A closer look at network structure:

- **network edge:**
  - hosts: clients and servers
  - servers often in data centers
- **access networks, physical media:** wired, wireless communication links
- **network core:**
  - interconnected routers
  - network of networks
What’s the Internet: a service view

- **Infrastructure that provides services to applications:**
  - Web, VoIP, email, games, e-commerce, social nets, …

- **provides programming interface to apps**
  - hooks that allow sending and receiving app programs to “connect” to Internet
  - provides service options, analogous to postal service
The network core

- mesh of interconnected routers
- packet-switching: hosts break application-layer messages into packets
  - forward packets from one router to the next, across links on path from source to destination
  - each packet transmitted at full link capacity
Packet-switching: store-and-forward

- Takes $L/R$ seconds to transmit (push out) $L$-bit packet into link at $R$ bps.
- **Store and forward**: entire packet must arrive at router before it can be transmitted on next link.
- End-end delay = $2L/R$ (assuming zero propagation delay).

**One-hop numerical example**:
- $L = 7.5$ Mbits
- $R = 1.5$ Mbps
- One-hop transmission delay = 5 sec

more on delay shortly …
Two key network-core functions

*routing*: determines source-destination route taken by packets

- *routing algorithms*

*forwarding*: move packets from router’s input to appropriate router output
**Introduction**

**Alternative core: circuit switching**

end-end resources allocated to, reserved for “call” between source & dest:

- In diagram, each link has four circuits.
  - call gets 2\(^{nd}\) circuit in top link and 1\(^{st}\) circuit in right link.

- dedicated resources: no sharing
  - circuit-like (guaranteed) performance

- circuit segment idle if not used by call *(no sharing)*

- Commonly used in traditional telephone networks
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 .0004

Q: How did we get value 0.0004?
Q: What happens if 50 users?
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 it 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)
- $d_{\text{trans}} = \frac{L}{R}$

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

$d_{\text{trans}}$ and $d_{\text{prop}}$ very different
Throughput: Internet scenario

- per-connection end-end throughput: min($R_c, R_s, R/10$)
- in practice: $R_c$ or $R_s$ is often bottleneck

10 connections (fairly) share backbone bottleneck link $R$ bits/sec
Why layering?

dealing with complex systems:

- explicit structure allows identification, relationship of complex system’s pieces
  - layered *reference model* for discussion

- modularization eases maintenance, updating of system
  - change of implementation of layer’s service transparent to rest of system
  - e.g., change in gate procedure doesn’t affect rest of system

- layering considered harmful?
application: supporting network applications
- FTP, SMTP, HTTP

transport: 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”
Client-server architecture

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

clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other
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, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
Sockets

- process sends/receives messages to/from its socket (SW interface)
- 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
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…
App-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
- allows for interoperability
- e.g., HTTP (RFC2616), SMTP (RFC5321)

proprietary protocols:
- e.g., Skype
Web and HTTP

First, a review...

- web page consists of objects
- object can be HTML file, JPEG image, Java applet, audio file,...
- web page consists of base HTML-file which includes several referenced objects
- each object is addressable by a URL (Uniform Resource Locator), e.g.,

  www.someschool.edu/someDept/pic.gif

  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

PC running Firefox browser

server running Apache Web server

iphone running Safari browser
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 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

**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
HTTP request message

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

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

carriage return character
line-feed character
request line
(GET, POST, HEAD, PUT, DELETE commands)
header lines
carriage return, line feed at start of line indicates end of header lines
HTTP response message

status line (protocol version 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 data ...

data, e.g., requested HTML file

header lines
User-server state: cookies

many Web sites use cookies

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 always access Internet from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID
Web caches (proxy server)

**goal:** satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client
Caching example

possible solution: install cache

- suppose hit rate is 0.4

consequence

- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 70 msec)
- total avg delay = Internet delay + access delay + LAN delay = 0.6*(2 + 0.14 + 0.02) secs + 0.4*(0 + 0 + 0.02) = 1.3 secs
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

HTTP request msg
If-modified-since: <date>

HTTP response
HTTP/1.0
304 Not Modified

HTTP request msg
If-modified-since: <date>

HTTP response
HTTP/1.0 200 OK
<data>

object not modified before <date>

object modified after <date>
FTP: the file transfer protocol

- transfer file to/from remote host
- client/server model
  - client: side that initiates transfer (either to/from remote)
  - server: remote host
- ftp: RFC 959
- ftp server: port 21
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
- control connection: “out of band”
Electronic 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, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server
Electronic 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
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: domain name system

