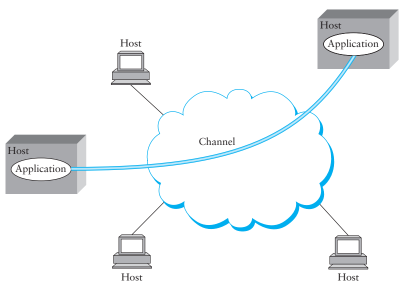
Outline

1. Networking overview
2. Systems issues
3. OS networking facilities
4. Implementing networking in the kernel

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.

Inter-process communication



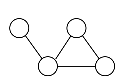
- **Want abstraction of inter-process (not just inter-node) communication**
- **Goal: two different applications, running on different computers, can exchange data as if they had a pipe between them.**

The 7-Layer and 4-Layer Models

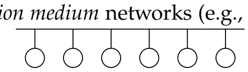
	OSI	TCP/IP
7	Application	Applications (FTP, SMTP, HTTP, etc.)
6	Presentation	
5	Session	
4	Transport	TCP (host-to-host)
3	Network	IP
2	Data link	Network access (usually Ethernet)
1	Physical	

Physical Layer

- **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):

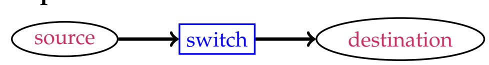


    - ▹ Any message can be seen by all nodes
    - ▹ Allows broadcast/multicast, but introduces contention
- **One implication: speed of light matters!**
  - $\sim 300,000$ km/sec in a vacum, slower in fiber

SF $\xrightarrow{\geq \sim 15\text{ msec}}$ NYC   Moore's law does not apply!

Link Layer, Indirect Connectivity

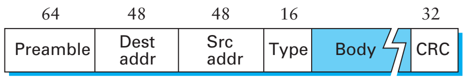
- **Rarely have direct physical connection to destination**
- **Instead, communications usually "hop" through multiple devices**



  - Allows links and devices to be shared for multiple purposes
  - Must determine which bits are part of which messages intended for which destinations
- ***Packet switched* networks**
  - Pack a bunch of bytes together intended for same destination
  - Slap a *header* on packet describing where it should go

Link Layer: Ethernet

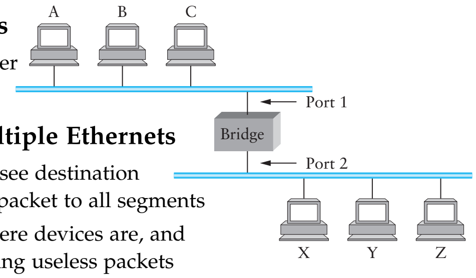
- **Originally designed for shared medium (coax), now generally not shared medium (switched)**
- **Vendors give each device a unique 48-bit *MAC address***
  - Specifies which card should receive a packet
- **Ethernet switches can scale to switch local area networks (thousands of hosts), but not much larger**
- **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

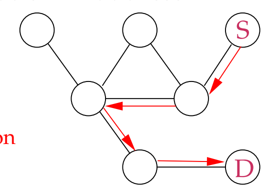
- **Ethernet Limits**
  - 2,500m diameter
  - 100 nodes
- **Can *bridge* multiple Ethernets**
  - First time you see destination address, send packet to all segments
  - Then learn where devices are, and avoid forwarding useless packets



- **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

Network Layer: 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



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

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

Principle: Packet Switching

- **A packet is a self contained unit of data which contains information necessary for it to reach its destination**
- **Packet switching: independently for each arriving packet, compute its outgoing link. If the link is free, send it. Otherwise, queue it for later (or drop).**
  - Makes forwarding very simple
  - Allows simple sharing of links

Principle: Layering

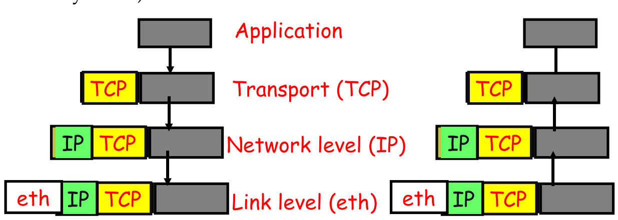
- **Break system functionality into a set of components**
- **Each component ("layer") provides a well-defined service**
- **Each layer uses only the service of the layer below it**
- **Layers communicate sequentially with the layers above or below**

The 7-Layer and 4-Layer Models

	OSI	TCP/IP
7	Application	Applications (FTP, SMTP, HTTP, etc.)
6	Presentation	
5	Session	
4	Transport	TCP (host-to-host)
3	Network	IP
2	Data link	Network access (usually Ethernet)
1	Physical	

Principle: Encapsulation

- **Stick packets inside packets**
- **How you realize packet switching and layering in a system**
  - 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 (not always true)



Outline

1. Networking overview
2. Systems issues
3. OS networking facilities
4. Implementing networking in the kernel

Unreliability of IP

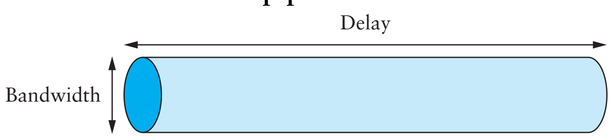
- **Network does not deliver packets reliably**
  - May drop packets, reorder packets, delay packets
  - May even corrupt packets, or duplicate them
- **How to implement reliable TCP on top of IP network?**
  - Note: This is entirely the job of the OS at the end nodes
- **Straw man: Wait for ack for 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)

Performance: 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 # bytes in flight $\geq$ bandwidth$\times$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**

Failure

- **Many more failure modes on net than w. local IPC**
- **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)

A little bit about TCP

- **Want to save network from congestion 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 [Jacobson], 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

Outline

1. Networking overview
2. Systems issues
3. OS networking facilities
4. Implementing networking in the kernel

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

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

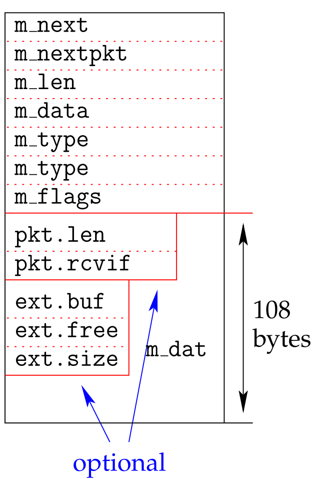
Outline

1. Networking overview
2. Systems issues
3. OS networking facilities
4. Implementing networking in the kernel

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_nextpkt` 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 `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 ≈230 data bytes (depends on pointers)**
  - 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)

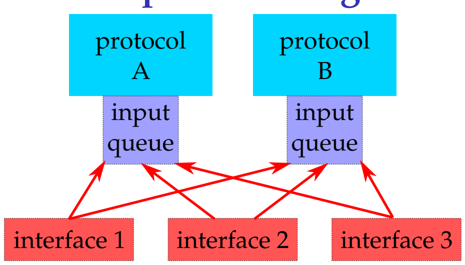
`protosw` structure

- **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)

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 OSes have features to 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