload that sender is willing to receive (in bytes)
- SETTINGS_INITIAL_WINDOW_SIZE: Indicates sender's initial window size (in bytes) for stream-level flow control
- SETTINGS_MAX_CONCURRENT_STREAMS: Indicates maximum number of concurrent streams that sender will allow
- SETTINGS_ENABLE_PUSH: This setting can be used to disable server push
- …

Example – GET request

- HTTP/1.1 requests and responses, and equivalent HTTP/2 requests and responses
- An **HTTP GET** request includes request header fields and no payload body and is therefore transmitted as a single **HEADERS frame**

```
GET /resource HTTP/1.1
Host: example.org
Accept: image/jpeg
```

HEADERS frame

```
HEADERS
    + END_STREAM
    + END_HEADERS
      :method = GET
      :scheme = https
      :path = /resource host = example.org
      host: example.org
      accept = image/jpeg
```

+ END_STREAM: end stream flag set
+ END_HEADERS: end header flag set

Example – HTTP response to GET request

- An HTTP response to GET request includes response header fields and payload/no payload body and is therefore transmitted as HEADERS frame(s) and DATA frame(s)

```
HTTP/1.1 304 Not Modified
ETag: "xyzzy"
Expires: Thu, 23 Jan ...
```

HEADERS frame

```
HEADERS
    + END_STREAM
    + END_HEADERS
      :status = 304
      etag = "xyzzy"
      expires = Thu, 23 Jan ...
```

Example – Post request

- HTTP/2: 1- HEADERS frame, 2- zero or more CONTINUATION frames containing request header fields, 3- one or more DATA frames, (CONTINUATION (or HEADERS) frame having END_HEADERS flag set and final DATA frame having END_STREAM flag set)

```
POST /resource HTTP/1.1
Host: example.org
Content-Type: image/jpeg
Content-Length: 123
{binary data}
```

1- HEADERS frame

```
HEADERS
    - END_STREAM
    - END_HEADERS
     :method = POST
     :path = /resource {binary data}
     :scheme = https
```

2- CONTINUATION frame

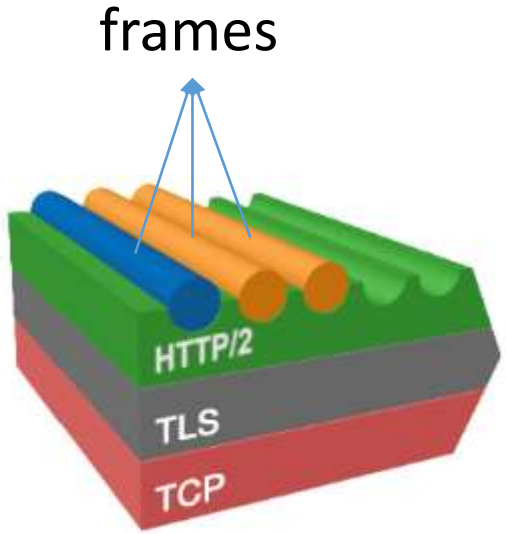
```
CONTINUATION
    + END_HEADERS
    content-type = image/jpeg
    host = example.org
    content-length = 123
```

3- DATA frame

```
DATA
  + END_STREAM {binary data}
```

Security

- HTTP/2 uses TLS (SSL) for security over TCP connection
- TLS (SSL) **acts on frames** in source and destination
- For any individual frame, TLS provides: encryption, data integrity, and end-point authentication



HTTP/2 - Shortcoming

HTTP/2 over single TCP connection means:

- Recovery from packet loss still stalls all object transmissions
  - as in HTTP 1.1, browsers have incentive to open **multiple parallel** TCP connections to reduce stalling (staying on object transmission), increase overall throughput

HTTP/2 to HTTP/3

- HTTP/3: per object error- and congestion-control (more pipelining) over QUIC (QUIC: Quick UDP Internet Connection), QUIC has TLS inside

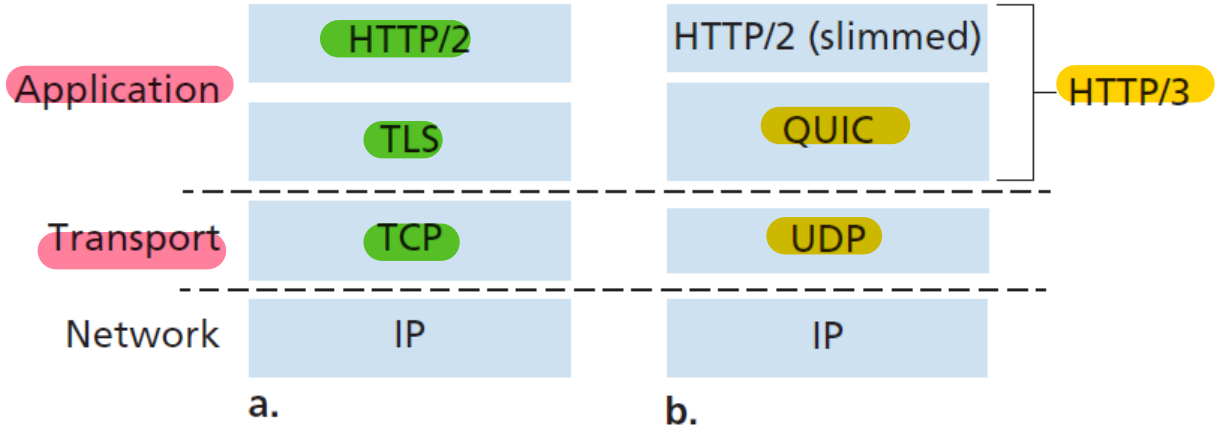


Figure 3.58 (a) traditional secure HTTP protocol stack, (b) secure QUIC-based HTTP/3 protocol stack

- more on HTTP/3 in transport layer (Chapter 3)

HTTP/3

- HTTP/3 uses QUIC transport protocol and an internal framing layer similar to HTTP/2.
- QUIC is a new “transport” protocol, **implemented in application layer** over UDP
- QUIC has message interleaving, per-stream flow control, and low-latency connection establishment
- **HTTP/3 has not yet been fully standardized**
- Many of HTTP/2 features (such as message interleaving) are subsumed by QUIC, (a simpler HTTP/2), allowing for a simpler, streamlined design for HTTP/3
- Once a client knows HTTP/3 server exists at a certain endpoint, it **opens a QUIC connection**
- QUIC incorporates **TLS 1.3**, offering comparable security to running TLS over TCP

2.2.4 Cookies

First time a user visits a site, server creates a cookie file and sent to client

Cookie has four components:

1. Set cookie: a cookie header line in HTTP response message
2. Cookie: a cookie header line in HTTP request message
3. a cookie file kept on user's computer and managed by user's browser
4. a back-end database at Web site

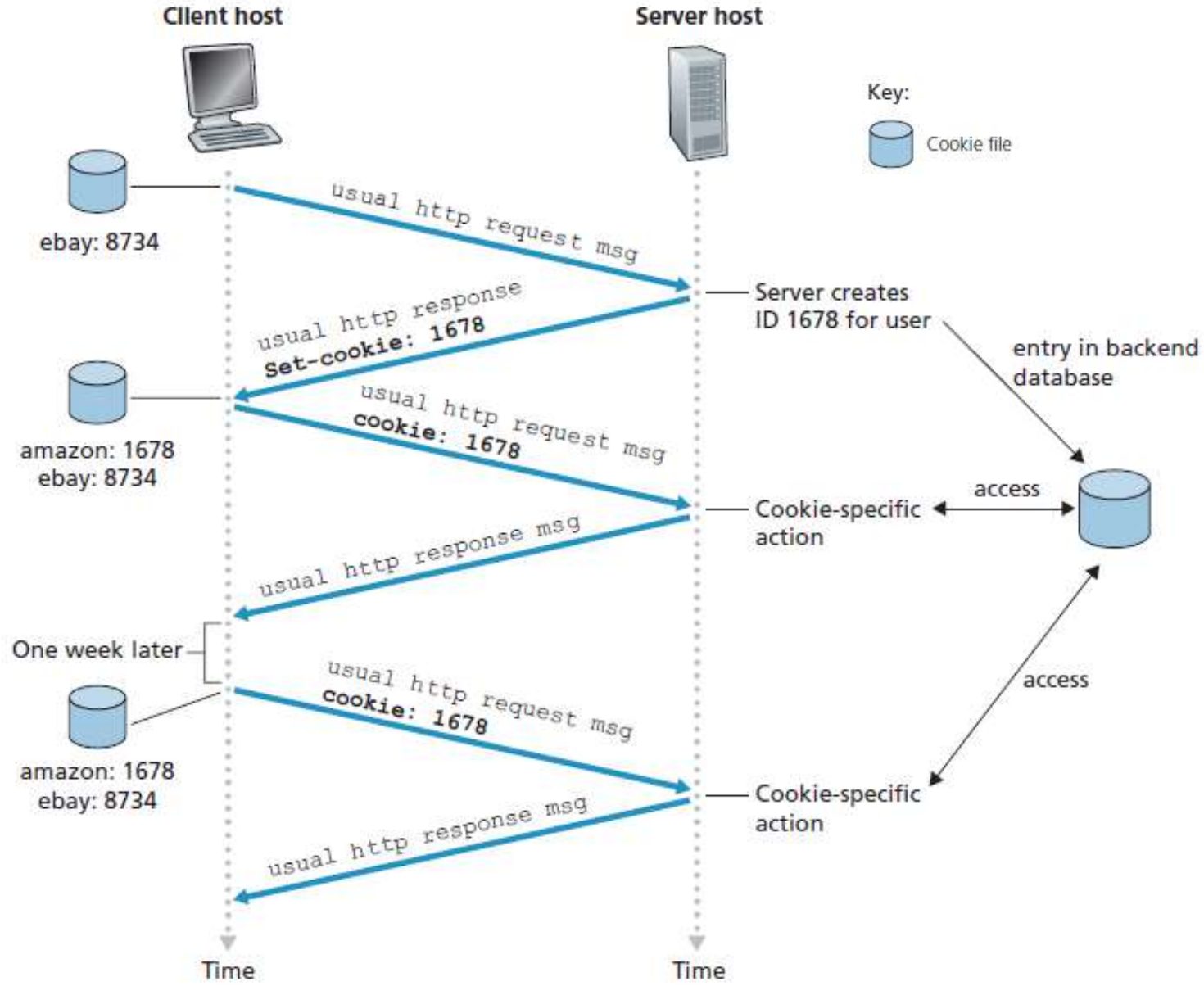


Figure 2.10 Keeping user state with cookies

Cookies

- Cookies can be used to create a user session layer on top of stateless HTTP
- Cookies can be used for
  - authorization
  - shopping
  - recommendations
  - user session state (Web e-mail)

- User privacy: cookies permit sites to *learn* a lot about you on their site
  - Using a combination of **cookies** and **user-supplied account information**, a Web site can learn a lot about a user and potentially sell this information to a third party

Cookie: Example

- `Set-Cookie`: `<cookie-name>=<cookie-value>; <cookie-name>=<cookie-value>; …`
- cookie-name: name, id, domain, send for, created, expires, …

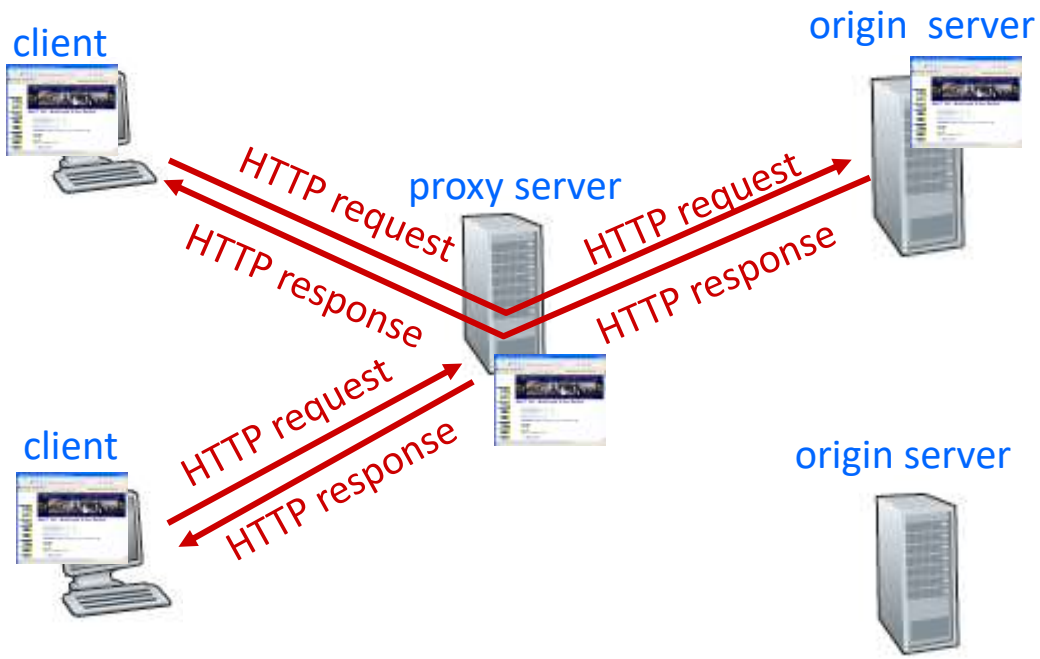
Example

- Set-Cookie: name=`X-OWA-CANARY;` id=`ow36HTOyAPq18Hn3/AlEZlh8PEpXBkH97_;`
  Domain=`email.iust.ac.ir;` Send for= `Secure connections only;` Created=
  `Wednesday, February 19, 2020 at 2:18:17 PM;` Expires= `Thursday,`
  `January 27, 2022 at 4:59:34 PM`

2.2.5 Web Caching

Satisfy client request without involving origin server

- user configures browser by (IP address and port) of a *Web cache process*
- browser sends all HTTP requests to cache process
  - *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)

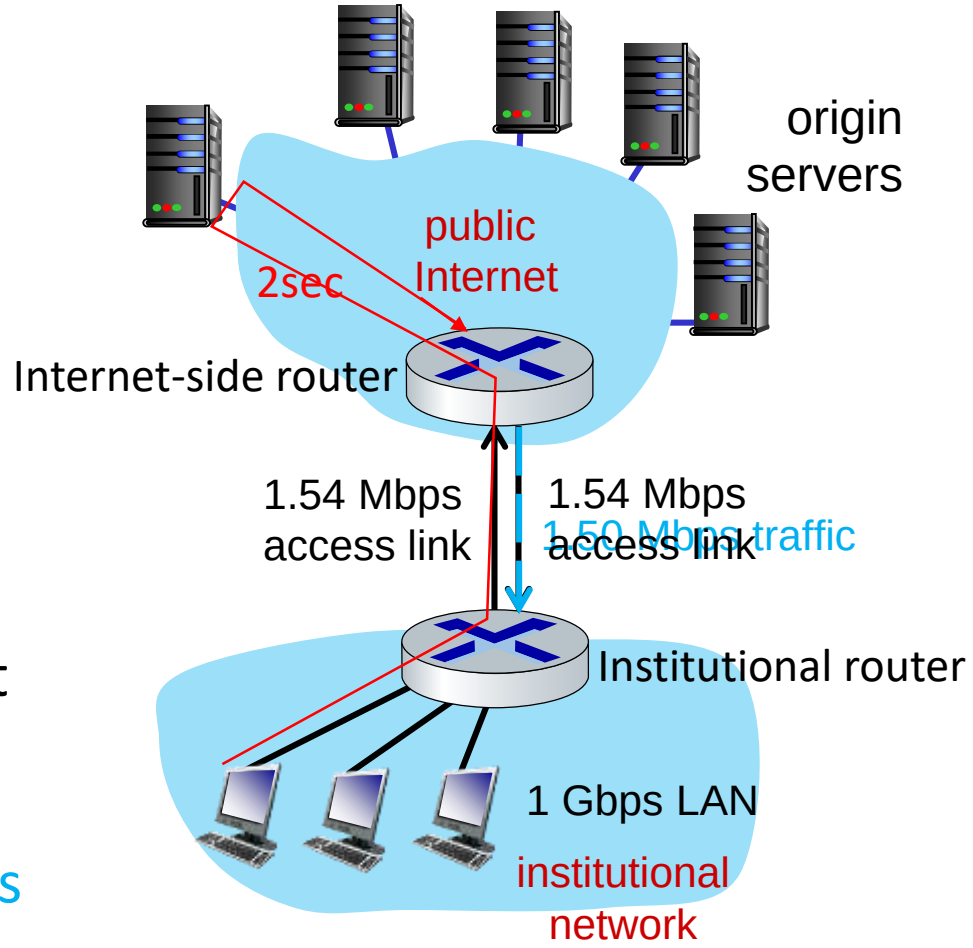
- Web cache server acts as both client and server
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

- *Why* Web caching?
- reduce response time for client request
  - cache is closer to client
- reduce traffic on an institution's access link
- Internet is dense with caches
  - enables "poor" content providers to more effectively deliver content

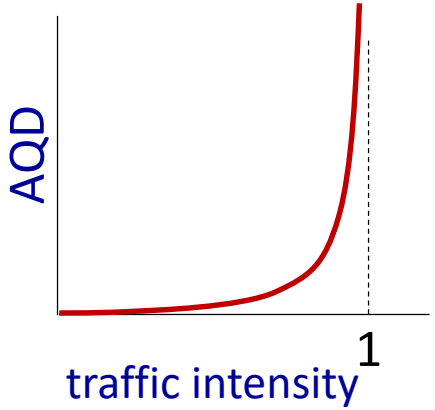
Caching example

Suppose:

- access link rate: 1.54 Mbps
- time from clients sending http request until message's packets inter to Internet-side router is 2sec on average
- average file size: 100K bits
- average request rate from browsers in client PCs to origin servers: 15requests/sec
  - average data rate to browsers: 100Kbits*15 bits-request/sec=1.50 Mbps



Caching example



Performance:

LAN utilization= 1.5Mbps/1Gbps = 0.0015

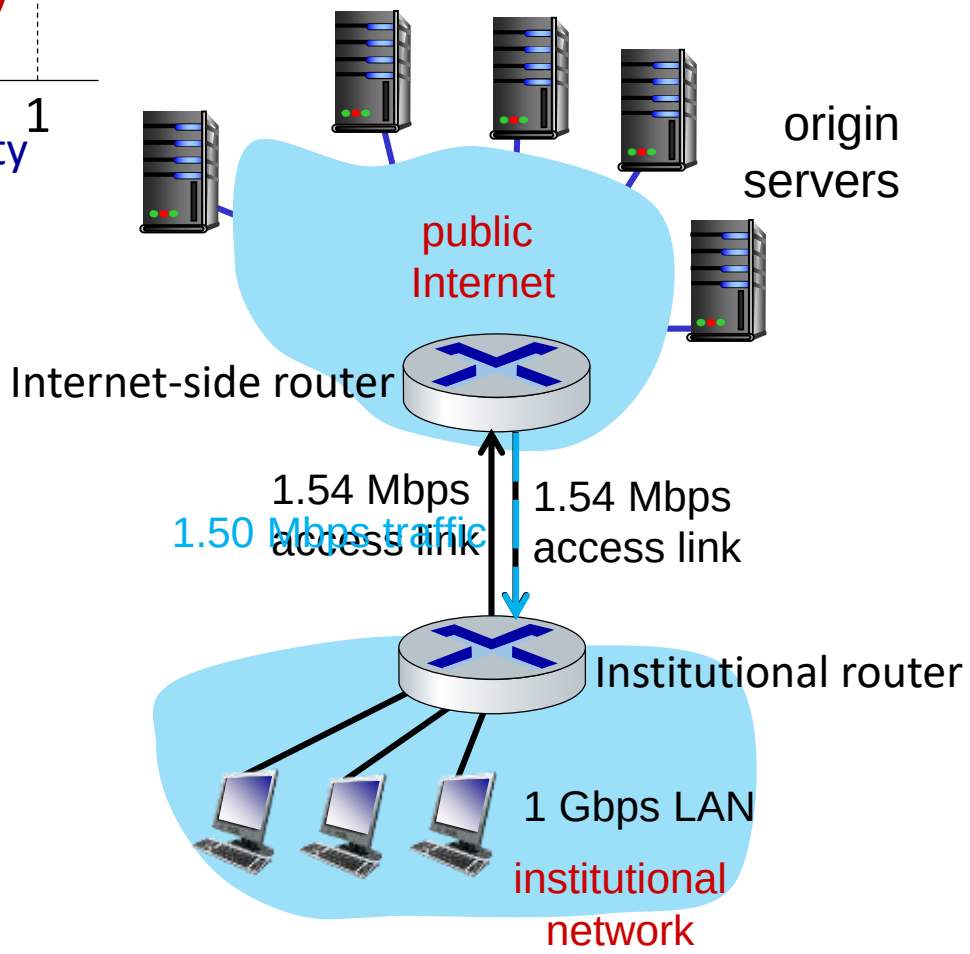
- Traffic Intensity on Internet-side router =1.5/1.54=0.97=97%

problem: large queuing delays

- end-end delay =

Internet delay + access link delay + LAN delay =

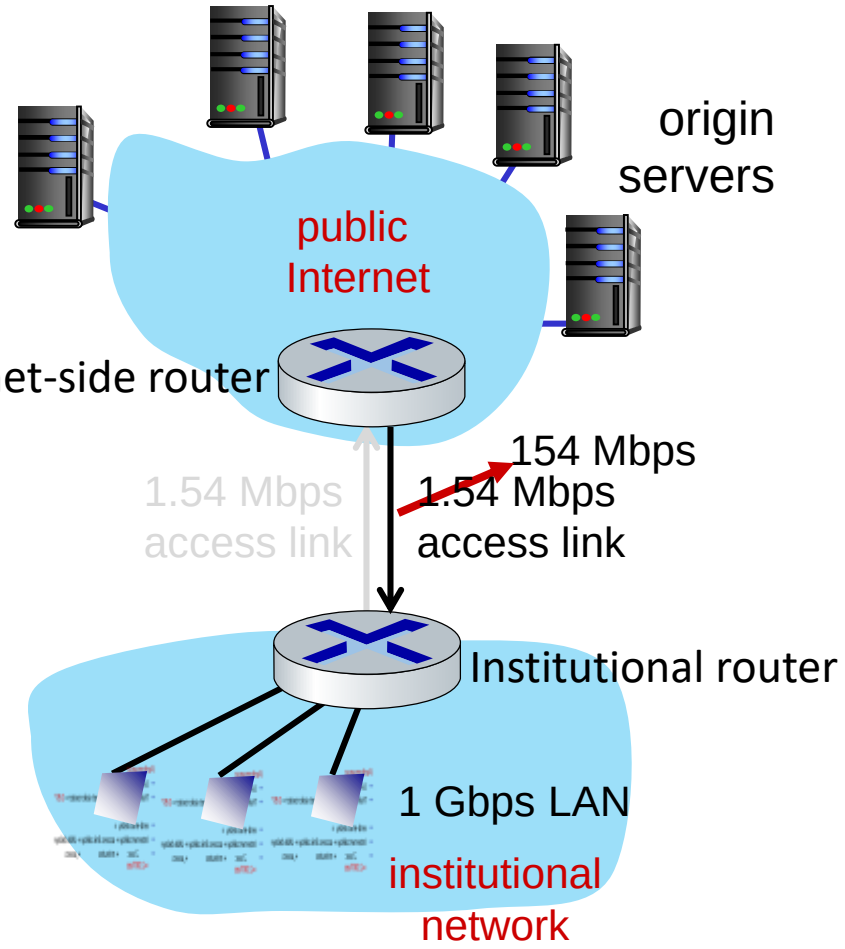
2 sec + minutes + μsecs



Caching example: buy a faster access link

Up grade Scenario: 154 Mbps

- access link rate: 1.54 Mbps
- *Cost:* faster access link (expensive)