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

**Internet hosts, routers:**
- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., www.yahoo.com - 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: a distributed, hierarchical database**

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
Local DNS name server

- 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 example

- host at cis.poly.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 example

recursive query:
- puts burden of name resolution on contacted name server

Diagram:
- Requesting host: cis.poly.edu
- Local DNS server: dns.poly.edu
- Root DNS server
- TLD DNS server: dns.cs.umass.edu
- Authoritative DNS server: dns.cs.umass.edu
- gaia.cs.umass.edu

Application Layer 2-39
DNS records

DNS: distributed db 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 mailserver associated with **name**
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 two RRs into .com TLD server:
    (networkutopia.com, dns1.networkutopia.com, NS)
    (dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server type A record for www.networkuptopia.com; type MX record for networkutopia.com
Transport services and protocols

- provide **logical communication** between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP
Transport vs. network layer

- **network layer**: logical communication between hosts
- **transport layer**: logical communication between processes
  - relies on, enhances, network layer services

**household analogy:**

12 kids in Ann’s house sending letters to 12 kids in Bill’s house:
- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service
Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
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

Transport Layer 3-45
Connectionless demultiplexing

- **recall**: created socket has host-local port #:
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  ```

- **recall**: when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

  IP datagrams with same dest. port #, but different source IP addresses and/or source port numbers will be directed to same socket at dest.
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request
UDP: User Datagram Protocol [RFC 768]

- “no frills,” “bare bones” Internet transport protocol
- “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
- SNMP

reliable transfer over UDP:
- add reliability at application layer
- application-specific error recovery!
UDP: segment header

UDP segment format

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

32 bits

length, in bytes of UDP segment, including header

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired
**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

*Transport Layer 3-50*
Internet checksum: example

Example: add two 16-bit integers

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

Wraparound

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

Sum

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

Checksum

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result.
Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
**rdt1.0: 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)
udt_send(packet)
```

```
Wait for call from below

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

sender

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
rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?
- sender doesn’t know what happened at receiver!
- can’t just retransmit: possible duplicate

handling duplicates:
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds sequence number to each pkt
- receiver discards (doesn’t deliver up) duplicate pkt

stop and wait
sender sends one packet, then waits for receiver response
**rdt2.1: sender, handles garbled ACK/NAKs**

```
<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt_send(data)</td>
<td>sndpkt = make_pkt(0, data, checksum)</td>
</tr>
<tr>
<td>udt_send(sndpkt)</td>
<td>rdt_send(data)</td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt) &amp;&amp; (corrupt(rcvpkt)</td>
<td></td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt) &amp;&amp; notcorrupt(rcvpkt) &amp;&amp; isACK(rcvpkt)</td>
<td>udt_send(sndpkt)</td>
</tr>
<tr>
<td>Wait for call 0 from above</td>
<td>Wait for ACK or NAK 0</td>
</tr>
<tr>
<td>Wait for call 1 from above</td>
<td>Wait for ACK or NAK 1</td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt) &amp;&amp; notcorrupt(rcvpkt) &amp;&amp; isACK(rcvpkt)</td>
<td>rdt_send(data)</td>
</tr>
<tr>
<td>sndpkt = make_pkt(1, data, checksum)</td>
<td>udt_send(sndpkt)</td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt) &amp;&amp; (corrupt(rcvpkt)</td>
<td></td>
</tr>
</tbody>
</table>
```
rdt2.1: receiver, handles garbled ACK/NAKs

- When receiving a packet, check if it is corrupted and has the correct sequence number.
- If the packet is not corrupt and has the correct sequence number, extract the data and deliver it.
- If the packet is corrupt or has the wrong sequence number, make a new packet with a NAK and send it.
- The receiver waits for either an ACK or a NAK from the sender.
- If an ACK is received, extract the data and deliver it.
- If a NAK is received, send a new NAK.
rdt3.0: channels with errors and loss

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
**rdt3.0 sender**

- **Wait for call 0 from above**
  - `rdt_rcv(rcvpkt)`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_send(data)`
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))`
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 1)`
  - `stop_timer`

