Interconnecting Computers: Architecture, Technology, and Economics[1]

Butler W. Lampson

Systems Research Center, Digital Equipment Corporation
One Kendall Sq., Bldg 700, Cambridge, MA 02138
lampson@src.dec.com

Abstract. Modern computer systems have a recursive structure of processing and storage elements that are interconnected to make larger elements:

Functional units connected to registers and on-chip cache.
Multiple processors and caches connected to main memories.
Computing nodes connected by a message-passing local area network.
Local area networks bridged to form an extended LAN.
Networks connected in a wide-area internet.
All the computers in the world exchanging electronic mail.

Above the lowest level of transistors and gates, the essential character of these connections changes surprisingly little over about nine orders of magnitude in time and space. Connections are made up of nodes and links; their important properties are bandwidth, latency, connectivity, availability, and cost. Switching is the basic mechanism for connecting lots of things. There are many ways to implement it, all based on multiplexing and demultiplexing. This paper describes some of them and gives many examples. It also considers the interactions among the different levels of complex systems.

1 Introduction

A point of view is worth 80 points of IQ.
Alan Kay

A computing system is part of a complex web of interconnections. We impose order on this web by organizing it hierarchically: a system is made up of connected subsystems, and is itself connected to other parts of the larger system that contains it. Figure 1 shows some of this structure as it exists today, ranging from an individual processor register to the world-wide Internet. It is striking that there is a range of at least seven orders of magnitude in both the number of components (shown on the right) and the minimum time for communication (shown on the left).

In spite of this enormous variation, the interconnections themselves are remarkably similar in design. Both the interface that an interconnection offers to a system and the structure of the interconnection itself are taken from a small set of design alternatives. This paper describes many of these alternatives and illustrates them with examples drawn from every level of the figure.

[1] Presented at the Conference on Programming Languages and System Architectures, Zurich, March 1994. Published in Lecture Notes in Computer Science **782**, Springer, 1994, pp 1-20

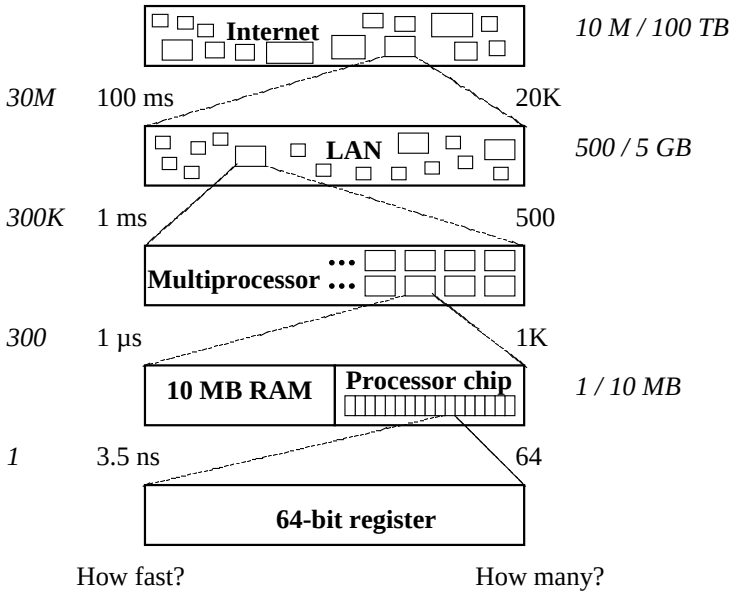


Fig. 1. Scales of interconnection. Relative speed and size are in italics.

An interface deals either with messages or with storage, and is characterized by a few performance parameters: bandwidth, latency, connectivity, and availability. Section 2 describes these variations.

An implementation of a connection is made up of links and nodes. We explain this scheme in Section 3 and study links in Section 4. A node in turn can be a converter, a multiplexer, or a switch; the next three sections are devoted to these components.

A critical property of both links and nodes is that they can be built by composing lower-level links and nodes according to uniform principles. Because different levels in this recursion are so similar, a study of interconnections in general reveals much about the details of any particular one. In the simplest case the interfaces in the composite are the same as those of the whole. This is the subject of Section 8, and Section 9 then treats the general case.

Section 10 gives a brief treatment of fault tolerance, and we end with a conclusion.

