A note on the use of these PowerPoint slides:
We’re making these slides freely available to all (faculty, students, readers). They’re in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part.
In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we’d like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2020
J.F Kurose and K.W. Ross, All Rights Reserved
Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS

- P2P applications
- Video streaming and content distribution networks
- Socket programming with UDP and TCP
Application layer: overview

Our goals:
- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm

- learn about protocols by examining popular application-layer protocols
  - HTTP
  - SMTP, IMAP
  - DNS

- programming network applications
  - socket API
Some network apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- P2P file sharing

- voice over IP (e.g., Skype)
- real-time video conferencing
- Internet search
- remote login
- ...

Q: your favorites?
Creating a network app

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

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

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

**clients:**
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other
- examples: HTTP, IMAP, FTP
Peer-peer architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing
Processes communicating

**process**: program running within a host

- within same host, two processes communicate using **inter-process communication** (defined by OS)

- processes in different hosts communicate by exchanging messages

**client process**: process that initiates communication

**server process**: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes
**Sockets**

- process sends/receives messages to/from its **socket**
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side
Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?
- *A:* no, *many* processes can be running on same host

- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly...
An application-layer protocol defines:

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:
- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:
- e.g., Skype
What transport service does an app need?

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

**throughput**
- some apps (e.g., multimedia) require minimum amount of throughput to be “effective”
- other apps (“elastic apps”) make use of whatever throughput they get

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

**security**
- encryption, data integrity, ...

## Transport service requirements: common apps

<table>
<thead>
<tr>
<th>application</th>
<th>data loss</th>
<th>throughput</th>
<th>time sensitive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>file transfer/download</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>e-mail</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>Web documents</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>audio: 5Kbps-1Mbps video:10Kbps-5Mbps</td>
<td>yes, 10’s msec</td>
</tr>
<tr>
<td>streaming audio/video</td>
<td>loss-tolerant</td>
<td>same as above</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>Kbps+</td>
<td>yes, 10’s msec</td>
</tr>
<tr>
<td>text messaging</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no</td>
</tr>
</tbody>
</table>
Internet transport protocols services

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

UDP service:
- **unreliable data transfer** between sending and receiving process
- **does not provide**: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

**Q:** why bother? Why is there a UDP?
# Internet transport protocols services

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

Vanilla TCP & UDP sockets:
- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)
- provides encrypted TCP connections
- data integrity
- end-point authentication

TSL implemented in application layer
- apps use TSL libraries, that use TCP in turn

TLS socket API
- cleartext sent into socket traverse Internet encrypted
- see Chapter 8
Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System (DNS)
- P2P applications
- Video streaming and content distribution networks
- Socket programming with UDP and TCP
Web and HTTP

First, a quick review...

- web page consists of *objects*, each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,...
- web page consists of *base HTML-file* which includes *several referenced objects*, each addressable by a *URL*, e.g.,

  \[ \text{www.someschool.edu/someDept/pic.gif} \]

  \hline
  host name & path name \hline

HTTP overview

HTTP: hypertext transfer protocol

- Web’s application layer protocol
- client/server model:
  - **client**: browser that requests, receives, (using HTTP protocol) and “displays” Web objects
  - **server**: Web server sends (using HTTP protocol) objects in response to requests
HTTP overview (continued)

**HTTP uses TCP:**
- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

**HTTP is “stateless”**
- server maintains no information about past client requests

---

**Aside**
- protocols that maintain “state” are complex!
- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled
HTTP connections: two types

**Non-persistent HTTP**
1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

Downloading multiple objects required multiple connections

**Persistent HTTP**
- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed
Non-persistent HTTP: example

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

1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80

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

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

3. HTTP server receives request message, forms response message containing requested object, and sends message into its socket
Non-persistent HTTP: example (cont.)

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

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects

4. HTTP server closes TCP connection.
Non-persistent HTTP: response time

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

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

Non-persistent HTTP response time = 2RTT + file transmission time
Persistent HTTP (HTTP 1.1)

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

Persistent HTTP (HTTP 1.1):
- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects (cutting response time in half)
HTTP request message

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

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/
```
HTTP request message: general format