- **Wait for call 1 from above**
  - `rdt_rcv(rcvpkt)`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_send(data)`
  - `sndpkt = make_pkt(1, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 0))`
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)`
  - `stop_timer`

- **Wait for ACK0**
  - `timeout`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_send(data)`
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))`

- **Wait for ACK1**
  - `timeout`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_send(data)`
  - `sndpkt = make_pkt(1, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 0))`
Pipelined protocols: overview

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

**Selective Repeat:**
- sender can have up to N unacknowledged packets in the pipeline
- receiver sends *individual ack* for each packet
- sender maintains a timer for each unacknowledged packet
  - when the timer expires, retransmit only that unacknowledged packet
Go-Back-N: sender

- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed

- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window
GBN: sender extended FSM

- rdt_send(data)
- if (nextseqnum < base+N) {
  sndpkt[nextseqnum] = make_pkt(nextseqnum, data, chksum)
  udt_send(sndpkt[nextseqnum])
  if (base == nextseqnum)
    start_timer
    nextseqnum++
} else
  refuse_data(data)
- rdt_rcv(rcvpkt) && refuse_data(rcvpkt)
- timeout
- start_timer
- udt_send(sndpkt[base])
- udt_send(sndpkt[base+1])
- ...
- udt_send(sndpkt[nextseqnum-1])
- rdt_rcv(rcvpkt) &&
- notcorrupt(rcvpkt)
- base = getacknum(rcvpkt)+1
- If (base == nextseqnum)
  stop_timer
  else
  start_timer
GBN: receiver extended FSM

ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember *expectedseqnum*

- out-of-order pkt:
  - discard (don’t buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

Transport Layer 3-63
Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - $N$ consecutive seq #’s
  - limits seq #’s of sent, unACKed pkts
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Transport Layer 3-65
Selective repeat

sender

data from above:
- if next available seq # in window, send pkt

timeout(n):
- resend pkt n, restart timer

ACK(n) in [sendbase, sendbase+N]:
- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N, rcvbase-1]
- ACK(n)
otherwise:
- ignore
TCP segment structure

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum** (as in UDP)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number of data</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>head len</td>
<td>Length of header</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>receive window</td>
<td>Number of bytes receiver willing to accept</td>
</tr>
<tr>
<td>checksum</td>
<td>Internet checksum</td>
</tr>
<tr>
<td>options (variable length)</td>
<td>Options for variable length</td>
</tr>
<tr>
<td>application data</td>
<td>Application data</td>
</tr>
<tr>
<td>(variable length)</td>
<td></td>
</tr>
</tbody>
</table>
TCP seq. numbers, ACKs

**Sequence Numbers:**
- Byte stream “number” of first byte in segment’s data

**Acknowledgements:**
- Seq # of next byte expected from other side
- Cumulative ACK

**Q:** How receiver handles out-of-order segments
- **A:** TCP spec doesn’t say, - up to implementor
EstimatedRTT = (1 - \alpha)*EstimatedRTT + \alpha*SampleRTT

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

- **timeout interval**: $\text{EstimatedRTT}$ plus “safety margin”
  - large variation in $\text{EstimatedRTT}$ $\rightarrow$ larger safety margin
- estimate SampleRTT deviation from $\text{EstimatedRTT}$:
  \[
  \text{DevRTT} = (1-\beta)\times\text{DevRTT} + \beta\times|\text{SampleRTT}-\text{EstimatedRTT}|
  \]
  (typically, $\beta = 0.25$)

$\text{TimeoutInterval} = \text{EstimatedRTT} + 4\times\text{DevRTT}$
TCP sender events:

**data rcvd from app:**
- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: TimeOutInterval

**timeout:**
- retransmit segment that caused timeout
- restart timer

**ack rcvd:**
- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments
### TCP ACK generation

**[RFC 1122, RFC 2581]**

<table>
<thead>
<tr>
<th><strong>event at receiver</strong></th>
<th><strong>TCP receiver action</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>arrival of out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>immediately send <em>duplicate ACK</em>, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>arrival of segment that partially or completely fills gap</td>
<td>immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
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

Flow control: receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast.
TCP flow control

