CMPE 150/L: Introduction to Computer Networks

Chen Qian
Computer Engineering
UCSC Baskin Engineering
Lecture 19
Final Exam

- Thursday March 22\textsuperscript{nd}.
- 4:00-7:00pm
- Cheat sheet is allowed (same requirement of midterm)
Synthesis: a day in the life of a web request

- journey down protocol stack complete!
  - application, transport, network, link

- putting-it-all-together: synthesis!
  - **goal:** identify, review, understand protocols (at all layers) involved in seemingly simple scenario: requesting www page
  - **scenario:** student attaches laptop to campus network, requests/receives www.google.com
A day in the life: scenario

Comcast network
68.80.0.0/13

Google’s network
64.233.160.0/19

web server
64.233.169.105

DNS server

school network
68.80.2.0/24

browser

web page
A day in the life… connecting to the Internet

- connecting laptop needs to get its own IP address, addr of first-hop router, addr of DNS server: use **DHCP**

- DHCP request *encapsulated* in **UDP**, encapsulated in **IP**, encapsulated in **802.3** Ethernet

- Ethernet frame *broadcast* (dest: FFFFFFFFFFFFFFFF) on LAN, received at router running **DHCP** server

- Ethernet *demuxed* to IP demuxed, UDP demuxed to DHCP
A day in the life… connecting to the Internet

- DHCP server formulates **DHCP ACK** containing client’s IP address, IP address of first-hop router for client, name & IP address of DNS server
- encapsulation at DHCP server, frame forwarded (**switch learning**) through LAN, demultiplexing at client
- DHCP client receives DHCP ACK reply

**Client now has IP address, knows name & addr of DNS server, IP address of its first-hop router**
A day in the life… ARP (before DNS, before HTTP)

- before sending **HTTP** request, need IP address of www.google.com: **DNS**
- DNS query created, encapsulated in UDP, encapsulated in IP, encapsulated in Eth. To send frame to router, need MAC address of router interface: **ARP**
- **ARP query** broadcast, received by router, which replies with **ARP reply** giving MAC address of router interface
- client now knows MAC address of first hop router, so can now send frame containing DNS query
**A day in the life... using DNS**

- IP datagram containing DNS query forwarded via LAN switch from client to 1st hop router

- IP datagram forwarded from campus network into comcast network, routed (tables created by **RIP, OSPF, IS-IS** and/or **BGP** routing protocols) to DNS server

- demux’ed to DNS server

- DNS server replies to client with IP address of www.google.com
A day in the life…TCP connection carrying HTTP

- to send HTTP request, client first opens *TCP socket* to web server
- TCP *SYN segment* (step 1 in 3-way handshake) *inter-domain routed* to web server
- web server responds with *TCP SYNACK* (step 2 in 3-way handshake)
- *TCP connection established!*
A day in the life... HTTP request/reply

- web page finally (!!!) displayed

- **HTTP request** sent into TCP socket
- IP datagram containing HTTP request routed to www.google.com
- web server responds with **HTTP reply** (containing web page)
- IP datagram containing HTTP reply routed back to client
1: Connecting (few) computers: e-mail, file transfer, remote login.
2: Connecting larger number of computers: sharing information (WWW).
3: Connecting wireless and mobile devices.
4: Connecting people: social networks.
5: Connecting objects: Information-Centric Networks (ICNs), Internet of Things (IoT), Context-Aware Networking.
Host: sends packets of data

- **Host sending function:**
  - Takes application message
  - Breaks into smaller chunks, known as packets, of length $L$ bits
  - Transmits packet into access network at **transmission rate $R$**
    - Link transmission rate, aka link capacity, aka link bandwidth

$$\text{packet transmission delay} = \frac{L \text{ (bits)}}{R \text{ (bits/sec)}}$$
Packet Switching: queueing delay, loss

**queueing and loss:**
- If arrival rate (in bits) to link exceeds transmission rate of link for a period of time:
  - packets will queue, wait to be transmitted on link
  - packets can be dropped (lost) if memory (buffer) fills up
Packet switching versus circuit switching

packet switching allows more users to use network!

**example:**
- 1 Mb/s link
- each user:
  - 100 kb/s when “active”
  - active 10% of time

- **circuit-switching:**
  - 10 users

- **packet switching:**
  - with 35 users, probability > 10 active at same time is less than 0.0004 *

Q: how did we get value 0.0004?
Q: what happens if > 35 users?
Packet switching versus circuit switching

is packet switching a “slam dunk winner?”