```
HTTP/1.1 GET /index.html HTTP/1.1

Host: example.com

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Connection: close

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Page Title</title>
</head>
<body>
<h1>Welcome to Our Website</h1>
<p>This is the content of our page.</p>
</body>
</html>
```
Other HTTP request messages

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

**GET method** (for sending data to server):
- include user data in URL field of HTTP GET request message (following a ‘?’):
  www.somesite.com/animalsearch?monkeys&banana

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

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

status line (protocol
status code status phrase)

HTTP/1.1 200 OK
Date: Sun, 26 Sep 2010 20:09:20 GMT
Server: Apache/2.0.52 (CentOS)
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT
ETag: "17dc6-a5c-bf716880"
Accept-Ranges: bytes
Content-Length: 2652
Keep-Alive: timeout=10, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1

data data data data data data ...

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/
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
Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:
   ```
   telnet gaia.cs.umass.edu 80
   ```
   - opens TCP connection to port 80 (default HTTP server port) at gaia.cs.umass.edu.
   - anything typed in will be sent to port 80 at gaia.cs.umass.edu

2. type in a GET HTTP request:
   ```
   GET /kurose_ross/interactive/index.php HTTP/1.1
   Host: gaia.cs.umass.edu
   ```
   - by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. look at response message sent by HTTP server!
   (or use Wireshark to look at captured HTTP request/response)
Maintaining user/server state: cookies

Recall: HTTP GET/response interaction is *stateless*

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

*Q:* what happens if network connection or client crashes at $t'$?
Maintaining user/server state: cookies

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

**four components:**
1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user’s host, managed by user’s browser
4) back-end database at Web site

**Example:**
- Susan uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID (aka “cookie”)
  - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to “identify” Susan
Maintaining user/server state: cookies

Amazon server creates ID 1678 for user

create entry

cookie-specific action

access

access

cookie-specific action

usual HTTP request msg

usual HTTP response

set-cookie: 1678

usual HTTP request msg

cookie: 1678

usual HTTP response msg

usual HTTP request msg

cookie: 1678

usual HTTP response msg

usual HTTP request msg

cookie: 1678

usual HTTP response msg

one week later:
HTTP cookies: comments

What cookies can be used for:
- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

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

cookies and privacy:
- cookies permit sites to learn a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites
Web caches (proxy servers)

Goal: satisfy client request without involving origin server

- user configures browser to point to a Web cache
- browser sends all HTTP requests to cache
  - if object in cache: cache returns object to client
  - else cache requests object from origin server, caches received object, then returns object to client
Web caches (proxy servers)

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

**Why Web caching?**
- Reduce response time for client request
  - 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

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

Performance:
- LAN utilization: .0015
- access link utilization = .97
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs
Caching example: buy a faster access link

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

**Performance:**
- LAN utilization: 0.0015
- access link utilization = .97
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs

**Cost:** faster access link (expensive!)
Caching example: install a web cache

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

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

Cost: web cache (cheap!)
Caching example: install a web cache

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

- Suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin.
- Access link: 60% of requests use access link.
- Data rate to browsers over access link = 0.6 * 1.50 Mbps = .9 Mbps.
- Utilization = 0.9/1.54 = .58.
- Average end-end delay = 0.6 * (delay from origin servers) + 0.4 * (delay when satisfied at cache) = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs.

Lower average end-end delay than with 154 Mbps link (and cheaper too!)
Conditional GET

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

- **cache:** specify date of cached copy in HTTP request
  - If-modified-since: <date>

- **server:** response contains no object if cached copy is up-to-date:
  - HTTP/1.0 304 Not Modified

  ![Diagram](image-url)
HTTP/2

*Key goal:* decreased delay in multi-object HTTP requests

**HTTP1.1:** introduced *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) stalls object transmission
HTTP/2

*Key goal:* decreased delay in multi-object HTTP requests

**HTTP/2:** [RFC 7540, 2015] increased flexibility at server in sending objects to client:

- methods, status codes, most header fields unchanged from HTTP 1.1
- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking
HTTP/2: mitigating 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

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

$O_2$, $O_3$, $O_4$ delivered quickly, $O_1$ slightly delayed
HTTP/2 to HTTP/3

*Key goal:* decreased delay in multi-object HTTP requests

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, increase overall throughput
- no security over vanilla TCP connection
- **HTTP/3:** adds security, per object error- and congestion-control (more pipelining) over UDP
  - more on HTTP/3 in transport layer
Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System (DNS)

- P2P applications
- Video streaming and content distribution networks
- Socket programming with UDP and TCP
E-mail

Three major components:
- user agents
- mail servers
- simple mail transfer protocol: SMTP

User Agent
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- outgoing, incoming messages stored on server
E-mail: mail servers

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
E-mail: the RFC (5321)

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
- direct transfer: sending server (acting like client) to receiving server
- three phases of transfer
  - 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 ASCII
Scenario: Alice sends e-mail 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 opens TCP connection with Bob’s mail server

4) SMTP client sends Alice’s message over the TCP connection

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

6) Bob invokes his user agent to read message
Sample SMTP interaction

S: 220 hamburger.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@hamburger.edu>
S: 250 bob@hamburger.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 hamburger.edu closing connection
Try SMTP interaction for yourself:

telnet <servername> 25
  ▪ see 220 reply from server
  ▪ enter HELO, MAIL FROM:, RCPT TO:, DATA, QUIT commands
above lets you send email without using e-mail client (reader)

Note: this will only work if <servername> allows telnet connections to port 25 (this is becoming increasingly rare because of security concerns)
SMTP: closing observations

**comparison with HTTP:**

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

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message
Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 531 (like HTTP)

RFC 822 defines syntax for e-mail message itself (like HTML)

- header lines, e.g.,
  - To:
  - From:
  - Subject:
    these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!

- Body: the “message”, ASCII characters only
Mail access protocols

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

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS

- P2P applications
- Video streaming and content distribution networks
- Socket programming with UDP and TCP
DNS: Domain Name System

**people:** many identifiers:
  - SSN, name, passport 

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

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

**Domain Name System:**

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

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

Q: Why not centralize DNS?
- single point of failure
- traffic volume
- distant centralized database
- maintenance

A: doesn’t scale!
- Comcast DNS servers alone: 600B DNS queries per day
Client wants IP address for www.amazon.com; 1st approximation:

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

- official, contact-of-last-resort by name servers that can not resolve name

- *incredibly important* Internet function
  - Internet couldn’t function without it!
  - DNSSEC – provides security (authentication and message integrity)

- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name “servers” worldwide each “server” replicated many times (~200 servers in US)
TLD: authoritative servers

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

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

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

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

Iterated query:
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”
DNS name resolution: recursive query

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

**Recursive query:**
- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?
Caching, 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 (best-effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire!

- update/notify mechanisms proposed IETF standard
  - RFC 2136
DNS records

DNS: distributed database storing resource records (RR)

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

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

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

**type=CNAME**
- name is alias name for some “canonical” (the real) name
- value is canonical name

**type=MX**
- value is name of mailserver associated with name
DNS protocol messages

DNS query and reply messages, both have same format:

message header:
- **identification**: 16 bit # for query, reply to query uses same #
- **flags**:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative
## DNS protocol messages

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

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td># questions</td>
<td># answer RRs</td>
</tr>
<tr>
<td># authority RRs</td>
<td># additional RRs</td>
</tr>
</tbody>
</table>

- name, type fields for a query
- RRs in response to query
- records for authoritative servers
- additional “helpful” info that may be used
Inserting records into DNS

Example: new startup “Network Utopia”

- register name networkuptopia.com at DNS registrar (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts NS, A RRs into .com TLD server:
    (networkutopia.com, dns1.networkutopia.com, NS)
    (dns1.networkutopia.com, 212.212.212.1, A)

- create authoritative server locally with IP address 212.212.212.1
  - type A record for www.networkuptopia.com
  - type MX record for networkutopia.com
DNS security

DDoS attacks
- bombard root servers with traffic
  - not successful to date
  - traffic filtering
  - local DNS servers cache IPs of TLD servers, allowing root server bypass
- bombard TLD servers
  - potentially more dangerous

Redirect attacks
- man-in-middle
  - intercept DNS queries
- DNS poisoning
  - send bogus relies to DNS server, which caches

Exploit DNS for DDoS
- send queries with spoofed source address: target IP
- requires amplification

DNSSEC
[ RFC 4033 ]
Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS

- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP
Peer-to-peer (P2P) architecture

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

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

- peer upload/download capacity is limited resource

\[ u_s: \text{server upload capacity} \]
\[ u_i: \text{peer i upload capacity} \]
\[ d_i: \text{peer i download capacity} \]

network (with abundant bandwidth)
File distribution time: client-server

- **Server transmission**: must sequentially send (upload) $N$ file copies:
  - time to send one copy: $F/u_s$
  - time to send $N$ copies: $NF/u_s$
- **Client**: each client must download file copy
  - $d_{\text{min}} = \min$ client download rate
  - min client download time: $F/d_{\text{min}}$

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

increases linearly in $N$
File distribution time: P2P

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

\[
D_{\text{P2P}} \geq \max\left\{ F/u_s, F/d_{\text{min}}, NF/(u_s + \sum u_i) \right\}
\]

... but so does this, as each peer brings service capacity
Client-server vs. P2P: example

client upload rate = \( u \), \( F/u = 1 \) hour, \( u_s = 10u \), \( d_{min} \geq u_s \)
P2P file distribution: BitTorrent

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

**tracker:** tracks peers participating in torrent

**torrent:** group of peers exchanging chunks of a file

Alice arrives ...
... obtains list of peers from tracker
... and begins exchanging file chunks with peers in torrent
P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- **churn:** peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent
BitTorrent: requesting, sending file chunks

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

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

(1) Alice “optimistically unchokes” Bob
(2) Alice becomes one of Bob’s top-four providers; Bob reciprocates
(3) Bob becomes one of Alice’s top-four providers

*higher upload rate*: find better trading partners, get file faster!
Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System (DNS)

- P2P applications
- Video streaming and content distribution networks
- Socket programming with UDP and TCP
Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- challenge: scale - how to reach ~1B users?
  - single mega-video server won’t work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- solution: distributed, application-level infrastructure
Multimedia: video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy within and between images to decrease # 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 (\( N \))

temporal coding example: instead of sending complete frame at \( i+1 \), send only differences from frame \( i \)
Multimedia: video

- **CBR**: (constant bit rate): video encoding rate fixed
- **VBR**: (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- **examples:**
  - MPEG 1 (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 ($N$)

**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 bandwidth 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

1. video recorded (e.g., 30 frames/sec)
2. video sent
3. video received, played out at client (30 frames/sec)

*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
Streaming multimedia: DASH

- **DASH**: Dynamic, Adaptive Streaming over HTTP

  - **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
    - 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)
Streaming multimedia: DASH

“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
Content distribution networks (CDNs)

- **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 video sent over outgoing link

....quite simply: this solution *doesn’t scale*
Content distribution networks (CDNs)

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

- **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: 240,000 servers deployed in more than 120 countries (2015)
  - *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
Content distribution networks (CDNs)

OTT: “over the top”

Internet host-host communication as a service

OTT challenges: coping with a congested Internet

- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?
CDN content access: a closer look

Bob (client) requests video http://netcinema.com/6Y7B23V
- video stored in CDN at http://KingCDN.com/NetC6y&B23V
Case study: Netflix

1. Bob manages Netflix account.
2. Bob browses Netflix video.
3. Manifest file, requested returned for specific video.
4. DASH server selected, contacted, streaming begins.

Network connections:
- Netflix registration, accounting servers
- Amazon cloud
- CDN server
- Multiple CDN servers
- DASH server

Upload copies of multiple versions of video to CDN servers.
Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System (DNS)
- P2P applications
- Video streaming and content distribution networks
- Socket programming with UDP and TCP
Socket programming

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

**socket:** door between application process and end-end-transport protocol
Socket programming

Two socket types for two transport services:

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

Application Example:

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

UDP: no “connection” between client & server
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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

Application viewpoint:
- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server
Client/server socket interaction: UDP

**server** (running on serverIP)

- create socket, port=x:
  - serverSocket = socket(AF_INET,SOCK_DGRAM)
- read datagram from serverSocket
- write reply to serverSocket specifying client address, port number

**client**

- create socket:
  - clientSocket = socket(AF_INET,SOCK_DGRAM)
- Create datagram with server IP and port=x; send datagram via clientSocket
- read datagram from clientSocket
- close clientSocket

Application Layer: 2-103
Example app: UDP client

Python UDPClient

include Python’s socket library → from socket import *
serverName = ‘hostname’
serverPort = 12000
clientSocket = socket(AF_INET,
SOCK_DGRAM)
message = raw_input(‘Input lowercase sentence:’)
clientSocket.sendto(message.encode(),
(serverName, serverPort))
modifiedMessage, serverAddress =
clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
Example app: UDP server

Python UDPServer

