Chapter 2: Application Layer

Applications
… built on ...
Reliable (or unreliable) transport
… built on ...
Best-effort global packet delivery
… built on ...
Best-effort local packet delivery
… built on ...
Physical transfer of bits

The source PowerPoint slides are public available, provided by Authors (JFK/KWR). They are revised for CS536@Purdue.
Application layer: goals

- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm

- learn about protocols by examining popular application-level protocols
  - HTTP
  - SMTP / POP3 / IMAP
  - DNS
  - P2P
  - Video streaming

- creating network applications
  - socket API
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
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

Application Layer: 2-5
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

- Example: to send HTTP message to google.com web server:
  - IP address: 142.250.191.100
  - 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, Zoom
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

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

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

**security**
- Encryption, data integrity, ...

Application Layer: 2-11
## 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
- **connection-oriented:** setup required between client and server processes
- **does not provide:** timing, minimum throughput guarantee, security

**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?
<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>
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

- Most popular or a killer application on the Internet
- Client-server model

PC, phone, pad
running web browser
(chrome, Firefox, Safari, …)

e.g., running
Apache Web server

Google
4 Questions in Web and HTTP

- What content to transfer?
  - Web pages

- How to transfer?
  - HTTP connections

- In what messages?
  - HTTP messages

- Advanced features
Web and Web browser

- On the client
  - Loading web pages in web browser

A web page for google.com (right-click: view source)
Web page source (example)

Say "Ok Google" to start a voice search.

Donate to help refugees and migrants in urgent need.

We'll match your donation.
Web page

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.,

  http://www.google.com

  /images/branding/googlelogo/2x/googlelogo_color_272x92dp.png

  host name

  path name
HTTP overview

HTTP: hypertext transfer protocol

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

**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)
Non-persistent HTTP vs. persistent HTTP

A web page with 10 image objects. How long to complete all?
  • Ignore the object transmission time

• non-persistent HTTP (with 1 object one time)
  22 RTTs = 2RTTs × (1 + 10) (2 RTTs for base HTML file, 2 RTTs per object)

• Persistent HTTP
  12 RTTs = 2RTTs + 10 RTTs (1 RTT per object)

• non-persistent HTTP (5 objects in parallel)
  6RTTs = 2RTTs + 2RTTs + 2RTTs (2 RTTs for base html file, 2 more RTTs for the first 5 objects, 2 more RTTs for the last 5 objects)
4 Questions in Web and HTTP

- What content to transfer?
  - Web pages

- How to transfer?
  - HTTP connections

- In what messages?
  - HTTP messages

- Advanced features
HTTP request message

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

request line (GET, POST, HEAD commands)

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

- **HTTP request message**
  - **method**
  - **sp**
  - **URL**
  - **sp**
  - **version**
  - **cr**
  - **lf**
  - **header field name**
  - **value**
  - **cr**
  - **lf**
  - **header field name**
  - **value**
  - **cr**
  - **lf**
  - **cr**
  - **lf**
  - **entity body**

**Diagram notes**
- Request line
- Header lines
- Body

Application Layer: 2-32
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
HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

  200 OK
  - request succeeded, requested object later in this message

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

  400 Bad Request
  - request msg not understood by server

  404 Not Found
  - requested document not found on this server

  505 HTTP Version Not Supported
HTTP response status codes (more, optional)

- **1xx**: Informational means the request was received and the process is continuing.
- **2xx**: Success means the action was successfully received, understood, and accepted.
- **3xx**: Redirection means further action must be taken in order to complete the request.
- **4xx**: Client Error means the request contains incorrect syntax or cannot be fulfilled.
- **5xx**: Server Error means the server failed to fulfill an apparently valid request.

Reference of 200 OK status: [https://restfulapi.net/http-status-200-ok/](https://restfulapi.net/http-status-200-ok/)
Trying out HTTP (client side) for yourself

1. netcat to your favorite Web server:

   % nc -c -v gaia.cs.umass.edu 80

   % 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)
4 Questions in Web and HTTP

- What content to transfer?
  - Web pages

- How to transfer?
  - HTTP connections

- In what messages?
  - HTTP messages

- Advanced features
Maintaining user/server state: cookies

Recall: HTTP GET/response interaction is *stateless*

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

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

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

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

backend database

one week later:

ebay 8734
cookie file

ebay 8734
amazon 1678

usual HTTP request msg
usual HTTP response msg
set-cookie: 1678

usual HTTP request msg
cookie: 1678

usual HTTP request msg
cookie: 1678

usual HTTP response msg

Application Layer: 2-41
HTTP cookies: comments

What cookies can be used for:

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

Challenge: How to keep state?

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

**Goal:** satisfy client requests without involving origin server

- user configures browser to point to a (local) *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 (aka proxy servers)

- Web cache acts as both client and server
  - server for original requesting client
  - client to origin server

- server tells cache about object’s allowable caching in response header:
  
  | Cache-Control: max-age=<seconds> |
  | Cache-Control: no-cache |

**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
  - avg data rate to browsers: 1.50 Mbps

Performance:
- access link utilization = .97
- LAN utilization: .0015
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs

problem: large queueing delays at high utilization!
Option 1: 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
- average request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

Performance:
- access link utilization = 0.97
- LAN utilization: 0.0015
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs

Cost: faster access link (expensive!)
Option 2: install a web cache

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
  - avg data rate to browsers: 1.50 Mbps

Cost: web cache (cheap!)