- great for bursty data
  - resource sharing
  - simpler, no call setup

- excessive congestion possible: packet delay and loss
  - protocols needed for reliable data transfer, congestion control
Internet structure: network of networks

Option: connect each access ISP to a global transit ISP? Customer and provider ISPs have economic agreement.
Internet structure: network of networks

- at center: small # of well-connected large networks
  - “tier-1” commercial ISPs (e.g., Level 3, Sprint, AT&T, NTT), national & international coverage
  - content provider network (e.g, Google): private network that connects its data centers to Internet, often bypassing tier-1, regional ISPs
Four sources of packet delay

\[ d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}} \]

\( d_{\text{proc}} \): nodal processing
- check bit errors
- determine output link
- typically < msec

\( d_{\text{queue}} \): queueing delay
- time waiting at output link for transmission
- depends on congestion level of router
Four sources of packet delay

transmission delay:
- $L$: packet length (bits)
- $R$: link bandwidth (bps)
- $d_{trans} = L/R$

propagation delay:
- $d$: length of physical link
- $s$: propagation speed in medium ($\sim 2 \times 10^8$ m/sec)
- $d_{prop} = d/s$

\[ d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop} \]
Throughput (more)

- $R_s < R_c$ What is average end-end throughput?
  No higher than $R_s$!

- $R_s > R_c$ What is average end-end throughput?
  No higher than $R_c$!

*bottleneck link*

link on end-end path that constrains end-end throughput
Internet protocol stack

- **application**: supporting network applications
  - FTP, SMTP, HTTP
- **transport**: process-process data transfer
  - TCP, UDP
- **network**: routing of datagrams from source to destination
  - IP, routing protocols
- **link**: data transfer between neighboring network elements
  - Ethernet, 802.11 (WiFi), PPP
- **physical**: bits “on the wire”
Bad guys: put malware into hosts via Internet

- malware can get in host from:
  - *virus*: self-replicating infection by receiving/executing object (e.g., e-mail attachment)
  - *worm*: self-replicating infection by passively receiving object that gets itself executed

- spyware malware can record keystrokes, web sites visited, upload info to collection site

- infected host can be enrolled in botnet, used for spam. DDoS attacks
Client/server versus P2P

- **Throughput and Scalability:**
  - P2P wins!
  - Because a server can only serve limited number of clients
  - P2P allows clients exchange data among them.
  - That’s why P2P became popular in early 2000

- **Management**
  - C/S wins!
  - Because users in P2P are HIGHLY unreliable.
  - In the recent years, throughput are not a big problem, management became the main issue.
  - That’s why we now switch back to C/S
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
## 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</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, 100’s msec</td>
</tr>
<tr>
<td>stored 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>few kbps up</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td>text messaging</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no</td>
</tr>
</tbody>
</table>
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)

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 connections

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

persistent HTTP

- multiple objects can be sent over single TCP connection between client, server
More about Web caching

- 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
- reduce traffic on an institution’s access link

**When is cache not good?**

- Every client of the ISP requests different content.
  - Waste time on visiting cache server
FTP: separate control, data connections

- FTP client contacts FTP server at port 21, using TCP
- Client authorized over control connection
- Client browses remote directory, sends commands over control connection
- When server receives file transfer command, server opens 2nd TCP data connection (for file) to client
- After transferring one file, server closes data connection
- Server opens another TCP data connection to transfer another file
- FTP server maintains “state”: current directory, earlier authentication
Scenario: Alice sends message to Bob

1) Alice uses UA to compose 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
Mail access protocols

- **SMTP**: delivery/storage to receiver’s server
- **mail access protocol**: retrieval from server
  - **POP**: Post Office Protocol [RFC 1939]: authorization, download
  - **IMAP**: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
  - **HTTP**: gmail, Hotmail, Yahoo! Mail, etc.
**DNS: a distributed, hierarchical database**

- **Root DNS Servers**
  - com DNS servers
  - org DNS servers
  - edu DNS servers

