Administrivia

- **Project 3 due Thursday**
  - As usual, 4:15pm due time
  - Extension to midnight if you come to class
  - For longer extensions must cs140-staff beforehand
- **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.

Addressing

- **Each node typically has unique *address***
  - (or at least is made to think it does when there is shortage)
- ***Routing* is process of delivering data to destination**
- **For *packet switched networks***
  - Data transmitted in small packets (e.g., 1,500 bytes)
  - Each packet must have a destination address
- **For *circuit switched* networks**
  - Use address to set up circuit
  - Stream data between two nodes across circuit
- **Special addresses can exist for broadcast/multicast**

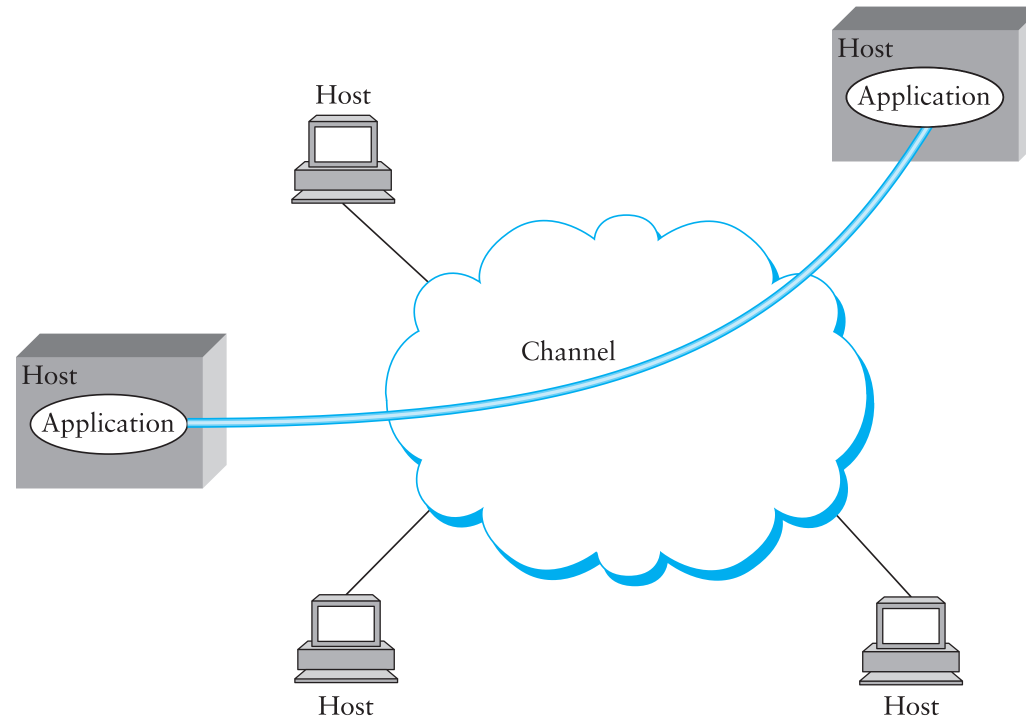
Internet protocol

- **Most computer nets connected by Internet protocol**
  - Runs over a variety of physical networks, so can connect Ethernet, Wireless, people behind modem lines, etc.
- **Every host has[a] a unique 4-byte IP address**
  - E.g., www.ietf.org → 132.151.6.21
  - Given a node's IP address, the network knows how to route a packet (we will discuss in much more detail)
- **But how do you build something like the web?**
  - Need naming (look up www.ietf.org) – DNS
  - Need interface for browser & server software (later)
  - Need demultiplexing within a host—E.g., which packets are for web server, which for mail server, etc.?

