23. Networks — Links and Switches[1]

This handout presents the basic ideas for transmitting digital data over links, and for connecting links with switches so that data can pass from lots of sources to lots of destinations. You may wish to read chapter 7 of Hennessy and Patterson for a somewhat different treatment, more focused on interconnecting the components of a multiprocessor computer.

Links

A link is an unreliable FIFO channel. As we mentioned in handout 21, it is an abstraction of a point-to-point wire or of a simple broadcast LAN. It is unreliable because noise or other physical problems can corrupt messages.

There are many kinds of physical links, with cost and performance that vary based on length, number of drops, and bandwidth. Here are some current examples. Bandwidth is in bytes/second[2], and the "+" signs mean that software latency must be added. The nature of the messages reflects the origins of the link. Computer people prefer variable-size packets, which are good for bursty traffic. Communications people have historically preferred bits or bytes, which are good for fixed-bandwidth voice traffic and minimize the latency and buffering added by collecting voice samples into a message.

A physical link can be unidirectional ('simplex') or bidirectional ('duplex'). A duplex link may operate in both directions at once ('full-duplex'), or in one direction at a time ('half-duplex'). A pair of simplex links in opposite directions forms a full-duplex link. So does a half-duplex link in which the time to reverse direction is negligible, but in this case the peak full-duplex bandwidth is half the half-duplex bandwidth. If most of the traffic is in one direction, however, the usable bandwidth of a half-duplex link may be nearly the same as that of a full-duplex link.

To increase the bandwidth of a link, run several copies of it in parallel. This goes by different names; 'space division multiplexing' and 'striping' are two of them. Common examples are:

Parallel busses, as in the first four lines of the table.

Switched networks: the telephone system and switched LANs.

Multiple disks, each holding part of a data block, that can transfer in parallel.

Cellular telephony, using spatial separation to reuse the same frequencies.

In the latter two cases the parallelism is being added to links that were originally designed to operate alone, so there must be physical switches to connect the parallel links.

Another use for multiple links is fault tolerance, discussed earlier.

[1] My thanks to Alex Shvartsman for some of the figures in this handout.
[2] Beware: communications people usually quote bits/sec, so network bandwidth tends to be quoted this way. All the numbers in the table are in bytes, however, except for the bus width in bits.

Medium	*Link*	*Bandwidth*		*Latency*		*Width*	*Message*
Alpha EV7 chip	on-chip bus	10	GB/s	.8	ns	64	word
PC board	Rambus bus	1.6	GB/s	75	ns	16	memory packet
	PCI I/O bus	266	MB/s	250	ns	32/64	DMA block
Wires	Fibre channel[3]	125	MB/s	200	ns	1	packet
	IEEE 1394[4]	50	MB/s	1	µs	1	packet
	USB 2	50	MB/s	1	µs	1	?
	SCSI	40	MB/s	500	ns	16	32
	USB	1.5	MB/s	5	µs	1	?
LAN	Gigabit Ethernet	125	MB/s	1 +	µs	1	packet, 64-1500 B
	Fast Ethernet[5]	12.5	MB/s	10 +	µs	1	packet, 64-1500 B
	Ethernet	1.25	MB/s	100 +	µs	1	packet, 64-1500 B
Wireless	802.11a	6	MB/s	100 +	µs	1	packet, < 1500 B
Fiber (Sonet)	OC-48	300	MB/s	5	µs/km	1	byte or 48 B cell
Coax cable	T3	6	MB/s	5	µs/km	1	byte
Copper pair	T1	0.2	MB/s	5	µs/km	1	byte
Copper pair	ISDN	16	KB/s	5	µs/km	1	byte
Broadcast	CAP 16	3	MB/s	3	µs/km	6 MHz	byte or cell

Flow control

Many links do not have a fixed bandwidth that is known to the sender, because the link is being shared (that is, there is multiplexing inside the link) or because the receiver can't always accept data. In particular, fixed bandwidth is bad when traffic is bursty, because it will be either too small or too large. If the sender doesn't know the link bandwidth or can't be trusted to stay below it, some kind of *flow control* is necessary to match the flow of traffic to the link's or the receiver's capacity. A link can provide this in two ways, by contention or by scheduling. In this case these general strategies take the form of *backoff* or *backpressure*.

Backoff

In backoff the link drops excess traffic and signals 'trouble' to the sender, either explicitly or by failing to return an acknowledgment. The sender responds by waiting for a while and then retransmitting. The sender increases the wait by some factor (say 2) after every trouble signal and decreases it with each trouble-free send. This is called 'exponential backoff'; when the

[3] M. Sachs and A. Varman, Fibre channel and related standards. *IEEE Communications* **34**, 8 (Aug. 1996), pp 40-49.
[4] G. Hoffman and D. Moore, IEEE 1394: A ubiquitous bus. *Digest of Papers, IEEE COMPCON '95*, 1995, pp 334-338.
[5] M. Molle and G. Watson, 100Base-T/IEEE 802.12/Packet switching. *IEEE Communications* **34**, 8 (Aug. 1996), pp 63-73.

increasing factor is 2, it is 'binary exponential backoff'. It is used in the Ethernet[6] and in TCP[7], and is analyzed in some detail in a later section.

Exponential backoff works because it adjusts the rate of sending so that most packets get through. If every sender does this, then every sender's delay will jiggle around the value at which the network is just managing to carry all the traffic. This is because a wait that is too short will overload the network, some packets will be lost, and the sender will increase the wait. On the other hand, a wait that is too long will always succeed, and the sender will decrease it. Of course these statements are probabilistic: sometimes a conservative sender will lose a packet because someone else overloaded the network.

The precise details of how the wait should be lengthened (backed off) and shortened depend on the properties of the channel. If the 'trouble' signal comes back very quickly and the cost of trouble is small, senders can shorten their waits aggressively; this happens in the Ethernet, where collisions are detected in at most 64 byte times and abort the transmission immediately, so that senders can start with 0 wait for each new message. Under the opposite conditions, senders must shorten their waits cautiously; this happens in TCP, where the 'trouble' signal is only the lack of an acknowledgment, which can only be detected by timeout and which cannot abort the transmission immediately. The timeout should be roughly one round-trip time; the fact that in TCP it's often impossible to get a good estimate of the round-trip time is a serious complication.

An obvious problem with backoff is that it requires all the senders to cooperate. A sender who doesn't play by the rules can get an unfair share of the link resource, and in many cases two such senders can cause the total throughput of the entire link to become very small.

Backpressure

In backpressure the link tells the sender explicitly how much it can send without suffering losses. This can take the form of start and stop signals, or of 'credits' that allow a certain amount of additional traffic to be sent. The number of unused credits the sender has is called its 'window'. Let b be the bandwidth at which the sender can send when it has permission and r be the time for the link to respond to new traffic from the sender. A start–stop scheme can allow rb units of traffic between a start and a stop; a link that has to buffer this traffic will overrun and lose traffic if r is too large. A credit scheme needs rb credits when the link is idle to keep running at full bandwidth; a link will underrun and waste bandwidth if r is too large.[8]

Start–stop is used in the Autonet[9] (handout 22), and on RS-232 serial lines under the name XON-XOFF. The Ethernet, although it uses backoff to control acquiring the channel, also uses backpressure, in the form of carrier sense, to keep a sender from interrupting another sender that has already acquired the channel; this is called 'deference'. TCP uses credits to allow the receiver to control the flow. It also uses backoff to deal with congestion within the link itself (that is, in the underlying packet network). Having both mechanisms is confusing, and it's even more confusing (though clever) that the waits required by backoff are coded by fiddling with credits.

The failure modes of the two backpressure schemes are different. A lost 'stop' may cause lost data. A lost credit may reduce the bandwidth but doesn't cause data to be lost. On the other hand, 'start' and 'stop' are idempotent, so that a good state is restored just be repeating them. This is not true for credits of the form "send n more messages". There are several ways to get around this problem with credits:

> Number the messages, and send credits in the form "n messages after message k". Such a credit resets the sender's window completely. TCP uses this scheme, counting bytes rather than messages. On an unreliable channel, however, it only works if each message carries its own number, and this is extra overhead that is serious if the messages are small (for instance, ATM cells are only 53 bytes, and only 48 bytes of this are payload).
>
> Stop sending messages and send a 'resync' request. When the receiver gets this it returns an absolute rather than an incremental credit. Once the sender gets this it resets its window and starts sending again. There are various schemes for avoiding a hiccup during the resync.
>
> Know the round-trip time between sender and receiver, and keep track of m, the number of messages sent during the last round-trip time. The receiver sends an absolute credit n, and the sender sets its window to $n - m$, since there are m messages outstanding that the receiver didn't know about when it issued n credits. This works well for links with no buffering (for example, simple wires), because the round-trip time is constant. It works poorly if the link has internal buffering, because the round-trip time varies.