Another unifying theme is how the evolution of silicon and fiber optic technology affects interconnection.

As more devices can fit on a single chip, it becomes feasible to use wide on-chip data paths, and to depend on control that is both complex and fast as long as the speed can be obtained by using more gates. The second fact tends to make designs similar at different scales, since it means that a fast, low-level implementation does not have to be extremely simple.

As the bandwidth available on a single fiber rises toward the tens of terabits/second that seems to be feasible in the long run, a big system can increasingly have the same interconnection bandwith as a small one. This too tends to make designs similar at different scales.

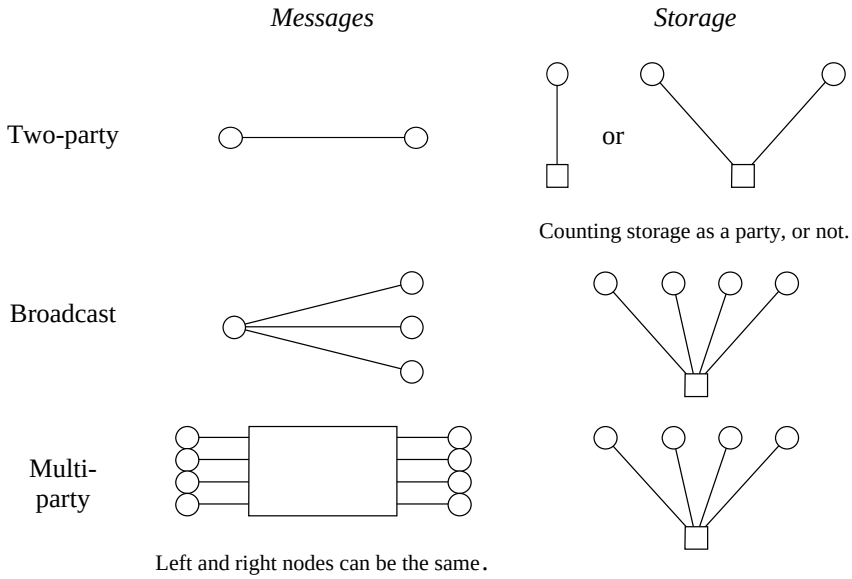


Fig. 2. Communication styles: messages and storage.
Legend: ◯ = active node, □ = storage node

2 Interfaces for communication

In the Turing tarpit everything is possible, but nothing is easy.
Alan Perlis

The duality between states and events is a recurring theme in computer science. For us it is the choice between messages and storage as the style of communication. Both are universal: you can do anything with one that you can do with the other, and we will see how to implement each one using the other. But the differences are important.

Figure 2 shows the system structures for various communication patterns using messages and using storage. The pictures reflect the role of storage as a passive, possible shared repository for data.

Multi-party communication requires addresses, which can be flat or hierarchical. A flat address has no structure: the only meaningful operation (other than communication) is equality. A hierarchical address, sometimes called a path name, is a sequence of flat addresses or simple names, and if one address is a prefix of another, then in some sense the party with the shorter address contains, or is the parent of, the party with the longer one. Usually there is an operation to enumerate the children of an address. Flat addresses are usually fixed size and hierarchical ones variable, but there are exceptions. An address may be hierarchical in the implementation but flat at the inter-

face, for instance an Internet address or a URL in the World Wide Web. The examples below should clarify these points.

2.1 Messages

The interface for messages is the familiar `send` and `receive` operations. The simplest form is a blocking `receive`, in which a process or thread of control doing a `receive` waits until a message arrives. If the receiver wants to go on computing it forks a separate thread to do the `receive`. The alternative is an interrupt when a message arrives; this is much more complicated to program but may be necessary in an old-fashioned system that has no threads or expensive ones.