- com DNS servers
  - yahoo.com DNS servers
  - amazon.com DNS servers

- org DNS servers
  - pbs.org DNS servers

- edu DNS servers
  - poly.edu DNS servers
  - umass.edu DNS servers

---

**client wants IP for www.amazon.com; 1st approx:**

- 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: services, structure

DNS services
- hostname to IP address translation
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?
- single point of failure
- traffic volume
- distant centralized database
- maintenance

A: doesn’t scale!
DNS: caching, updating 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
P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

**examples:**
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)

- However, most of them requires a central server to manage the peers
**Multiplexing/demultiplexing**

**Multiplexing at sender:**
- Handle data from multiple sockets, add transport header (later used for demultiplexing).

**Demultiplexing at receiver:**
- Use header info to deliver received segments to correct socket.

---

Diagram:
- **Application** layers:
  - **P1**, **P2**, **P3**, **P4**
- **Transport** layer
- **Network** layer
- **Link** layer
- **Physical** layer

**Socket** positioning:
- **Socket** icons are shown at the top of the diagram, associated with each process (P1, P2, P3, P4).

**Process** icons:
- **Process** icons are shown at the bottom of the diagram, labeled with each process's name (P1, P2, P3, P4).
How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses **IP addresses & port numbers** to direct segment to appropriate socket

TCP/UDP segment format:
- source port #
- dest port #
- other header fields
- application data (payload)
- 32 bits
UDP: User Datagram Protocol [RFC 768]

- “best effort” service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

UDP use:
- streaming multimedia apps (loss tolerant, rate sensitive)
- DNS
- Simple Network Management Protocol (SNMP)
**UDP checksum**

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

**sender:**
- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

**receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless? More later* ....
**Internet checksum: example**

example: add two 16-bit integers

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\]

wraparound

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\]

sum

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

checksum

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result.

At the receiver, adding all words and checksum, the result should be all ones. If there is a 0, some error must happen.
**rdt 1.0: reliable transfer over a reliable channel**

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

```
Wait for call from above

rdt_send(data)
packet = make_pkt(data)
uc_send(packet)

sender
```

```
Wait for call from below

rdt_rcv(packet)
extract (packet, data)
deliver_data(data)

receiver
```
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender
rdt2.1: idea

- Sender puts a seq num 0 or 1 to each segment.
- It sends a segment with 0 and then wait for an ACK.
- If receives ACK
  - Sends a segment with 1
- If receives NAK or corrupted ACK
  - Resends the segment with 0.

- Receiver receives a segment with 0.
  - Replies an ACK.
- Then if it receives a segment with 1.
  - The sender must received the ACK.
- If receives a segment with 0.
  - The sender did not receive the ACK.
rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*
new assumption: underlying channel can also lose packets (data, ACKs)
  - checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

approach: sender waits “reasonable” amount of time for ACK
  - retransmits if no ACK received in this time
  - if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate, but seq. #’s already handles this
    - receiver must specify seq # of pkt being ACKed
  - requires countdown timer
Pipelined protocols: overview

Go-back-N:
- sender can have up to $N$ unacked packets in pipeline
- receiver only sends cumulative ack
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit all unacked packets

Selective Repeat:
- sender can have up to $N$ unack’ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet
TCP round trip time, timeout

**Q:** how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

**Q:** how to estimate RTT?
- *SampleRTT:* measured time from segment transmission until ACK receipt
  - ignore retransmissions
- *SampleRTT* will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current *SampleRTT*
TCP round trip time, timeout

Estimated\text{RTT} = (1 - \alpha) \times \text{Estimated\text{RTT}} + \alpha \times \text{Sample\text{RTT}}

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)
TCP round trip time, timeout

- **timeout interval**: EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT → larger safety margin

- estimate SampleRTT deviation from EstimatedRTT:
  
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
  \]
  
  (typically, \( \beta = 0.25 \))

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

**TCP fast retransmit**

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #
  - likely that unacked segment lost, so don’t wait for timeout
TCP flow control

receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast
TCP 3-way handshake

**client state**

- **LISTEN**
  - choose init seq num, $x$
  - send TCP SYN msg

- **SYNSENT**
  - received SYNACK($x$) indicates server is live;
  - send ACK for SYNACK;
  - this segment may contain client-to-server data

- **ESTAB**

**server state**

- **LISTEN**

- **SYN RCVD**
  - choose init seq num, $y$
  - send TCP SYNACK msg, acking SYN

- **ESTAB**
  - received ACK($y$) indicates client is live
Throughput:
- Data rate at the receiver