Another form of flow control that is similar to backpressure is called 'rate-based'. It assigns a maximum transmission bandwidth or 'rate' to each sender, undertakes to deliver traffic up to that bandwidth with high probability, and is free to discard excess traffic. The rate is measured by taking a moving average across some time window.[10]

Framing

The idea of framing (sometimes called 'acquiring sync') is to take a stream of `X`'s and turn it into a stream of `Y`'s. An `X` might be a bit and a `Y` a byte, or an `X` might be a byte and a `Y` a packet. This is a parsing problem. It occurs repeatedly in communications, at every level from analog signals through bit streams, byte streams, and streams of cells up to encoded procedure calls. We looked at this problem abstractly and in the absence of errors when we studied encoding and decoding in handout 7. For communication the parsing has to work even though physical problems such as noise can generate an arbitrary prefix of `X`'s before a sequence of `X`'s that correctly encodes some `Y`'s.

If an `X` is big enough to hold a label, framing is easy: You just label each `X` with the `Y` it is part of, and the position it occupies in that `Y`. For example, to frame (or encapsulate) an IP packet on the Ethernet, just make the 'protocol type' field of the packet be 'IP', and if the packet is too big to

[6] R. Metcalfe and D. Boggs: Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM* **19**, 395-404 (1976)
[7] V. Jacobsen: Congestion avoidance and control. *ACM SigComm Conference*, 1988, pp 314-329. C. Lefelhocg et al., Congestion control for best-effort service. *IEEE Network* **10**, 1 (Jan 1996), pp 10-19.
[8] H. Kung and R. Morris, Credit-based flow control for ATM networks. *IEEE Network* **9**, 2 (Mar. 1995), pp 40-48.
[9] M. Schroeder et al., Autonet: A high-speed self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communication* **9**, 8 (Oct. 1991), pp 1318-1335.
[10] F. Bonomi and K. Fendick, The rate-based flow control framework for the available bit rate ATM service. *IEEE Network* **9**, 2 (Mar. 1995), pp 25-39.

fit in an Ethernet packet, break it up into 'fragments' and add a part number to each part. The receiver collects all the parts and puts them back together.[11] The jargon for the entire process is 'fragmentation/re-assembly'.

If X is small, say a bit or a byte, or even the measurement of a signal's voltage level, more cleverness is needed. There are many possibilities, all based on the idea of a 'sync' pattern that allows the receiver to recognize the start of a Y no matter what the previous sequence of X's has been.

Certain *values* of X can be reserved to mark the beginning or the end of a Y. In FDDI[12], for example, 4 bits of data are coded in 5 bits on the wire (this is called a 4/5 code). This is done because the wire doesn't work if there are too many 0's or too many 1's in a row, so it's not possible to simply send the data bytes. However, the wire's demands are weak enough that there are more than 16 allowable 5-bit combinations, and one of these is used as the sync mark for the start of a packet.[13] If a 'sync' appears in the middle of a packet, that is taken as an error, and the next legal symbol is the start of a new packet. A simpler version of this idea requires at least one transition on every bit (in 10 Mb Ethernet) or byte (in RS-232); the absence of a transition for a bit or byte time is a sync.

Certain *sequences* of X can be reserved to mark the beginning of a Y. If these sequences occur in the data, they must be 'escaped' or coded in some other way. A familiar example is C's literal strings, in which `'\'` is used as an escape, and to represent a `'\'` you must write `'\\'`. In HDLC an X is a bit, the rule is that more than *n* 0 bits is a sync for some small value of *n*, and the escape mechanism, called 'bit-stuffing', adds a 1 after each sequence of *n* data zeros when sending and removes it when receiving. In RS-232 an X is a high or low voltage level, sampled at say 10 times the bit rate, a Y is (usually) 8 data bits plus a 'start bit' which must be high and a 'stop bit' which must be low. Thus every Y begins with a low-high transition which determines the phase for the rest of the Y (this is called 'clock recovery'), and a sequence of 9 or more bit-times worth of low is a sync.

The sequences used for sync can be detected *probabilistically*. In telephony T-1 signaling there is a 'frame' of 193 bits, one sync bit and 192 data bits. The data bits can be arbitrary, but they are xored with a 'scrambling' sequence to make them pseudo-random. The encoding specifies a definite pattern (say "010101") for the sync bits of successive frames (which are not scrambled). The receiver decodes by guessing the start of a frame and checking a number of frames for the sync pattern. If it's not there, the receiver makes a different guess. After at most 193 tries it will have guessed right. This takes a lot longer than the previous schemes to acquire sync, but it uses a constant amount of extra bandwidth (unlike escape schemes), and much less than fixed sync schemes: 1/193 for T-1 instead of 1/5 for FDDI, 1/2 for Ethernet, or 1/10 for RS-232.

[11] Actually fragmentation is usually done at the IP level itself, but the idea is the same.
[12] F. Ross: An overview of FDDI: The fiber distributed data interface. *IEEE Journal on Selected Areas in Communication* 7 (1989)
[13] Another symbol is used to encode a token, and several others are used for somewhat frivolous purposes.

Multiplexing

Multiplexing is a way to share a link among multiple senders and receivers. It raises two issues:

Arbitration (for the sender)—when to send.

Addressing (for the receiver)—when to receive.

A 'multiplexer' implements arbitration; it combines traffic from several input links onto one output link. A 'demultiplexer' implements addressing; it separates traffic from one input link onto several output links. The multiplexed links are called 'sub-channels' of the one link, and each one has an address. Figure 1 shows various examples; the ovals are buffers.

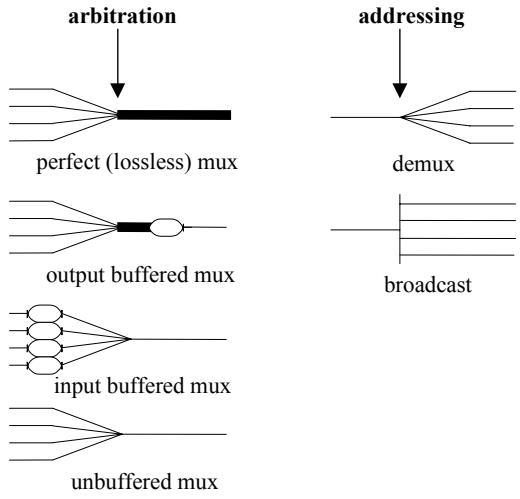


Fig. 1. Multiplexers and demultiplexers. Traffic flows from left to right. Fatter lines are faster channels.

There are three main reasons for multiplexers:

- Traffic may flow between one node and many on a single wire, for example when the one node is a busy server or the head end of a cable TV system.
- One wide wire may be cheaper than many narrow ones, because there is only one thing to install and maintain, or because there is only one connection at the other end. Of course the wide wire is more expensive than a single narrow one, and the multiplexers must also be paid for.
- Traffic aggregated from several links may be more predictable than traffic from a single one. This happens when traffic is bursty (varies in bandwidth) but uncorrelated on the input links. An extreme form of bursty traffic is either absent or present at full bandwidth. This is

standard in telephony, where extensive measurements of line utilization have shown that it's very unlikely for more than 10% of the lines to be active at one time, at least for voice calls.