A range of examples serves to illustrate the possibilities:

System	*Address*	*Sample address*	*Delivery*	
			Ordered	*Reliable*
J-machine[4]	source route	4 north, 2 east	yes	yes
IEEE 802 LAN	6 byte flat	FF F3 6E 23 A1 92	no	no
IP	4 byte hierarchical	16.12.3.134	no	no
TCP	IP + port	16.12.3.134 / 3451	yes	yes
RPC	TCP + procedure	16.12.3.134 / 3451 / Open	yes	yes
E-mail	host name + user	lampson@src.dec.com	no	yes

Usually there is some buffering for messages that have been sent but not yet received, and the sender is blocked (or perhaps gets an error) when the buffering is exhausted. This kind of "back-pressure" is important in many implementations of links as well; see Section 4. If there is no buffering the system is said to be "synchronous" because the `send` and the `receive` must wait for each other; this scheme was introduced in CSP [7] but is unpopular because the implementation must do extra work on every message. The alternative to blocking the `send` is to discard extra messages; the 802 and IP interfaces do this. Except in real-time systems where late messages are useless, an interface that discards messages is usually papered over using backoff and retry; see Section 4.

Message delivery may be ordered, reliable, both, or neither. Messages which may be reordered or lost are often cheaper to implement in a large system, so much cheaper, in fact, that ordered reliable messages are best provided by end-to-end sequence numbering and retransmission on top of unordered unreliable messages [10]. TCP on IP is an example of this, and there are many others.

Often messages are used in an asymmetrical "request–response", "client–server", or "remote procedure call" pattern [2] in which the requester always follows a `send` immediately with a `receive` of a reply, and the responder always follows a `receive` with some computation and a `send` of a response. This pattern simplifies programming because it means that communicating parties don't automatically run in parallel; concurrency can be programmed explicitly as multiple threads if desired.

A message interface may allow broadcast or multicast to a set of receivers. This is useful for barrier synchronization in a multiprocessor; the bandwidth is negligible, but the low latency of the broadcast is valuable. At the opposite extreme in performance

broadcast is also useful for publishing, where latency is unimportant but the bandwidth may be considerable. And broadcast is often used to discover the system configuration at startup; in this application performance is unimportant.

It's straightforward to simulate storage using messages by implementing a storage server that maintains the state of the storage and responds suitably to `load` and `store` messages. This is the simplest and most popular example of the client–server pattern.

2.2 Storage

The interface for storage is the familiar `load` and `store` operations. Like `receive` and `send`, these operations take an address, and they also return or take a data value. Normally they are blocking, but in a high-performance processor the operations the programmer sees are implemented by non-blocking `load` and write-buffered `store` together with extra bookkeeping.

Again, some examples show the range over which this interface is useful:

System	*Address*	*Sample address*	*Data value*
Main memory	32-bit flat	04E72A39	1, 2, 4, or 8 bytes
File system [20]	path name	/udir/bwl/Mail/inbox/214	0-4 Gbytes
World Wide Web	protocol + host name + path name	http://src.dec.com/SRC/docs.html	typed, variable size

Storage has several nice properties as a communication interface. Like request–response, it introduces no extra concurrency. It provides lazy broadcast, since the contents of the storage is accessible to any active party. And most important, it allows caching as an optimization that reduces both the latency of the interface and the bandwidth consumed. Of course nothing is free, and caching introduces the problem of cache coherence [9, 11], the treatment of which is beyond the scope of this paper.

There are two ways to simulate messages using storage. One is to construct a queue of waiting messages in storage, which the receiver can poll. A common example is an input/output channel that takes its commands from memory and delivers status reports to memory. The other method is to intercept the `load` and `store` operations and interpret them as messages with some other meaning. This idea was first used in the Burroughs B-5000 [1], but became popular with the PDP-11 Unibus, in which an input/output device has an assigned range of memory addresses, sees `load` and `store` operations with those addresses, and interprets the combination of address and data as an arbitrary message, for instance as a tape rewind command. Most input/output systems today use this idea under the name "programmed I/O". An example at a different level is the Plan 9 operating system [16], the Unibus of the '90s, in which everything in the system appears in the file name space, and the display device interprets loads and stores as commands to copy regions of the bitmap or whatever. The World Wide Web does the same thing less consistently but on a much larger scale.

2.3 Performance

The performance parameters of a connection are:

— *Latency*: how long a minimum communication takes. We can measure the latency in bytes by multiplying the latency time by the bandwidth; this gives the capacity penalty for each separate operation. There are standard methods for minimizing the effects of latency:

> Caching reduces latency when the cache hits.
> Prefetching hides latency by the distance between the prefetch and the use.
> Concurrency tolerates latency by giving something else to do while waiting.