---

[a] or thinks it has

Inter-process communication



- **Want abstraction of inter-process (not just inter-node) communication**
- **Solution: *Encapsulate* another protocol within IP**

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 to processes on two different machines
  - Handles congestion & flow control

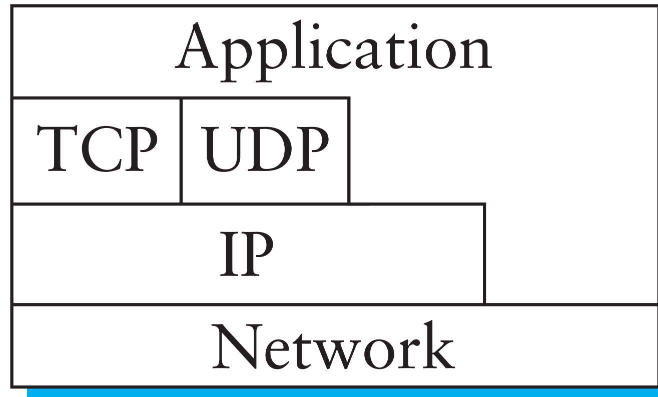
Failure

- **Several types of error can affect packet delivery**
  - Bit errors (e.g., electrical interference, cosmic rays)
  - Packet loss (overload)
  - Link and node failure
- **In addition, properly delivered frames can be delayed and reordered**

Uses of TCP

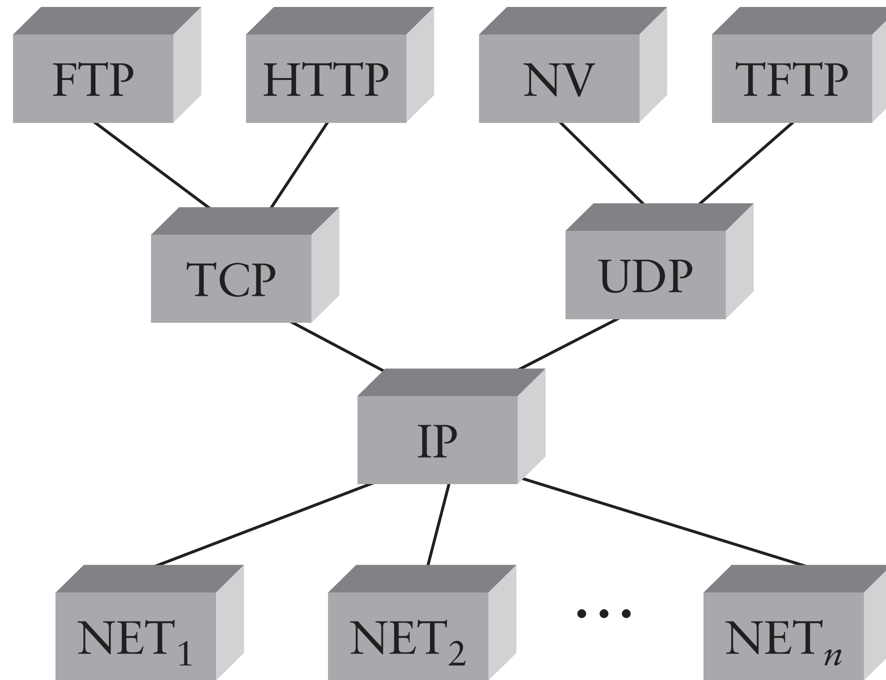
- **Most applications use TCP**
  - Easier interface to program to (reliability)
  - Automatically avoids congestion (don't need to worry about taking down network)
- **Servers typically listen on well-known ports**
  - SSH: 22
  - Email: 25
  - Finger: 79
  - Web / HTTP: 80
- **Example: Interacting with www.stanford.edu**
  - Browser resolves IP address of www.stanford.edu (171.67.22.34)
  - Browser connects to TCP port 80 on 171.67.22.34
  - Over TCP connection, browser requests and gets home page