There are many techniques for multiplexing. In the analog domain:

- *Frequency division* (FDM) uses a separate frequency band for each sub-channel, taking advantage of the fact that e^{int} is a convenient basis set of orthogonal functions. The address is the frequency band of the sub-channel. FDM is used to subdivide the electromagnetic spectrum in free space, on cables, and on optical fibers. On fibers it's usually called 'wave division multiplexing', and they talk about wavelength rather than frequency, but of course it's the same thing.
- *Code division* (CDM, usually called CDMA for 'code division multiple access') uses a different coordinate system in which a basis vector is a time-dependent sequence of frequencies. This smears out the cross-talk between different sub-channels. The address is the 'code', the sequence of frequencies. CDM is used for military communications and in newer varieties of cellular telephony. Figure 2 illustrates the simplest form of CDM, in which *n* senders share a digital channel. Bits on the channel have length 1, each sender's bits have length *n* (5 in the figure), and a sender has an *n*-bit 'code' (10010 in the figure) which it xor's with its current data bit. The receiver xor's the code in again and looks for either all zeros or all ones. If it sees something intermediate, that is interference from a sender with a different code. If the codes are sufficiently orthogonal (agree in few enough bits), the contributions of other senders will cancel out. Clearly longer code words work better.

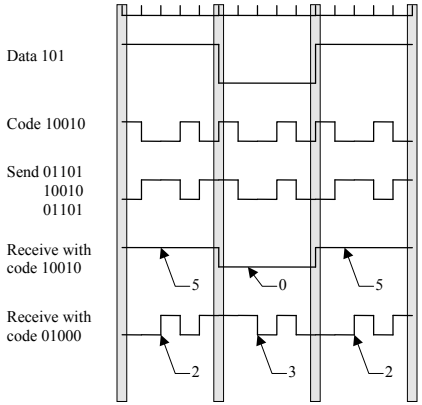


Fig 2: Simple code division multiplexing

In the digital domain time-division multiplexing (TDM) is the standard method. It comes in two flavors:

—*Fixed* TDM, in which *n* sub-channels are multiplexed by dividing the data sequence on the main channel into fixed-size slots (single bits, bytes, or whatever) and assigning every *n*th slot to the same sub-channel. Usually all the slots are the same size, but it's sufficient for the sequence of slot sizes to be fixed. The 1.5 Mbit/sec T1 line that we discussed earlier, for example, has 24 sub-channels and 'frames' of 193 bits. One bit marks the start of the frame, after which the first byte belongs to sub-channel 1, the second to sub-channel 2, and so forth. Slots are numbered from the start of the frame, and a sub-channel's slot number is its address. Note that this scheme requires framing to find the start of the frame (hence the name). But the addressing has no direct code (there is an "internal fragmentation" cost if the fixed channels are not fully utilized).

—*Variable* TDM, in which the data sequence on the main channel is divided into 'packets'. One packet carries data for one sub-channel, and the address of the sub-channel appears explicitly in the packet. If the packets are fixed size, they are often called 'cells', as in the Asynchronous Transfer Mode (ATM) networking standard. Fixed-size packets are used in other contexts, however, for instance to carry load and store messages on a programmed I/O bus. Variable sized packets (up to some maximum that either is fixed or depends on the link) are usual in computer networking, for example on the Ethernet, token ring, FDDI, or Internet, as well as for DMA bursts on I/O busses.

All these methods fix the division of bandwidth among sub-channels except for variable TDM, which is thus better suited to handle the burstiness of computer traffic. This is the only architectural difference among them. But there are other architectural differences among multiplexers, resulting from the different ways of coding the basic function of *arbitrating* among the input channels. The fixed schemes do this in a fixed way that is determined which the sub-channels are assigned. This is illustrated at the top of figure 1, where the wide main channel has enough bandwidth to carry all the traffic the input channels can offer. Arbitration is still necessary when a sub-channel is assigned to an input channel; this operation is usually called 'circuit setup'.

With variable TDM there are many ways to arbitrate, but they fall into two main classes, which parallel the two methods of flow control described in the section on links above:

- *Collision* (parallel to backoff): an input channel simply sends its traffic, but has some way to tell whether the traffic was accepted. If not, it 'backs off' by waiting for a while, and then retries. The input channel can get an explicit and immediate collision signal, as on the Ethernet, it can get a delayed collision signal in the form of a 'negative acknowledgment', or it can infer a collision from the lack of an acknowledgment, as in TCP.
- *Scheduling* (parallel to backpressure): an input channel makes a request for service and the multiplexer eventually grants it; I/O busses and token rings work this way. Granting can be centralized, as in many I/O busses, or distributed, as in a daisy-chained bus or a token ring like FDDI.

Flow control means buffering, as we saw earlier, and there are several ways to arrange buffering around a multiplexer, shown on the left side of figure 1. Having the buffers near the arbitration point is good because it reduces the round-trip time *r* and hence the size of the buffers. Output buffering is good because it allows arbitration to ignore contention for the output until the buffer fills up, but the buffer may cost more because it has to accept traffic at the total bandwidth of all the inputs. A switch implemented by a shared memory pays this cost automatically, and the shared memory acts as a shared buffer for all the outputs.

A multiplexer can be centralized, like a T1 multiplexer or a crosspoint in a crossbar switch, or it can be distributed along a bus. It seems natural to use scheduling with a centralized multiplexer and collision with a distributed one, but the examples of the Monarch memory switch[14] and the token ring described below show that the other combinations are also possible.

Multiplexers can be cascaded to increase the fan-in. This structure is usually combined with a converter. For example, 24 voice lines, each with a bandwidth of 64 Kb/s, are multiplexed to one 1.5 Mb/s T1 line, 30 of these are multiplexed to one 45 Mb/s T3 line, and 50 of these are multiplexed to one 2.4 Gb/s OC-48 fiber which carries 40,000 voice sub-channels. In the Vax 8800, 16 Unibuses are multiplexed to one BI bus, and 4 of these are multiplexed to one internal processor-memory bus.

Demultiplexing uses the same physical mechanisms as multiplexing, since one is not much use without the other. There is no arbitration, however; instead, there is *addressing*, since the input channel must select the proper output channel to receive each sub-channel. Again both centralized and distributed versions are possible, as the right side of figure 1 shows. A distributed implementation broadcasts the input channel to all the output channels, and an address decoder picks off the sub-channel as its data fly past. Either way it's easy to broadcast a sub-channel to any number of output channels.

Broadcast networks

From the viewpoint of the preceding discussion of links, a broadcast network is a link that carries packets, roughly one at a time, and has lots of receivers, all of which see all the packets. Each packet carries a *destination address*, each receiver knows its own address, and a receiver's job is to pick out its packets. It's also possible to view a broadcast network as a special kind of switched network, taking the viewpoint of the next section.

Viewed as a link, a broadcast network has to solve the problems of arbitration and addressing. Addressing is simple, since all the receivers see all the packets. All that is needed is 'address filtering' in the receiver. If a receiver has more than one address the code for this may get tricky, but a simple, if costly, fallback position is for the receiver to accept all the packets, and rely on some higher-level mechanism to sort out which ones are really meant for it.

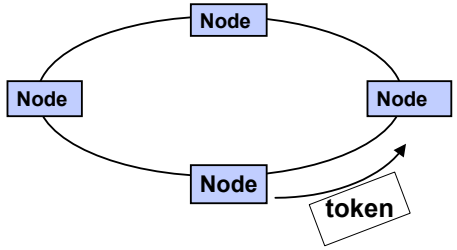
The tricky part is arbitration. A computer's I/O bus is an example of a broadcast network, and it is one in which each device requests service, and a central 'arbiter' *grants* bus access to one device at a time. In nearly all broadcast networks that are called networks, it is an article of religion that there is no central arbiter, because that would be a single point of failure, and another scheme would be required so that the distributed nodes could communicate with it[15]. Instead, the task is distributed among all the senders. As with link arbitration in general, there are two ways to do it: scheduling and contention.

[14] R. Rettberg et al.: The Monarch parallel processor hardware design. *IEEE Computer* 23, 18-30 (1990)

[15] There are times when this religion is inappropriate. For instance, in a network based on cable TV there is a highly reliable place to put the central arbiter: at the head end (or, in a fiber-to-the-neighborhood system, in the fiber-to-coax converter. And by measuring the round-trip delays between the head end and each node, the head end can broadcast "node *n* can make its request now" messages with timing which ensures that a request will never collide with another request or with other traffic.

