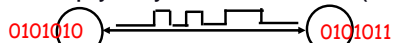
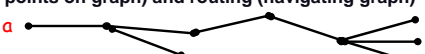



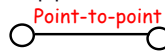
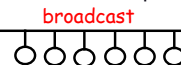
# CS 140: Operating Systems Lecture 24: Link Layer

Mendel Rosenblum

## The Big Picture

- ◆ Divide network into three layers:
  - link level: physically encode bits on "wire" (line segment)
 
  - network layer: connecting segments, addressing (locating points on graph) and routing (navigating graph)
 
  - end-to-end layer: making network simple and reliable
 
- ◆ Last time: networking from 50,000 feet.
- ◆ Today: Link level (ground zero). Reading: Lampson.

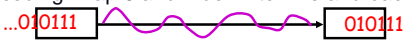
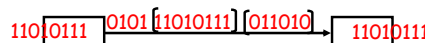
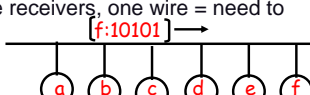
## Links

- ◆ Link = pipe to send information. Examples:
  - Point-to-point 
  - broadcast 
- ◆ Can build out of:
  - Twisted pair (the wire your phone connects to).
  - Coaxial cable (the wire your tv connects to).
  - Optical fiber (the stuff we all want to connect to).
  - Space (the stuff that does not require laying pipe: voice, radio waves, microwaves, laser beams)
- ◆ What do higher layers require of a link?
  - Very little: send bits well enough that we don't give up
  - assume: links lose, corrupt, reorder, and duplicate pkts.

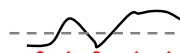
## Example links ("b" = bit, "B" = byte)

Medium	link	bandwidth	message
alpha chip	on-chip bus	4GB/s	8 bytes
pc board	RAMbus	500MB/s	packet < 100B
	PCI I/O bus	133MB/s	packet
wires	SCSI	20MB/s	packet
LAN	Ethernet	1.25MB/s	packet 64-1500B
	Fast Ethernet	12.5MB/s	packet, 64-1500B
	Gigabit Eth	125MB/s	packet, 64-1500B
Wireless	WaveLan	.25MB/s	< 1500 B
copper pair	ISDN	64Kb/s	1 byte
copper pair	T1	1.5Mb/s (24 ISDN's)	1 byte
coax cable	T3	44.736Mb/s (30 T1's)	1 byte
fiber	STS-1	51.8Mb/s	1 byte or cell
	STS-48	2.5Gb/s (48 STS-1's)	1 byte or cell
	(STS-* also called OC-*)		

## Link level issues

- ◆ Encoding: map 0 and 1 down to wire and back
  - 
- ◆ Framing: delineate bit stream into frames (packets) receiver knows where things begin
  - 
- ◆ Arbitration: multiple senders, one resource = need to coordinate
- ◆ Addressing: multiple receivers, one wire = need to indicate which one
  - 

## Encoding

- ◆ Goal: map 0's and 1's onto wire and back.
- ◆ Simple scheme:
  - 0 = low voltage
  - 1 = high voltage
  - 
  - Called "non-return to zero" (NRZ)
  - Doesn't work too well. As dumb as the name
- ◆ Problem 1: How to tell errors from packets?
  - What does a dead link look like?
  - What does a jammed link look like?
  - (fix: map data in such a way that always has transitions)
- ◆ Problem 2: More basic: when should we sample?!

### The big sampling problem: Clock drift

- ◆ To send bits: every clock cycle  
 Sender encodes a bit.  
 Receiver decodes a bit.  
 But: don't have same clock!
- 
- What happens out of phase?
- ◆ Sol'n: Clock recovery: derive clock from data  
 Insight: whenever signal changes from 0->1 or 1->0 receiver knows it is on a clock boundary.  
 So, encode data so that it always has transitions!  
 E.g.: Manchester encoding (used by Ethernet).

### Manchester encoding

- ◆ Goal: want both 0 and 1 to give a transition  
 Map 0 to low-high transition.  
 Map 1 to high-low transition.
- 
- ◆ Good: can detect:  
 Dead link, short, and collisions (how?)
  - ◆ Bad: reduces bandwidth  
 Could have sent two bits using NRZ for each 1 bit here  
 If wire can do N transitions per second, what is NRZ's bandwidth?  
 Manchester encodings?  
 Other encoding schemes fight this.

### Framing: splitting bits into packets

- ◆ Why packets?  
 Look for address in stream, know when to stop reading.
- ◆ Simplest approach: sentinel  
 send "begin packet" and "end packet" tokens.  
 e.g., if sentinel is "1111":  
 1111 0101010010101 1111 10100101 1111 1001101 1111  
 Data
- ◆ Main problem: what if sentinel appears in data?  
 The usual approach: "escape" sentinel (map to something else; receiver will have to map back) "bit stuffing".  
 Example: in C, '\ is a special character, if you need it, have to include it twice "\\".

### HDLC (high-level data link control protocol)

- ◆ Same sentinel for begin and end: 0111 1110
- ◆ packet format:  

0111 1110	header	data	CRC	0111 1110
-----------	--------	------	-----	-----------
- ◆ bit stuffing  
 Sender: if data contains five consecutive 1's, insert a 0 after it:  
 0111 1110 → 0111 1101 0  
 Receiver: if data = five 1's followed by a 0, remove 0.  
 0111 1101 0 → 0111 1110  
 Otherwise read next bit. A 0 implies? A 1 implies?  
 Packet size now depends on contents!