Goodput:
- Rate at the receiver for data without duplicate!
TCP Congestion Control: details

TCP sending rate:

- roughly: send cwnd bytes, wait RTT for ACKS, then send more bytes

\[ \text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec} \]

- sender limits transmission:

  \[
  \text{LastByteSent - LastByteAcked} \leq \text{cwnd}
  \]

- cwnd is dynamic, function of perceived network congestion
TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially $cwnd = 1$ MSS
  - double $cwnd$ every RTT
  - done by incrementing $cwnd$ for every ACK received

- **summary**: initial rate is slow but ramps up exponentially fast
TCP: detecting, reacting to loss

- loss indicated by **timeout**:
  - set a threshold $\text{ssthresh}$ to half of the $\text{cwnd}$;
  - $\text{cwnd}$ set to 1 MSS (by both TCP Tahoe and Reno);
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

- TCP Tahoe always sets $\text{cwnd}$ to 1 (**timeout** or 3 duplicate acks)

- TCP RENO: loss indicated by 3 duplicate ACKs
  - dup ACKs indicate network capable of delivering some segments
  - $\text{cwnd}$ is cut in half window then grows linearly
TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?
A: when cwnd gets to 1/2 of its value before timeout.

Implementation:
- variable ssthresh
- on loss event, ssthresh is set to 1/2 of cwnd just before loss event
### Network layer service models:

<table>
<thead>
<tr>
<th>Network Architecture</th>
<th>Service Model</th>
<th>Guarantees?</th>
<th>Congestion Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Bandwidth</td>
<td>Loss</td>
</tr>
<tr>
<td>Internet</td>
<td>best effort</td>
<td>none</td>
<td>no</td>
</tr>
<tr>
<td>ATM</td>
<td>CBR</td>
<td>constant rate</td>
<td>yes</td>
</tr>
<tr>
<td>ATM</td>
<td>VBR</td>
<td>guaranteed rate</td>
<td>yes</td>
</tr>
<tr>
<td>ATM</td>
<td>ABR</td>
<td>guaranteed minimum</td>
<td>no</td>
</tr>
<tr>
<td>ATM</td>
<td>UBR</td>
<td>none</td>
<td>no</td>
</tr>
</tbody>
</table>

ATM has various guarantees. Internet has almost none.
Longest prefix matching

*longest prefix matching*
when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

<table>
<thead>
<tr>
<th>Destination Address Range</th>
<th>Link interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>11001000 00010111 00010*** *********</td>
<td>0</td>
</tr>
<tr>
<td>11001000 00010111 00011000 *********</td>
<td>1</td>
</tr>
<tr>
<td>11001000 00010111 00011*** *********</td>
<td>2</td>
</tr>
<tr>
<td>otherwise</td>
<td>3</td>
</tr>
</tbody>
</table>

**examples:**

DA: 11001000 00010111 00010110 10100001 which interface?
DA: 11001000 00010111 00011000 10101010 which interface?
Datagram or VC network: why?

**Internet (datagram)**
- data exchange among computers
  - “elastic” service, no strict timing req.
- many link types
  - different characteristics
  - uniform service difficult
- “smart” end systems (computers)
  - can adapt, perform control, error recovery
  - *simple inside network, complexity at “edge”*

**ATM (VC)**
- evolved from telephony
- human conversation:
  - strict timing, reliability requirements
  - need for guaranteed service
- “dumb” end systems
  - telephones
  - *complexity inside network*
**IP datagram format**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP protocol version</td>
<td>32 bits</td>
</tr>
<tr>
<td>header length (bytes)</td>
<td>32 bits</td>
</tr>
<tr>
<td>“type” of data</td>
<td>16-bit identifier</td>
</tr>
<tr>
<td>max number remaining</td>
<td>time to live</td>
</tr>
<tr>
<td>remaining hops</td>
<td>header layer</td>
</tr>
<tr>
<td>(decremented at each</td>
<td>incremental number of hops</td>
</tr>
<tr>
<td>router)</td>
<td>options (if any)</td>
</tr>
<tr>
<td>32 bit source IP</td>
<td>data</td>
</tr>
<tr>
<td>address</td>
<td>(variable length, typically a TCP or UDP segment)</td>
</tr>
<tr>
<td>32 bit destination IP</td>
<td>total datagram length (bytes)</td>
</tr>
<tr>
<td>address</td>
<td>for fragmentation/reassembly</td>
</tr>
<tr>
<td>upper layer protocol</td>
<td>e.g. timestamp, record route taken, specify list of routers to visit.</td>
</tr>
<tr>
<td>to deliver payload to</td>
<td>how much overhead?</td>
</tr>
<tr>
<td></td>
<td>20 bytes of TCP</td>
</tr>
<tr>
<td></td>
<td>20 bytes of IP</td>
</tr>
<tr>
<td></td>
<td>= 40 bytes + app layer overhead</td>
</tr>
</tbody>
</table>
IP addressing: introduction