Performance: 9.7%

- LAN utilization: 0.0015
- Traffic Intensity on Internet-side router = 0.97

- end-end delay =

Internet delay + access link delay + LAN delay=

2 sec + minutes + μsecs ≈2.007sec

msecs

Caching example: install a web cache

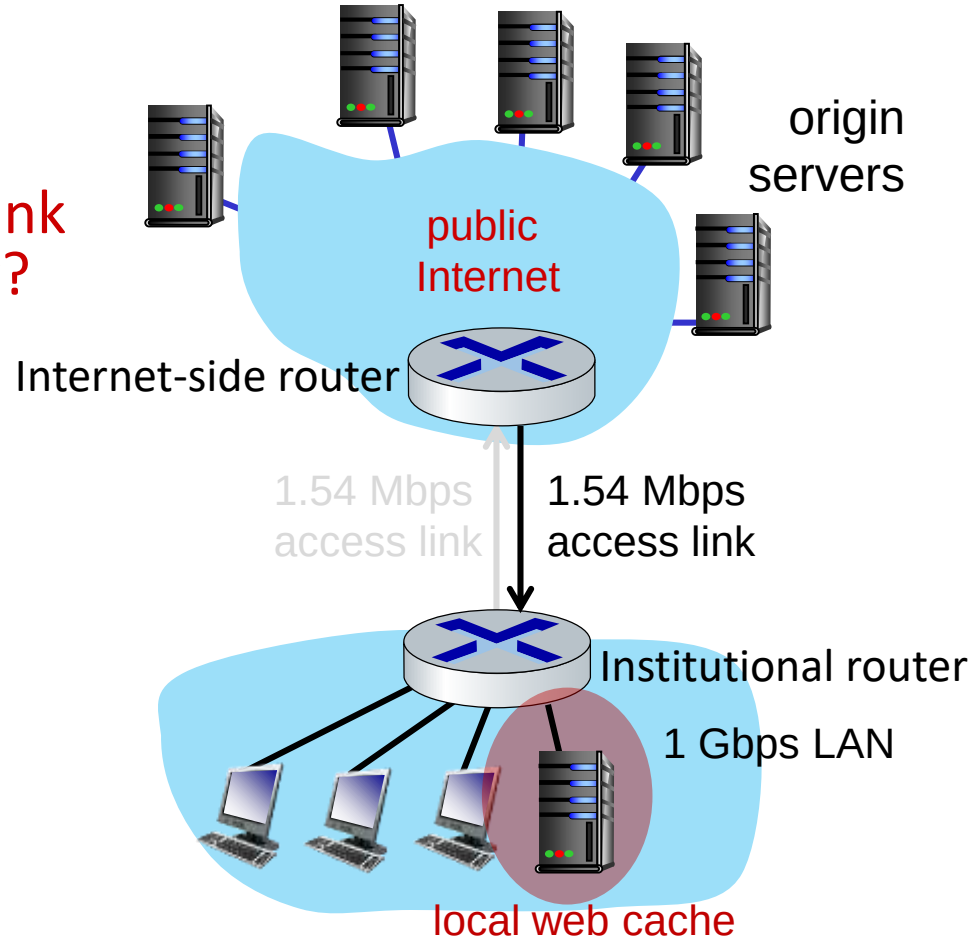
Performance:

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

How to compute link utilization, delay?

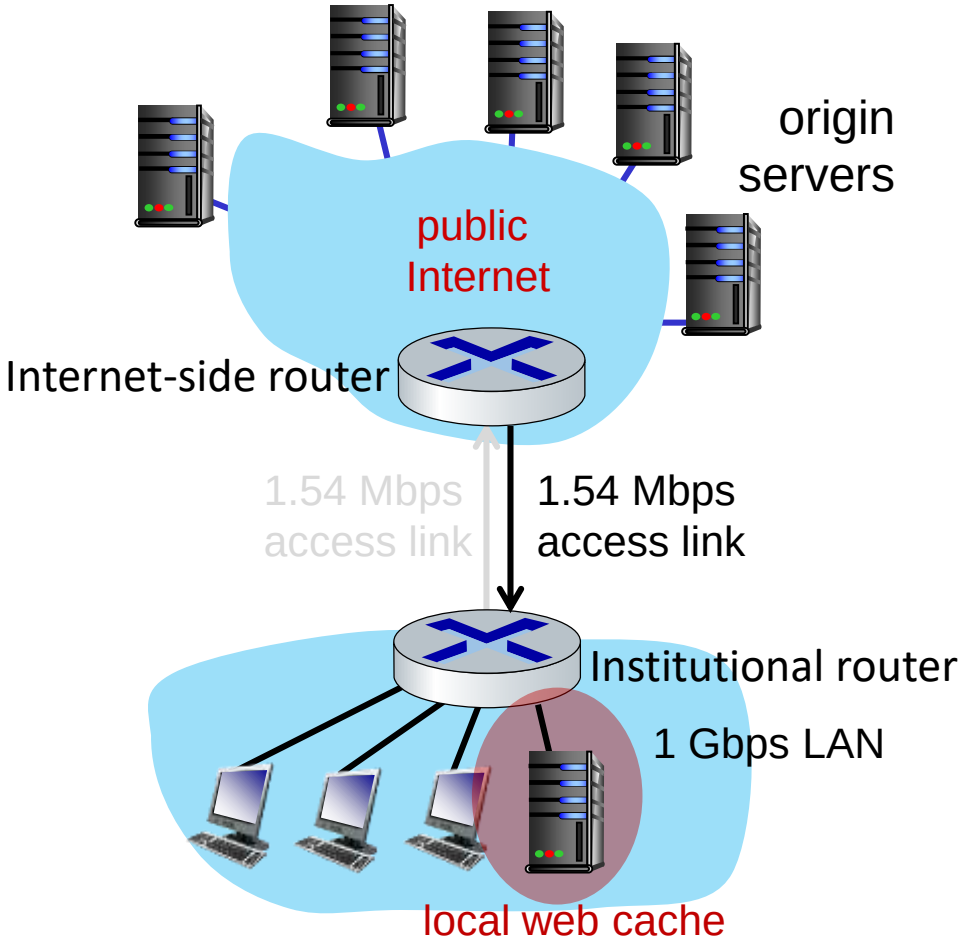
- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin

Cost: web cache (cheap)



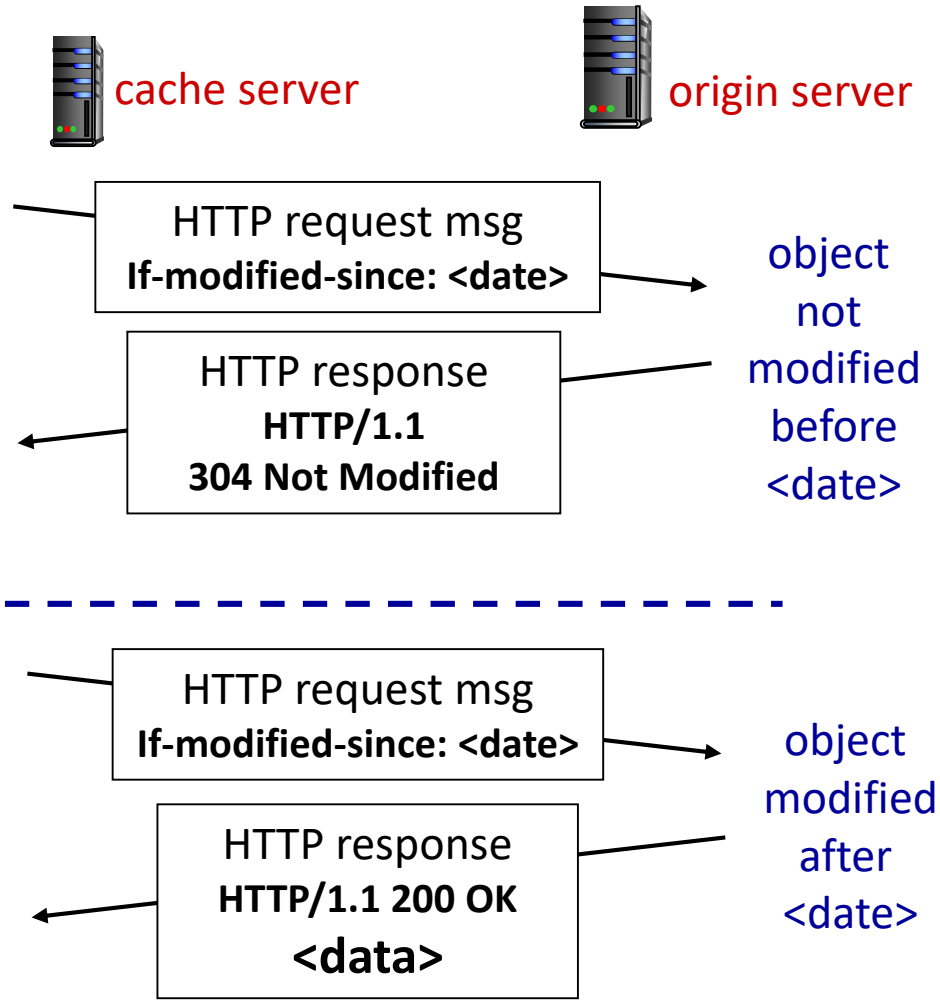
Caching example: install a web cache

- access link: 60% of requests use access link
- data rate to browsers over access link=
  = 0.6 * 1.50 Mbps = 0.9 Mbps
- Traffic Intensity = 0.9/1.54 = 0.58
- lower average end-end delay than with 154 Mbps link (and cheaper too)
- 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



Conditional GET and Cache

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



Contents

2.3 Electronic Mail in the Internet

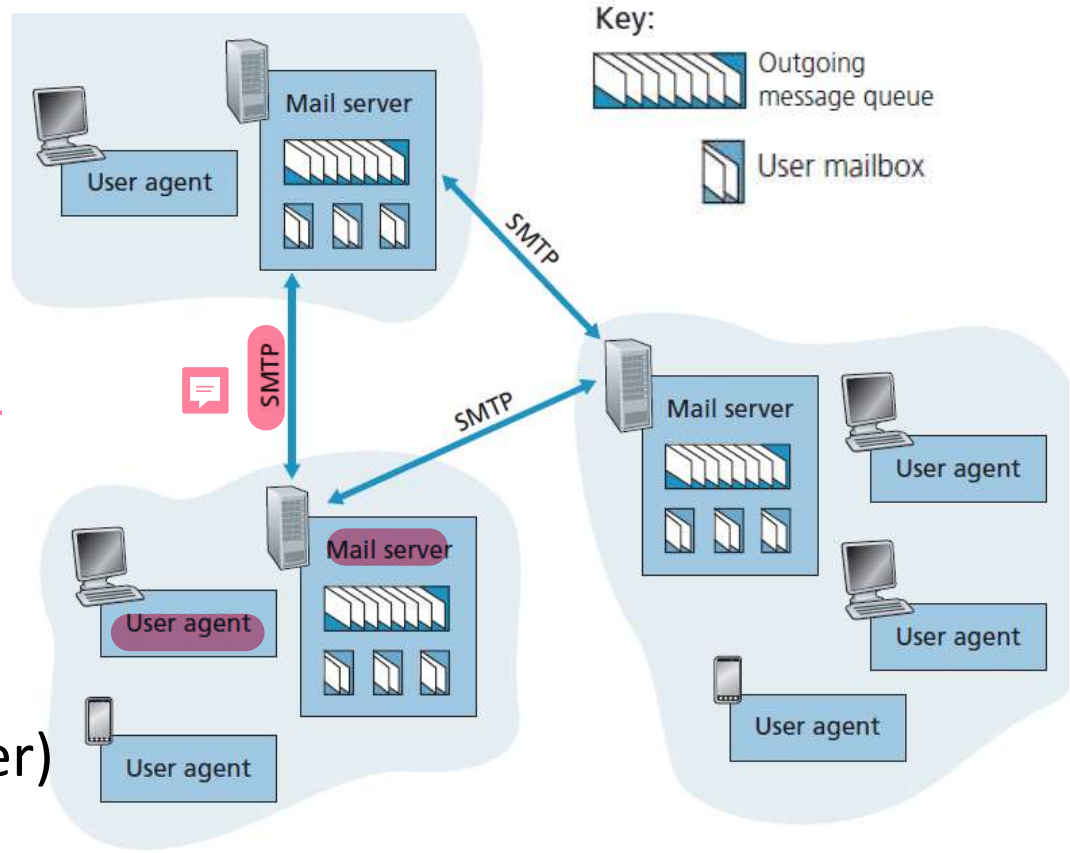
Three major components:

1. user agents
2. mail servers
3. simple mail transfer protocol: SMTP

1. User Agent (a.k.a. “mail reader”)

- composing, editing, reading mail messages (e.g., Outlook, Apple Mail (iPhone mail client))

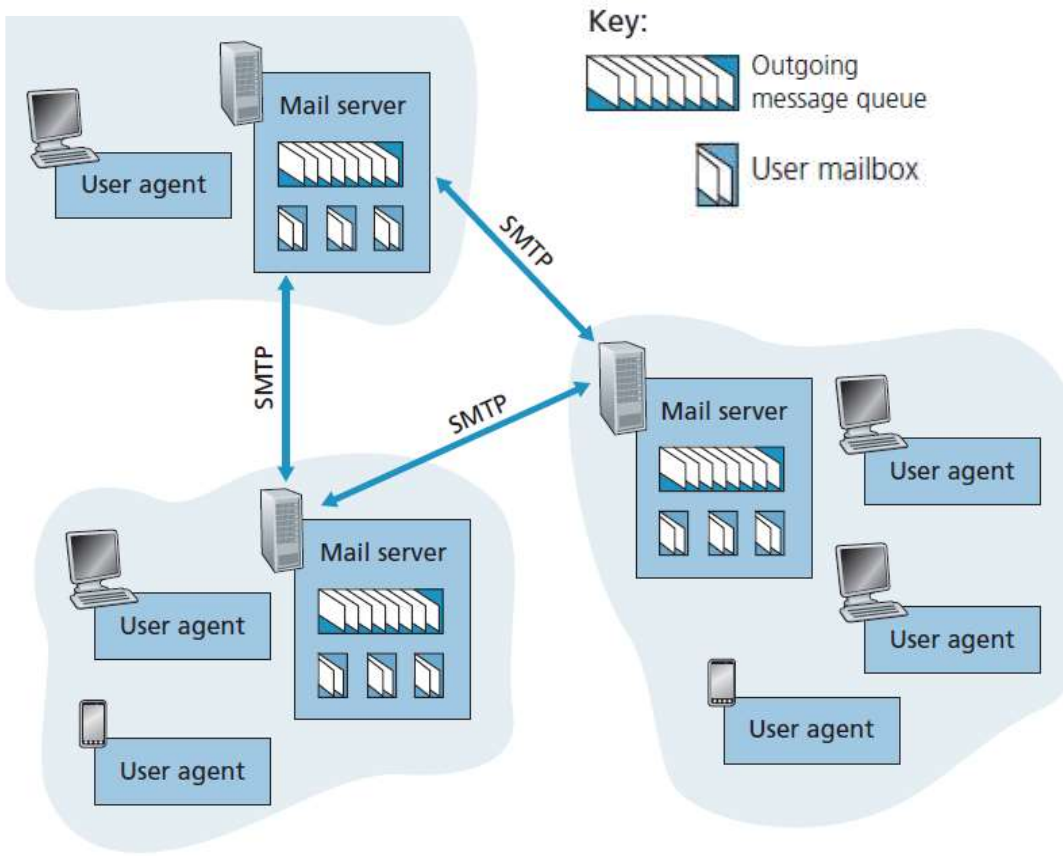
(outgoing, incoming messages stored on server)



2.3 Electronic Mail in the Internet

2. Mail servers

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



SMTP, RFC (5321)

3. SMTP

- Uses TCP to reliably transfer email message from client (mail server initiating connection) to destination mail server, port 25
- Direct transfer: **Sending server** to **Receiving server**
- Three phases of transfer:
  - handshaking (greeting)
  - transfer of messages
  - closure
- Command/Response interaction (like HTTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCI

Figure 2.15 Alice sends a message to Bob

1) Alice uses UA to compose e-mail message "to" bob@someschool.edu

2) Alice's UA sends message to her mail server; message placed in message queue

3) client side of SMTP (crepes.fr) opens TCP connection with Bob's mail server

4) SMTP client sends Alice's message over TCP connection

5) Bob's mail server places the message in Bob's mailbox

6) Bob invokes his user agent to read message

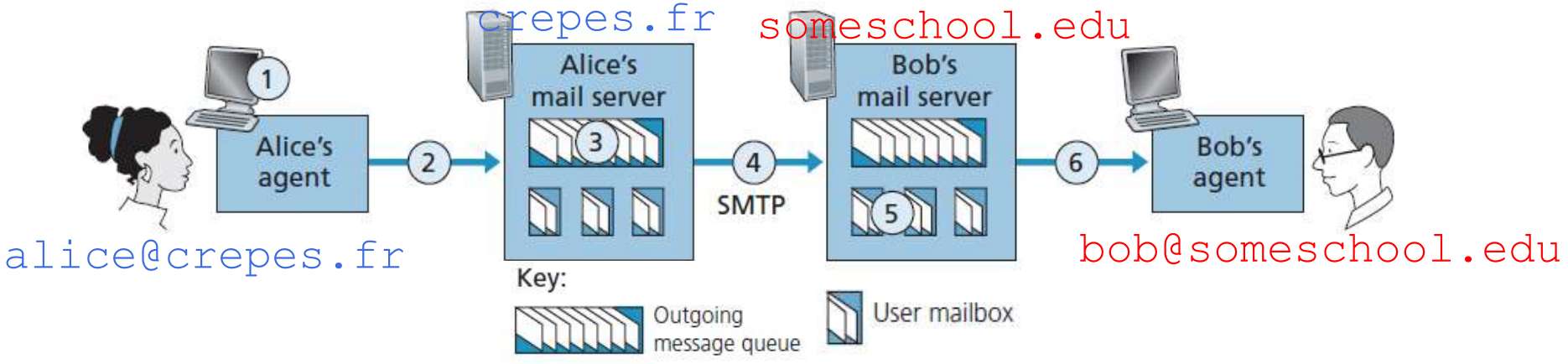


Figure 2.15 Alice sends a message to Bob

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

S=someschool.edu
C=crepes.fr

alice@crepes.fr

Send an email to

bob@someschool.edu

SMTP: closing observations

Comparison with HTTP1.1:

- HTTP1.1: pull (HTTP/2: pull, push)
- SMTP: push
- Both have ASCII command/response interaction, status codes
- HTTP1.1: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message
- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 531

RFC 822 defines syntax for e-mail message itself

- header lines
  - To:
  - From:
  - Subject:

  these lines are different from SMTP MAIL FROM:, RCPT TO: commands
- body: "message" , ASCII characters only