— *Bandwidth*: how communication time grows with data size. Usually this is quoted for a two-party link. The "bisection bandwidth" is the minimum bandwidth across a set of links that partition the system if they are removed; it is a lower bound on the possible total rate of communication. There are standard methods for minimizing the cost of bandwidth:

> Caching saves bandwidth when the cache hits.
> More generally, locality saves bandwidth when cost increases with distance.
> Combining networks reduce the bandwidth to a hot spot by combining several operations into one, several loads or increments for example [17].

— *Connectivity*: how many parties you can talk to. Sometimes this is a function of latency, as in the telephone system, which allows you to talk to millions of parties but only one at a time.

— *Predictability*: how much latency and bandwidth vary with time. Variation in latency is called "jitter"; variation in bandwidth is called "burstiness". The biggest difference between the computing and telecommunications cultures is that computer communication is basically unpredictable, while telecommunications service is traditionally highly predictable.

— *Availability*: the probability that an attempt to communicate will succeed.

Uniformity of performance at an interface is often as important as absolute performance, because dealing with non-uniformity complicates programming. Thus performance that depends on locality is troublesome, though often rewarding. Performance that depends on congestion is even worse, since congestion is usually much more difficult to predict than locality. By contrast, the Monarch multiprocessor [17] provides uniform, albeit slow, access to a shared memory from 64K processors, with a total bandwidth of 256 Gbytes/sec and a very simple programming model. Since all the processors make memory references synchronously, it can use a combining network to eliminate many hot spots.

3 Implementation components

An engineer can do for a dime what any fool can do for a dollar.
Anonymous

Communication systems are made up of links and nodes. Data flows over the links. The nodes connect and terminate the links. Of course, a link or a node can itself be a communication system made up of other links and nodes; we study this recursive structure in Sections 8 and 9.

There are two kinds of nodes: converters and switches. A converter connects two links of different types, or a terminal link and the client interface. The simplest switches connect one link to many; they are called multiplexers and demultiplexers depending on whether the one link is an output or an input. A general switch connects any one of a set of input links to any one of a set of output links.

The bandwidth of a connection is usually the minimum bandwidth of any link or node. The latency is the sum of several terms:

link time, which consists of
    time of flight for one bit, usually at least half the speed of light, plus
    message size (number of bits) divided by bandwidth;
switching time;
buffer delays;
conversion time (especially at the ends).

The cost of a connection is the total cost of its links and nodes. We study the cost of physical links in Section 4. Nodes are made of silicon and software, and software runs on silicon. Hence the cost of a node is governed by the cost of silicon, which is roughly proportional to area in a mature process. Since 1960 the width of a device (transistor or wire) on a silicon die has been cut in half every five years. The number of devices per unit area increases with the square of the width, and the speed increases linearly. Thus the total amount of computing per unit area, and hence per unit cost (measured in device-cycles), grows with the cube of the width and doubles three times every five years, or every 20 months [6, 13].

The cost of a node therefore tends to zero as long as it effectively uses the ever-increasing number of devices on a chip. There are two consequences:

— Concurrency on the chip is essential, in the form of wide busses and multiple function units.

— Complex control can be made fast as long as it can take advantage of lots of gates. Instead of a large number of sequential steps, each doing a small amount of work, it's possible to have lots of concurrent finite state machines, and to use lots of combinational logic to do more work in a single cycle.

We see typical results in Ethernet interface chips that cost $10, or in a high-bandwidth, low-latency, robust, reliable, low-cost switched network like Autonet [22].

In addition, dramatic improvements in fiber optics mean that almost as much bandwidth is available on long distance links as locally, and at much lower cost that in the past [5].

4 Links

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, and the "+" signs mean that software latency must be added.

Medium	*Link*	*Bandwidth*		*Latency*		*Width*
Alpha chip	on-chip bus	2.2	GB/s	3.6	ns	64
PC board	RAMbus	0.5	GB/s	150	ns	8
	PCI I/O bus	133.0	MB/s	250	ns	32
Wires	HIPPI	100	MB/s	100	ns	32
	SCSI	20	MB/s	500	ns	16
LAN	FDDI	12.5	MB/s	20 +	µs	1
	Ethernet	1.25	MB/s	100 +	µs	1
Wireless	WaveLAN	.25	MB/s	100 +	µs	1
Fiber	OC-48	300	MB/s	5	µs/km	1
Coax cable	T3	6	MB/s	5	µs/km	1
Copper pair	T1	0.2	MB/s	5	µs/km	1
Copper pair	IDSN	16	KB/s	5	µs/km	1
Broadcast	CAP 16	3	MB/s	3	µs/km	6 MHz

A physical link can be unidirectional ("simplex") or bidirectional ("duplex"). A duplex link may operate in both directions at the same time ("full-duplex"), or in one direction at a time ("half-duplex"). A pair of simplex links running in opposite directions form a full-duplex link, as does a half-duplex link in which the time to reverse direction is negligible.