IP layering



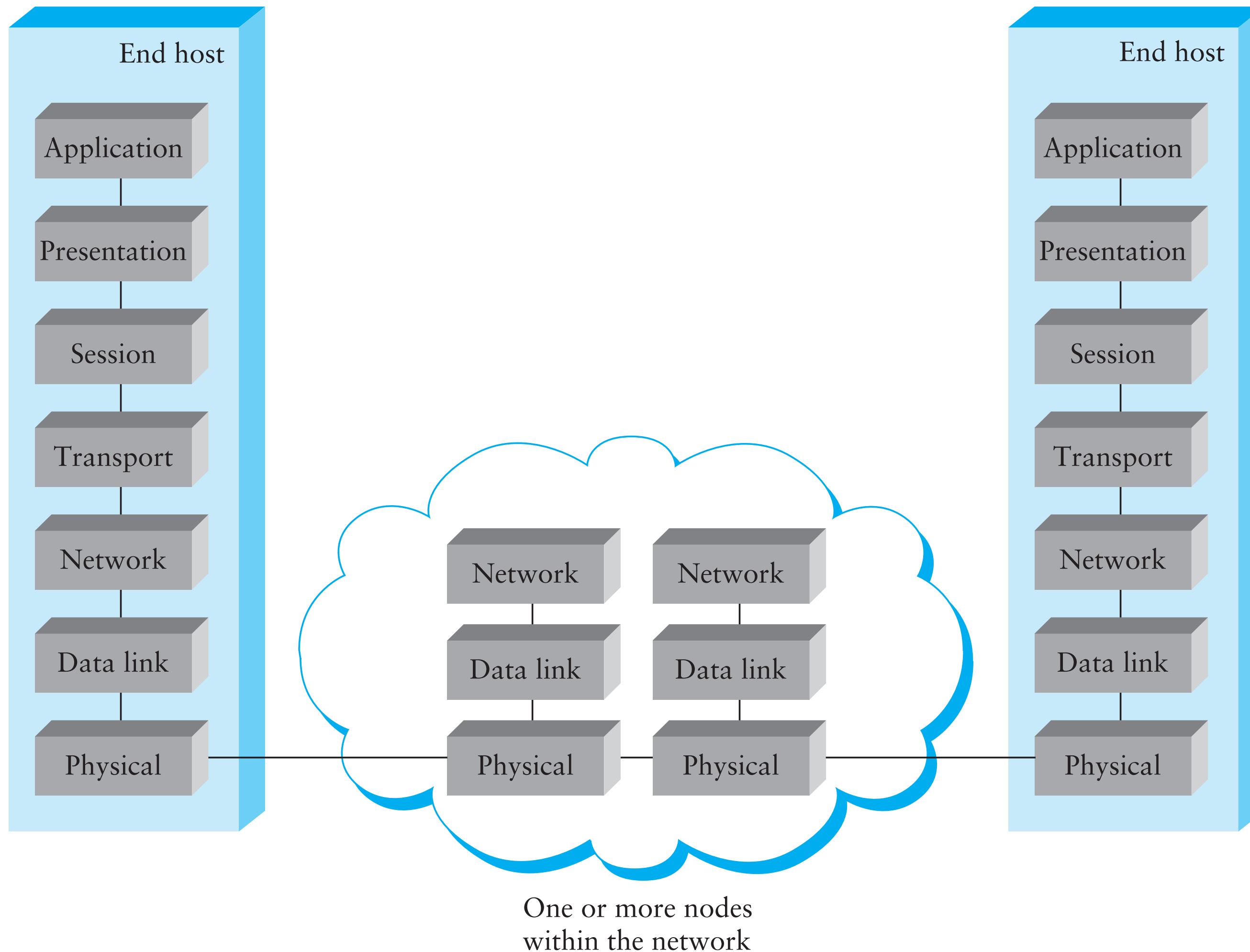
- **Can view network encapsulation as a stack**
  - Each layer produces packets that become the payload of the lower-layer's packets
  - This is almost correct, but TCP/UDP "cheat" to detect certain errors in IP-level information like address

Hourglass



- **Many application protocols over TCP & UDP**
- **IP works over many types of network**
- **This is "Hourglass" philosophy of Internet**
  - Idea: If everybody just supports IP, can use many different applications over many different networks

OSI layers



- **Layers typically fall into 1 of 7 categories**

Layers

1. **Physical – sends individual bits**
2. **Data link – sends *frames*, handles access control to shared media (e.g., coax)**
3. **Network – delivers packets, using *routing***
4. **Transport – demultiplexes, provides reliability & flow control**
5. **Session – can tie together multiple streams (e.g., audio & vidio)**
6. **Presentation – crypto, conversion between representations**
7. **Application – what end user gets, e.g., HTTP (web)**