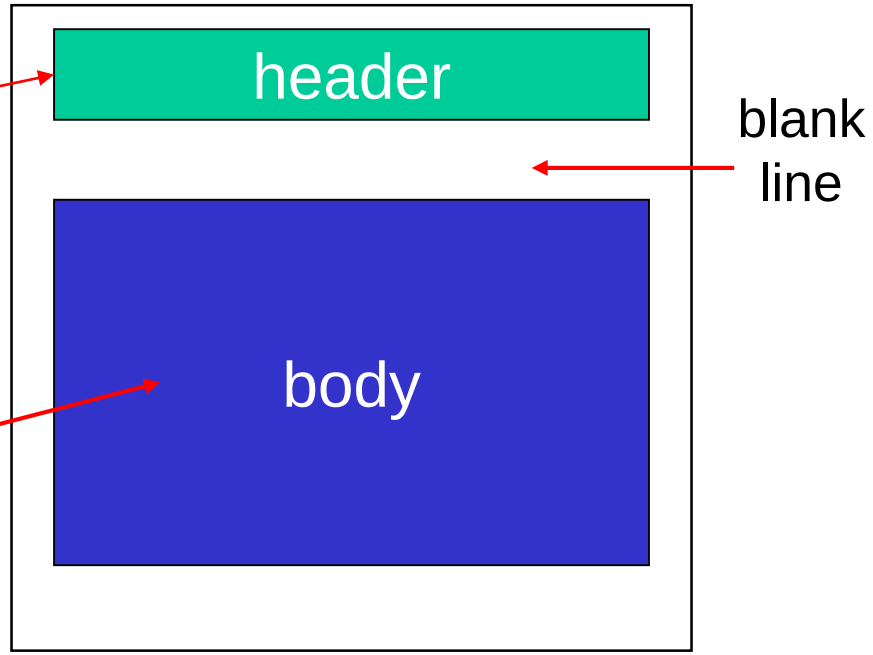
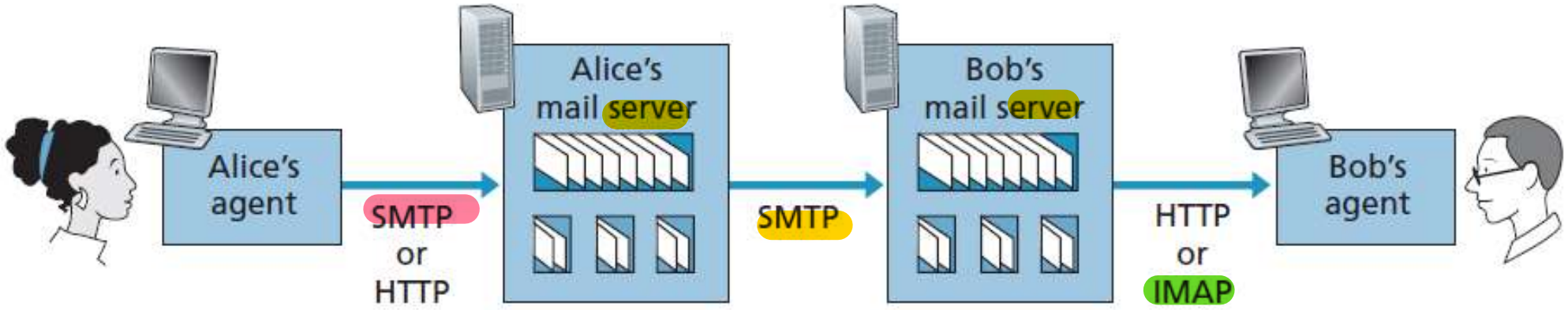
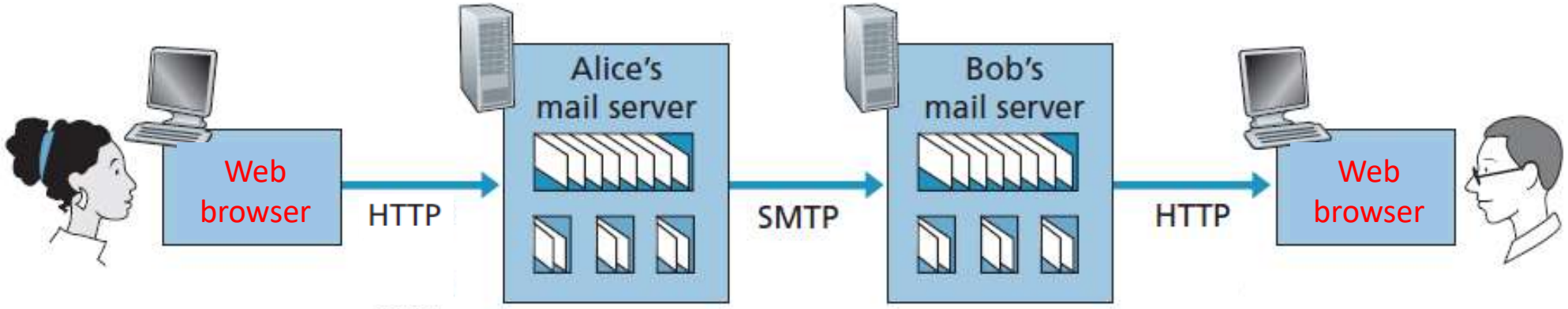


Figure 2.16



- Bob' s UA can't use SMTP to obtain messages because obtaining messages is a pull operation, whereas SMTP is a push protocol
- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
  - POP3: Post Office Protocol-Version3, is extremely simple protocol
  - HTTP

Web-Based E-Mail



- Sending and accessing through Web browsers
- Hotmail introduced Web-based E-Mail in mid 1990s
- Now: Google, Yahoo , universities and corporations, …
- UA is a Web browser (user communicates with its remote mailbox via HTTP)
- Mail server still sends messages to, and receives messages from, other mail servers using SMTP

Contents

DNS: Domain Name System, RFC:1035

- People: many identifiers:
  - Name, SSN (کد ملی), Passport number, …
- Internet hosts, routers:
  - IP address (32/128 bit) - used for addressing an interface of a host or router
  - "domain name", e.g., www.iust.ac.ir and gw-vlan-2451.cs.umass.edu used by humans for addressing a host and an interface of a router

Q: map between IP address and "domain name", and vice versa?

A: Domain Name System:

- distributed database implemented in hierarchy of many name servers
- application-layer protocol: "host" - "name server" communication to resolve names (address/name translation)

2.4.1 Services Provided by DNS

DNS services

- Hostname to IP address translation and vice versa
  - host domain name ↔ host IP address

- A host CAN have a **real** (**canonical**) domain name and one or several **alias** (نام مستعار) names
- A domain name for multiple servers: many IP addresses correspond to one name (for load distribution)
  - How? A set of IP addresses is associated with one alias hostname

Host aliasing

- A host with a complicated hostname can have one or more alias names
- For example, a hostname such as relay1.west-coast.enterprise.com could have, two aliases such as enterprise.com and www.enterprise.com
- relay1.west-coast.enterprise.com is said to be a canonical hostname

- DNS can be invoked by an application to obtain canonical hostname for a supplied alias hostname as well as IP address of host

IUST mail server aliasing

- IUST provides email service to its people
  - For obvious reasons, desirable addresses is @iust.ac.ir
- However, canonical (real) hostname of IUST mail server is mxgate.iust.ac.ir (so email address should be …@mxgate.iuat.ac.ir)
- **MX record** permits a IUST's mail server and Web server have identical (aliased) hostnames: iust.ac.ir

Example

- jupitr.iust.ac.ir is authoritative Name Server of Iran University of Science and Technology
- Here is a name resolution response from jupitr.iust.ac.ir for iust.ac.ir

Type	Hostname	IP Address	TTL
A	iust.ac.ir	194.225.238.111 Iran University Of Science and Technology (AS41620)	48 hrs
MX	mxgate.iust.ac.ir	194.225.230.66 Iran University Of Science and Technology (AS41620)	48 hrs
MX	mxgate2.iust.ac.ir	194.225.230.74 Iran University Of Science and Technology (AS41620)	48 hrs
MX	mxgate3.iust.ac.ir	194.225.230.190 Iran University Of Science and Technology (AS41620)	48 hrs

- MX record permits a company's mail server and Web server to have identical (aliased) hostnames

Load distribution

- Busy sites, such as tsetmc.com, are replicated over multiple servers, having a different IP address
- A set of IP addresses is associated with one alias hostname
- DNS database contains this set of IP addresses
- When clients make a DNS query for a name mapped to a set of addresses, server responds with entire set of IP addresses, but rotates ordering of addresses within each reply
- Client sends its HTTP request message to IP address that is listed first in set, DNS rotation distributes traffic among replicated servers
- DNS rotation is also used for e-mail so that multiple mail servers can have same alias name

2.4.2 Overview of How DNS Works

- DNS implemented as a Distributed data base

Q: Why not centralize DNS?

- Single point of failure
- Traffic volume
- Distant centralized database: A single DNS server cannot be "close to" all
- Maintenance
- doesn‘t scale : Comcast DNS servers alone: 600Billion DNS queries per day (208.67.222.222 and 208.67.220.220)

A Distributed, Hierarchical Database

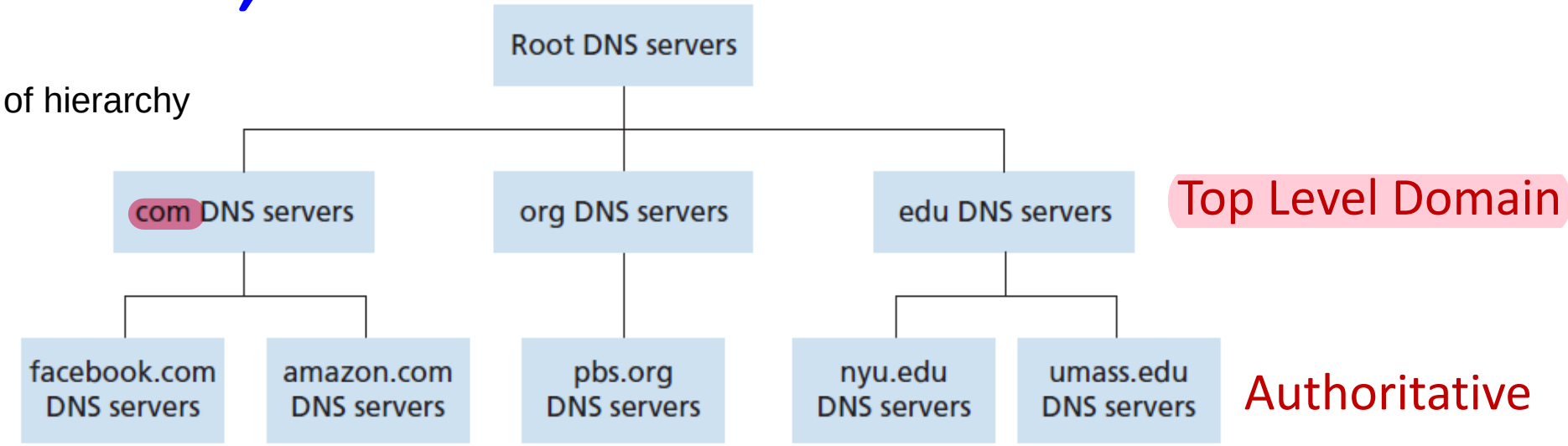


Figure 2.17 Portion of hierarchy of DNS servers

- Client queries local name server to get IP address for www.amazon.com
- Local name server queries root server to find .com DNS TLD server
- Local name server queries .com DNS server to get amazon.com DNS authoritative server
- Local name server queries amazon.com DNS authoritative server to get IP address for www.amazon.com and cache and send address to client.

Figure 2.19 Interaction of various DNS servers

- Example: host at domain name: cse.nyu.edu wants IP address for gaia.cs.umass.edu
- Recursive Query: 1,6
  - Send me answer
- Iterated Query: 2, 4
  - How knows answer?
  - puts burden of name resolution on contacting name server
  - root and TLD DNS replies with IP address of server to contact
  - 3, 5 : "I don't know answer, but ask this server"

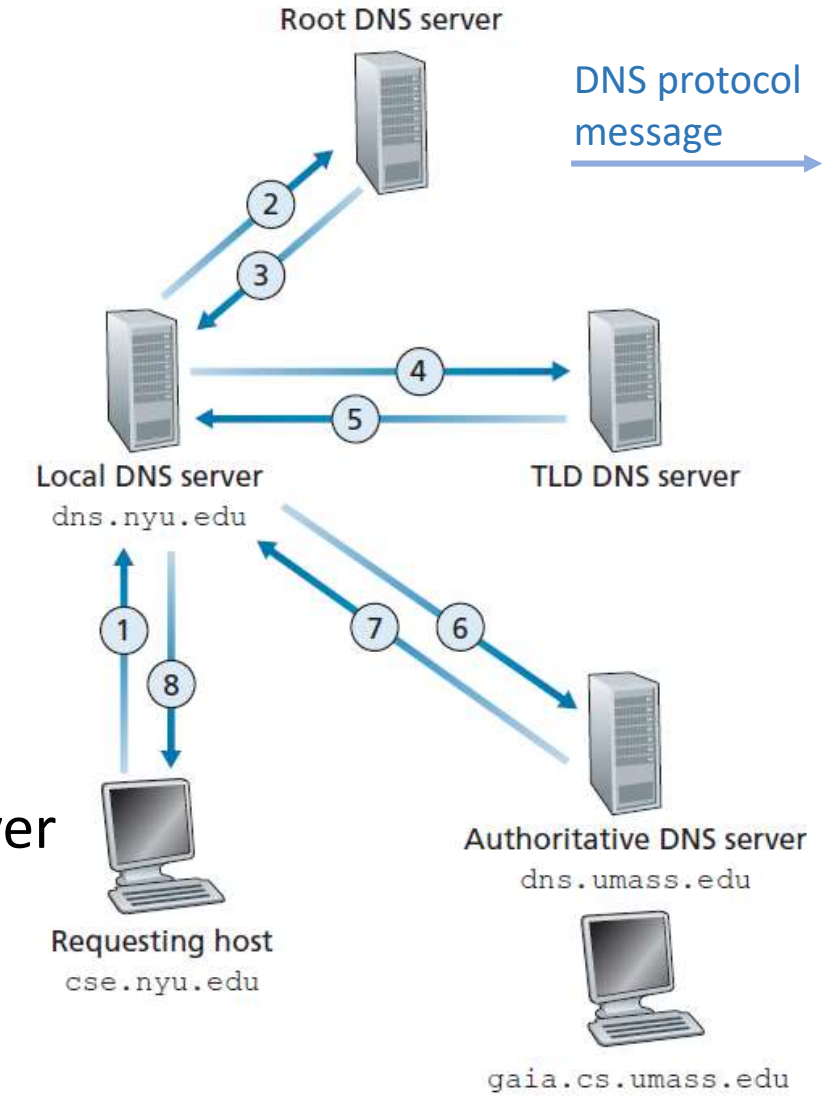
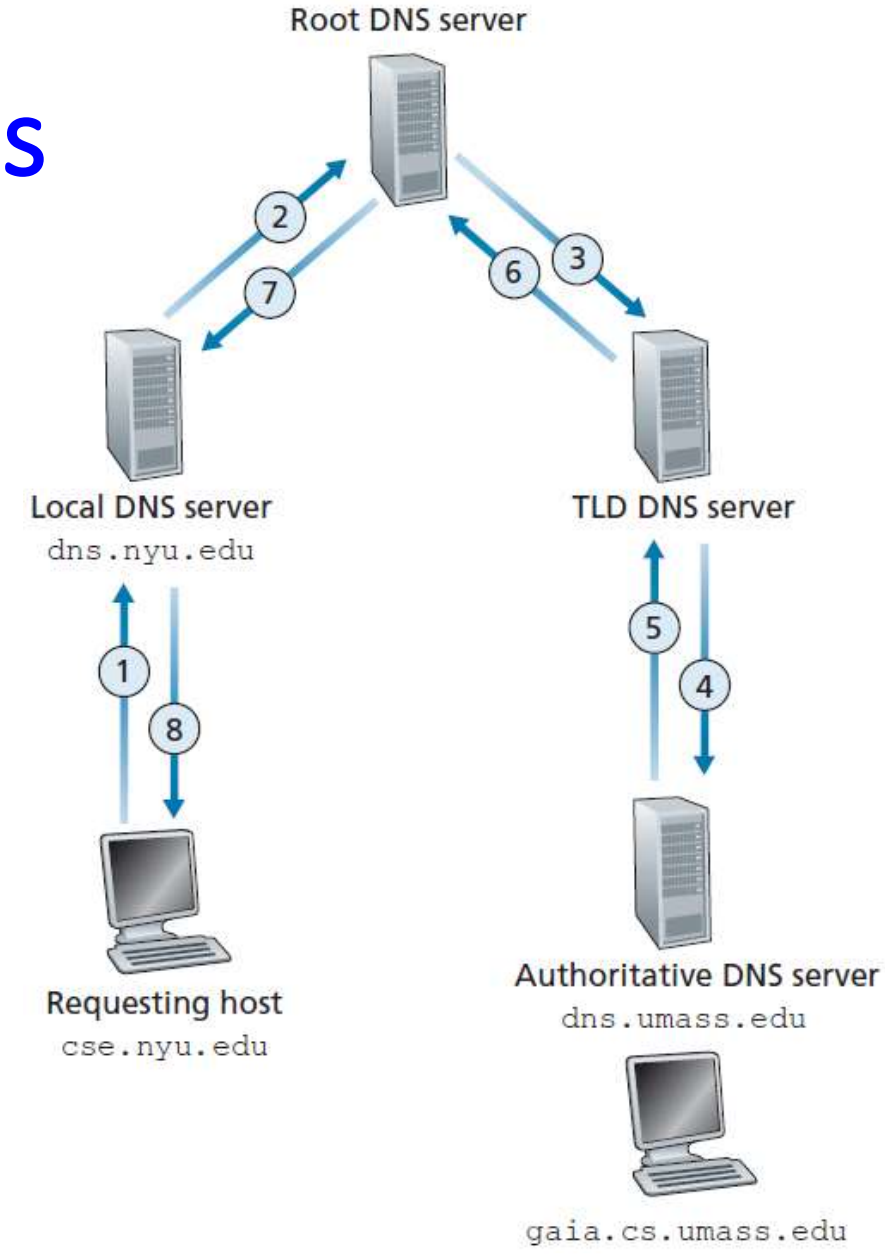


Figure 2.20 Recursive Queries

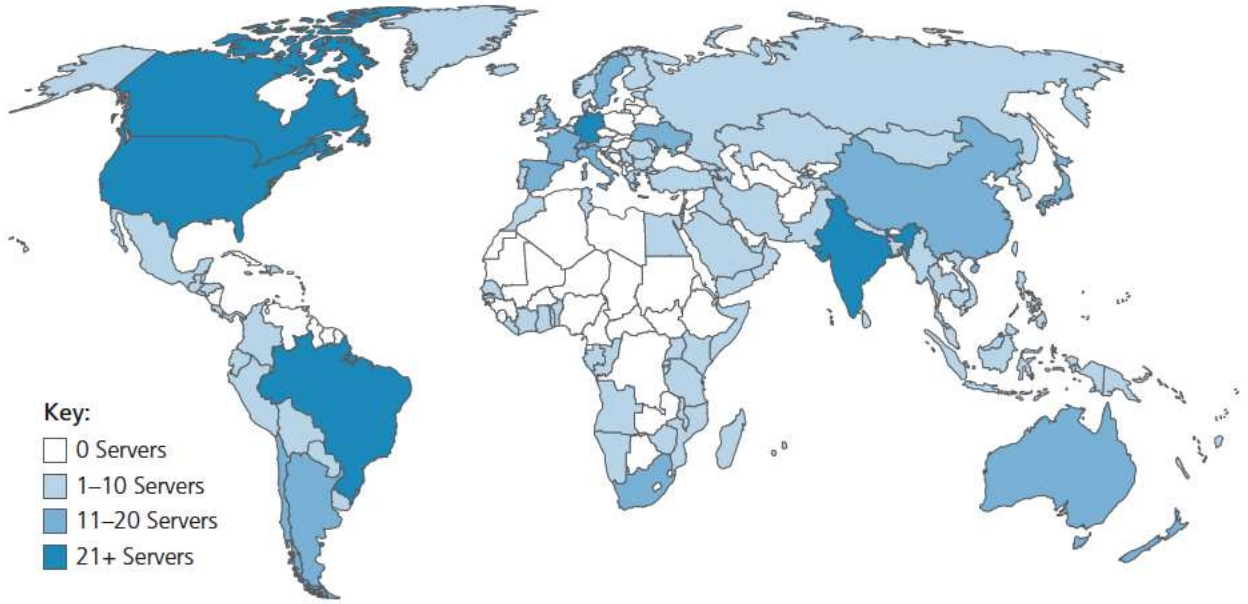
- Recursive Query: 1, 2, 3, 4
- puts burden of name resolution on contacted name server
- heavy load at roots and TLD DNS servers



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 (Domain Name System Security Extensions) 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)



https://www.iana.org/domains/root/servers

HOSTNAME	IP ADDRESSES	OPERATOR
a.root-servers.net	198.41.0.4, 2001:503:ba3e::2:30	Verisign, Inc.
b.root-servers.net	199.9.14.201, 2001:500:200::b	University of Southern California, Information Sciences Institute
c.root-servers.net	192.33.4.12, 2001:500:2::c	Cogent Communications
d.root-servers.net	199.7.91.13, 2001:500:2d::d	University of Maryland
e.root-servers.net	192.203.230.10, 2001:500:a8::e	NASA (Ames Research Center)
f.root-servers.net	192.5.5.241, 2001:500:2f::f	Internet Systems Consortium, Inc.
g.root-servers.net	192.112.36.4, 2001:500:12::d0d	US Department of Defense (NIC)
h.root-servers.net	198.97.190.53, 2001:500:1::53	US Army (Research Lab)
i.root-servers.net	192.36.148.17, 2001:7fe::53	Netnod
j.root-servers.net	192.58.128.30, 2001:503:c27::2:30	Verisign, Inc.
k.root-servers.net	193.0.14.129, 2001:7fd::1	RIPE NCC
l.root-servers.net	199.7.83.42, 2001:500:9f::42	ICANN
m.root-servers.net	202.12.27.33, 2001:dc3::35	WIDE Project

Top Level Domain

Top-Level Domain (TLD) servers:

- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .ir, .cn, .uk, .fr, .ca, .jp
- Registrar IRNIC (مرکز ثبت دامنه کشوری ایران): authoritative registry for .ir, maintain .ir TLD
- Registrar is a commercial entity that verifies uniqueness of domain name, enters domain name into LTD name server
- Complete list of registrars: http:// www. internic. net

TLD name server

- When IUST register domain name iust.ac.ir with IRNIC, it provides IRNIC with names and IP addresses of IUST's primary and secondary authoritative DNS servers
  - jupitr.iust.ac.ir, 194.225.228.4
  - ns.iust.ac.ir, 194.225.228.8

- IRNIC enters a Type NS and a Type A record into TLD ir servers

iust.ac.ir.	86400	IN	NS	jupitr.iust.ac.ir.
iust.ac.ir.	86400	IN	NS	ns.iust.ac.ir.
jupitr.iust.ac.ir.	86400	IN	A	194.225.228.4
ns.iust.ac.ir.	86400	IN	A	194.225.228.8

Country-code top-level domain: .ir

- ccTLD Manager: Institute for Research in Fundamental Sciences (IPM)
- .ir TLD name servers:

University of Vienna
Austria

HOST NAME	IP ADDRESS(ES)
ns5.univie.ac.at	193.171.255.77 2001:628:453:4305:0:0:0:53
a.nic.ir	193.189.123.2
b.nic.ir	193.189.122.83
ir.cctld.authdns.ripe.net	193.0.9.85 2001:67c:e0:0:0:0:0:85

- Registry Information
  - URL for registration services: http://www.nic.ir مرکز ثبت دامنه های ایران
  - WHOIS Server: whois.nic.ir

Authoritative DNS server

Who is responsible for Authoritative DNS server:

- 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
- A TLD DNS server keeps name and IP address of any Authoritative DNS server

Local DNS name servers

- Does not strictly belong to hierarchy
- Each ISP (residential ISP, company, university) has one
  - “default name server” or
  - Use public local name servers such as: Google public DNS servers: 8.8.8.8 and 8.8.4.4.
- 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

Caching, Updating DNS Records

- 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
  - If a host changes its IP address, may not be known Internet-wide until all TTLs expire
- Update/notify mechanisms proposed IETF standard
  - RFC 2136

2.4.3 DNS Records and Messages

DNS: distributed database storing resource records (RR)

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

type=A
- name is hostname
- value is IP address

type=NS
- name is domain (e.g., iust.ac.ir)
- value is hostname of authoritative name server for this domain

type=CNAME
- name is alias name (nickname)
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical (real) name

type=MX
- value is name of mail server associated with name

DNS types

type	type id (decimal)	Description	Function
A	1	IPv4 address record	Returns a 32-bit IPv4 address
AAAA	28	IPv6 address record	Returns a 128-bit IPv6 address
CNAME	5	Canonical name record	Alias of one name to another
MX	15	Mail exchange record	Maps a domain name to a list of mail servers
NS	2	Name server record	Returns **DNS** servers responsible (authoritative) for a domain name (zone)

iust.ac.ir

Example: .ir TLD authority places NS records for iust.ac.ir domain in .ir TLD name server:

```
iust.ac.ir.    86400   IN    NS    jupitr.iust.ac.ir.
iust.ac.ir.    86400   IN    NS    ns.iust.ac.ir.
```

Windows Command Prompt

>nslookup iust.ac.ir a.nic.ir

```
Name:    iust.ac.ir
Served by:
- jupitr.iust.ac.ir
          194.225.228.4
          2001:14e8:4:99:194:225:228:4
          iust.ac.ir
- ns.iust.ac.ir
          194.225.228.5
          2001:14e8:4:99:194:225:228:5
          iust.ac.ir
```

Authoriative Name Servers

Windows Command Prompt

>nslookup -q=MX iust.ac.ir jupitr.iust.ac.ir

Mail Servers: for x@iust.ac.ir

```
Server:  jupitr.iust.ac.ir
Address:  194.225.228.4

iust.ac.ir      MX preference = 20, mail exchanger = mxgate3.iust.ac.ir
iust.ac.ir      MX preference = 5, mail exchanger = mxgate.iust.ac.ir
iust.ac.ir      MX preference = 5, mail exchanger = mxgate2.iust.ac.ir
iust.ac.ir      nameserver = jupitr.iust.ac.ir
iust.ac.ir      nameserver = ns1.nic.ir
iust.ac.ir      nameserver = ns.nic.ir
iust.ac.ir      nameserver = ns.iust.ac.ir
mxgate.iust.ac.ir       internet address = 194.225.230.66
mxgate2.iust.ac.ir      internet address = 194.225.230.74
mxgate3.iust.ac.ir      internet address = 194.225.230.190
ns.nic.ir       AAAA IPv6 address = 2001:960:618:70::89
ns.iust.ac.ir   internet address = 194.225.228.5
ns.iust.ac.ir   AAAA IPv6 address = 2001:14e8:4:99:194:225:228:5
ns1.nic.ir      internet address = 194.225.70.83
jupitr.iust.ac.ir       internet address = 194.225.228.4
jupitr.iust.ac.ir       AAAA IPv6 address = 2001:14e8:4:99:194:225:228:4
```

Windows Command Prompt

>nslookup -q=A mxgate.iust.ac.ir jupitr.iust.ac.ir

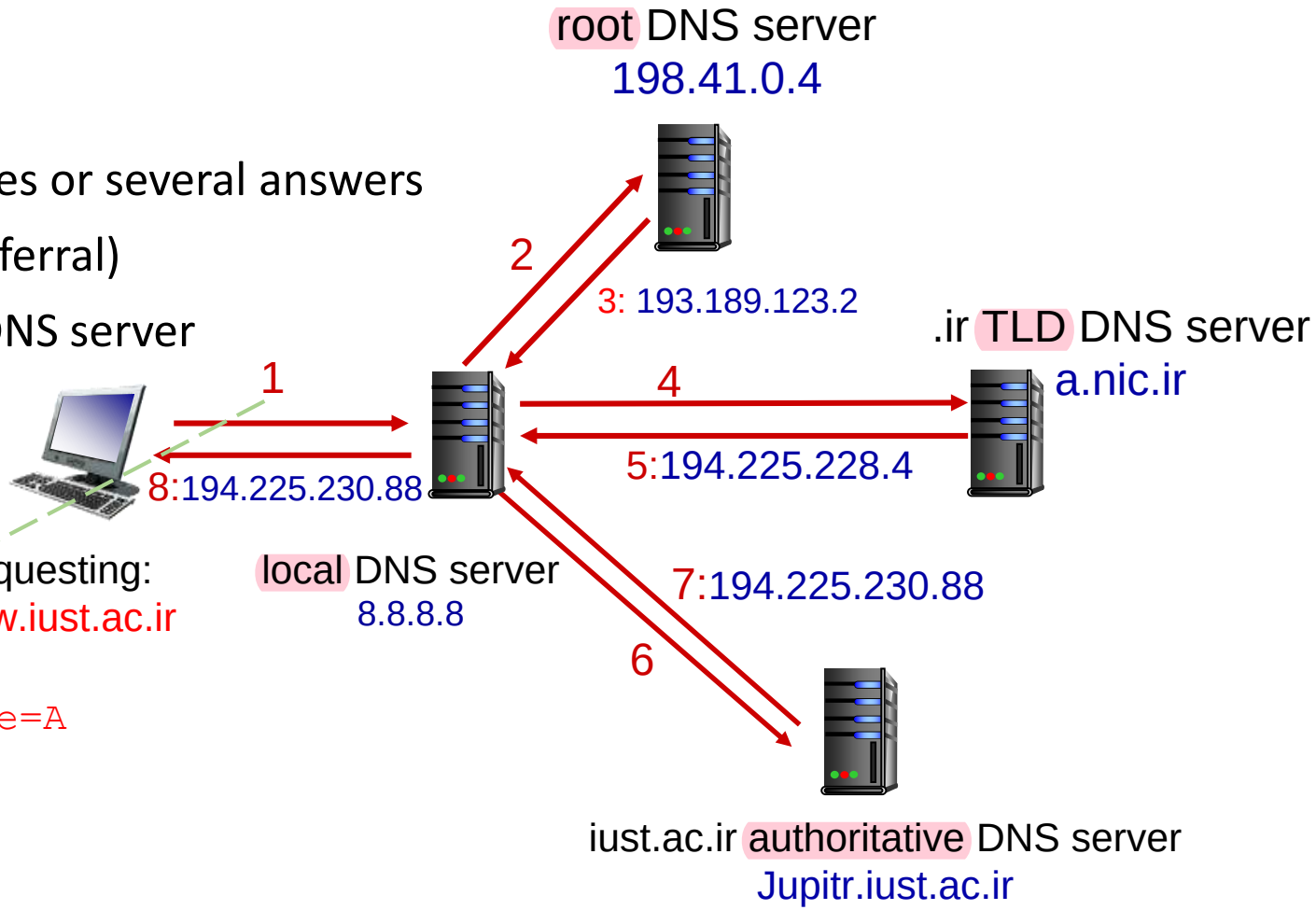
```
Server:  jupitr.iust.ac.ir
Address:  194.225.228.4

Name:    mxgate.iust.ac.ir
Address:  194.225.230.66
```

DNS Protocol

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

- DNS protocol uses UDP
- A massage may contain several queries or several answers
- Messages: Query, Reply(answer or referral)
- Query provides name and type to DNS server
- Reply returns value



Query:
name=www.iust.ac.ir, type=1

type=A

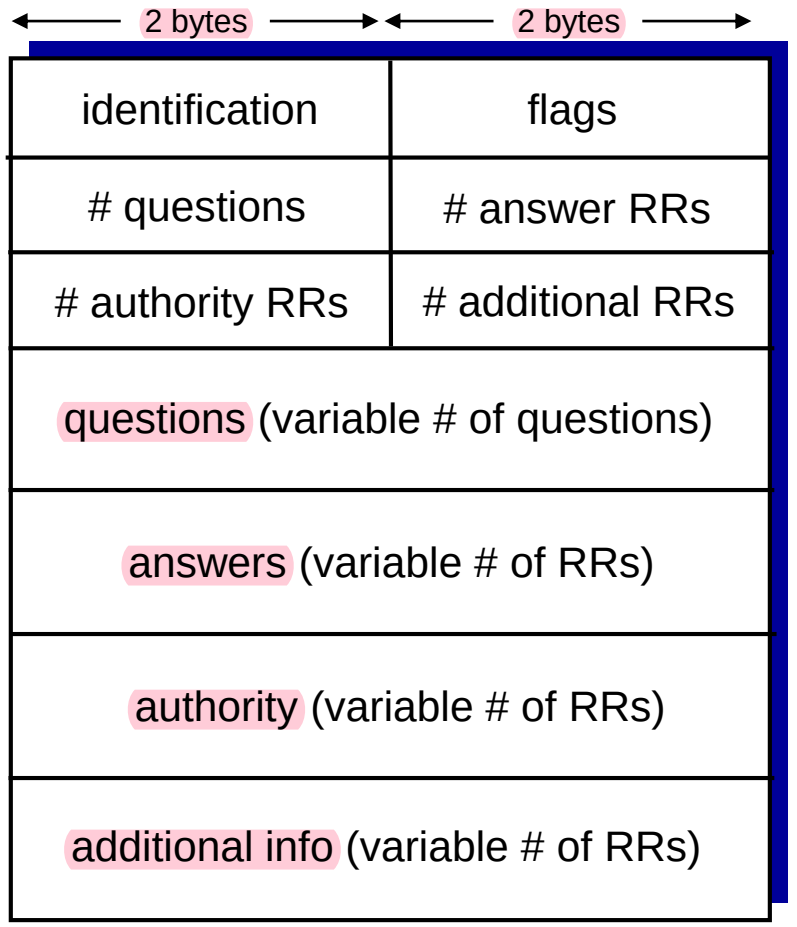
- Query: 1, 2, 4, 6
- Reply (referral): 3, 5
- Reply (answer): 8

DNS protocol messages

DNS query and reply messages, both have same format:

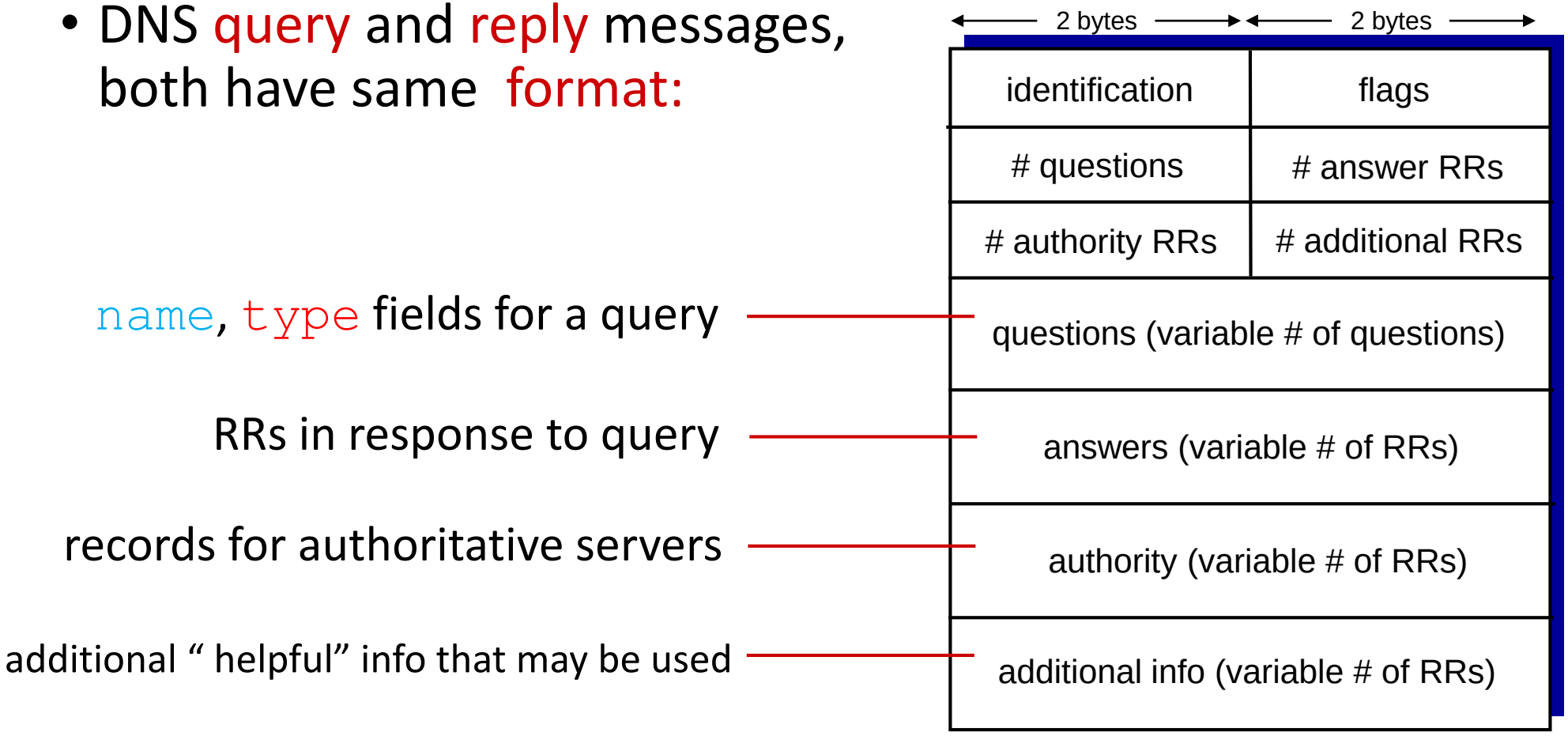
message header:

- identification: 16 bit number for query, reply to query uses same number
- flags:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



DNS protocol messages

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



Inserting records into DNS

Example: new startup "Network Utopia"

- Register name networkutopia.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.          86400  IN  NS  dns1.networkutopia.com
dns1.networkutopia.com.     86400  IN  A   212.212.212.1
networkutopia.com.          86400  IN  NS  dns2.networkutopia.com
dns2.networkutopia.com.     86400  IN  A   212.212.212.2
```

- Create primary authoritative server (dns1.networkutopia.com, 212.212.212.1)