- receiver “advertises” free buffer space by including \texttt{rwnd} value in TCP header of receiver-to-sender segments
  - \texttt{RcvBuffer} size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust \texttt{RcvBuffer}
- sender limits amount of unacked (“in-flight”) data to receiver’s \texttt{rwnd} value
- guarantees receive buffer will not overflow
Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

```
Socket clientSocket = newSocket("hostname","port number");

Socket connectionSocket = welcomeSocket.accept();
```
TCP 3-way handshake

**client state**

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

- **SYNSENT**
  - received SYNACK(x)
  - ACKbit=1, ACKnum=x+1
  - indicates server is live
  - send ACK for SYNACK
  - this segment may contain client-to-server data

- **ESTAB**
  - SYNbit=1, Seq=x
  - ACKbit=1, ACKnum=y+1

**server state**

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

- **SYNSENT**
  - SYNbit=1, Seq=y

- **ESTAB**
  - received ACK(y)
  - indicates client is live
TCP: closing a connection

**client state**

- ESTAB
- FIN_WAIT_1
  - clientSocket.close()
  - can no longer send but can receive data
  - wait for server close
- FIN_WAIT_2
  - timed wait for 2*max segment lifetime
- TIMED_WAIT
- CLOSED

**server state**

- ESTAB
- CLOSE_WAIT
  - can still send data
- LAST_ACK
  - can no longer send data
- CLOSED
Approaches towards congestion control

two broad approaches towards congestion control:

end-end congestion control:
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

network-assisted congestion control:
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at
TCP congestion control: additive increase multiplicative decrease

- **approach**: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **additive increase**: increase $cwnd$ by 1 MSS every RTT until loss detected
  - **multiplicative decrease**: cut $cwnd$ in half after loss

AIMD saw tooth behavior: probing for bandwidth

![Graph showing AIMD saw tooth behavior](image)
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:
  - $cwnd$ set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - $cwnd$ is cut in half window then grows linearly

- TCP Tahoe always sets $cwnd$ to 1 (timeout or 3 duplicate acks)
TCP: switching from slow start to CA

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

Implementation:
- variable \textit{ssthresh}
- on loss event, \textit{ssthresh} is set to 1/2 of \textit{cwnd} just before loss event
Network layer

- transport segment from sending to receiving host
- on sending side encapsulates segments into datagrams
- on receiving side, delivers segments to transport layer
- network layer protocols in every host, router
- router examines header fields in all IP datagrams passing through it
Two key network-layer functions

- **forwarding**: move packets from router’s input to appropriate router output

- **routing**: determine route taken by packets from source to dest.
  - **routing algorithms**

**analogy:**

- **routing**: process of planning trip from source to dest
- **forwarding**: process of getting through single interchange
Datagram networks

- no call setup at network layer
- routers: no state about end-to-end connections
  - no network-level concept of “connection”
- packets forwarded using destination host address
Datagram forwarding table

4 billion IP addresses, so rather than list individual destination address list range of addresses (aggregate table entries)

**Local forwarding table**

<table>
<thead>
<tr>
<th>dest address</th>
<th>output link</th>
</tr>
</thead>
<tbody>
<tr>
<td>address-range 1</td>
<td>3</td>
</tr>
<tr>
<td>address-range 2</td>
<td>2</td>
</tr>
<tr>
<td>address-range 3</td>
<td>2</td>
</tr>
<tr>
<td>address-range 4</td>
<td>1</td>
</tr>
</tbody>
</table>

IP destination address in arriving packet’s header
### 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 1010001 1010001
- 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*
**Router architecture overview**

**two key router functions:**
- run routing algorithms/protocol (RIP, OSPF, BGP)
- *forwarding* datagrams from incoming to outgoing link

---

**Diagram:**
- **Routing processor**
- **High-speed switching fabric**
- **Router input ports**
- **Router output ports**
- **Forwarding tables computed, pushed to input ports**
- **Routing, management control plane (software)**
- **Forwarding data plane (hardware)**

---

Network Layer 4-90
**Input port functions**

Physical layer: bit-level reception

Data link layer: e.g., Ethernet

- Decentralized switching:
  - Given datagram dest., lookup output port using forwarding table in input port memory ("match plus action")
  - Goal: complete input port processing at 'line speed'
  - Queuing: if datagrams arrive faster than forwarding rate into switch fabric