Arbitration by scheduling: Token rings

Scheduling is deterministic, and the broadcast networks that use it are called 'token rings'. The idea is that each node is connected to two neighbors, and the resulting line is closed into a circle or ring by connecting the two ends. Bits travel around the ring in one direction. Except when it is sending or receiving its own packets, a node retransmits every bit it receives. A single 'token' circulates around the ring, and a node can send when the token arrives at the node. After sending one or more packets, the node regenerates the token so that the next node can send. When its packets have traveled all the way around the ring and returned, the node 'strips' them from the ring. This results in round-robin scheduling, although there are various ways to add priorities and semi-synchronous service.



Rings are difficult to engineer because of the closure properties they need to have:

- *Clock synchronization*: each node transmits everything that it receives except for sync marks and its own packets. It's not possible to simply use the receive clock for transmitting because errors in decoding the clock will accumulate, so the node must generate its own clock. However, it must keep this clock very close to the clock of the preceding node on the ring to keep from having to add sync marks or buffer a lot of data.
- *Maintaining the single token*: with multiple tokens the broadcasting scheme fails. With no tokens, no one can send. So each node must monitor the ring. When it finds a bad state, it cooperates with other nodes to clear the ring and elect a 'leader' who regenerates the token. The strategy for election is that each node has a unique ID. A node starts an election by broadcasting its ID. When a node receives the ID of another node, it forwards it unless its own ID is larger, in which case it sends its own ID. When a node receives its own ID, it becomes the leader; this works because every other node has seen the leader's ID and determined that it is larger than its own. Compare this with the Paxos scheme for electing a leader (in handout 18).
- *Preserving the ring connectivity* in spite of failures. In a simple ring, the failure of a single node or link breaks the ring and stops the network from working at all. A 'dual-attachment' ring is actually two rings, which can run in parallel when there are no failures. If a node fails, splicing the two rings together as shown in figure 3 restores a single ring. Tolerating a single failure can be useful for a ring that runs in a controlled environment like a machine room, but is not of much value for a LAN where there is no reason to believe that only one node or link

will fail. FDDI has dual attachment because it was originally designed as a machine room interconnect; today this feature adds complexity and confuses customers.

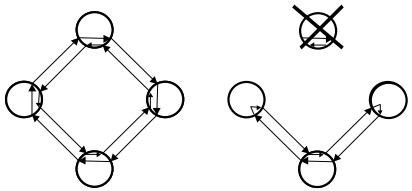


Fig. 3: A dual-attachment ring tolerates failure of one node

- A practical way to solve this problem is to connect all the nodes to a single 'hub' in a so-called 'star' configuration, as shown in figure 4. The hub detects when a node fails and cuts it out of the ring. If the hub fails, of course, the entire ring goes down, but the hub is a simple, special-purpose device installed in a wiring closet or machine room, so it's much less likely to fail than a node. The drawback of a hub is that it contains much of the hardware needed for the switches discussed in the next lecture, but doesn't provide any of the performance gains that switches do.

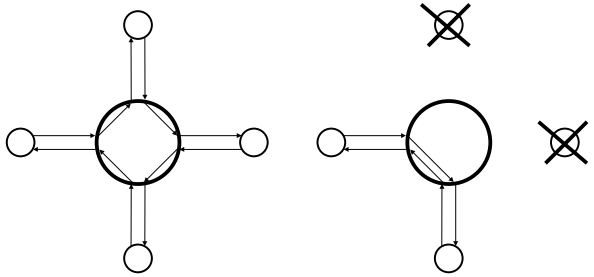


Fig. 4: A ring with a hub tolerates multiple failures

In spite of these problems, two token rings are in wide use (though much less wide than Ethernet, and rapidly declining): the IBM token ring and FDDI. In the case of the IBM token ring this happened because of IBM's marketing prowess; the salesmen persuaded bankers that they didn't want precious packets carrying dollars to collide on the Ethernet. In the case of FDDI it happened because most people were busy deploying Ethernet and developing Ethernet bridges and switches; the FDDI standard gained momentum before anyone noticed that it's not very good.

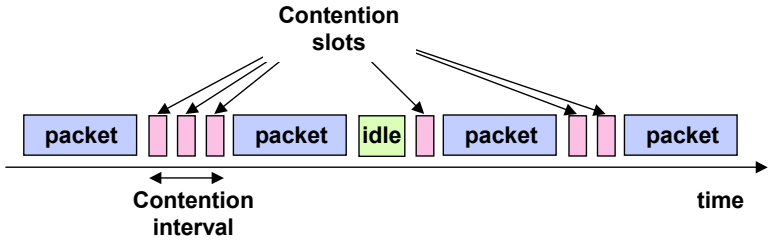
Arbitration by contention: Ethernet

Contention, using backoff, is probabilistic, as we saw when we discussed backoff on links. It wastes some bandwidth in unsuccessful transmissions. In the case of a broadcast LAN,

bandwidth is wasted whenever two packets overlap at the receiver; this is called a 'collision'. How often does it happen?

In a 'slotted Aloha' network a node can't tell that anyone else is sending; this model is appropriate for the radio transmission from feeble terminals to a central hub that was used in the original Aloha network. If everyone sends the same size packet (desirable in this situation because long packets are more likely to collide) and the senders are synchronized, we can think of time as a sequence of 'slots', each one packet long. In this situation exponential backoff gives an efficiency of $1/e = .37$ (see below).

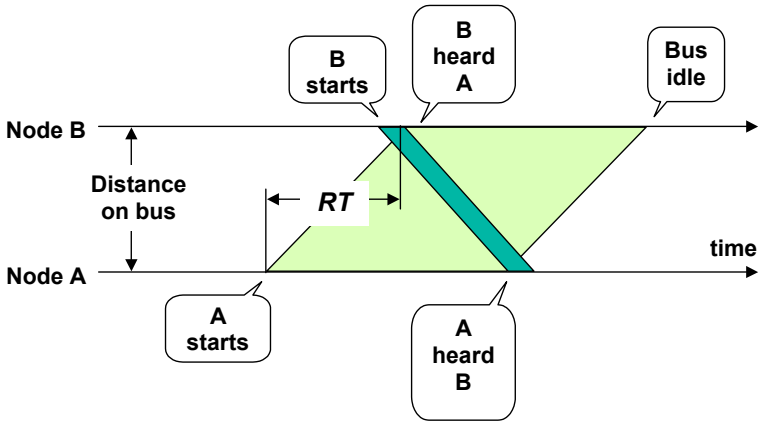
If a node that isn't sending can tell when someone else is sending ('carrier sense'), then a potential sender can 'defer' to a current sender. This means that once a sender's signal has reached all the nodes without a collision, it has 'acquired' the medium and will be able to send the rest of its packet without further danger of collision. If a sending node can tell when someone else is sending ('collision detection') both can stop immediately and back off. Both carrier sense and collision detection are possible on a shared bus and are used in the Ethernet. They are also possible in a system with a head end that can hear all the nodes, even if the nodes can't hear each other: the head end sends a collision signal whenever it hears more than one sender.



The critical parameter for a 'CSMA/CD' (Carrier Sense Multiple Access/Collision Detection) network like the Ethernet is the round-trip time for a signal to get from one node to another and back; see the figure below. After a maximum round-trip time *RT* without a collision, a sender knows it has acquired the medium. For the Ethernet this time is about 50 μs = 64 bytes at the 10 Mbits/sec transmission time; this comes from a maximum diameter of 2 km = 10 μs (at 5 μs/km for signal propagation in cable), 10 μs for the time a receiver needs to read the 'preamble' of the packet and either synchronize with the clock or detect a collision, and 5 μs to pass through a maximum of two repeaters, which is 25 μs, times 2 for the round trip. A packet must be at least this long or the sender might finish sending it before detecting a collision, in which case it wouldn't know whether the transmission was successful.

The 100 Mbits/sec fast Ethernet has the same minimum packet size, and hence a maximum diameter of 5 μs, 10 times smaller. Gigabit Ethernet has a maximum diameter of .5 μs or 100 m. However, it normally operates in 'full-duplex' mode, in which a wire connects only two nodes and is used in only one direction, so that two wires are needed for each pair of nodes. With this arrangement only one node ever sends on a given wire, so there is no multiplexing and hence no need for arbitration. The CSMA/CD stuff is still in the standard because any change to the standard would mean a whole new standards process, during which lots of people would try to

introduce their own crazy ideas. It's much faster and safer to leave in the unused features. In any case, the logic for CSMA/CD must be in the chips so that they can run at the slower speeds as well, in order to ensure that the network will still work no matter how it's wired up.