- **OS responsible for 3–4, must also know about 2 & 5+**

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:**

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)

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, so 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?**
- **Recall pipes:** `int pipe (int fds[2]);`
  - Allow Inter-process communication on one machine
  - Writes to `fds[1]` will be read on `fds[0]`
  - 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
- **Well-known services "listen" on standard ports:**
  - e.g., finger—79, HTTP—80, mail—25, ssh—22
  - Clients connect from arbitrary ports to well known ports
- **A *connection* can be named by 5 components**
  - Protocol (TCP), local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP

System calls for using TCP

Client	Server
	`socket` – make socket
	`bind` – assign address
	`listen` – listen for clients
`socket` – make socket	
`bind`* – assign address	
`connect` – connect to listening socket	
	`accept` – accept connection

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

Client interface

```
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

```
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

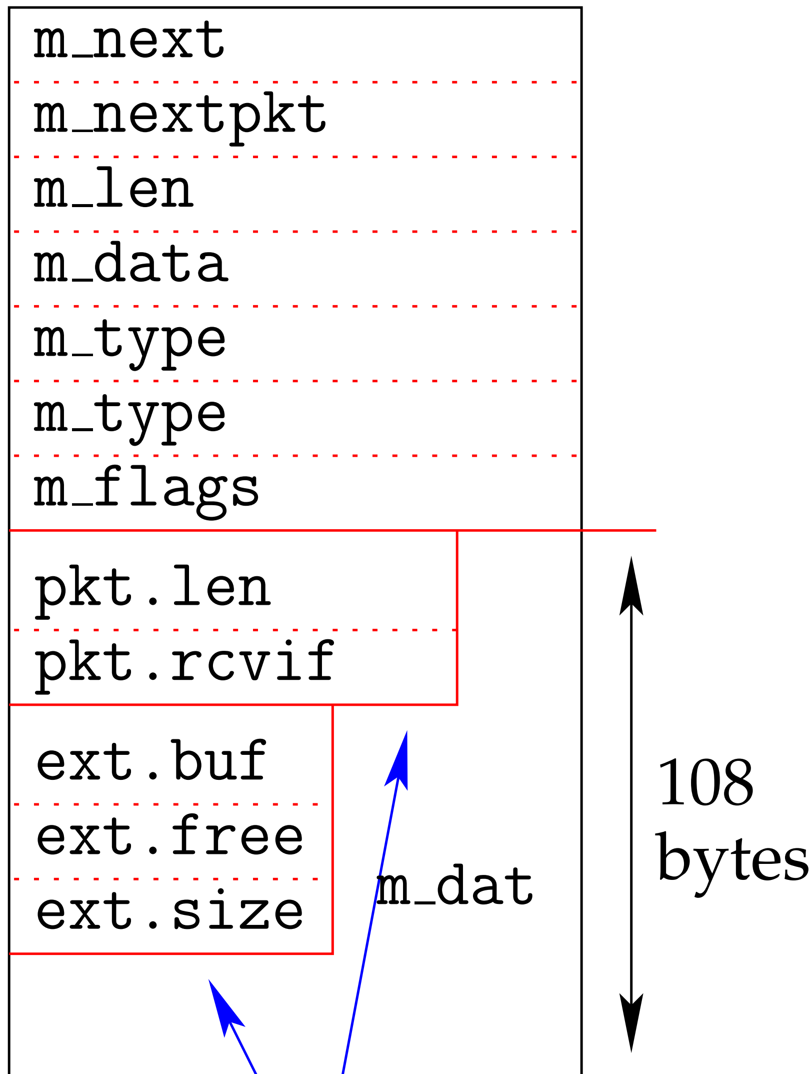
Uses of connected UDP sockets

- **Kernel demultplexes packets based on port**
  - So can have different processes getting UDP packets from different peers
  - For security, ports < 1024 usually can't be bound
  - But can safely inherit UDP port below that connected to one particular peer
- **Feedback based on ICMP messages**
  - Say no process has bound UDP port you sent packet to…
  - With `sendto`, you might think network dropping packets
  - Server sends port unreachable message, but only detect it when using connected sockets