```python
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)
```

- create UDP socket
- 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

Application Layer: 2-105
Socket programming with TCP

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

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

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

Application viewpoint

TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server
Client/server socket interaction: TCP

**server** (running on hostid)

- create socket, port=x, for incoming request:
  - serverSocket = socket()

- wait for incoming connection request
  - connectionSocket = serverSocket.accept()

- read request from connectionSocket

- write reply to connectionSocket

- close connectionSocket

**TCP connection setup**

**client**

- create socket, connect to hostid, port=x
  - clientSocket = socket()

- send request using clientSocket

- read reply from clientSocket

- write reply to clientSocket

- close clientSocket
Example app: TCP client

Python TCPClient

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

Python TCPServer

```python
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((' ',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```
Chapter 2: Summary

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P

- application service requirements:
  - reliability, bandwidth, delay

- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP

- specific protocols:
  - HTTP
  - SMTP, IMAP
  - DNS
  - P2P: BitTorrent

- video streaming, CDNs

- socket programming:
  TCP, UDP sockets
Chapter 2: Summary

Most importantly: learned about protocols!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code

- message formats:
  - headers: fields giving info about data
  - data: info(payload) being communicated

- important themes:
  - centralized vs. decentralized
  - stateless vs. stateful
  - scalability
  - reliable vs. unreliable message transfer
  - “complexity at network edge”
Additional Chapter 2 slides