Here is how to calculate the throughput of Ethernet. If there are k nodes trying to send, p is the probability of one station sending, and r is the round trip time, then the probability that one of the nodes will succeed is $A = kp(1-p)^{k-1}$. This has a maximum at $p=1/k$, and the limit of the maximum for large k is $1/e = .37$. So if the packets are all of minimum length this is the efficiency. The expected number of tries is $1/A = e = 2.7$ at this maximum, including the successful transmission. The waste, also called the 'contention interval', is therefore $1.7r$. For packets of length l the efficiency is $l/(l + 1.7r) = 1/(1 + 1.7r/l) \sim 1 - 1.7r/l$ when $1.7r/l$ is small. The biggest packet allowed on the Ethernet is 1.5 Kbytes = 20 r, and this yields an efficiency of 91.5% for the maximum r. Most networks have a much smaller r than the maximum, and correspondingly higher efficiency.

But how do we get all the nodes to behave so that $p=1/k$? This is the magic of exponential backoff. A is quite sensitive to p, so if several nodes are estimating k too small they will fail and increase their estimate. With carrier sense and collision detect, it's OK to start the estimate at 0 each time as long as you increase it rapidly. An Ethernet node does this, doubling its estimate at each backoff by doubling its maximum backoff time, and making it smaller by resetting its backoff time to 0 after each successful transmission. Of course each node must chose its actual backoff time randomly in the interval [0 .. maximum backoff]. As long as all the nodes obey the rules, they share the medium fairly, with one exception: if there are very few nodes, say two, and one has lots of packets to send, it will tend to 'capture' the network because it always starts with 0 backoff, whereas the other nodes have experienced collisions and therefore has a higher backoff.

The TCP version of exponential backoff doesn't have the benefit of carrier sense or collision detection. On the other hand, routers have some buffering, so it's not necessary to avoid collisions completely. As a result, TCP has 'slow start'; it transmits slowly until it gets some acknowledgments, and then speeds up. When it starts losing packets, it slows down. Thus each

sender's estimate of k oscillates around the true value (which of course is always changing as well).

All versions of backoff arbitration have the problem that a selfish sender that doesn't obey the rules can get more than its share. This isn't a problem for Ethernet because there are very few sources of interface chips, and each one has been careful engineered to behave correctly. For TCP there are similarly few sources of widely used code, but on the other hand the code can fairly easily be patched to misbehave. This doesn't have much benefit for clients in the Internet, however, since most traffic is from servers to clients. It might have some benefit for servers, but they are usually run by organizations that can be made to suffer if detected in misbehavior. So in both cases social mechanisms keep things working.

Since the Ethernet works by sharing a passive medium, a failing node can only cause trouble by 'babbling', transmitting more than the protocol allows. The most likely form of babbling is transmitting all the time, and Ethernet interfaces have a very simple way of detecting this and shutting off the transmitter.

Most Ethernet installations do not use a single wire with all the nodes attached to it. Although this configuration is possible, the hub arrangement shown in figure 5 is much more common (contrary to the expectations of the Ethernet's designers). An Ethernet hub just repeats an incoming signal to all the nodes. Hub wiring has three big advantages:

It's easier to run Ethernet wiring in parallel with telephone wiring, which runs to a hub.

The hub is a good place to put sensors that can measure traffic from each node and switches that can shut off faulty or suspicious nodes.

Once wiring goes to a hub, it's easy to replace the simple repeating hub with a more complicated one that does some amount of switching and thus increases the total bandwidth. It's even possible to put in a multi-protocol hub that can detect what protocol each node is using and adjust itself accordingly. This arrangement is standard for fast Ethernet, which runs at 100 Mbits/sec instead of 10, but is otherwise very similar. A fast Ethernet hub automatically handles either speed on each of its ports.

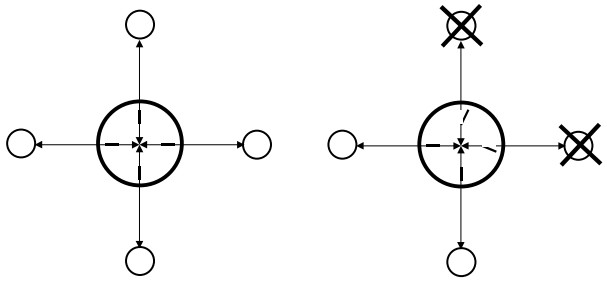
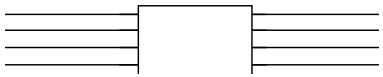
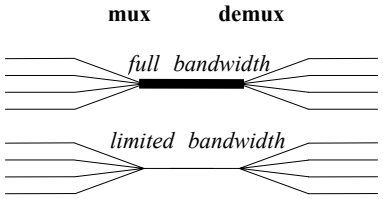


Fig. 5: An Ethernet with a hub can switch out failed nodes

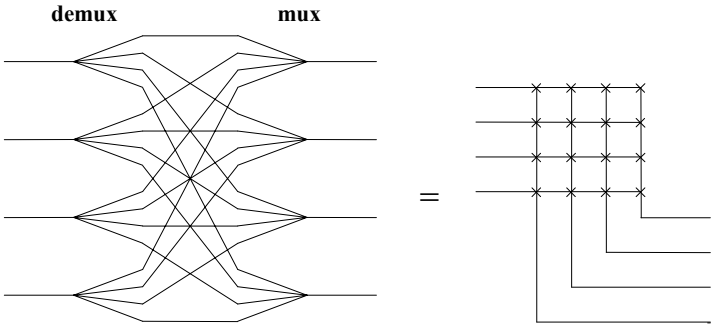
A drawback is that the hub is a single point of failure. Since it is very simple, this is not a major problem. It would be possible to connect each network interface to two hubs, and switch to a backup if the main hub fails, but people have not found it necessary to do this. Instead, nodes that need very high availability of the network have two network interfaces connected to two different hubs.



(a) The usual representation of a switch



(b) Mux–demux code



(c) Demux–mux code, usually drawn as a crossbar

Fig. 6. Switches.

Switches

The modern trend in local area networks, however, is to abandon broadcast and replace hubs with switches. A switch has much more silicon than a hub, but silicon follows Moore's law and gets cheaper by 2x every 18 months. The cost of the wires, connectors, and packaging is the same, and there is much more aggregate bandwidth. Furthermore, a switch can have a number of slow ports and a few fast ones, which is exactly what you want to connect a local group of clients to a higher bandwidth 'backbone' network that has more global scope.

In the rest this handout we describe the different kinds of switches, and consider ways of connecting switches with links to form a larger link or switch.

A switch is a generalization of a multiplexer or demultiplexer. Instead of connecting one link to many, it connects many links to many. Figure 6(a) is the usual drawing for a switch, with the input links on the left and the output links on the right. We view the links as simplex, but usually they are paired to form full-duplex links so that every input link has a corresponding output link which sends data in the reverse direction. Often the input and output links are connected to the same nodes, so that the switch allows any node to send to any other.

A basic switch can be built out of multiplexers and demultiplexers in the two ways shown in figure 6(b) and 6(c). The latter is sometimes called a 'space-division' switch since there are separate multiplexers and demultiplexers for each link. Such a switch can accept traffic from every link provided each is connected to a different output link. With full-bandwidth multiplexers this restriction can be lifted, usually at a considerable cost. If it isn't, then the switch must arbitrate among the input links, generalizing the arbitration done by its component multiplexers, and if input traffic is not reordered the average switch bandwidth is limited to 58% of the maximum by 'head-of-line blocking'.[16]

Some examples reveal the range of current technology. The range in latencies for the LAN switches and IP routers is because they receive an entire packet before starting to send it on. For Email routers, latency is not usually considered important.