Socket implementation: buffering

- **Need to be able to encapsulate data easily**
  - E.g., add UDP header to data
  - Add IP header to UDP packet
  - Add Ethernet header to IP packet
- **Need to de-encapsulate as well**
  - Strip off headers before sending data up the layer stack
- **Solution: Don't store packets in contiguous memory**
- **BSD solution: mbufs**
  - Small, fixed-size (256 byte) structures
  - Makes allocation/deallocation easy (no fragmentation)
- **Mbufs working example for this lecture**
  - Linux uses `sk_buffs`, which are similar idea

mbuf details



- **Pkts made up of multiple mbufs**
  - *Chained* together by `m_next`
  - Such linked mbufs called *chains*
- **Chains linked w.** `m_nextpkt`
  - Linked chains known as *queues*
  - E.g., device output queue
- **Most mbufs have 108 data byes**
  - First in chain has `pkt` header
- ***Cluster* mbufs have more data**
  - `ext` header points to data
  - Up to 2 KB not collocated w. mbuf
  - `m_dat` not used
- `m_flags` **or of various bits**
  - E.g., if cluster, or if `pkt` header used

Adding/deleting data w. mbufs

- **`m_data` always points to start of data**
  - Can be `m_dat`, or `ext.buf` for cluster mbuf
  - Or can point into middle of that area
- **To strip off a packet header (e.g., TCP/IP)**
  - Increment `m_data`, decrement `m_len`
- **To strip off end of packet**
  - Decrement `m_len`
- **Can add data to mbuf if buffer not full**
- **Otherwise, add data to chain**
  - Chain new mbuf at head/tail of existing chain

mbuf utility functions

- `mbuf *m_copym(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

Socket implementation

- **Each socket fd has associated socket structure with:**
  - Send and receive buffers
  - Queues of incoming connections (on listen socket)
  - A *protocol control block* (PCB)
  - A *protocol handle* (`struct protosw *`)
- **PCB contains protocol-specific info. E.g., for TCP:**
  - Pointer to IP TCB w. source/destination IP address and port
  - Information about received packets & position in stream
  - Information about unacknowledged sent packets
  - Information about timeouts
  - Information about connection state (setup/teardown)

`protosw` structure

- **Goal: abstract away differences between protocols**
  - In C++, might use virtual functions on a generic socket struct
  - 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

Routing

- **Routing is deciding where to send a packet**
  - Machine may have multiple NICs – which to use?
  - Machine may be on shared net and need to chose next hop
    (E.g., if I connect to `cnn.com`, send packet to Stanford's router)
- **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 & mask → 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

ARP

- **Must map IP addresses into physical addresses**
  - E.g., Ethernet address of destination host
  - Or ethernet address of next hop router
- **Techniques**
  - Encode physical address in host part of IP address (IPv6)
  - Each network node maintains a lookup table (phys→IP)
- **ARP – *address resolution protocol***
  - Table of IP to physical address bindings
  - Broadcast request if IP address not in table
  - Everybody learns physical address of requesting node (broadcast)
  - Target machine responds with its physical address
  - Table entries are discarded if not refreshed

Arp Ethernet packet format

<table>
<tr><th>0</th><th>8</th><th>16 … 31</th></tr>
<tr><td colspan="2">Hardware type = 1</td><td>ProtocolType = 0x0800</td></tr>
<tr><td>HLen = 48</td><td>PLen = 32</td><td>Operation</td></tr>
<tr><td colspan="3">SourceHardwareAddr (bytes 0–3)</td></tr>
<tr><td colspan="2">SourceHardwareAddr (bytes 4–5)</td><td>SourceProtocolAddr (bytes 0–1)</td></tr>
<tr><td colspan="2">SourceProtocolAddr (bytes 2–3)</td><td>TargetHardwareAddr (bytes 0–1)</td></tr>
<tr><td colspan="3">TargetHardwareAddr (bytes 2–5)</td></tr>
<tr><td colspan="3">TargetProtocolAddr (bytes 0–3)</td></tr>
</table>