```
www.networkuptopia.com      86400  IN  A    212.212.212.3
networkutopia.com           86400  IN  MX   mail1.networkutopia.com
```

- Create an authoritative server as secondary (212.212.212.2)

.....

DNS security

- DDoS attacks on Root
  - bombard root servers with ICMP messages
    - Internet Control Message Protocol, routers sends ICMP error messages to source IP address when network problems prevent delivery of IP packets
  - Unsuccessful attack: Root name servers filter ICMP messages
  - local DNS servers cache IPs of TLD servers, allowing root server bypass

- DDoS attacks on TLD
  - send flood of DNS queries to TLD, harder to filter DNS queries, so bombard TLD servers potentially more dangerous

DNS security

- Redirect attacks (redirect a user to attacker's Web site)
  - man-in-middle: attacker intercepts queries from hosts and returns false replies
- DNS poisoning
  - attacker sends false replies to a local DNS server, tricking local DNS server into accepting bogus records into its cache.

- DNS has demonstrated itself to be surprisingly robust against attacks

Contents

Peer-to-Peer (P2P) Applications

- Users-to-User direct communication
- User (peer) requests service from other users (peers)
- User (peer) provides service to other users (peers)
- Self scalability: new peers bring new service capacity, and new service demands
- P2P applications are more complex than client-server Apps
- Peers are intermittently connected and change IP addresses

- Some P2P Apps: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)

File sharing: Client-Server vs P2P

- How much time for distributing a file (size F) from one server to N peers?

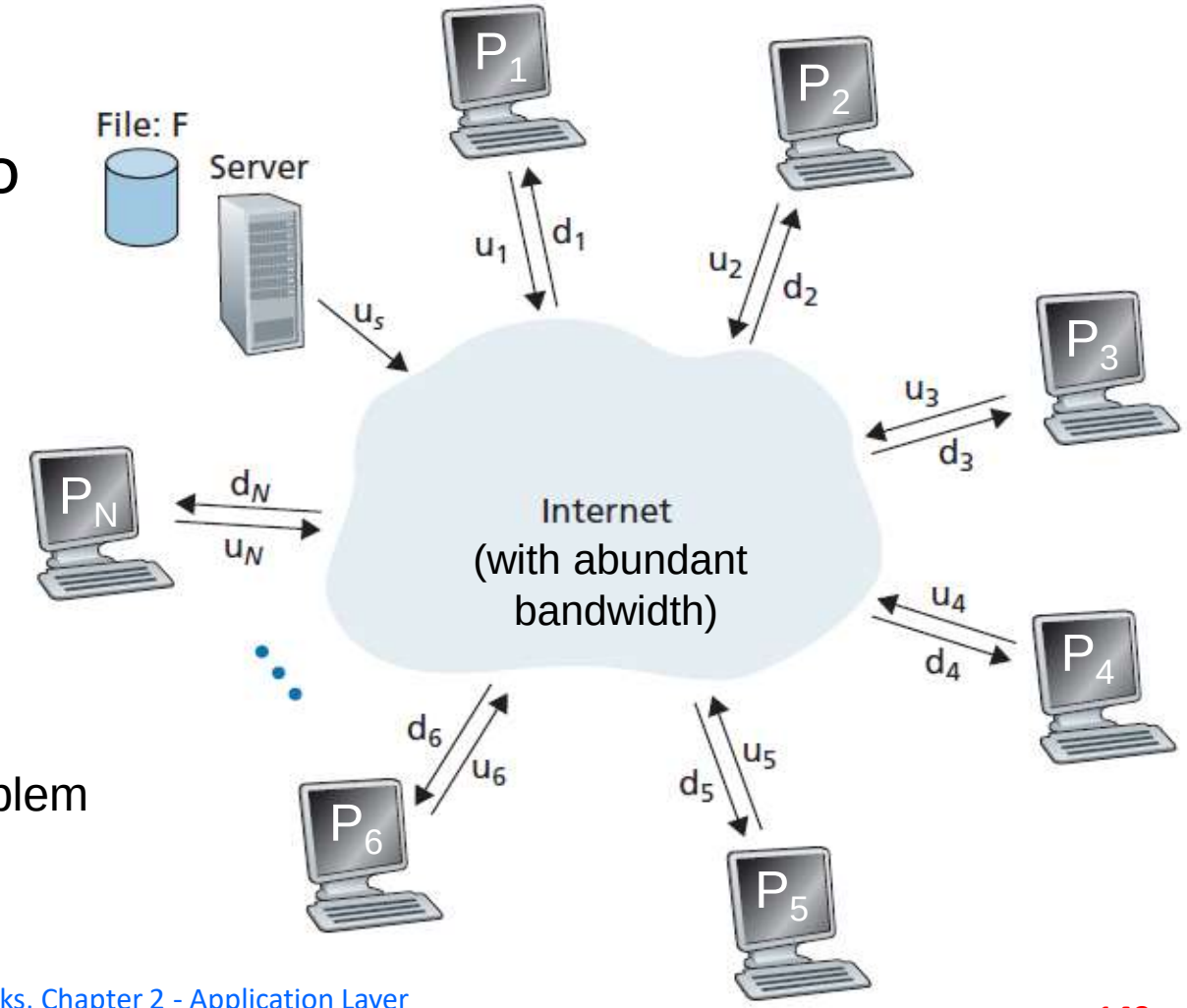


Figure 2.22 An illustrative file sharing problem

File sharing: Client-Server

- Server transmission: must sequentially send (upload) N file copies:
  - time to send one copy: F/us
  - time to send N copies: NF/us
- client: each client must download file
  - d_{min} = minimum client download rate (bps)
  - minimum client download time: F/d_{min}

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

$$D_{C-S} \geq max\left\{\frac{NF}{u_S}, \quad \frac{F}{d_{min}}\right\} \qquad (2.1)$$

total demand (increases linearly in N)

total upload capacity

File sharing : P2P

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

time to distribute F to N clients using P2P approach

$$D_{P2P} \geq max\left\{\frac{F}{u_S}, \frac{F}{d_{min}}, \frac{NF}{u_S + \sum_{i=1}^{N} u_i}\right\} \quad (2.2)$$

total demand (increases linearly in N)

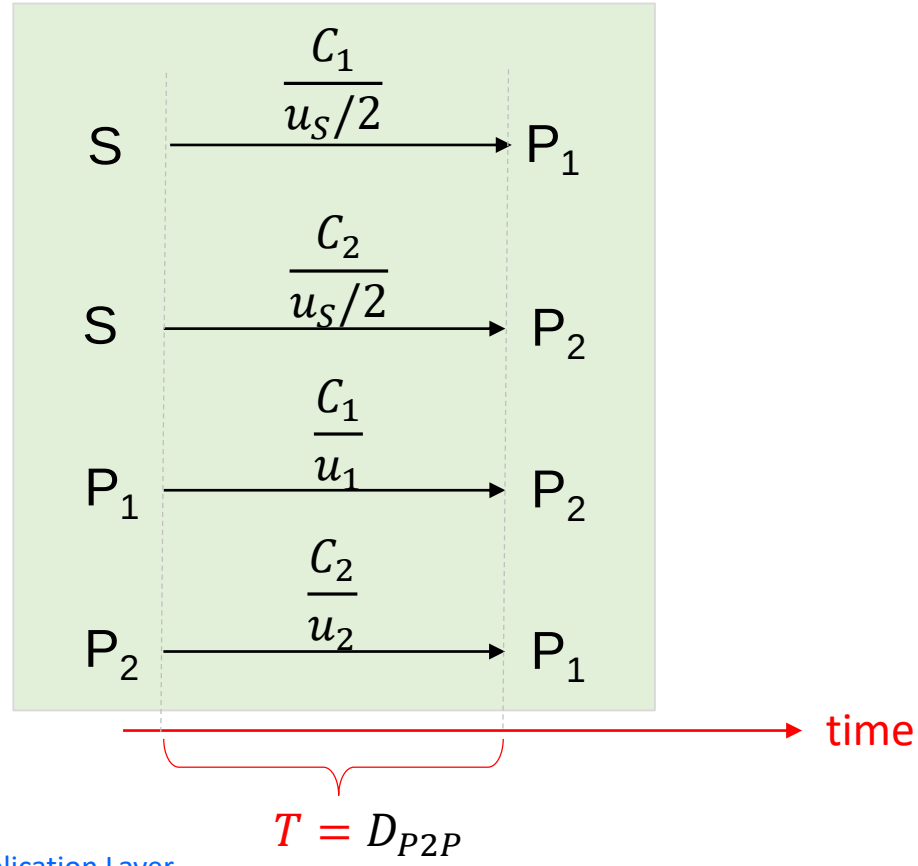
total upload capacity each peer brings service capacity

File distribution time: P2P Examples

total demand=$2F$
total upload capacity=$2u_s$

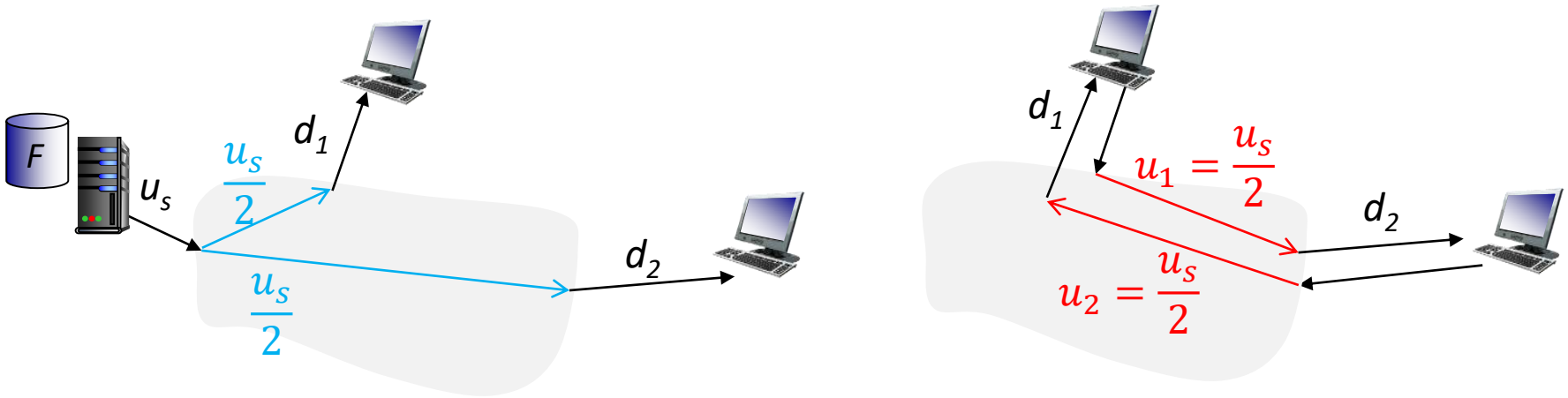
- A file chunked into $C_1, C_2, \ldots, C_N$, all in equal size
- Suppose: $u_1 = u_2 = \cdots = u_N$

- For N=2, $u_1 = u_2 = u_S/2$,
- $(u_1 + u_2 + u_s = 2u_s)$,
- $d_1 = d_2 = d$
- $T = \frac{F/2}{u_s/2} = \frac{F/2}{u_1} = \frac{F/2}{u_2} = \frac{F}{u_s}$
- $d \geq \frac{u_s}{2} + (2-1)\frac{u_s}{2} = u_s$



File distribution time: P2P Examples

$$D_{P2P} > max\left\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{NF}{u_s + u_1 + u_2}\right\} = \frac{F}{u_s}$$



$$d_1 \geq \frac{u_s}{2} + (2-1)\frac{u_s}{2} = u_s$$

$$d_2 \geq \frac{u_s}{2} + (2-1)\frac{u_s}{2} = u_s$$

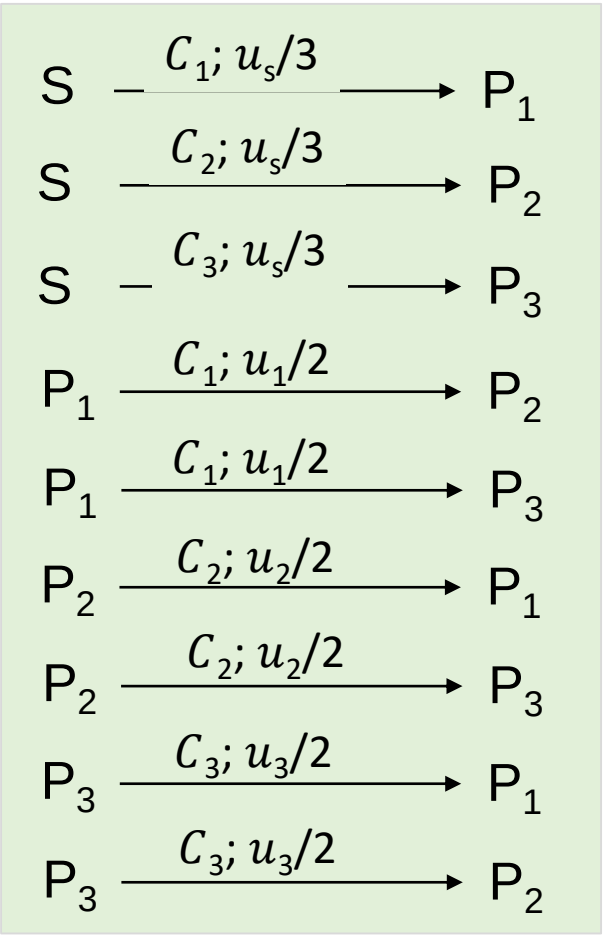
File distribution time: P2P Examples

total demand=$3F$
total upload capacity=$3u_s$

- A file chunked into $C_1, C_2, \ldots, C_N$, all in equal size
- Suppose: $u_1 = u_2 = \cdots = u_N$

- For N=3, $u_1 = u_2 = u_3 = 2u_S/3$
- $(u_1 + u_2 + u_3 + u_s = 3u_s), d_1 = d_2 = d_3 = d \geq u_S$

- $T = \frac{F/3}{u_s/3} = \frac{F/3}{u_1/2} = \frac{F/3}{u_2/2} = \frac{F}{u_s}$
- $d \geq \frac{u_s}{3} + (3-1)\frac{u_s}{3} = u_s$



File distribution time: P2P Examples

total demand=NF
total upload capacity=Nu_s

- A file chunked into $C_1, C_2, \ldots, C_N$, all in equal size
- Suppose: $u_1 = u_2 = \cdots = u_N$

- For N, $u_1 = u_2 = \cdots = u_N = \frac{(N-1)\times u_S}{N}$
- $(u_1 + u_2 + \cdots + u_N + u_s = Nu_s)$, $d_1 = d_2 = \cdots = d_N = d \geq u_S$

- $T = \frac{F/N}{u_s/N} = \frac{F/N}{u_1/(N-1)} = \frac{F/N}{u_2/(N-1)} + \cdots + \frac{F/N}{u_N/(N-1)} = \frac{F}{u_s}$
- $d \geq \frac{u_s}{N} + (N-1)\frac{u_s}{N} = u_s \Longrightarrow F/d \leq F/u_S$

Client-server vs. P2P: example

client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$

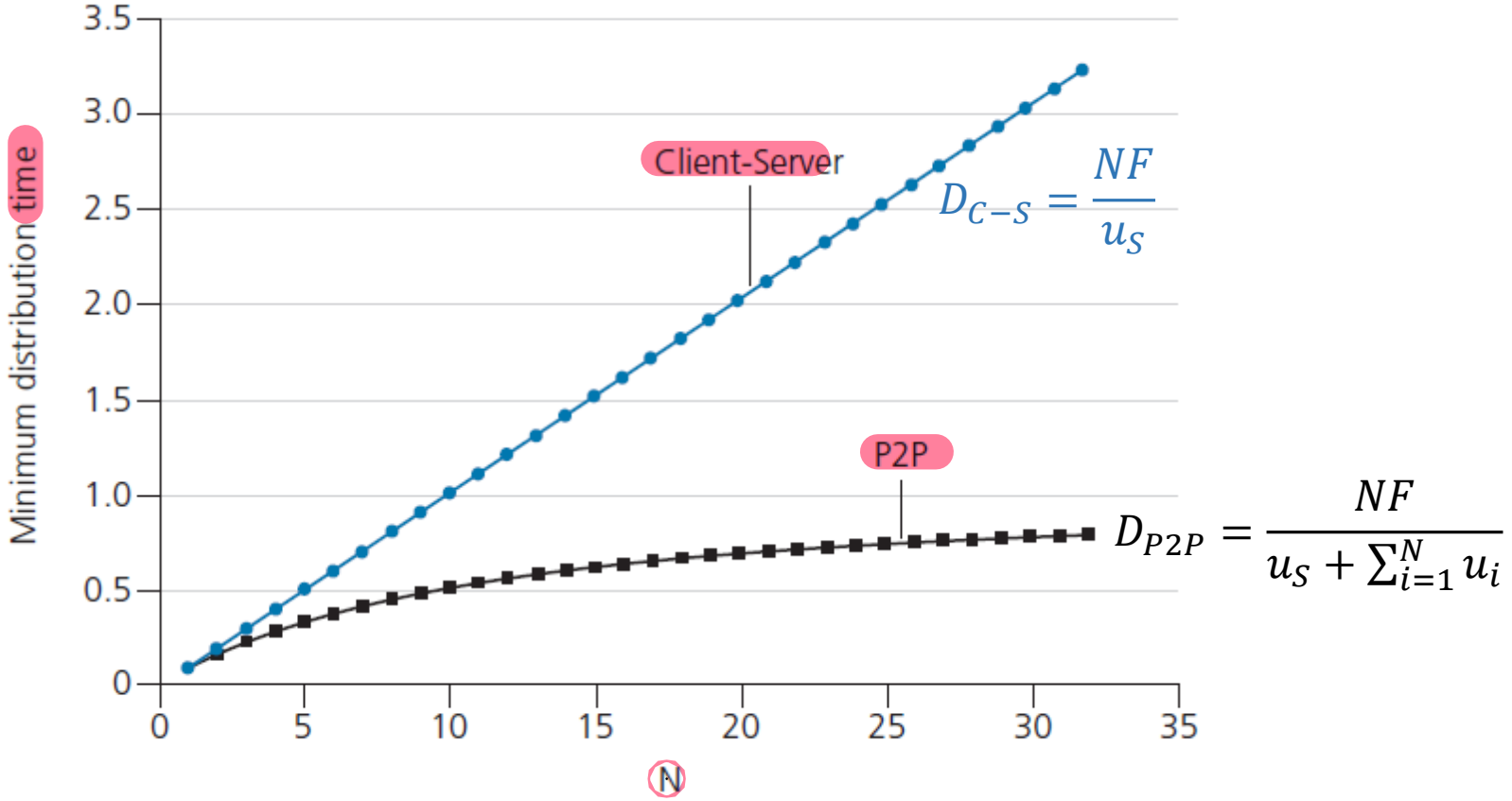
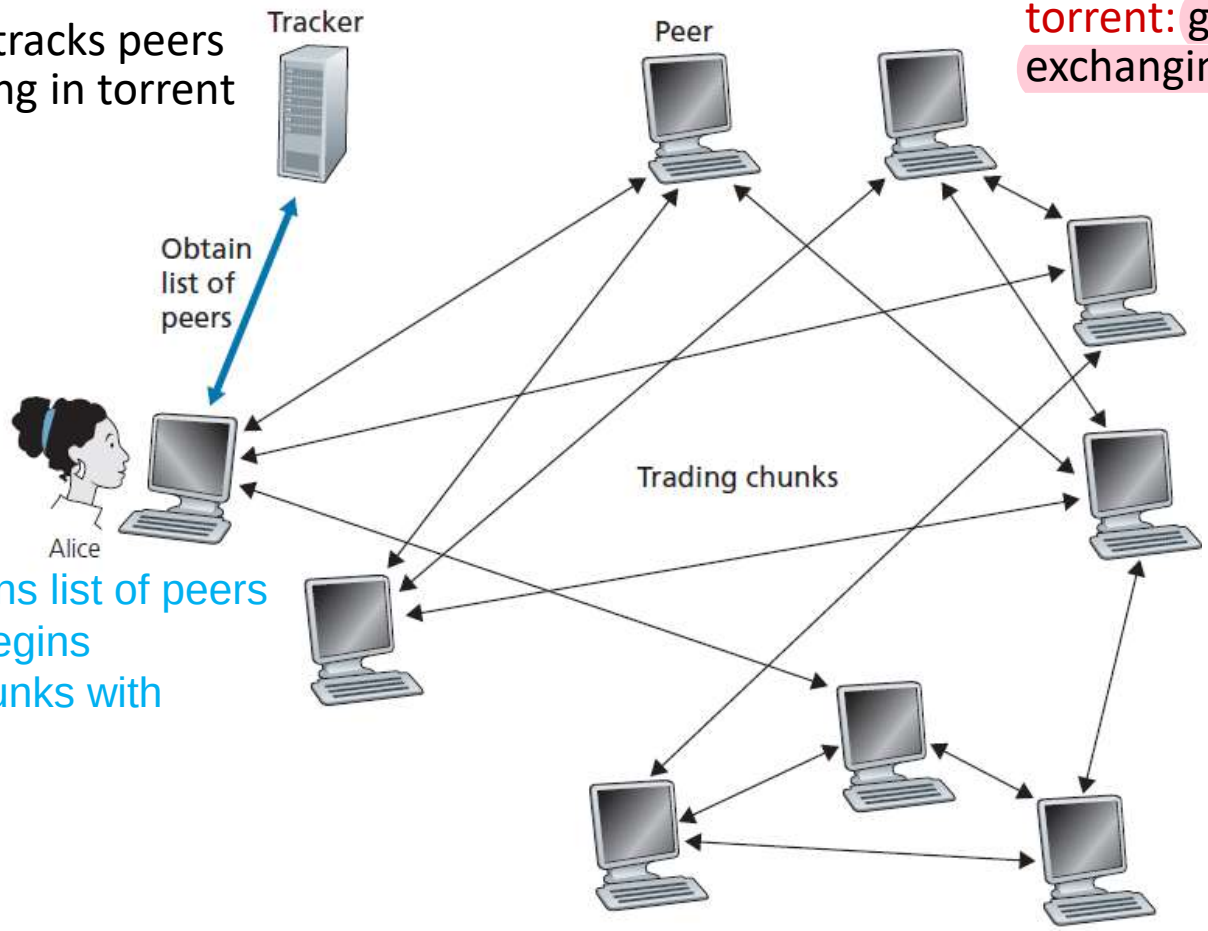


Figure 2.23 Distribution time for P2P and client-server architectures

BitTorrent

tracker: tracks peers participating in torrent



torrent: group of peers exchanging chunks of a file

file divided into 256 or 512 or 1024 Kbyte chunks, peers in torrent send/receive file chunks

Alice connects, 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, (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

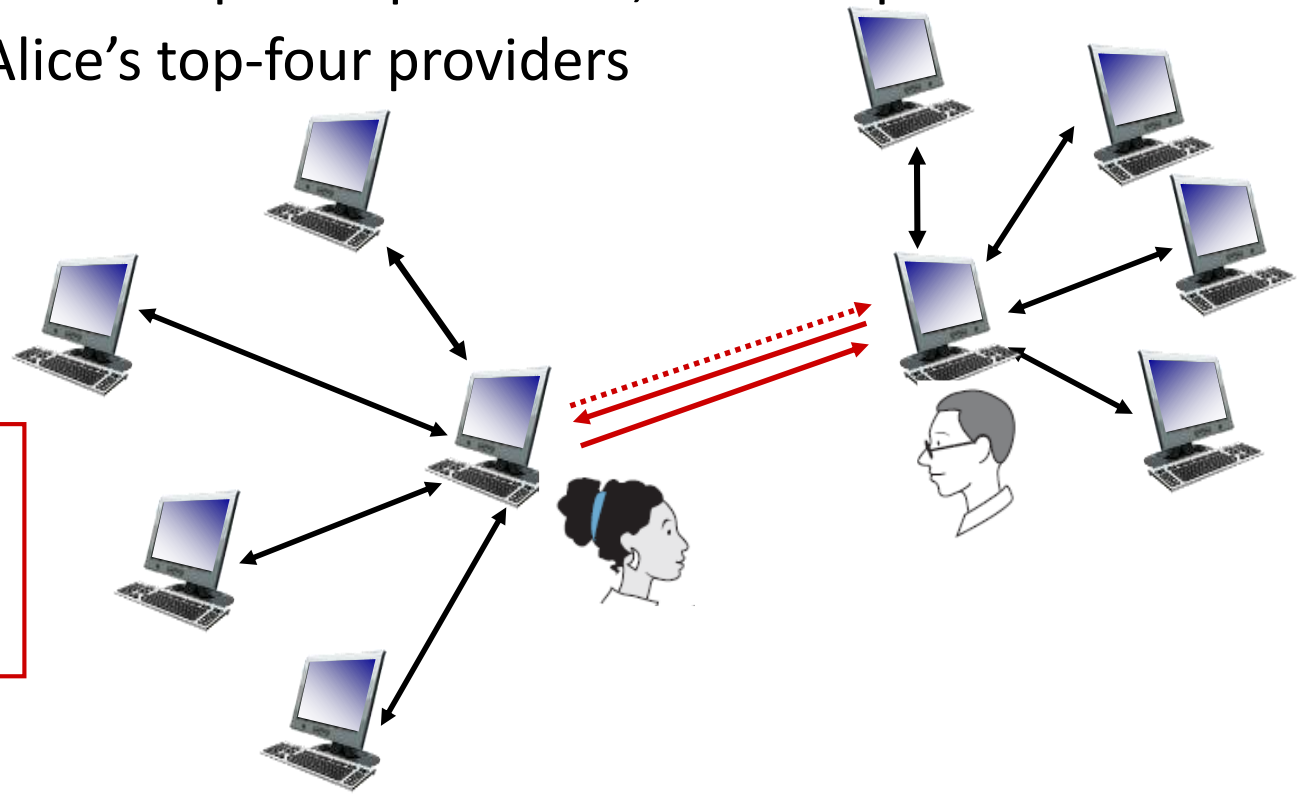
- Alice sends chunks to those 4 peers currently sending her chunks at highest rate
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer (5_{th}), starts sending chunks
  - "optimistically unchoke" this peer
  - 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



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

How to get first chunk?

- Alice joins BitTorrent without any chunks
- She cannot become a top-four uploader for any of other peers, since she has nothing to upload.
- **Q:** How then will Alice get her first chunk?

- **A:** Alice connects to Tracker and obtains list of peers' IP addresses from tracker, and establish concurrent TCP connections with all the peers on the list
- Peers that are successfully connected to Alice are called as neighboring peers of Alice
- One or more of neighbors of Alice select randomly Alice as optimistically unchoked peer

Contents

2.6.1 Internet Video

- Stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, PrimeVideo (Amazon): 80% of residential ISP traffic (2020)
- Challenge1: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
  - Solution: Dynamic, Adaptive Streaming over HTTP (Application level solution)
- Challenge2: scale
  - how to reach ~1Bilion users? single mega-video server won't work (why?)
  - Solution: Content Distribution Network (Infrastructural solution)

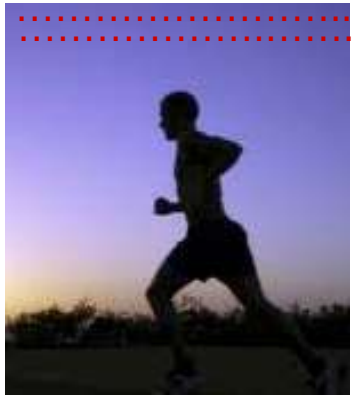
Multimedia: video

- Video: sequence of images displayed at constant rate
  - e.g., 24 frames/sec, 30 frames/sec
- Digital frame (image): array of pixels
  - HD frame: Horizontal pixels=1920, Vertical Pixels 1080
  - each pixel represented by 3 Bytes
  - 3Byts/pixel:
    HD=1920*1080*3=6,220,800Bytes/frame≈ 6MBytes/frame
- Raw HD video
  - 24 frame/sec, 3Byts/pixel = 24*6,220,800 =
    149,299,200Bytes/sec = 1,194,393,600 bits/sec ≈1.2Gbps

Multimedia: video coding

- **Coding:** use redundancy within and between images to decrease number of bits used to encode image
  - Spatial (within image)
  - Temporal (from one image to next)

spatial coding example: instead of sending N values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)



frame i

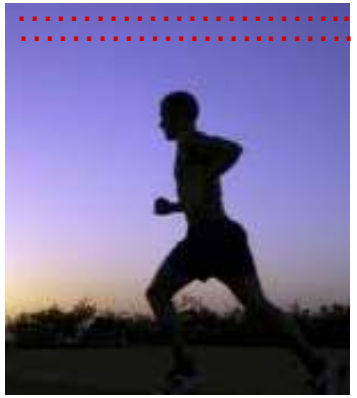
temporal coding example: instead of sending complete frame at $i+1$, send only differences from frame i

frame $i+1$

Multimedia: video encoding

- CBR: (constant bit rate): video encoding rate fixed
- VBR: (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- examples:
  - MPEG1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, 64Kbps – 12 Mbps)

spatial coding example: instead of sending N values of same color (all purple), send only two values: color value (*purple) and number of repeated values* (N)