Medium	*Link*	*Bandwidth*		*Latency*		*Links*
Alpha chip	register file	60	GB/s	.8	ns	6
Wires	Cray T3D	122	GB/s	1	µs	2K
LAN	Switched gigabit Ethernet	4	GB/s	5-100	µs	32
	FDDI Gigaswitch	275	MB/s	10–400	µs	22
	Switched Ethernet	40	MB/s	100–1200	µs	32
IP router	many	1-6400	MB/s	50–5000	µs	16
Email router	SMTP	10-1000	KB/s	1-100	s	many
Copper pair	Central office	80	MB/s	125	µs	50K

Storage can serve as a switch of the kind shown in figure 6(b). The storage device is the common channel, and queues keep track of the addresses that input and output links should use.

[16] M. Karol et al., Input versus output queuing on a space-division packet switch. *IEEE Transactions on Communications* **35**, 12 (Dec. 1987), pp 1347-1356.

If the switching is coded in software, the queues are kept in the same storage, but sometimes they are maintained separately. Bridges and routers usually code their switches this way.

Pipelines

What can we make out of a collection of links and switches. The simplest thing to do is to concatenate two links using a connecting node, as in figure 7, making a longer link. This structure is sometimes called a 'pipeline'.

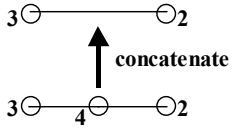
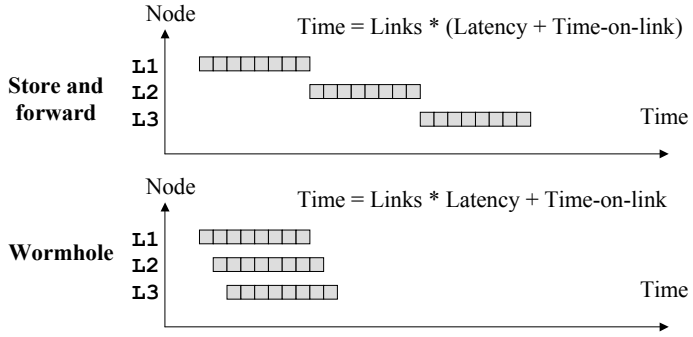


Fig. 7. Composing switches by concatenating.

The only interesting thing about it is the rules for forwarding a single traffic unit:

Can the unit start to be forwarded before it is completely received ('wormholes' or 'cut-through')[17], and

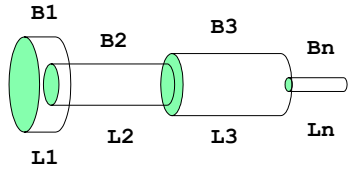
Can parts of two units be intermixed on the same link ('interleaving'), or must an entire unit be sent before the next one can start?



As we shall see, wormholes give better performance when the time to send a unit is not small, and often it is not because often a unit is an entire packet. Furthermore, wormholes mean that a switch need not buffer an entire packet.

The latency of the composite link is the total delay of its component links (the time for a single bit to traverse the link) plus a term that reflects the time the unit spends entering links (or leaving them, which takes the same time). With no wormholes a unit doesn't start into link *i* until all of it has left link *i*-1, so this term is the sum of the times the unit spends entering each link (the size of the unit divided by the bandwidth of the link). With wormholes and interleaving, it is the time entering the slowest link, assuming that the granularity of interleaving is fine enough. With wormholes but without interleaving, each point where a link feeds a slower one adds the difference in the time a unit spends entering them; where a link feeds a faster one there is no added time because the faster link gobbles up the unit as fast as the slower one can deliver it.

[17] L. Ni and P. McKinley: A survey of wormhole routing techniques in direct networks. *IEEE Computer* 26, 62-76 (1993).



Latency = L1 + L2 + L3 + Ln

This rule means that a sequence of links with increasing times is equivalent to the slowest, and a sequence with decreasing times to the fastest, so we can summarize the path as alternating slow and fast links $s_1 f_1\ s_2 f_2 \ldots s_n f_n$ (where f_n could be null), and the entering time is the total time to enter slow links minus the total time to enter fast links. We summarize these facts:

Wormhole	*Interleaving*	*Time on links*
No	—	$\Sigma\ t_i$
Yes	No	$\Sigma\ ts_i - \Sigma\ tf_i = \Sigma\ (ts_i - tf_i)$
Yes	Yes	max t_i

The moral is to use either wormholes or small units, and to watch out for alternating fast and slow links if you don't have interleaving. However, a unit shouldn't be too small on a variable TDM link because it must always carry the overhead of its address. Thus ATM cells, with 48 bytes of payload and 5 bytes of overhead, are about the smallest practical units (though the Cambridge slotted ring used cells with 2 bytes of payload). This is not an issue for fixed TDM, and indeed telephony uses 8 bit units.

There is no need to use wormholes for ATM cells, since the time to send 53 bytes is small in the intended applications. But Autonet, with packets that take milliseconds to transmit, uses wormholes, as do multiprocessors like the J-machine[18] which have short messages but care about every microsecond of latency and every byte of network buffering. The same considerations apply to pipelines.

[18] W. Dally: A universal parallel computer architecture. *New Generation Computing* 11, 227-249 (1993).

Meshes

If we replace the connectors with switch nodes, we can assemble a mesh like the one in figure 8. The mesh can code the bigger switch above it; note that this switch has the same nodes on the input and output links. The heavy lines in both the mesh and the switch show the path from node 3 to node 2. The pattern of links between internal switches is called the 'topology' of the mesh. The figure is oversimplified in at least two ways: Any of the intermediate nodes might also be an end node, and the Internet has 300 million nodes rather than 4.

The new mechanism we need to make this work is *routing*, which converts an address into a 'path', a sequence of decisions about what output link to use at each switch. Routing is done with a map from addresses to output links at each switch. In addition the address may change along the path; this is coded with a second map, from input addresses to output addresses.

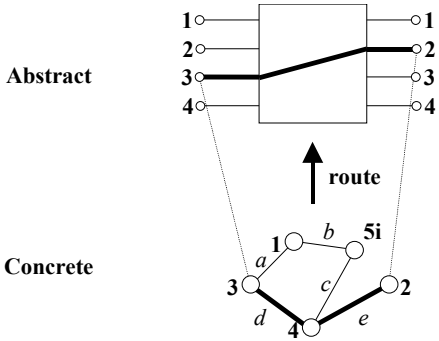


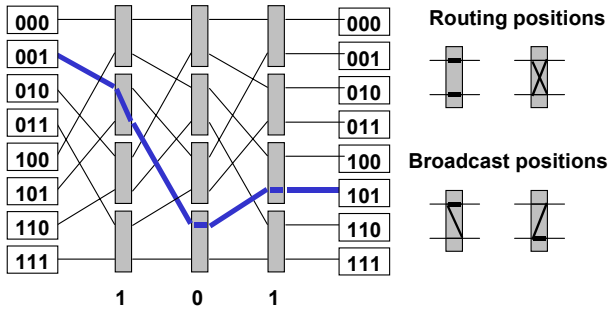
Fig. 8. Composing switches in a mesh.

What spec does a mesh network satisfy? We saw earlier that a broadcast network provides unreliable FIFO delivery. In general, a mesh provides unreliable unordered delivery, because the routes can change, allowing one packet to overtake another, even if the links are FIFO. This is fine for IP on the Internet, which doesn't promise FIFO delivery. When switches are used to extend a broadcast LAN transparently, however, great care has to be taken in changing routes to preserve the FIFO property, even though it has very little value to most clients. This use of switching is called 'bridging'.

Addresses

There are three kinds of addresses. In order of increasing cost to code the maps, and increasing convenience to the end nodes, they are:

- *Source* addresses: the address is just the sequence of output links to use; each switch strips off the one it uses. In figure 8, the source addresses of node 2 from node 3 are (*d, e*) and (*a, b, c, e*).The IBM token ring and several multiprocessors (including the MIT J-machine and the Cosmic Cube[19]) use this. A variation distributes the source route across the path; the address (called a 'virtual circuit') is local to a link, and each switch knows how to map the addresses on its incoming links. ATM uses this variation, and so does the 'shuffle-exchange' network shown below.



- *Hierarchical* addresses: the address is hierarchical. Each switch corresponds to one node in the address tree and knows what links to use to get to its siblings, children, and parent. The Internet[20] and cascaded I/O busses use this.
- *Flat* addresses: the address is flat, and each switch knows what links to use for every address. Broadcast networks like Ethernet and FDDI use this; the code is easy since every receiver sees all the addresses and can just pick off those destined for it. Bridged LANs also use flat routing, falling back on broadcast when the bridges lack information about where an end-node address is. The mechanism for routing 800 telephone numbers is mainly flat.