### Arbitration: coordinating multiple senders

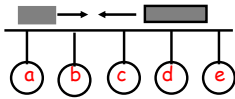
- ◆ Problem: one wire, multiple senders  
 Old problem (CPU, disk, memory). A new thing: no centralized control.  
 Many different approaches.
- ◆ Time-division multiplexing (telephone model)  
 Divide into time slices and round robin among senders.  
 Exceed capacity? Don't admit new senders (busy signal).
- ◆ Frequency division multiplexing (radio model)  
 Divide spectrum up into slices; give each sender a piece.  
 Capacity exceeded? Don't give any more slices.
- ◆ Both give fixed delays and constant data rate  
 Good for continuous data streams.  
 Doesn't work well for computer (bursty) traffic.

### Statistical multiplexing

- ◆ Two problems with TDM and FDM  
 Does not adapt. Sender has no data? Slot wasted. Lots of data? Can only send as much as slot allows.  
 Need to know maximum senders ahead of time.  
 (if net = memory, what do TDM/FDM correspond to?)
- ◆ Two key observations:  
 Links usually idle & computer network traffic bursty.
- ◆ Two key ideas:  
 Idea 1: transmit on demand (when sender has data)  
 result: link not wasted, get peak bandwidth  
 Idea 2: upper bound on packet that can be sent.  
 Result? (hint: view packet as process)
- ◆ Problem: congestion (too many senders)

### Classic Ethernet

- ◆ "Carrier Sense, Multiple Access with Collision Detect"
  - Statistical multiplexing: nodes send when they want.
  - Carrier sense = can distinguish between idle and busy.
  - Collision detect = if your packet hits another, can hear it.
- ◆ To send data, node:
  - Listens until link idle
  - Immediately sends, while listening for a collision
- ◆ To handle collision:
  - Jam wire so all senders hear. (determines min. pkt size)
  - When to retransmit?
  - Wait until idle and retry? But will hit again!

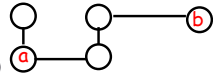


### Exponential backoff

- ◆ Pretty idea (from Aloha radio network):
  - Pick a random number distributed between [0..n).
  - Wait this amount. Then retransmit.
  - If collide again, pick a number between [0..2n), ...
  - Result: coordination without centralized control!
- ◆ Nice effects:
  - Adaptive: wait determined by how busy wire is.
  - If really busy, everyone will back off until they reach a point that contention goes away.
  - Exponential back off used in lots of places to handle contention without some central control (locks, TCP).
  - Note: doing retransmit at this level is just an optimization...

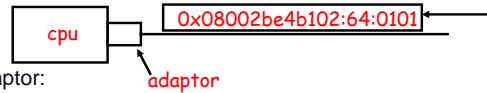
### Addressing

- ◆ Problem: how to send from a to b?
  - If point-to-point, trivial (only one possible receiver)
  - How to handle if multiple listeners?
- ◆ On a mesh:
  - Uniquely name nodes (how?)
  - Sender must know name (how?)
  - Nodes that join links ("switches" or "routers" depending on level) have to know if & where to forward pkts.
  - We'll examine this more when we look at IP.
- ◆ On a broadcast network, solution is simple:
  - Also give unique names.
  - Sender appends name to the beginning of each packet.
  - Receiver checks each packet for its name.



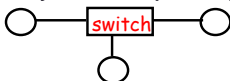
### Ethernet receiving example

- ◆ Node connected to wire by Ethernet adaptor:
  - cpu
  - adaptor
  - 0x08002be4b102:64:0101
- ◆ adaptor:
  - Has unique 6-byte address (e.g., 8:0:2b:e4:b1:2) put in ROM by manufacturer. Can change it.
  - Scans for packets with its address or with broadcast address (ff:ff:ff:ff:ff:ff)
  - When it finds a packet of interest, it buffers it and wakes up CPU.
  - Can view this as an example of moving computation to reduce bandwidth. (Otherwise CPU has to scan.)
  - Can also view as using parallelism to make system faster.



### Experience with Ethernet

- ◆ Work best under lightly loaded conditions
  - Over 30% and degrades severely due to contention.
  - Not much of a problem in practice.
- ◆ Why it works:
  - Ethernet first viewed as broadcast with lots of nodes, frequently more like a point-to-point!
- ◆ Higher level protocols implement "flow control" which prevents a single node from pounding on network.
- ◆ Why it succeeded: simple to administer + dirt cheap



### Bridges: connecting links

- ◆ Bridge:
  - Place between two links.
  - Put network adaptor in "promiscuous" mode (see all pkts) whenever receives packet, forwards to the other link.
- ◆ Learning bridge:
  - Rather than forward all packets, just forward those that are on other side.
  - How to figure out? Look at source address in packet!
  - Tells you who on receiving side.
  - Now when get packet: if destination on sending side, ignore, otherwise forward packet
  - (Since network changes, throw out entries over time)



### Link layer summary

---

- ◆ Links carry signals: have to encode bits on these.
- ◆ Given a sequence of 0's and 1's, have to break into packets (frames). Usually done using sentinels.
- ◆ Multiple senders? Need to coordinate them.
  - Statistical multiplexing allows you to take advantage of spasmodic traffic.**
  - exponential backoff counters contention.**
- ◆ Multiple receivers? Need to name them.
- ◆ Next lecture: the network layer
  - Issues in finding name of who you want to send to routing packet to them.**