Network Layer 4-91
Switching fabrics

- transfer packet from input buffer to appropriate output buffer
- switching rate: rate at which packets can be transferred from inputs to outputs
  - often measured as multiple of input/output line rate
  - N inputs: switching rate N times line rate desirable
- three types of switching fabrics
Output ports

- **buffering** required when datagrams arrive from fabric faster than the transmission rate
- **scheduling discipline** chooses among queued datagrams for transmission
Output port queueing

- buffering when arrival rate via switch exceeds output line speed
- queueing (delay) and loss due to output port buffer overflow!
Input port queuing

- fabric slower than input ports combined -> queueing may occur at input queues
  - queueing delay and loss due to input buffer overflow!
- Head-of-the-Line (HOL) blocking: queued datagram at front of queue prevents others in queue from moving forward

output port contention:
only one red datagram can be transferred.
*lower red packet is blocked*
IP datagram format

- IP protocol version number
- Header length (# of 32-bit words)
- "Type" of data
- 16-bit identifier
- Flags
- Time to live
- Upper layer protocol
- Total data length (bytes)
- Type of service
- Fragment offset
- Upper layer protocol to deliver payload to
- 32-bit destination IP address
- Options (if any)
- Data (variable length, typically a TCP or UDP segment)
- How much overhead?
  - 20 bytes of TCP
  - 20 bytes of IP
  - = 40 bytes + app layer overhead
- E.g. timestamp record route taken, specify list of routers to visit.

Network Layer 4-96
IP fragmentation, reassembly

- network links have MTU (max. transfer size) - largest possible link-level frame
  - different link types, different MTUs
- large IP datagram divided ("fragmented") within net
  - one datagram becomes several datagrams
  - "reassembled" only at final destination
  - IP header bits used to identify, order related fragments

fragmentation: in: one large datagram
out: 3 smaller datagrams

reassembly
**IP fragmentation, reassembly**

**example:**
- 4000 byte datagram
- MTU = 1500 bytes

1480 bytes in data field

offset = \( \frac{1480}{8} \)
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
```

Network Layer 4-99
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*
IP addressing: CIDR

CIDR: Classless InterDomain Routing

- subnet portion of address of arbitrary length
- address format: $a.b.c.d/x$, where $x$ is # bits in subnet portion of address