Deadlock

Traffic traversing a composite link needs a sequence of resources (most often buffer space) to reach the end. Usually it acquires a resource while holding on to existing ones, since you need to get the next buffer before you can free the current one. This means that deadlock is possible. The left side of figure 9 shows the simplest case: two nodes with a single buffer pool in each, and links connecting them. If traffic must acquire a buffer at the destination before giving up its buffer at the source, it is possible for all the messages to deadlock waiting for each other to release their buffers.[21]

The simple rule for avoiding deadlock is well known (see handout 14): define a partial order on the resources, and require that a resource cannot be acquired unless it is greater in this order than all the resources already held. In our application it is usual to treat the links as resources and require paths to be increasing in the link order. Of course the ordering relation must be big enough to ensure that a path exists from every sender to every receiver.

[19] C. Seitz: The cosmic cube. *Communications of the ACM* 28, 22-33 (1985)
[20] W. Stallings, IPV6: The new Internet protocol. IEEE Communications 34, 7 (Jul 1996), pp 96-109.
[21] Actually, this simple configuration can only deadlock if each node fills up with traffic going to the other node. This is very unlikely; usually some of the buffers will hold traffic for other nodes to the left or right, and this will drain out in time.

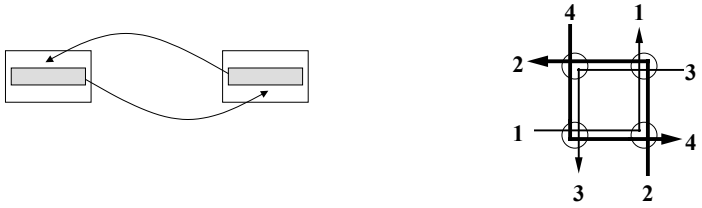


Fig. 9. Deadlock. The version on the left is simplest, but can't happen with more than 1 buffer/node

The right side of figure 9 shows what can happen even at one cell of a simple rectangular grid if this problem is ignored. The four paths use links as follows: 1—EN, 2—NW, 3—WS, 4—SE. There is no ordering that will allow all four paths, and if each path acquires its first link there is deadlock.

The standard order on a grid is: $l_1 < l_2$ iff they are head to tail, and either they point in the same direction, or l_1 goes east or west and l_2 goes north or south. So the rule is: "Go east or west first, then north or south." On a tree $l_1 < l_2$ iff they are head to tail, and either both go up toward the root, or l_2 goes down away from the root. The rule is thus "First up, then down." On a DAG impose a spanning tree and label all the other links up or down arbitrarily; the Autonet does this.

Note that this kind of rule for preventing deadlock may conflict with an attempt to optimize the use of resources by sending traffic on the least busy links.

Although figure 9 suggests that the resources being allocated are the links, this is a bit misleading. It is the buffers in the receiving nodes that are the physical resource in short supply. This means that it's possible to multiplex several 'virtual' links on a single physical link, by dedicating separate buffers to each virtual link. Now the virtual links are resources that can run out, but the physical links are not. The Autonet does not do this, but it could, and other mesh networks such as AN2[22] have done so, as do modern multiprocessor interconnects.

Topology

In the remainder of the handout, we study mechanisms for routing in more detail.[23] It's convenient to divide the problem into two parts: computing the topology of the network, and making routing decisions based on some topology. We begin with topology, in the context of a collection of links and nodes identified by index types `L` and `N`. A topology `T` specifies the nodes that each link connects. For this description it's not useful to distinguish routers from hosts or end-nodes, and indeed in most networks a node can play both roles.

[22] T. Anderson et al., High-speed switch scheduling for local area networks. *ACM Transactions on Computer Systems* **11**, 4 (Nov. 1993), pp 319-352.

[23] This is a complicated subject, and our treatment leaves out a lot. An excellent reference is R. Perlman, *Interconnections: Bridges and Routers* , Addison-Wesley, 1992. Chapter 4 on source routing bridges is best left unread.

These are simplex links, with a single sender and a single receiver. We have seen that a broadcast LAN can be viewed as a link with n senders and receivers. However, for our current purposes it is better to model it as a switch with $2n$ links to and from each attached node. Concretely, we can think of a link to the switch as the physical path from a node onto the LAN, and a link from the switch as the physical path the other way together with the address filtering mechanism.

Note that a path is not uniquely determined by a sequence of nodes (much less by endpoints), because there may be multiple links between two nodes. This is why we define a path as `SEQ L` rather than `SEQ N`. Note also that we are assuming a global name space `N` for the nodes; this is usually coded with some kind of UID such as a LAN address, or by manually assigned addresses like IP addresses. If the nodes don't have unique names, life becomes a lot more confusing.

We name links with local names that are relative to the sending node, rather than with global names. This reflects the fact that a link is usually addressed by an I/O device address. The link from a broadcast LAN node to another node connected to that LAN is named by the second node's LAN address.

```
MODULE Network[
    L                                                    % Link; local name
    N ]                                                  % Node; global name

TYPE Ns = SET N
    T  = N -> L -> N SUCHTHAT (\ t | t.dom={n|true})     % Topology; defined at each N
    P  = [n, r: SEQ L] WITH {"<=":=Prefix}               % Path starting at n
```

Here `t(n)(l)` is the node reached from node `n` on link `l`. For the network of figure 8,

```
t(3)(a) = 1
t(3)(d) = 4
t(1)(a) = 3
t(1)(b) = 5i
```

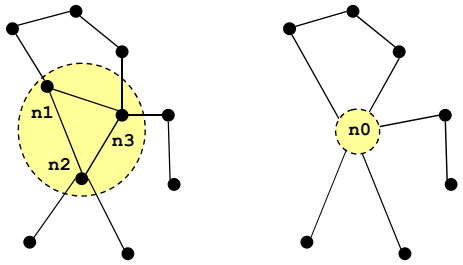
etc.

Note that a `T` is defined on every node, though there may not be any links from a node.

The `End` function computes the end node of a path. A `P` is actually a path if `End` is defined on it, that is, if each link actually exists. A path is acyclic if the number of distinct nodes on it is one more than the number of links. We can compute all the nodes on a path and all the paths between two nodes. All these notions only make sense in the context of a topology that says how the nodes and links are hooked up.

```
FUNC End(t, p) -> N = RET (p.r = {} => p.n [*] End(t, P{t(p.n)(p.r.head), p.r.tail})

FUNC IsPath(t, p) -> Bool = RET End!(t, p)

FUNC Prefix(p1, p2) -> Bool = RET p1.n = p2.n /\ p1.r <= p2.r

FUNC Nodes(t, p) -> Ns = RET {p' | p' <= p | End(t, p')}

FUNC IsAcyclic(t, p) -> Bool = RET IsPath(t, p) /\ Nodes(t, p).size = p.r.size + 1

FUNC Paths(t, n1, n2)  -> SET p =
    RET {p | p.n = n1 /\ End(t, p) = n2 /\ IsAcyclic(t, p)}
```

Like anything else in computing, a network can be recursive. This means that a connected sub-network can be viewed as a single node. To make this precise we define the restriction of a topology to a set of nodes, keeping only the links between nodes in the set. Then we can collapse a topology to a smaller one in which a connected `ns` appears as a single representative node `n0`, by replacing all the links into `ns` with links to `n0` and discarding all the internal links. The outgoing links have to be named by pairs `[n, ll]`, since the naming scheme is local to a node; here we use `ll` for the 'lower-level' links of the original `T`. Often collapsing is tied to hierarchical addressing, so that an entire subtree will be collapsed into a single node for the purposes of higher-level routing.



ns = {n1, n2, n3}
n0 IN ns

```
TYPE L = (L + [n, ll])

FUNC Restrict(t, ns) -> T =
     RET (\ n | (\ l | (n IN ns /\ (t(n)(l) IN ns => t(n)(l)) ))

FUNC IsConnected(t, ns) -> Bool =
     RET (ALL n1 :IN ns, n2 :IN ns | Paths(Restrict(t, ns), n1, n2) # {})

FUNC Collapse(t, ns, n0) -> T = n0 IN ns /\  IsConnected(t, ns) =>
     RET (\ n | (\ l |
          (    ~ n IN ns => (t(n)(l) IN ns => n0 [*] t(n)(l))
          [*] n = n0 /\ l IS [n, ll] /\ l.n IN n' /\ ~ t(l.n)(l.ll) IN ns =>
               t(l.n)(l.ll) ) ))
```

How does a network find out what its topology is? Aside from supplying it manually, there are two approaches. In both, each node learns which nodes are 'neighbors', that is, are connected to its links, by sending 'hello' messages down the links.