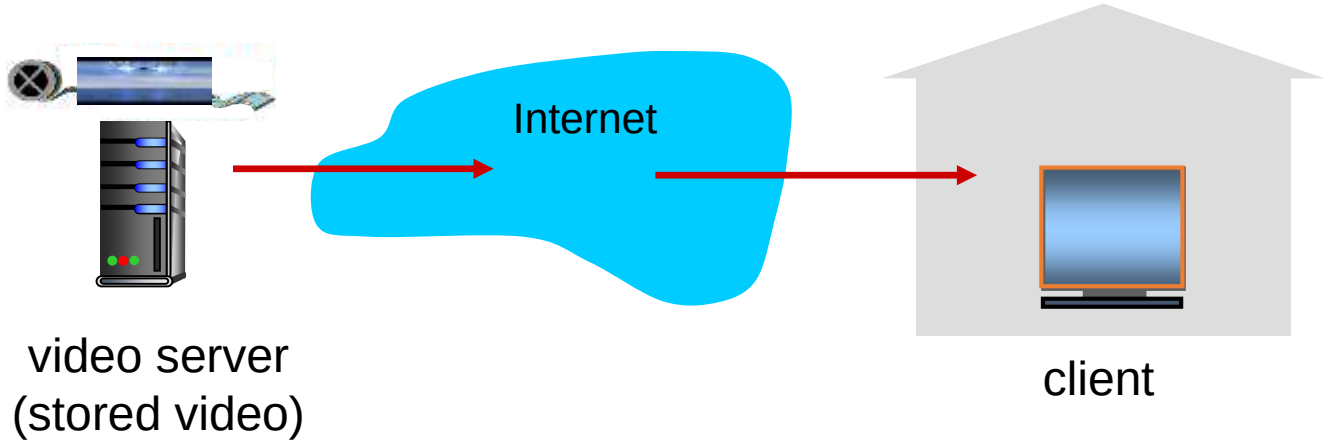


frame i

frame $i + 1$

temporal coding example: instead of sending complete frame at $i + 1$, send only differences from frame i

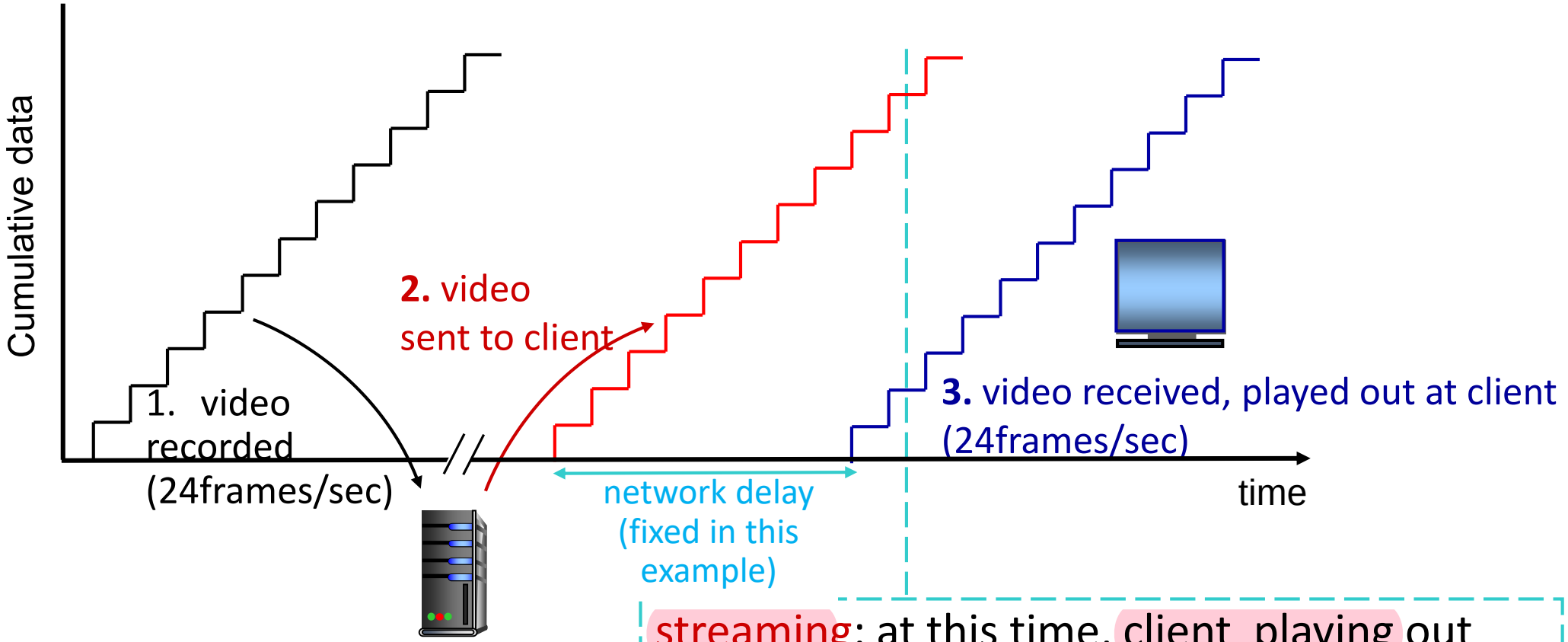
Streaming stored video: simple scenario



Main challenges:

- server-to-client available network bandwidth (network throughput) will vary over time, with changing network congestion levels (in house, in access network, in network core, at video server)
- packet loss and delay due to congestion will delay playout, or result in poor video quality

Streaming stored video

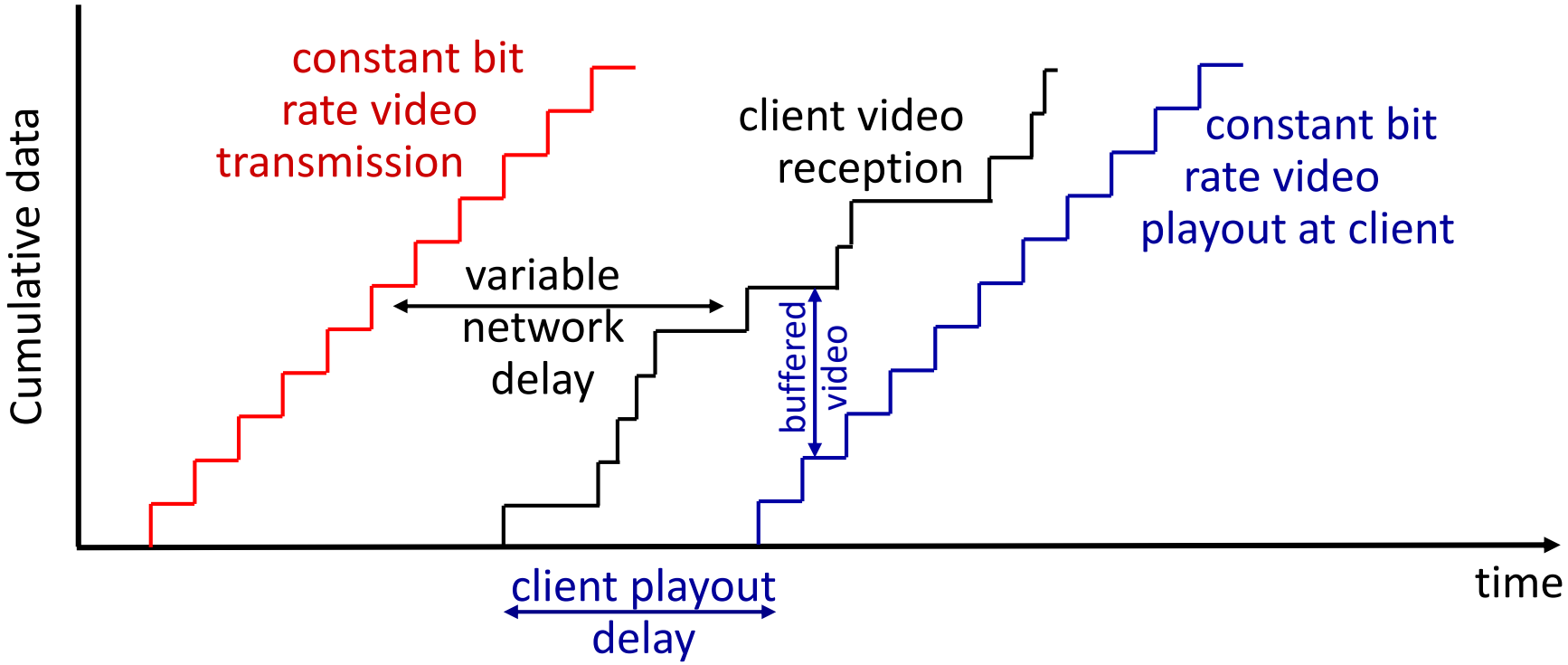


streaming: at this time, client playing out early part of video, while server still sending later part of video

Streaming stored video: challenges

- Continuous playout constraint: once client playout begins, playback must match original timing
  - … but network delays are variable (jitter), so will need **client-side buffer** to match playout requirements
- Other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted

Streaming stored video: playout buffering



client-side buffering and playout delay: compensate for network-added delay, delay jitter

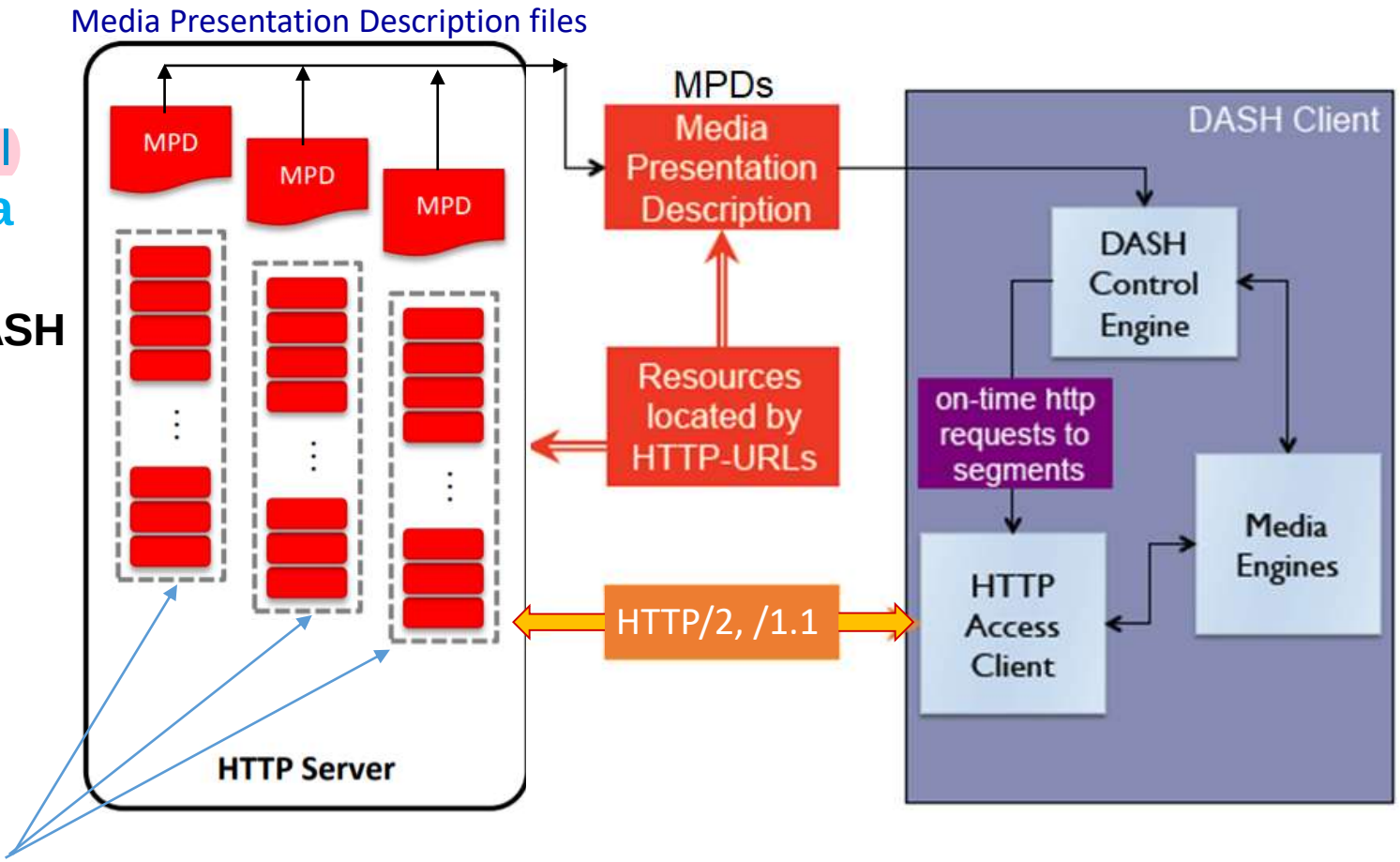
2.6.2 HTTP Streaming and DASH

DASH: a HTTP-based protocol designed for **streaming media**

DASH: also known as **MPEG-DASH**

manifest file:

- provides URLs for different chunks (playlist file for video chunks)
- MPDU (Media Presentation Description): manifest files



A Video segmented to several chunks each chunk encoded with different bit rates

DASH server/client

- Server:
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - manifest file: provides URLs for different chunks
- Client:
  - periodically measures server-to-client bandwidth (network throughput)
  - consulting manifest, requests one chunk at a time
    - chooses maximum coding rate sustainable given current network throughput
    - can choose different coding rates at different points in time (depending on network throughput at time)

DASH server

- Server:
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - manifest file=Media Presentation Description (MPD) :
    - Provides URLs for different chunks
    - Byte range for each accessible chunk
    - Chunk availability start and end time
    - Instructions on starting play out (for live service)
    - …

DASH - Chunks

- Chunks with variable durations
  - Duration of chunks can be varied
  - With live streaming, duration of next segment can also be signaled with delivery of current segment

- A video available at different servers at different rates
  - client can stream from any server to maximize available network bandwidth

Dash client

- Client:
  - Periodically measures server-to-client bandwidth
  - Consulting manifest, requests one chunk at a time
    - chooses maximum coding rate sustainable given current bandwidth
    - can choose different coding rates at different points in time (depending on available bandwidth at time)

DASH client

- "intelligence" at client: client determines
  - when to request chunk (so that buffer starvation, or overflow does not occur)
  - what encoding rate to request (higher quality when more bandwidth available)

- where to request chunk (can request from URL server that is "close" to client or has high available bandwidth)

Streaming video = encoding + DASH + playout buffering

http header for DASH

- Rang HTTP Request Header line: example

  Range: bytes=200-1000, 2000-6576, 19000-

- Range indicates part of a video that server should return
- Server may send back all requested ranges in a multipart document
- If server sends back ranges, it uses the 206 Partial Content for response
- If ranges are invalid, server returns the 416 Range Not Satisfiable error

Other HTTP-based adaptive bit rate streaming

- Beside MPEG-DASH, there are other HTTP-based adaptive bit rate streaming technologies
- Apple Inc. has **HTTP Live Streaming (HLS)**
- Microsoft has **Microsoft Smooth Streaming (MSS)**
- **…**

2.6.3 Content Distribution Networks

- Challenge: how to stream content (selected from millions of videos) to hundreds of thousands of simultaneous users?

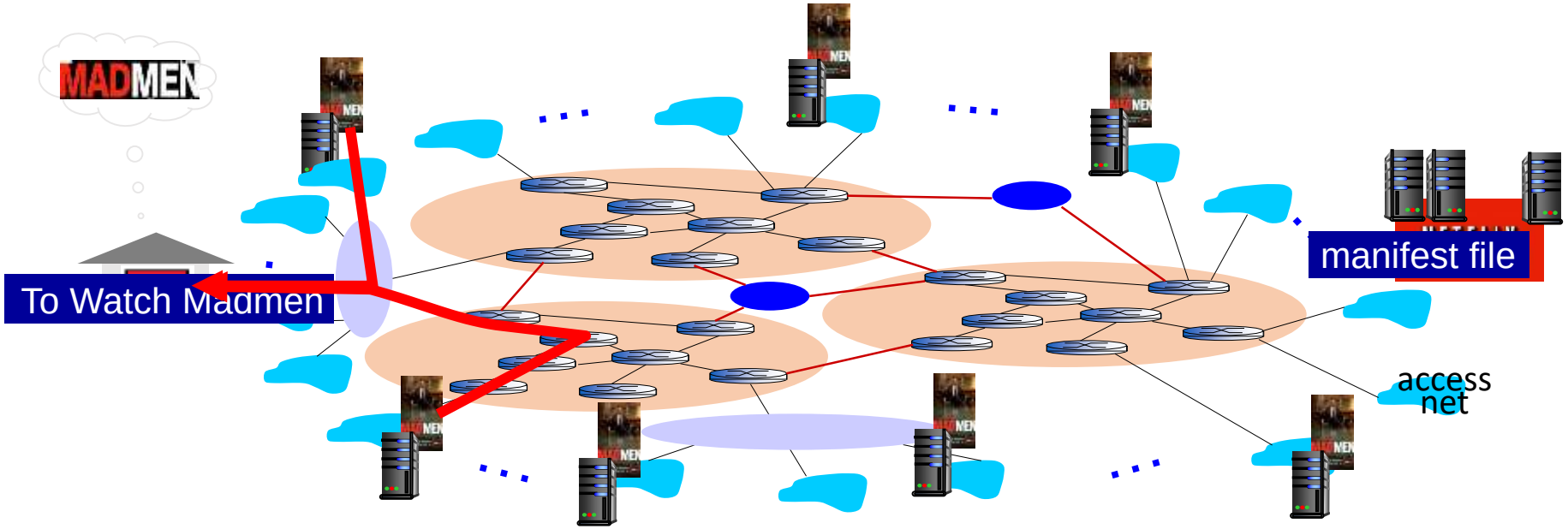
- Option 1: single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of a video sent over outgoing link
- mega-server doesn't scale

Content distribution networks (CDNs)

- Option 2: store/serve multiple copies of videos at multiple geographically distributed sites (CDN)
  - enter deep: push CDN servers deep into many access networks
    - close to users
    - Akamai: 300,000 servers deployed in more than 130 countries (2020)
  - bring home: smaller number (10's) of larger clusters in POPs near (but not within) access networks
    - used by Limelight

Content distribution networks (CDNs)

- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested



Example 1: CDN content access

Retrieving a video from NetCinema Company using URL:

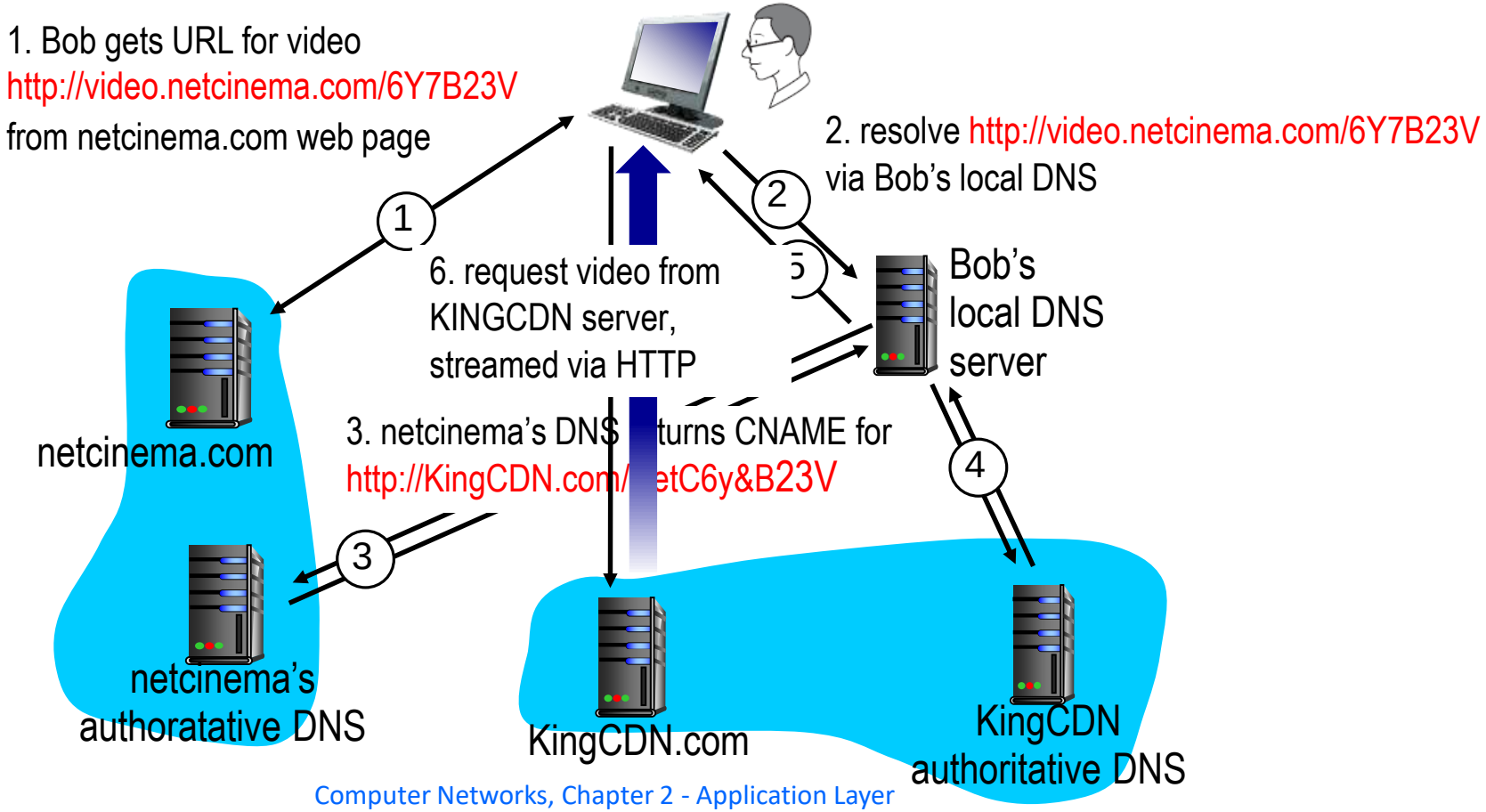
http://video.netcinema.com/6Y7B23V

- Client's Local DNS Server (LDNS) sends query to NetCinema's authoritative DNS server (ADNS) for: video.netcinema.com/6Y7B23V
- NetCinema's ADNS observes string "video" in hostname video.netcinema.com
- NetCinema's ADNS returns a hostname in KingCDN's domain, for example, a1105.kingcdn.com to LDNS
- LDNS sends a query to King's ADNS to resolve a1105.kingcdn.com
- King's ADNS returns IP addresses of a KingCDN content server to LDNS

Example 1: CDN content access

Bob (client) requests video http://video.netcinema.com/6Y7B23V

- video stored in CDN at http://KingCDN.com/NetC6y&B23V



Example 1: CDN content access

- If DASH is used, a1105.kingcdn.com sends to client a manifest file with a list of URLs, one for each version of video, and client will dynamically select chunks from different versions

- Most CDNs take advantage of DNS to intercept and redirect requests

Cluster Selection Strategies

- At core of any CDN deployment is a **cluster selection strategy**, a mechanism for dynamically directing clients to a server cluster or a data center within CDN
- CDN learns IP address of client's local DNS server via client's DNS lookup
- After learning this IP address, CDN needs to select an appropriate cluster based on this IP address
- Note: Some users are configured to use remotely located LDNSs, LDNS location may be far from the client's location

1- Geographically closest strategy

- Simple strategy: **geographically closest**
- Using commercial geo-location databases, each LDNS IP address is mapped to a geographic location. When a DNS request is received from a particular LDNS, CDN chooses geographically closest cluster
- This simple strategy ignores variation in delay and available bandwidth over time of Internet paths, always assigning same cluster to a particular client
- Geographically closest cluster may not be closest cluster in terms of length or number of hops of network path

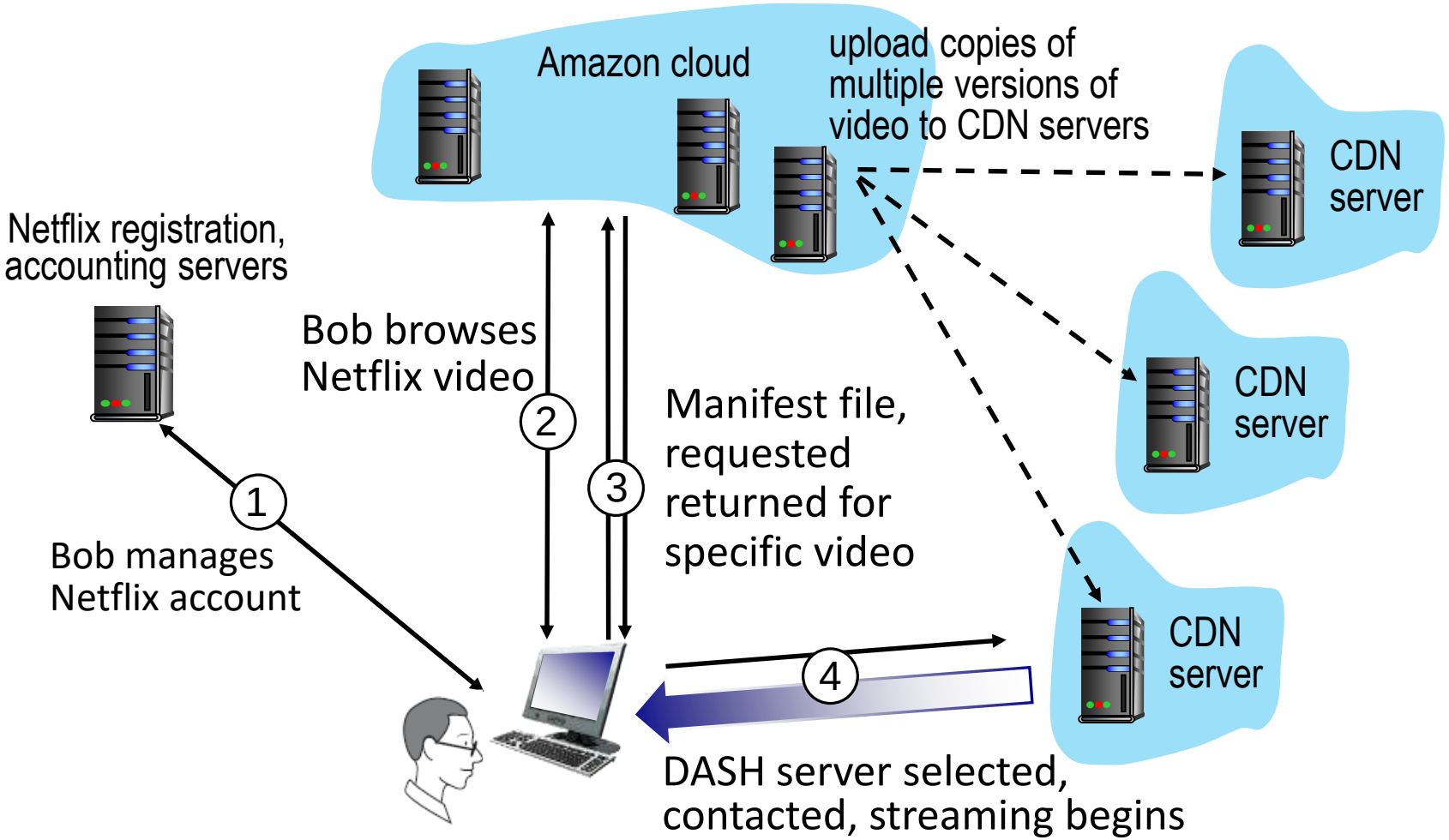
2- Real-time measurements strategy

- CDNs performs periodic **real-time measurements** of delay and loss between their clusters and clients
- For instance, a CDN can have each of its clusters periodically send probes (for example, ping messages or DNS queries) to all of LDNSs around world
- One drawback of this approach is that many LDNSs are configured to not respond to such probes

Case study: Netflix

- Netflix video distribution has two major components:
- 1- Amazon cloud
  - Netflix Web site is on Amazon servers in Amazon cloud
  - Netflix receives studio master versions of movies and uploads them to hosts in Amazon cloud
  - Hosts in Amazon cloud create different formats for each movie and manifest files. Clients receives manifest files form Amazon hosts