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

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

Switched networks: the telephone system and switched LANs (see Section 7).

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 there must be physical switches to connect the parallel links.

Another use for multiple links is fault tolerance, discussed in Section 10.

Many links do not have a fixed bandwidth that is known to the sender, because of multiplexing inside the link. Instead, some kind of *flow control* is necessary to match the flow of traffic to the link's capacity. A link can provide this in two ways:

— By dropping excess traffic and signaling "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 after every trouble signal and decreases it with each trouble-free send. In this "exponential back-off" scheme the sender is using the wait as an estimate of the link capacity. It is used in the Ethernet and in TCP [14, 8]

— By supplying "back-pressure" that tells the sender how much it can send without suffering losses. This can take the form of start and stop signals, or of "credits" that al-

low a certain amount of 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. The failure mode of the credit scheme is usually less serious. Start–stop is used in the Autonet [22] and on RS-232 serial lines under the name XON-XOFF; credits are used in TCP [8].

Either of these schemes can be implemented on an individual link. An alternative is to let internal links simply drop excess traffic and to implement backoff end-to-end [19]. TCP does this, and it confusingly also uses credits to keep the receiver's buffers from overflowing.

5 Converter nodes

Many converters from one kind of link to another connect a fast link to a slow one and are therefore part of a multiplexer or demultiplexer. Most other converters are for backward compatibility. They are usually cheap, because by the time an old link must be connected to a new one, hardware technology has improved so much that the old link is simple to implement. A glance at the back of a Macintosh or a stereo receiver shows how many different connectors you can buy for a small amount of money.

The converters that terminate connections are another matter. For a simple, synchronous link that is designed along with its end nodes, like the bus between a register file and a functional unit on a processor chip, the converter is implemented in simple, fast hardware and presents a design problem only if a lot of switching is involved.

Terminating a network link is much more complicated because of requirements for standardization and fault-tolerance. Furthermore, the link is usually specified without much attention to the problem of terminating it. A network converter consists of an "adapter" or "controller" together with "driver" software. In computer applications the driver is usually the main source of latency and often a serious bandwidth bottleneck as well, especially when individual messages are small. To see why this is true, consider Amdahl's rule that one instruction of useful work needs one bit of I/O. If a message is 20 bytes (a common size for multiprocessors) and we want to keep the driver overhead below 10%, there are only 16 instructions available for handling each message. It takes a lot of care to handle a message this cheaply [3, 4, 21]. Certainly it cannot be done unless the controller and the driver are designed together.

6 Multiplexer nodes

A multiplexer combines traffic from several input links onto one output link, and a demultiplexer 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 3 shows various examples.

There are three main reasons for multiplexers:

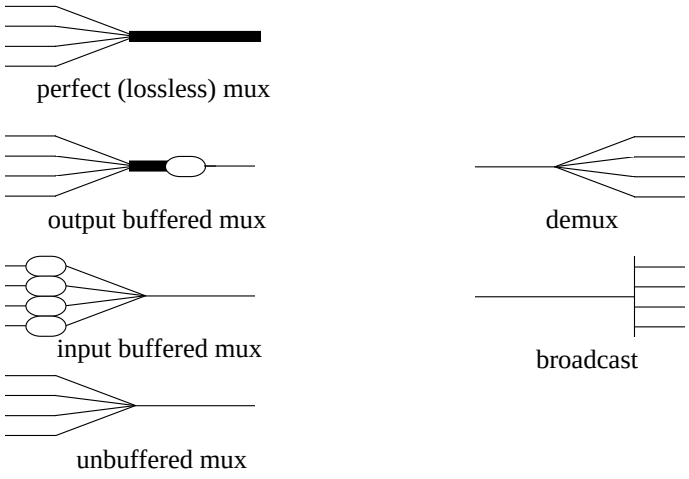


Fig. 3. Multiplexers and demultiplexers. Traffic flows from left to right.