1. Run a global computation in which one node is chosen to learn the whole topology by becoming the root of a spanning tree. The root collects all the neighbor information and broadcasts what it has learned to all the nodes. The Autonet uses this method.

2. Run a distributed computation in which each node periodically tells its neighbors everything it knows about the topology. In time, any change in a node's neighbors will spread throughout the network. There are some subtleties about what a node should do when it gets conflicting information. The Internet uses this method, which is called 'link-state routing', and calls it OSPF.

In a LAN with many connected nodes, usually most are purely end-nodes, that is, do not do any switching of other people's packets. The end-nodes don't participate in the neighbor computation, since that would be an n^2 process. Instead, only the routers on the LAN participate, and there is a separate scheme for the end-nodes. There are two mechanisms needed:

1. Routers need to know what end-nodes are on the LAN. Each end-node can periodically broadcast its IP address and LAN address, and the routers listen to these broadcasts and cache the results. The cache times out in a few broadcast intervals, so that obsolete information doesn't keep being used. Similarly, the routers broadcast the same information so that end-nodes can find out what routers are available. The Internet often doesn't do this, however. Instead, information about the routers and end-nodes on a LAN is manually configured.

2. An end-node `n1` needs to know which router can reach a node `n2` that it wants to talk to; that is, `n1` needs the value of `sw(n1)(n2)` defined below. To get it, `n1` broadcasts `n2` and expects to get back a LAN address. If node `n2` is on the same LAN, it returns its LAN address. Otherwise a router that can reach `n2` returns the router's LAN address. In the Internet this is done by the address resolution protocol (ARP). Of course `n1` caches this result and times out the cache periodically.

The Autonet paper describes a variation on this, in which end-nodes use an ARP protocol to map Ethernet addresses into Autonet short addresses. This is a nice illustration of recursion in communication, because it turns the Autonet into a 'generic LAN' that is essentially an Ethernet, on top of which IP protocols will do another level of ARP to map IP addresses to Ethernet addresses.

Routing

For traffic to make it through the network, each switch must know which link to send it on. We begin by studying a simplified situation in which traffic is addressed by the `N` of its destination node. Later we consider the relationship between these globally unique addresses and real addresses.

A `SW` tells for each node how to map a destination node into a link[24] on which to send traffic; you can think of it as the dual of a topology, which for each node maps a link to a destination node. Then a route is a path that is chosen by `sw`.

```
TYPE SW  = N -> N -> L

PROC Route(t, sw, n1, n2) -> P = VAR p :IN Paths(t, n1, n2) |
     (ALL p' | p' <= p /\ p'.r # {} ==>
               p'.r.last = sw(End(t, p'{r := p'.r.reml})(n2)) => RET p
```

Here `sw(n1)(n2)` gives the link on which to reach `n2` from `n1`. Note that if `n1 = n2`, the empty path is a possible result. There is nothing in this definition that says the route must be efficient. Of course, `Route` is not part of the code, but simply a spec.

[24] or perhaps a set of links, though we omit this complication here.

We could generalize `SW` to `N -> N -> SET L`, and then

```
PROC Route(t, sw, n1, n2) -> SET P = RET {p :IN Paths(t, n1, n2) |
     (ALL p' | p' <= p /\ p'.r # {} ==>
               p'.r.last IN sw(End(t, p'{r := p'.r.reml})(n2))}
```

We want consistency between `sw` and `t`: the path `sw` chooses actually gets to the destination and is acyclic. Ideally, we want `sw` to choose a cheapest path. This is easy to arrange if everyone knows the topology and the `Cost` function. For concreteness, we give a popular cost function: the length of the path.

```
FUNC IsConsistent(t, sw) -> Bool =
     RET ( ALL n1, n2 | Route(t, sw, n1, n2) IN Paths(t, n1, n2) )

FUNC IsBest(t, sw) -> Bool = VAR best := {p :IN Paths(t,n1,n2) | | Cost(p)}.min |
     RET ( ALL n1, n2 | Cost(Route(t, sw, n1, n2)) = best )

FUNC Cost(p) -> Int = RET p.r.size                    % or your favorite
```

Don't lose sight of the fact that this is not code, but rather the spec for computing `sw` from `t`. Getting `t`, computing `sw`, and using it to route are three separate operations.

There might be more than one suitable link, in which case `L` is replaced by `SET L`, or by a function that gives the cost of each possible `L`. We work out the former:

```
TYPE SW  = N -> N -> SET L

PROC Routes(t, sw, n1, n2) -> SET P = RET { p :IN Paths(t, n1, n2) |
     (ALL p' | p' <= p /\ p'.r # {} ==>
               p'.r.last IN sw(End(t, p'{r := p'.r.reml})(n2)) }

FUNC IsConsistent(t, sw) -> Bool =
     RET ( ALL n1, n2 | Routes(t, sw, n1, n2) <= Paths(t, n1, n2) )

FUNC IsBest(t, sw) -> Bool = VAR best := {p :IN Paths(t,n1,n2) | | Cost(p)}.min |
     RET ( ALL n1, n2 | (ALL p :IN Routes(t, sw, n1, n2) | Cost(p) = best) )
```

Addressing

In a broadcast network addressing is simple: since every node sees all the traffic, all that's needed is a way for each node to recognize its own addresses. In a mesh network the `sw` function in every router has to map each address to a link that leads there. The structure of the address can make it easy or hard for the router to do the switching, and for all the nodes to learn the topology. Not surprisingly, there are tradeoffs.

It's useful to classify addressing schemes as local (dependent on the source) or global (the same address works throughout the network), and as hierarchical or flat.

	Flat	*Hierarchical*
Local	—	Source routing Circuits = distributed source routing: route once, keep state in routers.
Global	LANs: router knows links to everywhere By broadcast By learning Fallback is broadcast, e.g. in bridges.	IP, OSI: router knows links to parent, children, and siblings.

Source routing is the simplest for the switches, since all work of planning the routes is unloaded on the sender and the resulting route is explicitly encoded in the address. The drawbacks are that the address is bigger and, more seriously, that changes to the topology of the network must be reflected in changes to the addresses.

Congestion control

As we have seen, we can view an entire mesh network as a single switch. Like any structure that involves multiplexing, it requires arbitration for its resources. This network-level arbitration is not the same as the link-level arbitration that is requires every time a unit is sent on a link. Instead, its purpose is to allocate the resources of the network as a whole. To see the need for network-level arbitration, consider what happens when some internal switch or link becomes overloaded.

As with any kind of arbitration, there are two possibilities: scheduling, or contention and backoff. Scheduling can be done statically, by allocating a fixed bandwidth to a path or 'circuit' from a sender to a receiver. The telephone system works this way, and it does not allow traffic to flow unless it can commit all the necessary resources. A variation that is proposed for ATM networks is to allocate a maximum bandwidth for each path, but to overcommit the network resources and rely on traffic statistics to make it unlikely that the bluff will be called.

Alternatively, scheduling can be done dynamically by backpressure, as in the Autonet and AN2. We studied this method in connection with links, and the issues are the same in networks. One difference is that the round-trip time may be longer, so that more buffering is needed to support a given bandwidth. In addition, the round-trip time is usually much more variable, because traffic has to queue at each switch. Another difference is that because a circuit that is held up by backpressure may be tying up resources, deadlock is possible.

Contention and backoff are also similar in links and networks; indeed, one of the backoff links that we studied was TCP, which is normally coded on top of a network. When a link or switch is overloaded, it simply drops some traffic. The trouble signal is usually coded by timeout waiting for an ack. There have been a number of proposals for an explicit 'congested' signal, but it's difficult to ensure that this signal gets back to the sender reliably.