  - All created versions are uploaded (push caching) into Netflix's CDN
- 2- Netflix CDN infrastructure
  - Netflix streams all of its videos from its CDN
  - Netflix has streaming DASH servers in IXPs and within residential ISPs

2.6.4 Case Studies: Netflix and YouTube



Case Studies: YouTube

- Similar to Netflix, Google uses its own private CDN to distribute YouTube videos
- Unlike Netflix, Google uses pull caching, as described in Section 2.2.5, and DNS redirect, as described in Section 2.6.3
- Google's cluster-selection strategy directs client to cluster for which RTT between client and cluster is lowest
- YouTube employs HTTP streaming, often making a small number of different versions available for a video, each with a different bit rate and corresponding quality level
- YouTube does not employ adaptive streaming (such as DASH), but instead requires user to manually select a version

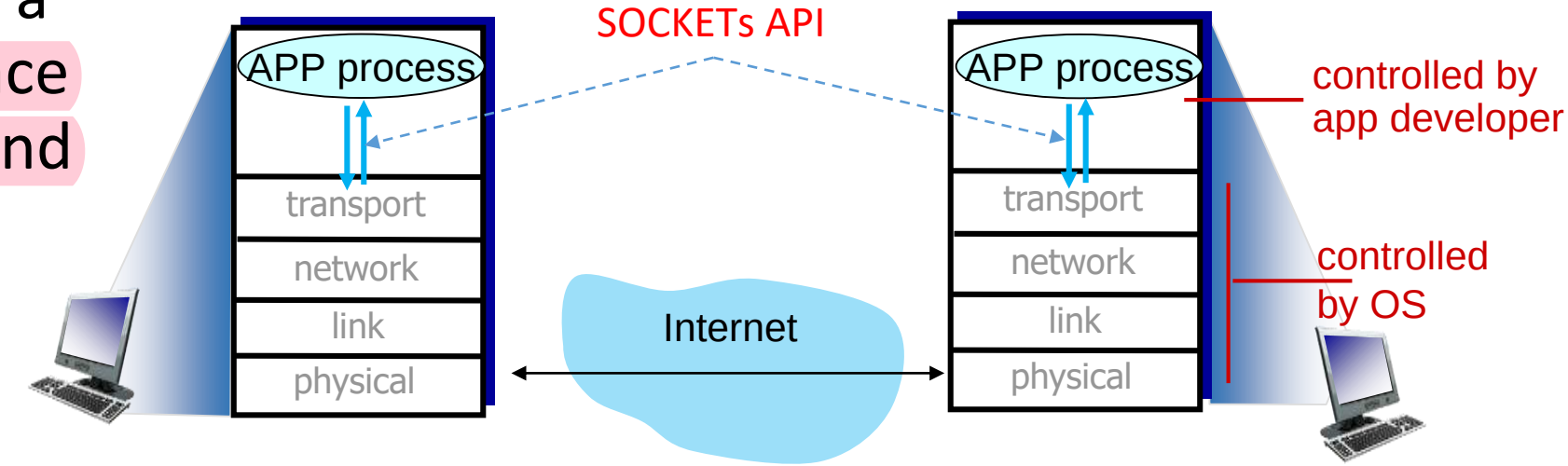
CDNs and DASH

- Akamai CDN supports DASH
- Amazon Web Services Elastic Transcoder has support for MPEG-DASH
- Amazon CloudFront CDN supports DASH
- Kollective Technology Inc. ECDN supports DASH
- Level 3 Communications CDN supports DASH
- Limelight Networks CDN supports DASH
- Tata Communications CDN supports DASH

Contents

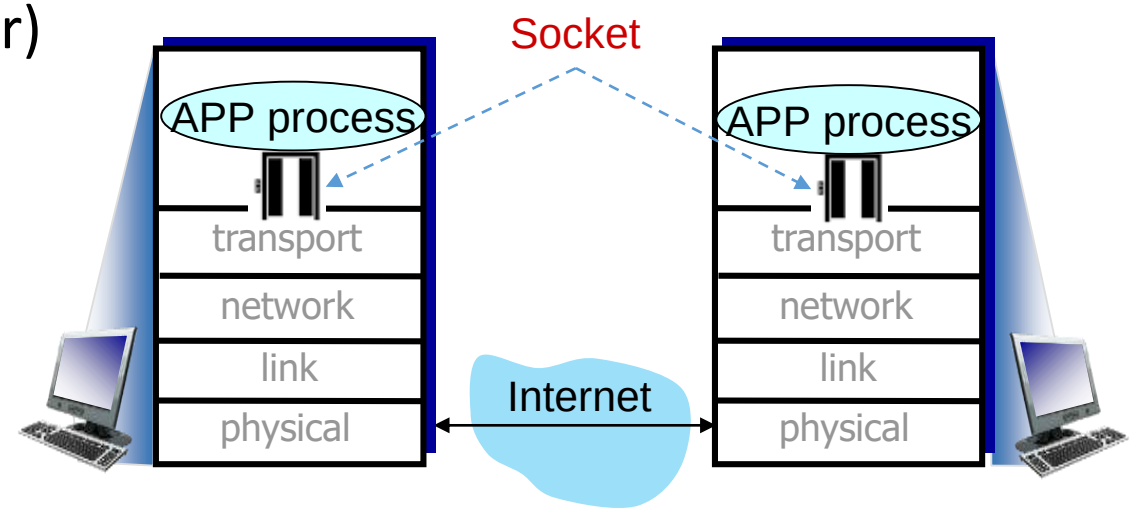
2.7 Socket Programming: Creating Network Applications

- Goal: how to build client/server applications that communicate using SOCKETs API
- SOCKETs API is a **collection of network system calls** that enable APPs to send and receive messages (to communicate)
- **System calls** provide a programming interface between a process and Operating System



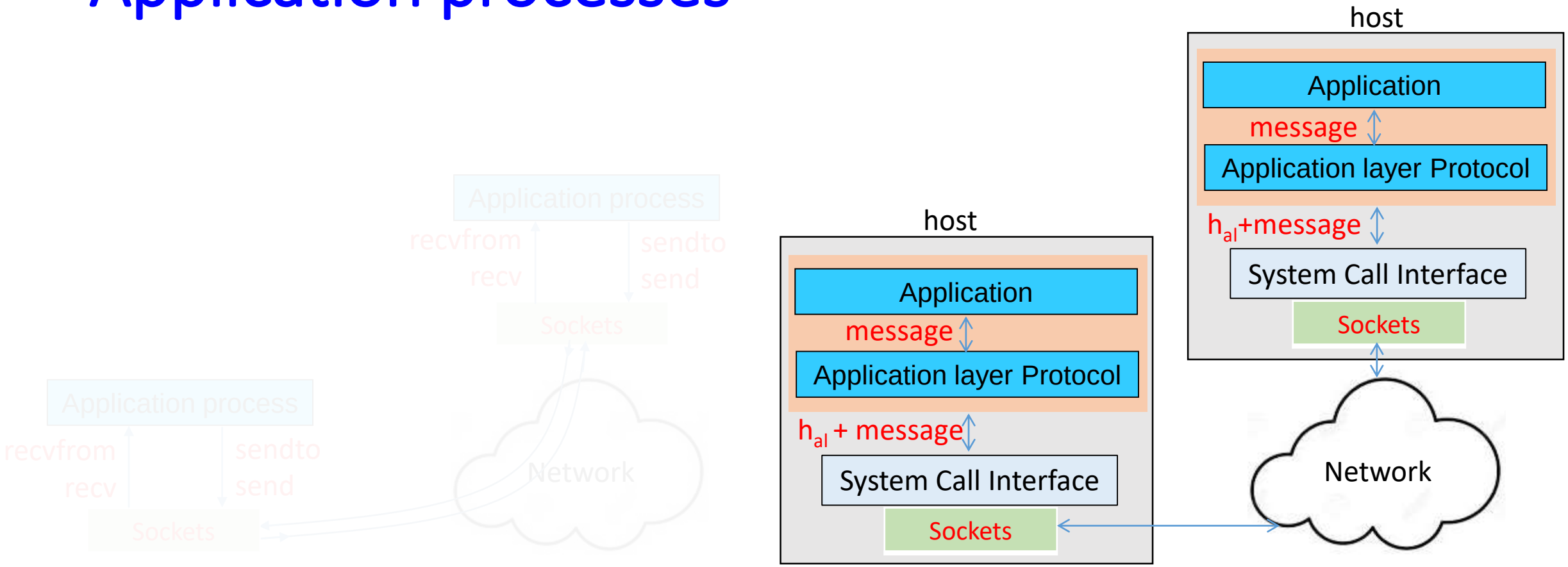
Network System Calls: SOCKETs API

- Socket( ): Create an endpoint (imaginary door) for communication
- Bind( ): Bind a IP and a port to a socket
- Listen( ): Listen for connections on a socket
- Accept( ): Accept a connection on a socket
- Connect( ): Initiate a connection on a socket
- Recvfrom( ): Receive a message from a socket
- Sendto( ): Send a message on a socket
- Close( ): Shut down part of a full-duplex connection
- …



Socket: an imaginary door between application process and end-end-transport protocol

A bidirectional logical channel between Application processes



Socket programming

Two socket types for two transport services:

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

Application Example:

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

2.7.1 Socket Programming with UDP

UDP: no “connection” between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port number to each packet
- receiver extracts sender IP address and port number 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 processes

Client/Server socket interaction: UDP

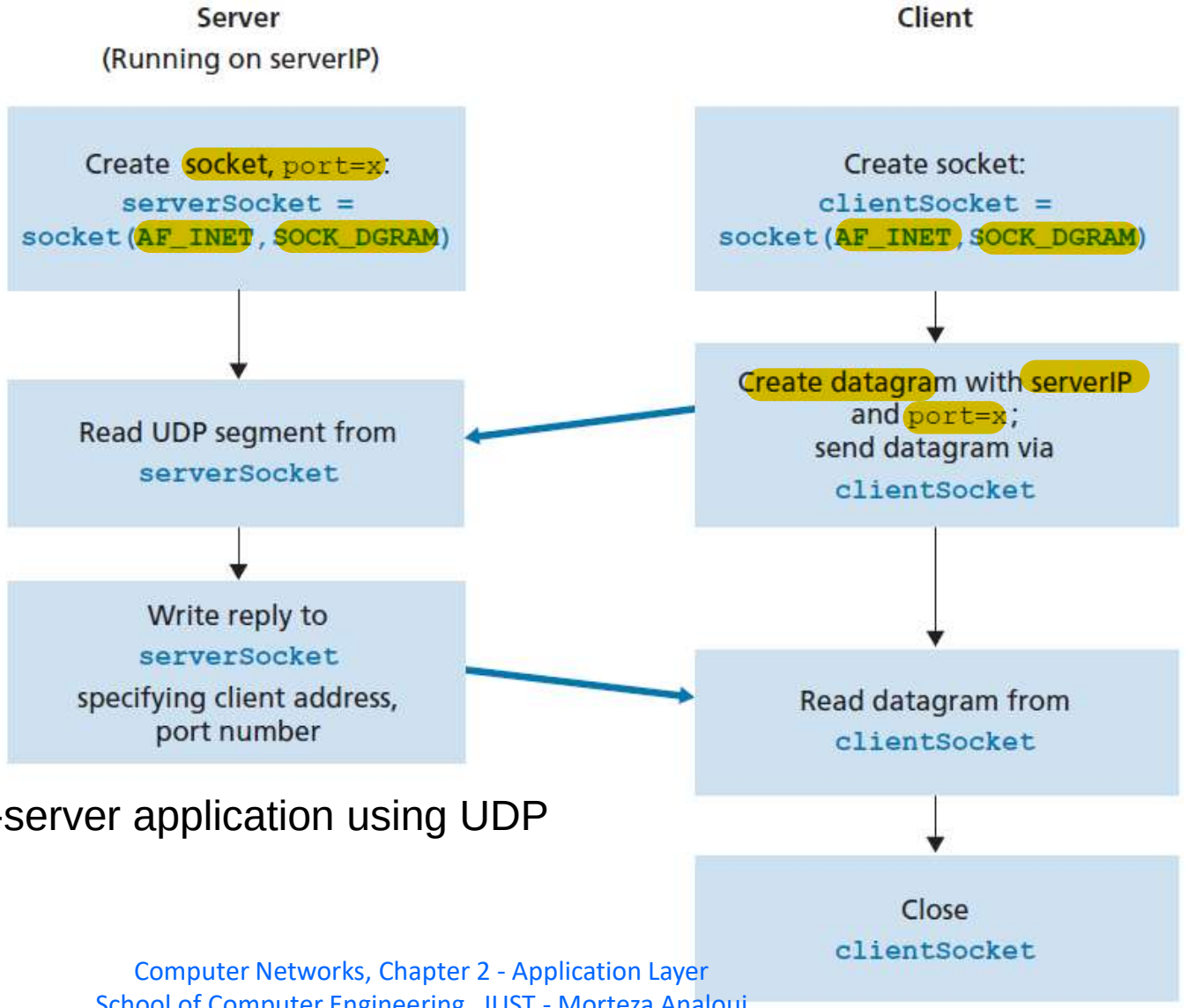


Figure 2.27 The client-server application using UDP

Example app: UDPClient.py

include Python's socket library
socket module forms basis of all network communications in Python. By including this line, we will be able to create sockets within our program.

create UDP socket , called clientSocket

input() is a built-in function in Python. When this command is executed, user is prompted with words "Input lowercase sentence:" User then uses keyboard to input a line, which is put into variable message

either IP address of server ("128.138.32.126")
or hostname of server ("cmp.iust.ac.ir")

Address Family: IP4 (AF_INET6: IP6

UDP

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

Example app: UDPClient.py

First convert message from string type to byte type, as we need to send bytes into a socket; this is done with encode() method. Method sendto() attaches destination address (serverName, serverPort) to message and sends resulting packet into clientSocket. Source address is also attached to packet, this is done automatically rather than explicitly by code.
Client waits to receive data from server

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

Receiving Buffer (Bytes)

Example app: UDPClient.py

When a packet arrives at client's socket, recvfrom method puts packet's data into variable modifiedMessage and packet's source address into variable serverAddress. serverAddress contains server's IP address and port number. UDPClient doesn't need this server address, since it already knows server address; but this line of Python provides server address nevertheless. Method recvfrom takes buffer size (here 2048) as input

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

Example app: UDPClient.py

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

Prints out modifiedMessage on user's display, after converting message from bytes to string

Close method closes socket, process then terminates

Example app: UDPServer.py

create UDP socket, called serverSocket

bind socket to local port number 12000

loop forever

Read from UDP socket into message, getting client's address (client IP and port)

send upper case string back to this client

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((' ', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

2.7.2 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
- Client **creates socket**, then establishes 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)

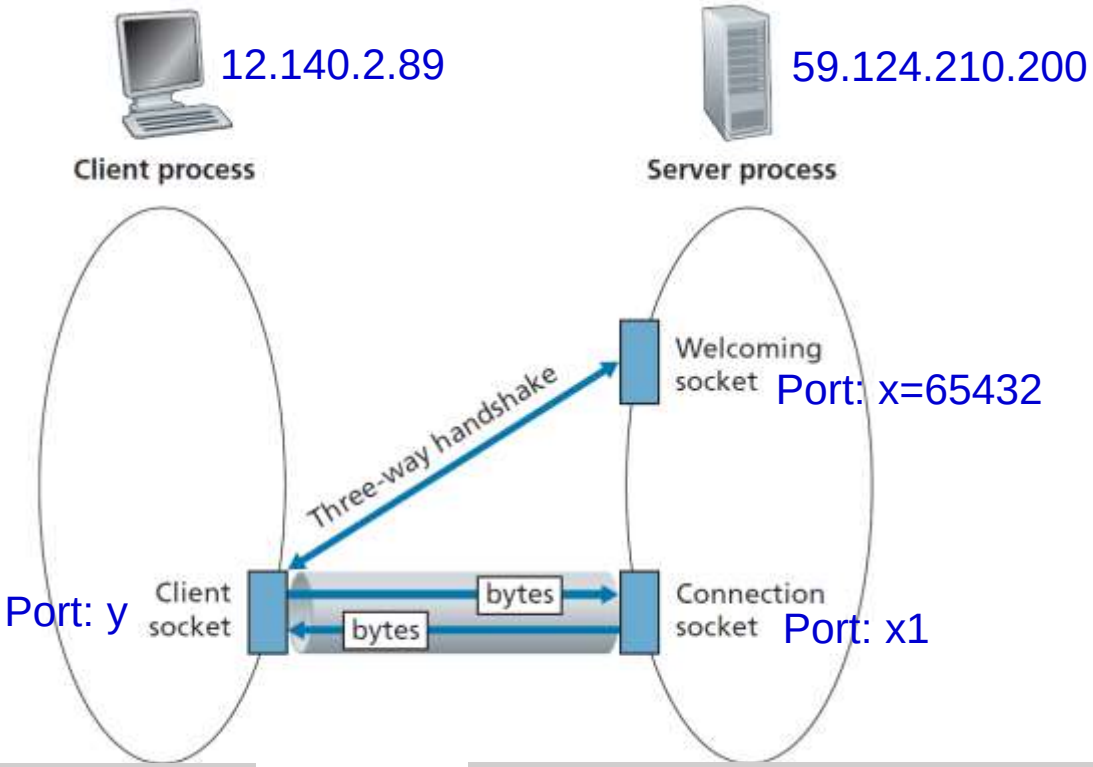
Application viewpoint

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

Sockets in TCP

- Server process creates a Welcoming socket (65432)
- Clients use (59.124.210.200; 65432) to connect to Server process
- Server process creates a private socket (Connection socket) for Client process
- Server and Client processes send and receive data using Connection socket

Welcoming socket address:
59.124.210.200; 65432



Client socket Address:
(12.140.2.89; x)

Connection socket Address:
(59.124.210.200; y, 12.140.2.89; x)

Can we use x1=65432?

TCP connection setup (is a 3-way handshake)

Client socket Address:
(12.140.2.89; y)

Client process

Welcoming socket address:
59.124.210.200; 65432

Server process

Connection request packet:
Src IP:12.140.2.89, Src PORT:y
Des IP:59.124.210.200, Des PORT:65432

Connection acceptance packet:
Src IP:59.124.210.200, Src PORT:x1
Des IP:12.140.2.89, Des PORT:y

Server creates a connection socket for this connection and send connection acceptance using this socket.

Connection Acknowledge packet:
Src IP:12.140.2.89, Src PORT:y
Des IP:59.124.210.200, Des PORT:x1

Server and Client processes send and receive data using x1 and y sockets

TCP: From application's perspective

- Client socket and Connection socket are directly connected by a virtual pipe
- Client process can send arbitrary bytes into Client Socket, and TCP guarantees that server process will receive (through Connection Socket) each byte in order sent
- Server process can send arbitrary bytes into Connection Socket, and TCP guarantees that Client process will receive (through Client Socket) each byte in order sent

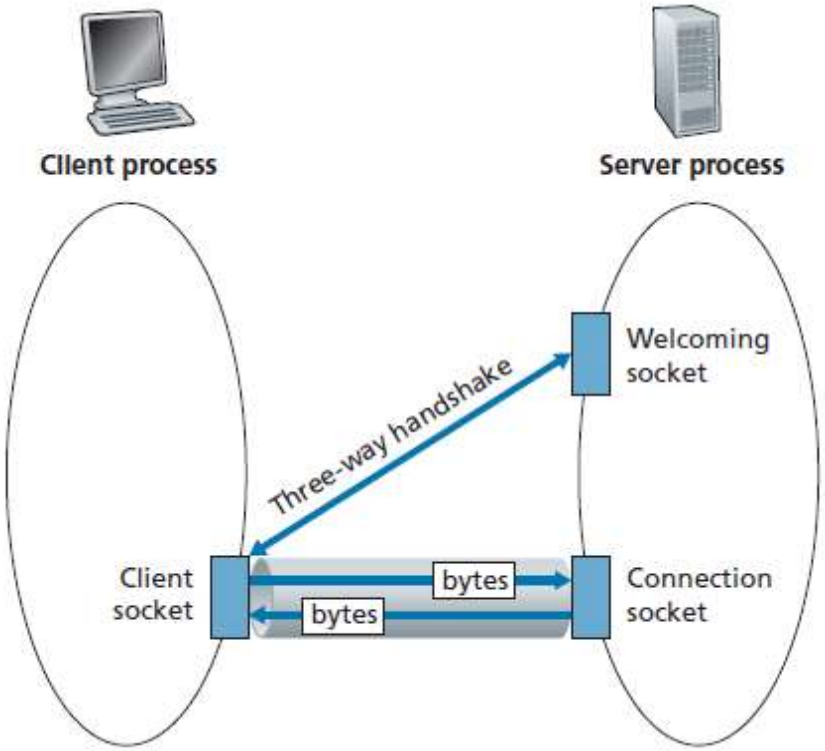


Figure 2.28 The TCP Server process has two sockets

Figure 2.29

- Client-server application using TCP
- Client sends one line of data to server, server capitalizes line and sends it back to client. Figure 2.29 highlights main socket-related activity of client and server that communicate over TCP transport service

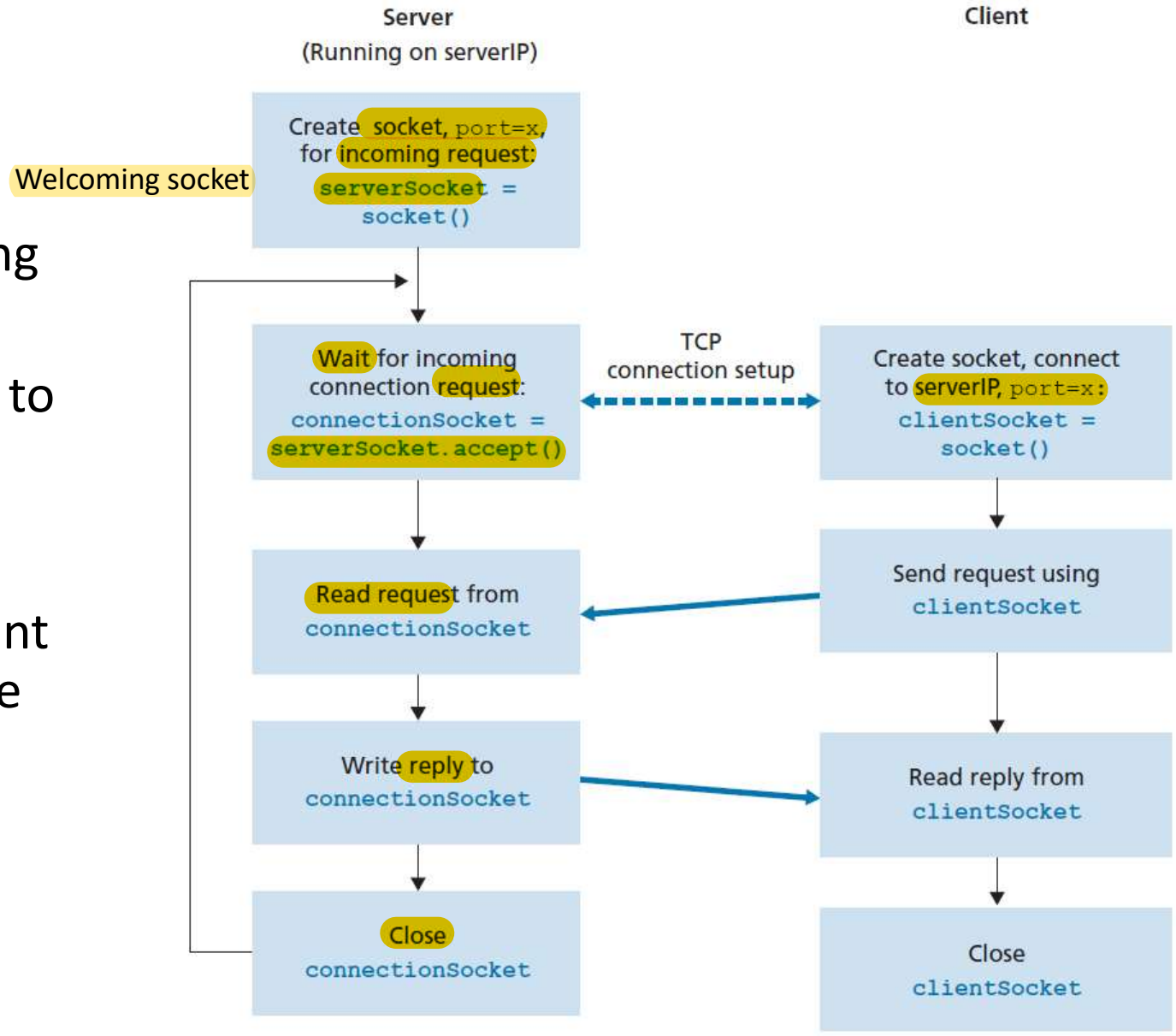
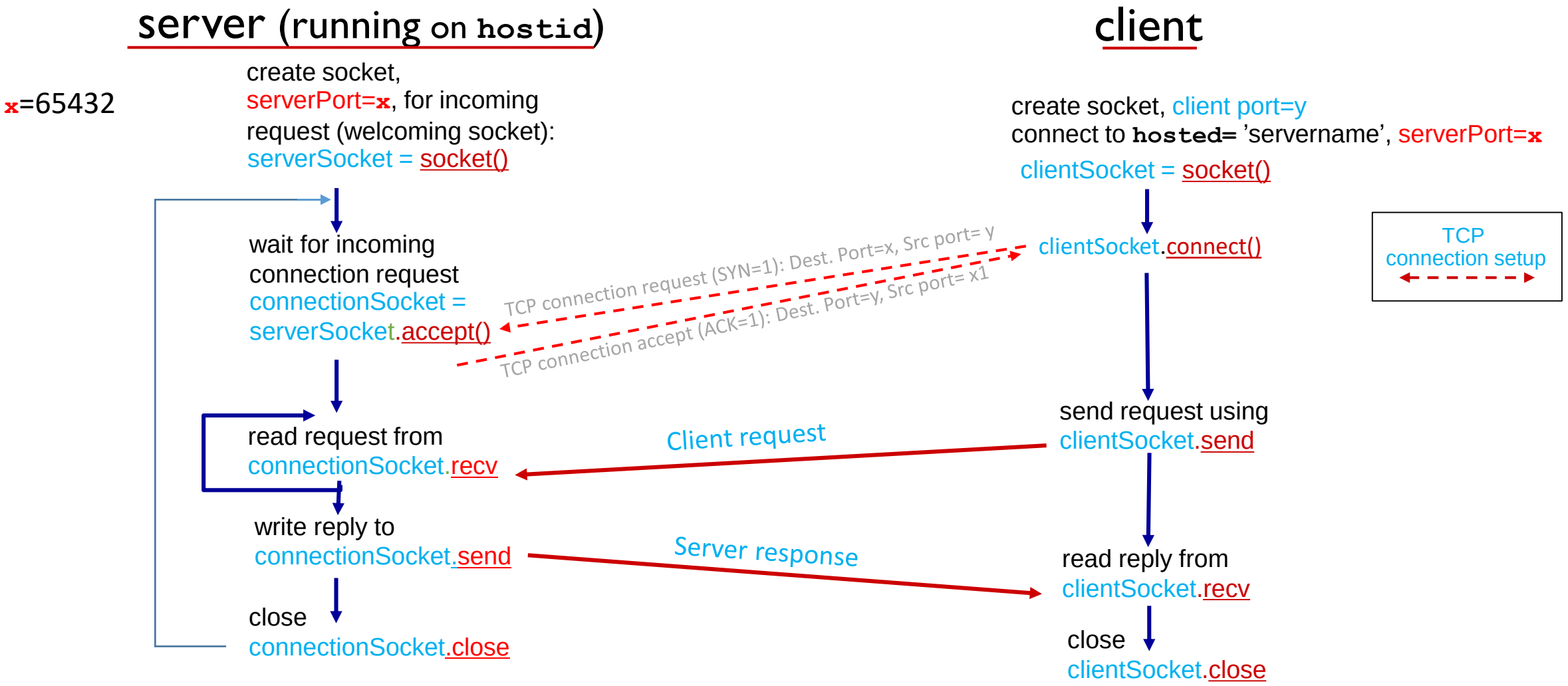


Figure 2.29

Client/server socket interaction: TCP



Example app: TCP client

create TCP socket, called clientSocket
Client socket port number is set by OS

TCP connection to server (three-way handshake is performed)

No need to attach server name, port