```
11001000  00010111  00010000  00000000
```

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
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
Routing algorithm classification

Q: *global* or *decentralized* information?

**global:**
- all routers have complete topology, link cost info
- “link state” algorithms

**decentralized:**
- router knows physically-connected neighbors, link costs to neighbors
- iterative process of computation, exchange of info with neighbors
- “distance vector” algorithms

Q: *static* or *dynamic*?

**static:**
- routes change slowly over time

**dynamic:**
- routes change more quickly
  - periodic update
  - in response to link cost changes
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
Dijsktra’s Algorithm

1 **Initialization:**
2 \[ N' = \{ u \} \]
3 for all nodes \( v \)
4 if \( v \) adjacent to \( u \)
5 then \( D(v) = c(u,v) \)
6 else \( D(v) = \infty \)

8 **Loop**
9 find \( w \) not in \( N' \) such that \( D(w) \) is a minimum
10 add \( w \) to \( N' \)
11 update \( D(v) \) for all \( v \) adjacent to \( w \) and not in \( N' \):
12 \[ D(v) = \min( D(v), D(w) + c(w,v) ) \]
13 /* new cost to \( v \) is either old cost to \( v \) or known shortest path cost to \( w \) plus cost from \( w \) to \( v \) */
15 until all nodes in \( N' \)
Dijkstra’s algorithm: another example

<table>
<thead>
<tr>
<th>Step</th>
<th>N'</th>
<th>D(v),p(v)</th>
<th>D(w),p(w)</th>
<th>D(x),p(x)</th>
<th>D(y),p(y)</th>
<th>D(z),p(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>u</td>
<td>2,u</td>
<td>5,u</td>
<td>1,u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ux</td>
<td>2,u</td>
<td>4,x</td>
<td>2,x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>uxy</td>
<td>2,u</td>
<td>3,y</td>
<td>4,y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>uxyv</td>
<td>2,u</td>
<td>3,y</td>
<td>4,y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>uxyvw</td>
<td></td>
<td>4,y</td>
<td>4,y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>uxyvwz</td>
<td></td>
<td></td>
<td>4,y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm: example (2)

resulting shortest-path tree from u:

resulting forwarding table in u:

<table>
<thead>
<tr>
<th>destination</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>(u,v)</td>
</tr>
<tr>
<td>x</td>
<td>(u,x)</td>
</tr>
<tr>
<td>y</td>
<td>(u,x)</td>
</tr>
<tr>
<td>w</td>
<td>(u,x)</td>
</tr>
<tr>
<td>z</td>
<td>(u,x)</td>
</tr>
</tbody>
</table>
**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 \( x \)
Distance vector algorithm

- $D_x(y) =$ estimate of least cost from $x$ to $y$
  - $x$ maintains distance vector $D_x = [D_x(y): y \in N]$

- node $x$:
  - knows cost to each neighbor $v$: $c(x,v)$
  - maintains its neighbors’ distance vectors. For each neighbor $v$, $x$ maintains $D_v = [D_v(y): y \in N]$
key idea:

- from time-to-time, each node sends its own distance vector estimate to neighbors
- when \( x \) receives new DV estimate from neighbor, it updates its own DV using B-F equation:

\[
D_x(y) \leftarrow \min_v \{ c(x,v) + D_v(y) \} \quad \text{for each node } y \in N
\]

- under minor, natural conditions, the estimate \( D_x(y) \) converge to the actual least cost \( d_x(y) \)
Distance vector algorithm

**iterative, asynchronous:**
- each local iteration caused by:
  - local link cost change
  - DV update message from neighbor

**distributed:**
- each node notifies neighbors *only* when its DV changes
  - neighbors then notify their neighbors if necessary

**each node:**
- wait for (change in local link cost or msg from neighbor)
- recompute estimates
- if DV to any dest has changed, *notify* neighbors
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
Interconnected ASes

- Forwarding table configured by both intra- and inter-AS routing algorithm
  - intra-AS sets entries for internal dests
  - inter-AS & intra-AS sets entries for external dests
Inter-AS tasks

- Suppose router in AS1 receives datagram destined outside of AS1:
  - Router should forward packet to gateway router, but which one?

AS1 must:
1. Learn which dests are reachable through AS2, which through AS3
2. Propagate this reachability info to all routers in AS1

*job of inter-AS routing!*
RIP (Routing Information Protocol)

- included in BSD-UNIX distribution in 1982
- distance vector algorithm
  - distance metric: # hops (max = 15 hops), each link has cost 1
  - DVs exchanged with neighbors every 30 sec in response message (aka advertisement)
  - each advertisement: list of up to 25 destination subnets (in IP addressing sense)

from router A to destination subnets:

<table>
<thead>
<tr>
<th>subnet</th>
<th>hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>1</td>
</tr>
<tr>
<td>v</td>
<td>2</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>
RIP: link failure, recovery

if no advertisement heard after 180 sec -->
neighbor/link declared dead

- routes via neighbor invalidated
- new advertisements sent to neighbors
- neighbors in turn send out new advertisements (if tables changed)
OSPF (Open Shortest Path First)

- “open”: publicly available
- uses link state algorithm