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

How to compute link utilization, delay?
Calculating access link utilization, end-end delay with cache:

suppose cache hit rate is 0.4:

- 40% requests served by cache, with low (msec) delay
- 60% requests satisfied at origin
  - rate to browsers over access link
    \[ = 0.6 \times 1.50 \text{ Mbps} = 0.9 \text{ Mbps} \]
  - access link utilization = 0.9/1.54 = 0.58 means low (msec) queueing delay at access link

average end-end delay:
\[ = 0.6 \times \text{(delay from origin servers)} + 0.4 \times \text{(delay when satisfied at cache)} \]
\[ = 0.6 \times (2.01) + 0.4 \times (\text{~msecs}) = \sim 1.2 \text{ secs} \]

lower average end-end delay than with 154 Mbps link (and cheaper too!)
Browser caching: Conditional GET

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

- no object transmission delay (or use of network resources)

- **client:** specify date of browser-cached copy in HTTP request
  
  \[\text{If-modified-since: } <\text{date}>\]

- **server:** response contains no object if browser-cached copy is up-to-date:
  
  \[\text{HTTP/1.0 304 Not Modified}\]
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

\[ O_2, O_3, O_4 \text{ delivered quickly, } O_1 \text{ slightly delayed } \]
HTTP/2 to HTTP/3

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 (optional):** 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
SMTP 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
  - SMTP handshaking (greeting)
  - SMTP transfer of messages
  - SMTP closure
- command/response interaction (like HTTP)
  - commands: ASCII text
  - response: status code and phrase
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 using SMTP; message placed in message queue

3) client side of SMTP at mail server 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

Application Layer: 2-65
Sample SMTP interaction

S: 220 hamburger.edu
SMTP: observations

comparison with HTTP:

- HTTP: client pull
- SMTP: client 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 5321 (like RFC 7231 defines HTTP)

RFC 2822 defines syntax for e-mail message itself (like HTML defines syntax for web documents)

- 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
Retrieving email: 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 (DNS):**
- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, DNS 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/day
- Akamai DNS servers alone: 2.2T DNS queries/day
Thinking about the DNS

humongous distributed database:
- ~ billion records, each simple

handles many *trillions* of queries/day:
- *many* more reads than writes
- *performance matters*: almost every Internet transaction interacts with DNS - msecs count!

organizationally, physically decentralized:
- millions of different organizations responsible for their records

“bulletproof”: reliability, security
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

![Diagram of DNS hierarchy with root servers and delegation to .com, .org, and .edu zones]
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, 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)
Top-Level Domain, and 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

- when host makes DNS query, it is sent to its *local* DNS server
  - Local DNS server returns reply, answering:
    - from its local cache of recent name-to-address translation pairs (possibly out of date!)
    - forwarding request into DNS hierarchy for resolution
  - each ISP has local DNS name server; to find yours:
    - MacOS: `% scutil --dns`
    - Windows: `>ipconfig /all`

- local DNS server *doesn’t* strictly belong to 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?
once (any) name server learns mapping, it *caches* mapping, and *immediately* returns a cached mapping in response to a query

- caching improves response time
- cache entries timeout (disappear) after some time (TTL)
- TLD servers typically cached in local name servers

cached entries may be *out-of-date*

- if named host changes IP address, may not be known Internet-wide until all TTLs expire!
- *best-effort name-to-address translation*!
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
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical name

type=MX
- value is name of SMTP mail server associated with name
Demo: Try DNS yourself (nslookup)

> nslookup amazon.com
> nslookup cs.purdue.edu

Authoritative name servers
> nslookup -type=NS cs.purdue.edu
> nslookup -type=A harbor.ecn.purdue.edu

Canonical name (alias)
> nslookup -type=CNAME amazon.com
> nslookup -type=CNAME purdue.edu

Mail server
> nslookup -type=MX gmail.com
> nslookup -type=MX purdue.edu
More at demo-nslookup

> nslookup -type=A cs.purdue.edu
Server: 128.210.11.57
Address: 128.210.11.57#53
Name: cs.purdue.edu
Address: 128.10.19.120

> nslookup -type=A www.cs.purdue.edu
Server: 128.210.11.57
Address: 128.210.11.57#53
Name: pythia.cs.purdue.edu
Address: 128.10.19.120
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

<table>
<thead>
<tr>
<th></th>
<th>identification</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td># questions</td>
<td></td>
<td># answer RRs</td>
</tr>
<tr>
<td># authority RRs</td>
<td></td>
<td># additional RRs</td>
</tr>
</tbody>
</table>

- **questions (variable # of questions)**
- **answers (variable # of RRs)**
- **authority (variable # of RRs)**
- **additional info (variable # of RRs)**
### DNS protocol messages

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>identification</td>
<td>flags</td>
<td></td>
</tr>
<tr>
<td># questions</td>
<td># answer RRs</td>
<td></td>
</tr>
<tr>
<td># authority RRs</td>
<td># additional RRs</td>
<td></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**

2 bytes

DNS protocol messages of Application Layer: 2-86.
Getting your info into the 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
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?

- 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, access network, network core, video server)
- packet loss, delay due to congestion will delay playout, or result in poor video quality
Streaming multimedia: DASH

server:
- divides video file into multiple chunks
- each chunk encoded at multiple different rates
- different rate encodings stored in different files
- files replicated in various CDN nodes
- manifest file: provides URLs for different chunks

client:
- periodically estimates 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), and from different servers
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
Streaming at Scale?

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

**Answer:** content distribution networks (CDNs)

- *Cache in the middle or edge of the Internet*
- *Optional: read 2.6.3 and 2.6.4*

**Used by almost all the video streaming companies, e.g. Youtube, Netflix,**
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