**Administrivia**

- **Project 3 due Friday noon**
  - For longer extensions, please email cs140-staff
- **Stop by “working office hours” this week**
  - Today (Monday) 4-6pm in Gates 498
  - Wednesday 3:30-5:30pm in Gates 167
- **Project 4 goes out at end of week**
- **This Friday will have section on project 4**

**Networks**

- **What is a network?**
  - A system of lines/channels that interconnect
  - E.g., railroad, highway, plumbing, communication, telephone, computer
- **What is a computer network?**
  - A form of communication network—moves information
  - Nodes are general-purpose computers
- **Computer networks are particularly interesting**
  - You can program the nodes
  - Very easy to innovate and develop new uses of network
  - Contrast: Telephone network—can’t program most phones, need FCC approval for new devices, etc.

**Physical connectivity**

- **Computers send bits over physical links**
  - E.g., Coax, twisted pair, fiber, radio, …
  - Bits may be encoded as multiple lower-level “chips”
- **Two categories of physical links**
  - Point-to-point networks (e.g., fiber, twisted pair):
  - Shared transmission medium networks (e.g., coax, radio):
    - Any message can be seen by all nodes
    - Allows broadcast/multicast, but introduces contention
- **One implication: speed of light matters!**
  - ~ 300,000 km/sec in a vacuum, slower in fiber
  - SF $\geq \sim 15$ msec $\rightarrow$ NYC Moore’s law does not apply!

**Indirect connectivity**

- Rarely have direct physical connection to destination
- Instead, communications usually “hop” through multiple devices
  - Allows links and devices to be shared for multiple purposes
  - Must determine which bits are part of which messages intended for which destinations
- **Circuit switched** networks provide virtual links
  - E.g., old telephone network when numbers dialed
- **Packet switched** networks
  - Pack a bunch of bytes together intended for same destination
  - Slap a header on packet describing where it should go
  - Most networks today are packet switched

**Bandwidth-delay**

- Network delay over WAN will never improve much
- But **throughput** (bits/sec) is constantly improving
- Can view network as a pipe
  - For full utilization want $\# \text{bytes in flight} \geq \text{bandwidth} \times \text{delay}$
  (But don’t want to overload the network, either)
- **What if protocol doesn’t involve bulk transfer?**
  - E.g., ping-pong protocol will have poor throughput
- **Another implication:** Concurrency & response time critical for good network utilization

**Example: Ethernet**

- Originally designed for shared medium (coax)
  - Medium Access Control (MAC) protocol governs access to coax
  - E.g., don’t transmit when someone else is
  - If you collide, used randomized backoff and try again
- **Vendors give each device a unique 48-bit MAC address**
  - Specifies which node should receive a packet
    - 64 48 48 16 32
    - Preamble Dest addr Src addr Type Body CRC
  - Packet format:
    - Preamble helps device recognize start of packet
    - CRC allows card to ignore corrupted packets
    - Body up to 1,500 bytes for same destination
    - All other fields must be set by sender’s OS
    (NIC cards tell the OS what the card’s MAC address is, Special addresses used for broadcast/multicast)
Why Ethernet is insufficient

- **Ethernet Limits**
  - 2,500m diameter
  - 100 nodes
- **Can bridge multiple Ethereports**
  - First time you see destination address, send packet to all segments
  - Then learn where devices are, and avoid forwarding useless packets
- **A switch is like a bridge with n > 2 ports**
  - Widely used within organizations
  - But could never scale to the size of the Internet
- **Moreover, need to communicate across networks**
  - E.g., laptop w. DSL or wireless contacting server w. Ethernet

Internet Protocol (IP)

- **IP used to connect multiple networks**
  - Runs over a variety of physical networks
  - Most computers today speak IP
- **Every host has a unique 4-byte IP address**
  - (Or at least thinks it has, when there is address shortage)
  - E.g., www.ietf.org → 132.151.6.21
- **Packets are routed based on destination IP address**
  - Address space is structured to make routing practical at global scale
  - E.g., 171.66.*.* goes to Stanford
  - So packets need IP addresses in addition to MAC addresses

Layering

- **Stick packets inside packets**
  - E.g., an Ethernet packet may encapsulate an IP packet
  - An IP router forwards a packet from one Ethernet to another, creating a new Ethernet packet containing the same IP packet
  - In principle, an inner layer should not depend on outer layers
  - E.g., IP packets should be independent of Ethernet
  - Annoyingly, TCP (next) has a checksum that violates this