- **IP address**: 32-bit identifier for host, router interface

- **interface**: connection between host/router and physical link
  - router's typically have multiple interfaces
  - host typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)

- **IP addresses associated with each interface**

  - 223.1.1.1 = 11011111 00000001 00000001 00000001
  - 223 1 1 1
Subnets

- **IP address:**
  - subnet part - high order bits
  - host part - low order bits

- **What's a subnet?**
  - device interfaces with same subnet part of IP address
  - can physically reach each other *without intervening router*

Network consisting of 3 subnets:

- 223.1.1.1
- 223.1.1.2
- 223.1.1.3
- 223.1.1.4
- 223.1.2.1
- 223.1.2.2
- 223.1.3.1
- 223.1.3.2
- 223.1.3.27
IP addressing: CIDR

CIDR: Classless InterDomain Routing
- subnet portion of address of arbitrary length
- address format: `a.b.c.d/x`, where `x` is the number of bits in the subnet portion of address

```
11001000  00010111  00010000  00000000
subnet part

11001000  00010111  00010000  00000000
host part

200.23.16.0/23
```
DHCP: Dynamic Host Configuration Protocol

**goal:** allow host to *dynamically* obtain its IP address from network server when it joins network

- can renew its lease on address in use
- allows reuse of addresses (only hold address while connected/“on”)
- support for mobile users who want to join network (more shortly)

**DHCP overview:**

- host broadcasts “DHCP discover” msg [optional]
- DHCP server responds with “DHCP offer” msg [optional]
- host requests IP address: “DHCP request” msg
- DHCP server sends address: “DHCP ack” msg
**NAT: network address translation**

*motivation:* local network uses just one IP address as far as outside world is concerned:

- range of addresses not needed from ISP: just one IP address for all devices
- can change addresses of devices in local network without notifying outside world
- can change ISP without changing addresses of devices in local network
- devices inside local net not explicitly addressable, visible by outside world (a security plus)
Traceroute and ICMP

- source sends series of UDP segments to dest
  - first set has TTL = 1
  - second set has TTL = 2, etc.
  - unlikely port number
- when $n$th set of datagrams arrives to $n$th router:
  - router discards datagrams
  - and sends source ICMP messages (type 11, code 0)
  - ICMP messages includes name of router & IP address

- when ICMP messages arrives, source records RTTs

**stopping criteria:**
- UDP segment eventually arrives at destination host
- destination returns ICMP “port unreachable” message (type 3, code 3)
- source stops
IPv6: motivation

- *initial motivation*: 32-bit address space soon to be completely allocated.

- additional motivation:
  - header format helps speed processing/forwarding
  - header changes to facilitate QoS

**IPv6 datagram format:**
- fixed-length 40 byte header
- no fragmentation allowed
Tunneling

logical view:

physical view:

IPv4 tunnel connecting IPv6 routers
Software Defined Networking (SDN)

Logically-centralized control

API to the data plane (e.g., OpenFlow)

Smart, slow

Dumb, fast

Switches
Data-Plane: Simple Packet Handling

- Simple packet-handling rules
  - Pattern: match packet header bits
  - Actions: drop, forward, modify, send to controller
  - Priority: disambiguate overlapping patterns
  - Counters: #bytes and #packets

1. src=1.2.*.*, dest=3.4.5.* → drop
2. src = **.*.*.*, dest=3.4.*.* → forward(2)
3. src=10.1.2.3, dest=**.*.*.* → send to controller
A Link-State Routing Algorithm

Dijkstra’s algorithm

- net topology, link costs known to all nodes
  - accomplished via “link state broadcast”
  - all nodes have same info
- computes least cost paths from one node (‘source’) to all other nodes
  - gives forwarding table for that node
- iterative: after k iterations, know least cost path to k dest.’s

notation:

- \( c(x, y) \): link cost from node \( x \) to \( y \); \( = \infty \) if not direct neighbors
- \( D(v) \): current value of cost of path from source to dest. \( v \)
- \( p(v) \): predecessor node along path from source to \( v \)
- \( N' \): set of nodes whose least cost path definitively known
Distance vector algorithm

Bellman-Ford equation (dynamic programming)

let
\[ d_x(y) := \text{cost of least-cost path from } x \text{ to } y \]
then
\[ d_x(y) = \min_v \{ c(x,v) + d_v(y) \} \]

- cost from neighbor \( v \) to destination \( y \)
- cost to neighbor \( v \)

\( \min \) taken over all neighbors \( v \) of \( x \)
Comparison of LS and DV algorithms

message complexity
- **LS**: with n nodes, E links, $O(nE)$ msgs sent
- **DV**: exchange between neighbors only
  - convergence time varies

speed of convergence
- **LS**: $O(n^2)$ algorithm requires $O(nE)$ msgs
  - may have oscillations
- **DV**: convergence time varies
  - may be routing loops
  - count-to-infinity problem

robustness: what happens if router malfunctions?
- **LS**:
  - node can advertise incorrect *link* cost
  - each node computes only its own table
- **DV**:  
  - DV node can advertise incorrect *path* cost
  - each node’s table used by others
    - error propagate thru network
Hierarchical routing

- aggregate routers into regions, "autonomous systems" (AS)
- routers in same AS run same routing protocol
  - "intra-AS" routing protocol
  - routers in different AS can run different intra-AS routing protocol

**gateway router:**
- at "edge" of its own AS
- has link to router in another AS
Intra-AS Routing

- also known as *interior gateway protocols (IGP)*
- most common intra-AS routing protocols:
  - RIP: Routing Information Protocol
  - OSPF: Open Shortest Path First
  - IGRP: Interior Gateway Routing Protocol (Cisco proprietary)
Internet inter-AS routing: BGP

- **BGP (Border Gateway Protocol):** the de facto inter-domain routing protocol
  - “glue that holds the Internet together”

- **BGP provides each AS a means to:**
  - **eBGP:** obtain subnet reachability information from neighboring ASs.
  - **iBGP:** propagate reachability information to all AS-internal routers.
  - determine “good” routes to other networks based on reachability information and policy.

- allows subnet to advertise its existence to rest of Internet: “I am here”
A,B,C are **provider networks**

X,W,Y are customer (of provider networks)

X is dual-homed: attached to two networks

- X does not want to route from B via X to C
- .. so X will not advertise to B a route to C
BGP routing policy (2)

- A advertises path AW to B
- B advertises path BAW to X
- Should B advertise path BAW to C?
  - No way! B gets no “revenue” for routing CBAW since neither W nor C are B’s customers
  - B wants to force C to route to w via A
  - B wants to route only to/from its customers!
Why different Intra-, Inter-AS routing?

**policy:**
- inter-AS: admin wants control over how its traffic routed, who routes through its net.
- intra-AS: single admin, so no policy decisions needed

**scale:**
- hierarchical routing saves table size, reduced update traffic

**performance:**
- intra-AS: can focus on performance
- inter-AS: policy may dominate over performance
In-network duplication

- **flooding**: when node receives broadcast packet, sends copy to all neighbors
  - problems: cycles & broadcast storm
- **controlled flooding**: node only broadcasts pkt if it hasn’t broadcast same packet before
  - node keeps track of packet ids already broadcasted
  - or reverse path forwarding (RPF): only forward packet if it arrived on shortest path between node and source
- **spanning tree**:  
  - no redundant packets received by any node
Link layer services

- **framing, link access:**
  - encapsulate datagram into frame, adding header, trailer
  - channel access if shared medium
  - “MAC” addresses used in frame headers to identify source, dest
    - different from IP address!