  - LS packet dissemination
  - topology map at each node
  - route computation using Dijkstra’s algorithm
- OSPF advertisement carries one entry per neighbor
- advertisements flooded to entire AS
  - carried in OSPF messages directly over IP (rather than TCP or UDP)
Hierarchical OSPF

- **two-level hierarchy:** local area, backbone.
  - link-state advertisements only in area
  - each nodes has detailed area topology; only know direction (shortest path) to nets in other areas.
- **area border routers:** “summarize” distances to nets in own area, advertise to other Area Border routers.
- **backbone routers:** run OSPF routing limited to backbone.
- **boundary routers:** connect to other AS’ s.
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.
**BGP basics**

- **BGP session:** two BGP routers ("peers") exchange BGP messages:
  - advertising *paths* to different destination network prefixes
  - exchanged over TCP connections

- **when AS3 advertises a prefix to AS1:**
  - AS3 *promises* it will forward datagrams towards that prefix
  - AS3 can aggregate prefixes in its advertisement
**BGP basics: distributing path information**

- Using eBGP session between 3a and 1c, AS3 sends prefix reachability info to AS1.
  - 1c can then use iBGP to distribute new prefix info to all routers in AS1
  - 1b can then re-advertise new reachability info to AS2 over 1b-to-2a eBGP session

- When router learns of new prefix, it creates entry for prefix in its forwarding table.
Path attributes and BGP routes

- advertised prefix includes BGP attributes
  - prefix + attributes = “route”
- two important attributes:
  - **AS-PATH**: contains ASs through which prefix advertisement has passed: e.g., AS 67, AS 17
  - **NEXT-HOP**: indicates specific internal-AS router to next-hop AS. (may be multiple links from current AS to next-hop-AS)
- gateway router receiving route advertisement uses import policy to accept/decline
  - e.g., never route through AS x
  - *policy-based* routing
BGP route selection

- router may learn about more than 1 route to destination AS, selects route based on:
  1. local preference value attribute: policy decision
  2. shortest AS-PATH
  3. closest NEXT-HOP router: hot potato routing
  4. additional criteria
Link layer services

- **framing, link access:**
  - encapsulate datagram into frame, adding header, tailer
  - 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)
  - wireless links: high error rates
    - Q: why both link-level and end-end reliability?
Link layer services (more)

- **flow control:**
  - pacing between adjacent sending and receiving nodes

- **error detection:**
  - errors caused by signal attenuation, noise.
  - receiver detects presence of errors:
    - signals sender for retransmission or drops frame

- **error correction:**
  - receiver identifies *and corrects* bit error(s) without resorting to retransmission

- **half-duplex and full-duplex**
  - with half duplex, nodes at both ends of link can transmit, but not at same time
**Parity checking**

**single bit parity:**
- detect single bit errors
- even parity and odd parity scheme

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

![Diagram showing parity checking](attachment://parity_diagram.png)

- **0111000110101011**

**Examples:**
- **1010111**
- **1111000**
- **0111011**
- **0010101**

**No errors**

**Correctable single bit error**

Link Layer 5-128
Cyclic redundancy check

- more powerful error-detection coding
- view data bits, \( D \), as a binary number
- choose \( r+1 \) bit pattern (generator), \( G \)
- goal: choose \( r \) CRC bits, \( R \), such that
  - \( <D,R> \) exactly divisible by \( G \) (modulo 2)
  - receiver knows \( G \), divides \( <D,R> \) by \( G \). If non-zero remainder: error detected!
  - can detect all burst errors less than \( r+1 \) bits
- widely used in practice (Ethernet, 802.11 WiFi, ATM)

\[
D: \text{data bits to be sent} \quad R: \text{CRC bits}
\]

\[D \times 2^r \text{ XOR } R\]

\text{bit pattern}
CRC example

want:
\[ D \cdot 2^r \text{ XOR } R = nG \]
equivalently:
\[ D \cdot 2^r = nG \text{ XOR } R \]
equivalently:
If we divide \( D \cdot 2^r \) by \( G \), want remainder \( R \) to satisfy:

\[ R = \text{ remainder}\left[ \frac{D \cdot 2^r}{G} \right] \]
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
Channel partitioning MAC protocols: TDMA

TDMA: time division multiple access

- access to channel in "rounds"
- each station gets fixed length slot (length = pkt trans time) in each round
- unused slots go idle
- example: 6-station LAN, 1,3,4 have pkt, slots 2,5,6 idle
**Channel partitioning MAC protocols: FDMA**

**FDMA: frequency division multiple access**
- channel spectrum divided into frequency bands
- each station assigned fixed frequency band
- unused transmission time in frequency bands go idle
- example: 6-station LAN, 1,3,4 have pkt, frequency bands 2,5,6 idle
**Slotted ALOHA**

**assumptions:**
- all frames same size
- time divided into equal size slots (time to transmit 1 frame)
- nodes start to transmit only slot beginning
- nodes are synchronized
- if 2 or more nodes transmit in slot, all nodes detect collision