ARP [RFC826]

- **When forwarding an IP packet from Ethernet A to Ethernet B, what MAC address should you use?**
  - If destination host physically connected, use its MAC address
  - Otherwise, use MAC address of next router (given IP address)
- **Must map IP addresses into physical addresses**
- **ARP – address resolution protocol**
  - Broadcast request for MAC address of IP address
  - Everybody learns the requesting node’s MAC address
  - (since broadcast request has requester’s MAC and IP addresses)
  - Target machine responds with its MAC address
- **OS keeps ARP cache w. IP→MAC addr. mappings**
  - Periodically discard entries that have not been refreshed
  - E.g., run “arp -a” on Unix to see contents of ARP cache

ARP Ethernet packet format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware type</td>
<td>0–3</td>
<td>1 = Ethernet</td>
</tr>
<tr>
<td>HLen = 48</td>
<td>8–16</td>
<td>Length of hardware address</td>
</tr>
<tr>
<td>ProtocolType = 0x0800</td>
<td>16–23</td>
<td>Protocol type</td>
</tr>
<tr>
<td>SourceHardwareAddr (0–3)</td>
<td>24–27</td>
<td>Source hardware address</td>
</tr>
<tr>
<td>SourceHardwareAddr (4–5)</td>
<td>28–31</td>
<td>Source hardware address</td>
</tr>
<tr>
<td>SourceProtocolAddr (0–3)</td>
<td>32–35</td>
<td>Source protocol address</td>
</tr>
<tr>
<td>SourceProtocolAddr (2–3)</td>
<td>36–39</td>
<td>Source protocol address</td>
</tr>
<tr>
<td>TargetHardwareAddr (0–1)</td>
<td>40–43</td>
<td>Target hardware address</td>
</tr>
<tr>
<td>TargetHardwareAddr (2–5)</td>
<td>44–47</td>
<td>Target hardware address</td>
</tr>
<tr>
<td>TargetProtocolAddr (0–3)</td>
<td>48–51</td>
<td>Target protocol address</td>
</tr>
<tr>
<td>TargetProtocolAddr (4–5)</td>
<td>52–55</td>
<td>Target protocol address</td>
</tr>
</tbody>
</table>

[Hardware address = MAC address]

Inter-process communication

- **Want abstraction of inter-process (not just inter-node) communication**
- **Solution: Encapsulate other protocols within IP**
**TCP**

- Want to save network from congestive collapse
  - Packet loss usually means congestion, so back off exponentially
- Several types of error can affect packet delivery
  - Bit errors (e.g., electrical interference, cosmic rays)
  - Packet loss (packets dropped when queues fill on overload)
  - Link and node failure
- In addition, properly delivered frames can be delayed, reordered, even duplicated
- How much should OS expose to application
  - Some failures cannot be masked (e.g., server dead)
  - Others can be (e.g., retransmit lost packet)
  - But masking errors may be wrong for some applications (e.g., old audio packet no longer interesting if too late to play)
- Many more failure modes on net than w. local IPC

**UDP and TCP**

- UDP and TCP most popular protocols on IP
  - Both use 16-bit port number as well as 32-bit IP address
  - Applications bind a port & receive traffic to that port
- UDP – unreliable datagram protocol
  - Exposes packet-switched nature of Internet
  - Sent packets may be dropped, reordered, even duplicated (but generally not corrupted)
- TCP – transmission control protocol
  - Provides illusion of a reliable “pipe” between two processes on two different machines
  - Masks lost & reordered packets so apps don’t have to worry
  - Handles congestion & flow control

**Unreliability of IP**

- Network does not deliver packets reliably
  - May drop packets, reorder packets, delay packets
  - May even corrupt packets, or duplicate them
- How to implement reliable TCP on top of IP network?
  - Note: This is entirely the job of the OS at the end nodes
- Straw man: Wait for ack for each packet
  - Send a packet, wait for acknowledgment, send next packet
  - If no ack, timeout and try again
- Problems:
  - Low performance over high-delay network (bandwidth is one packet per round-trip time)
  - Possible congestive collapse of network (if everyone keeps retransmitting when network overloaded)