- **reliable delivery between adjacent nodes**
  - we learned how to do this already (chapter 3)!
  - seldom used on low bit-error link (fiber, some twisted pair)
  - Used in wireless links: high error rates
    - Q: why both link-level and end-end reliability?
    - A: Reduce the frequency of end-end retrans
Parity checking

**single bit parity:**
- detect single bit errors

```
0111000110101011
```

d data bits  parity bit

**two-dimensional bit parity:**
- detect and correct single bit errors

```
<table>
<thead>
<tr>
<th>d_{1,1}</th>
<th>\ldots</th>
<th>d_{1,j}</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_{2,1}</td>
<td>\ldots</td>
<td>d_{2,j}</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>d_{i,1}</td>
<td>\ldots</td>
<td>d_{i,j}</td>
</tr>
<tr>
<td>d_{i+1,1}</td>
<td>\ldots</td>
<td>d_{i+1,j}</td>
</tr>
</tbody>
</table>
```

**row parity**

```
| d_{1, j+1} |
| d_{2, j+1} |
| \ldots |
| d_{i, j+1} |
| d_{i+1, j+1} |
```

**column parity**

```
101011
111100
011101
001010
```

**no errors**

```
101011
101100
011101
001010
```

**correctable single bit error**

```
101011
011100
011101
001010
```
MAC protocols: taxonomy

three broad classes:

- **channel partitioning**
  - divide channel into smaller “pieces” (time slots, frequency, code)
  - allocate piece to node for exclusive use

- **random access**
  - channel not divided, allow collisions
  - “recover” from collisions

- “taking turns”
  - nodes take turns, but nodes with more to send can take longer turns
Summary of MAC protocols

- **channel partitioning**, by time, frequency or code
  - Time Division, Frequency Division

- **random access** (dynamic),
  - ALOHA, S-ALOHA, CSMA, CSMA/CD
  - carrier sensing: easy in some technologies (wire), hard in others (wireless)
    - CSMA/CD used in Ethernet
    - CSMA/CA used in 802.11

- **taking turns**
  - polling from central site, token passing
  - bluetooth
MAC addresses and ARP

- **32-bit IP address:**
  - *network-layer* address for interface
  - used for layer 3 (network layer) forwarding

- **MAC (or LAN or physical or Ethernet) address:**
  - function: *used ‘locally’ to get frame from one interface to another physically-connected interface (same network, in IP-addressing sense)*
  - 48 bit MAC address (for most LANs) burned in NIC ROM, also sometimes software settable
  - e.g.: 1A-2F-BB-76-09-AD

  hexadecimal (base 16) notation
  (each “number” represents 4 bits)
**ARP: address resolution protocol**

**Question:** how to determine interface’s MAC address, knowing its IP address?

**ARP table:** each IP node (host, router) on LAN has table

- IP/MAC address mappings for some LAN nodes: `< IP address; MAC address; TTL>`
- TTL (Time To Live): time after which address mapping will be forgotten (typically 20 min)
Addressing: routing to another LAN

walkthrough: send datagram from A to B via R

- focus on addressing – at IP (datagram) and MAC layer (frame)
- assume A knows B’s IP address
- assume A knows IP address of first hop router, R (how?)
  - DHCP
- assume A knows R’s MAC address (how?)
  - ARP
Ethernet: physical topology

- **bus**: popular through mid 90s
  - all nodes in same collision domain (can collide with each other)

- **star**: prevails today
  - active *switch* in center
  - each “spoke” runs a (separate) Ethernet protocol (nodes do not collide with each other)
Ethernet: unreliable, connectionless

- **connectionless**: no handshaking between sending and receiving NICs
- **unreliable**: receiving NIC doesn't send acks or nacks to sending NIC
  - data in dropped frames recovered only if initial sender uses higher layer rdt (e.g., TCP), otherwise dropped data lost

- Ethernet's MAC protocol: unslotted *CSMA/CD with binary backoff*
Switch forwarding table

**Q:** how does switch know A’ reachable via interface 4, B’ reachable via interface 5?

- **A:** each switch has a switch table, each entry:
  - (MAC address of host, interface to reach host, time stamp)
  - looks like a routing table!

**Q:** how are entries created, maintained in switch table?

- something like a routing protocol?
Switches vs. routers

both are store-and-forward:

- **routers**: network-layer devices (examine network-layer headers)
- **switches**: link-layer devices (examine link-layer headers)

both have forwarding tables:

- **routers**: compute tables using routing algorithms, IP addresses
- **switches**: learn forwarding table using flooding, learning, MAC addresses
Data center networks

- rich interconnection among switches, racks:
  - increased throughput between racks (multiple routing paths possible)
  - increased reliability via redundancy