**operation:**
- when node obtains fresh frame, transmits in next slot
  - *if no collision:* node can send new frame in next slot
  - *if collision:* node retransmits frame in each subsequent slot with prob. p until success
Pros:
- single active node can continuously transmit at full rate of channel
- highly decentralized: only slots in nodes need to be in sync
- simple

Cons:
- collisions, wasting slots
- idle slots
- nodes may be able to detect collision in less than time to transmit packet
- clock synchronization
Pure (unslotted) ALOHA

- unslotted Aloha: simpler, no synchronization
- when frame first arrives
  - transmit immediately
- collision probability increases:
  - frame sent at $t_0$ collides with other frames sent in $[t_0-1,t_0+1]$
CSMA (carrier sense multiple access)

**CSMA:** listen before transmit:
- if channel sensed idle: transmit entire frame
  - if channel sensed busy, defer transmission
- human analogy: don’t interrupt others!
CSMA/CD (collision detection)

**CSMA/CD:** carrier sensing, deferral as in CSMA
- collisions *detected* within short time
- colliding transmissions aborted, reducing channel wastage

- collision detection:
  - easy in wired LANs: measure signal strengths, compare transmitted, received signals
  - difficult in wireless LANs: received signal strength overwhelmed by local transmission strength

- human analogy: the polite conversationalist
“Taking turns” MAC protocols

**polling:**
- master node “invites” slave nodes to transmit in turn
- typically used with “dumb” slave devices
- concerns:
  - polling overhead
  - latency
  - single point of failure (master)
"Taking turns" MAC protocols

**token passing:**

- control *token* passed from one node to next sequentially.
- token message
- concerns:
  - token overhead
  - latency
  - single point of failure (token)
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)
ARP protocol: same LAN

- A wants to send datagram to B
  - B’s MAC address not in A’s ARP table.
- A broadcasts ARP query packet, containing B's IP address
  - dest MAC address = FF-FF-FF-FF-FF-FF
  - all nodes on LAN receive ARP query
- B receives ARP packet, replies to A with its (B's) MAC address
  - frame sent to A’s MAC address (unicast)

- A caches (saves) IP-to-MAC address pair in its ARP table until information becomes old (times out)
  - soft state: information that times out (goes away) unless refreshed
- ARP is “plug-and-play”:
  - nodes create their ARP tables without intervention from net administrator
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?)
- assume A knows R’s MAC address (how?)
Ethernet

“dominant” wired LAN technology:

- cheap $20 for NIC
- first widely used LAN technology
- simpler, cheaper than token LANs and ATM
- kept up with speed race: 10 Mbps – 10 Gbps

Metcalfe’s Ethernet sketch
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)

*bus*: coaxial cable

*star*:

---

Link Layer 5-146
**Ethernet frame structure**

Sending adapter encapsulates IP datagram (or other network layer protocol packet) in Ethernet frame.

**Preamble:**
- 7 bytes with pattern 10101010 followed by one byte with pattern 10101011
- Used to synchronize receiver, sender clock rates

<table>
<thead>
<tr>
<th>preamble</th>
<th>dest. address</th>
<th>source address</th>
<th>data (payload)</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>type</td>
<td></td>
</tr>
</tbody>
</table>
Ethernet frame structure (more)

- **Addresses**: 6 byte source, destination MAC addresses
  - if adapter receives frame with matching destination address, or with broadcast address (e.g., ARP packet), it passes data in frame to network layer protocol
  - otherwise, adapter discards frame
- **Type**: indicates higher layer protocol (mostly IP but others possible, e.g., Novell IPX, AppleTalk)
- **CRC**: cyclic redundancy check at receiver
  - error detected: frame is dropped
Ethernet switch

- link-layer device: takes an active role
  - store, forward Ethernet frames
  - examine incoming frame’s MAC address, selectively forward frame to one-or-more outgoing links when frame is to be forwarded on segment, uses CSMA/CD to access segment

- transparent
  - hosts are unaware of presence of switches

- plug-and-play, self-learning
  - switches do not need to be configured
Switch forwarding table

**Q:** how does switch know A’s reachable via interface 4, B’s 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?
Switch: frame filtering/forwarding

when frame received at switch:

1. record incoming link, MAC address of sending host
2. index switch table using MAC destination address
3. if entry found for destination
   then {
     if destination on segment from which frame arrived
     then drop frame
     else forward frame on interface indicated by entry
   }
else flood /* forward on all interfaces except arriving interface */
Interconnecting switches

- switches can be connected together

Q: sending from A to G - how does S₁ know to forward frame destined to F via S₄ and S₃?

A: self learning! (works exactly the same as in single-switch case!)
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