**Uses of TCP**

- Most applications use TCP
  - Easier interface to program to (reliability)
  - Automatically avoids congestion (don’t need to worry about taking down network)
- Servers typically listen on well-known ports
  - SSH: 22
  - Email: 25
  - Finger: 79
  - Web / HTTP: 80
- Example: Interacting with www.stanford.edu
  - Browser resolves IP address of www.stanford.edu (171.67.216.15)
  - Browser connects to TCP port 80 on 171.67.216.15
  - Over TCP connection, browser requests and gets home page
- A little bit about TCP
  - Want to save network from congestive collapse
    - Packet loss usually means congestion, so back off exponentially
  - Want multiple outstanding packets at a time
    - Get transmit rate up to $n$-packet window per round-trip
  - Must figure out appropriate value of $n$ for network
    - Slowly increase transmission by one packet per acked window
    - When a packet is lost, cut window size in half
  - Connection set up and tear down complicated
    - Sender never knows when last packet might be lost
    - Must keep state around for a while after close
  - Lots more hacks required for good performance
    - Initially ramp $n$ up faster (but too fast caused collapse in 1986, TCP had to be changed)
    - Fast retransmit when single packet lost

**Lots of OS issues for TCP**

- Have to track unacknowledged data
  - Keep a copy around until recipient acknowledges it
  - Keep timer around to retransmit if no ack
  - Receiver must keep out of order segments & reassemble
- When to wake process receiving data?
  - E.g., sender calls write (fd, message, 8000);
    - First TCP segment arrives, but is only 512 bytes
    - Could wake recipient, but useless w/o full message
    - TCP sets “PUSH” bit at end of 8000 byte write data
  - When to send short segment, vs. wait for more data
    - Usually send only one unacked short segment
    - But bad for some apps, so provide NODELAY option
- Must ack received segments very quickly
  - Otherwise, effectively increases RTT, decreasing bandwidth
OS interface to TCP/IP

- What interface should OS provide to TCP/IP?
- Inspired by pipes (int pipe (int fds[2]);)
  - Allow Inter-process communication on one machine
  - Writes to fds[1] will be read on fds[0]
  - Can give each file descriptor to a different process (w. fork)
- Idea: Provide similar abstraction across machines
  - Write data on one machine, read it on the other
  - Allows processes to communicate over the network
- Complications across machines
  - How do you set up the file descriptors between processes?
  - How do you deal with failure?
  - How do you get good performance?

Sockets

- Abstraction for communication between machines
- Datagram sockets: Unreliable message delivery
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: send/recv
- Stream sockets: Bi-directional pipes
  - With IP, gives you TCP
  - Bytes written on one end read on the other
  - Reads may not return full amount requested—must re-read

Socket naming

- TCP & UDP name communication endpoints by
  - 32-bit IP address specifies machine
  - 16-bit TCP/UDP port number demultiplexes within host
- A connection is thus named by 5 components
  - Protocol (TCP), local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP
- OS keeps connection state in protocol control block (PCB) structure
  - Keep all PCB’s in a hash table
  - When packet arrives (if destination IP address belongs to host), use 5-tuple to find PCB and determine what to do with packet

System calls for using TCP

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket – make socket</td>
<td>socket – make socket</td>
</tr>
<tr>
<td>bind – assign address</td>
<td>bind* – assign address</td>
</tr>
<tr>
<td>listen – listen for clients</td>
<td>listen – listen for clients</td>
</tr>
<tr>
<td>connect – connect to listening socket</td>
<td>connect – connect to listening socket</td>
</tr>
<tr>
<td>accept – accept connection</td>
<td>accept – accept connection</td>
</tr>
</tbody>
</table>

*This call to bind is optional; connect can choose address & port.

Client interface

```c
struct sockaddr_in {
    short sin_family; /* = AF_INET */
    u_short sin_port; /* = htons (PORT) */
    struct in_addr sin_addr;
    char sin_zero[8];
} sin;

int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (13); /* daytime port */
sin.sin_addr.s_addr = htonl (IP_ADDRESS);
connect (s, (sockaddr *) &sin, sizeof (sin));
```