```
from socket import *
serverName = 'servername'
serverPort = 65432
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

TCP

Example app: TCP server

Accept connection from any IP address

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

queue up as many as 3 connect requests before refusing outside connections

```
from socket import *
serverPort = 65432
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(3)
print ('The server is ready to receive')
while True:
    connectionSocket, clientAddress = serverSocket.accept(5)

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

Example app: TCP server

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

Obtains a sentence from user. String sentence continues to gather characters until user ends line by typing a carriage return

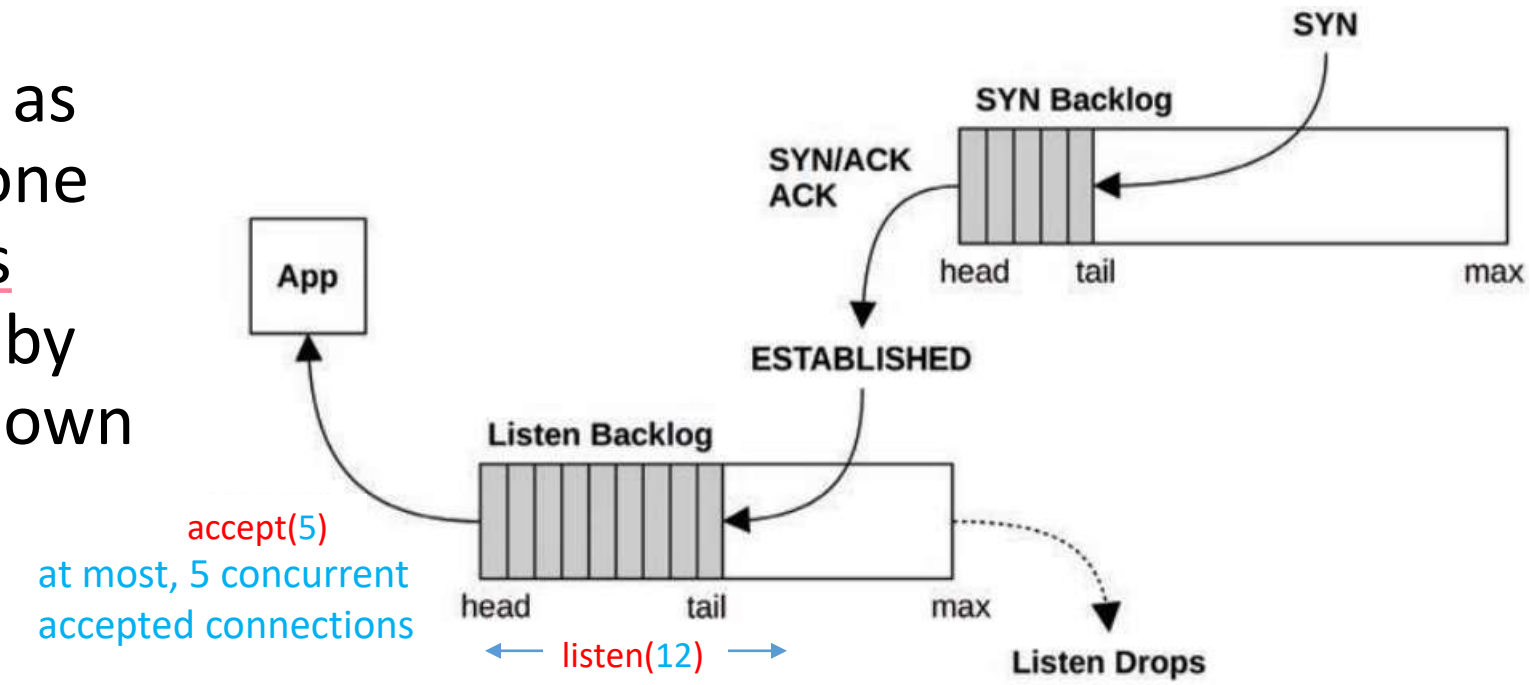
close connection socket to this client (but **not** welcoming socket)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((' ',serverPort))
serverSocket.listen(3)
print ('The server is ready to receive')
while True:
    connectionSocket, clientAddress = serverSocket.accept(5)

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

TCP Connection Queues: Server Side

- Bursts of inbound connections are handled by using backlog **queues**.
- There are **two such queues**, one for incomplete connections while the TCP handshake completes (also known as the SYN backlog), and one for established sessions waiting to be accepted by the application (also known as the listen backlog).



TCP Connection Queues: Server Side.

- With two queues, the first can act as a staging area for potentially bogus connections, which are promoted to the second queue only once the connection is established. The first queue can be made long to absorb SYN floods.
- The length of these queues can be tuned independently.
- The second can also be set by the application as the backlog argument to listen().

2.8 Summary

- Conceptual and implementation aspects of network applications
- Client-server architecture adopted by many Internet applications and seen its use in HTTP, SMTP, and DNS protocols
- P2P architecture
- Streaming video, and how modern video distribution systems leverage CDNs
- Socket API to build network applications.
- Service models that TCP and UDP offer to applications

Appendix

Host-Centric Networking

- Host-centric networking systems restrict applications to data transfer between end-hosts only
- APPs do not natively support:
  1. multi-party communication, e.g., multi-source/multi-destination communication
  2. a ubiquitous information ecosystem that is not restricted to end-host addresses

- Best current practice to manage growth in terms of data volume and number of devices is to **increase infrastructure investment, employ application-layer solutions such as CDNs and P2P**

Information-Centric Networking (ICN)

- Evolve Internet infrastructure to directly support information distribution by introducing **uniquely named data** as a core Internet principle
- Directly support accessing Named Data Objects (NDOs)

- Data becomes independent from
  - location
  - application
  - storage
  - means of transportation
- enabling or enhancing a number of desirable features, such as
  - security
  - user mobility
  - multicast
  - in-network caching

Adaptive Video Streaming over ICN

Topics on video distribution over ICN

- evolving DASH to work over ICN
- layering encoding over ICN
- P2P video distribution,
- adapting P2P Streaming Protocol (PPSP) for ICN
- Data creating more stringent requirements over ICN because of delay constraints added by Internet Protocol Television (IPTV)
- managing digital rights in ICN