Server interface

```c
struct sockaddr_in sin;
int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (9999);
sin.sin_addr.s_addr = htonl (INADDR_ANY);
bind (s, (struct sockaddr *) &sin, sizeof (sin));
listen (s, 5);

for (;;) {
    socklen_t len = sizeof (sin);
    int cfd = accept (s, (struct sockaddr *) &sin, &len);
    /* cfd is new connection; you never read/write s */
    do_something_with (cfd);
    close (cfd);
}
Using UDP

- Call `socket` with `SOCK_DGRAM`, bind as before
- New system calls for sending individual packets
  - int `sendto(int s, const void *msg, int len, int flags, const struct sockaddr *to, socklen_t tolen);
  - int `recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, socklen_t fromlen);
- Must send/get peer address with each packet
- Can use UDP in connected mode
  - `connect` assigns remote address
  - `send`/`recv` syscalls, like `sendto`/`recvfrom` w/o last 2 args

Socket implementation

- Need to implement layering efficiently
  - Add UDP header to data, Add IP header to UDP packet, …
  - De-encapsulate Ethernet packet so IP code doesn’t get confused by Ethernet header
- Don’t store packets in contiguous memory
  - Moving data to make room for new header would be slow
- BSD solution: mbufs [Leffler]
  (Note [Leffler] calls `m_nxt` by old name `m_act`)
  - Small, fixed-size (256 byte) structures
  - Makes allocation/deallocation easy (no fragmentation)
- BSD Mbufs working example for this lecture
  - Linux uses `skbuffers`, which are similar idea

mbuf utility functions

- `mbuf *m_copy(mbuf *m, int off, int len, int wait);
  - Creates a copy of a subset of an mbuf chain
  - Doesn’t copy clusters, just increments reference count
  - `wait` says what to do if no memory (wait or return NULL)
- `void m_adj(struct mbuf *mp, int len);
  - Trim [len] bytes from head or (if negative) tail of chain
- `mbuf *m_pullup(struct mbuf *n, int len);
  - Put first len bytes of chain contiguously into first mbuf
- Example: Ethernet packet containing IP datagram
  - Trim Ethernet header w. `m_adj`
  - Call `m_pullup(n, sizeof(ip_hdr));`
  - Access IP header as regular C data structure
**protosw structure**

- **Goal:** abstract away differences between protocols
  - In C++, might use virtual functions on a generic socket structure
  - Here just put function pointers in protosw structure
- **Also includes a few data fields**
  - `type, domain, protocol` — to match socket syscall args, so know which `protosw` to select
  - `flags` — to specify important properties of protocol
- **Some protocol flags:**
  - `ATOMIC` — exchange atomic messages only (like UDP, not TCP)
  - `ADDR` — address given w. messages (like unconnected UDP)
  - `CONNREQUIRED` — requires connection (like TCP)
  - `WANTRCVD` — notify socket of consumed data (e.g., so TCP can wake up a sending process blocked by flow control)

**protosw functions**

- `pr_slowtimo` — called every 1/2 sec for timeout processing
- `pr_drain` — called when system low on space
- `pr_input` — takes mbuf chain of data to be read from socket
- `pr_output` — takes mbuf chain of data written to socket
- `pr_usrreq` — multi-purpose user-request hook
  - Used for bind/listen/accept/connect/disconnect operations
  - Used for out-of-band data
  - Various other control operations

**Network interface cards**

- **Each NIC driver provides an ifnet data structure**
  - Like protosw, tries to abstract away the details
- **Data fields:**
  - Interface name (e.g., “eth0”)
  - Address list (e.g., Ethernet address, broadcast address, …)
  - Maximum packet size
  - Send queue
- **Function pointers**
  - `if_output` — prepend header, enqueue packet
  - `if_start` — start transmitting queued packets
  - Also ioctl, timeout, initialize, reset

**Input handling**

- NIC driver determines packet protocol
- Enqueues packet for appropriate protocol handler
  - If queue full, drop packet (can create livelock [Mogul])
- Posts “soft interrupt” for protocol-layer processing
  - Runs at lower priority than hardware (NIC) interrupt
  … but higher priority than process-context kernel code

**Routing**

- **An OS must route all transmitted packets**
  - Machine may have multiple NICs plus “loopback” interface
  - Which interface should a packet be sent to, and what MAC address should packet have?
- **Routing is based purely on the destination address**
  - Even if host has multiple NICs w. different IP addresses
  - (Though some packet filters can redirect based on source IP)
- **OS maintains routing table**
  - Maps IP address & prefix-length → next hop
- **Use radix tree for efficient lookup**
  - Branch at each node in tree based on single bit of target
  - When you reach leaf, that is your next hop
- **Most OSes provide packet forwarding**
  - Received packets for non-local address routed out another if