— Traffic must flow between one node and many, 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.

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.

— *Code division* multiplexing (CDM) 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 a new variety of cellular telephony.

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. A 1.5 Mbit/sec T1 line, 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.

— *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 which 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 implementing 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 3, 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 Section 4.

— *Collision*: an input channel simply sends its traffic, but has some way to tell whether it 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 [14], or it can infer a collision from the lack of an acknowledgment, as in TCP [8].

— *Scheduling*: 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 [18].

Flow control means buffering, as we saw in Section 4, and there are several ways to arrange buffering around a multiplexer, shown on the left side of Figure 3. Having the buffers near the arbitration point is good because it reduces *r* and hence the size of the buffers. Output buffering is good because it tolerates a larger *r* across the multiplexer, but the buffer may cost more because it has to accept traffic at the total bandwidth of all the inputs.

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 [17] and the token ring [18] 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 implementations are possible, as the right side of figure 3 shows. In a distributed implementation the input channel is broadcast to each output channel, 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.

7 Switch nodes

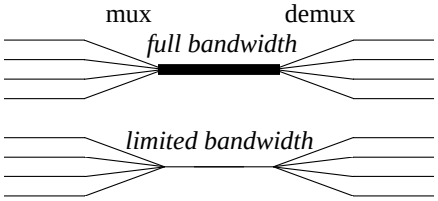
A switch is a generalization of a multiplexer or demultiplexer. Instead of connecting one link to many, it connects many links to many. Figure 4(a) is the usual drawing for a switch, with the input links on the left and the output links on the right. We view the links as simplex, but usually they are paired to form full-duplex links so that every input link has a corresponding output link which sends data in the reverse direction.

A basic switch can be built out of multiplexers and demultiplexers in the two ways shown in Figure 4(b) and 4(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".

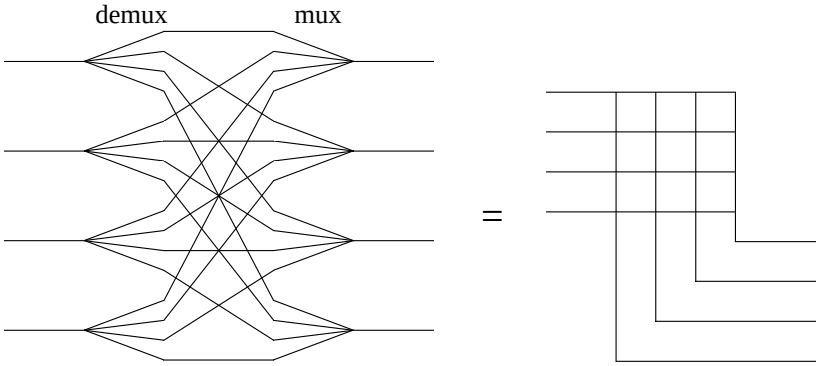
Some examples reveal the range of current technology. The range in latencies for the LAN switches is because they receive an entire packet before starting to send it on.

Medium	*Link*	*Bandwidth*		*Latency*		*Links*
Alpha chip	register file	13.2	GB/s	3.6	ns	6
Wires	Cray T3D	85	GB/s	1	µs	2K
	HIPPI	1.6	GB/s	1	µs	16
LAN	FDDI Gigaswitch	275	MB/s	10–400	µs	22
	Switched Ethernet	10	MB/s	100–1200	µs	8
Copper pair	Central office	80	MB/s	125	µs	50K

(a) The usual representation of a switch



(b) A mux–demux implementation



(c) A demux–mux implementation, often drawn as a crossbar

Fig. 4. Switches.

It is also possible to use storage as a switch of the kind shown in Figure 4(b). The storage device is the common channel, and queues keep track of the addresses that input and output links should use. If the switching is implemented in software the queues are kept in the same storage, but sometimes they are maintained separately. Bridges and routers usually implement their switches this way.

8 Composing switches

Any idea in computing is made better by being made recursive.
Brian Randell

Having studied the basic elements out of which interconnections are made, we can now look at how to compose them. We begin by looking at how to compose switches to make a larger switch with the same interface; the next section examines the effect of changing the interface.

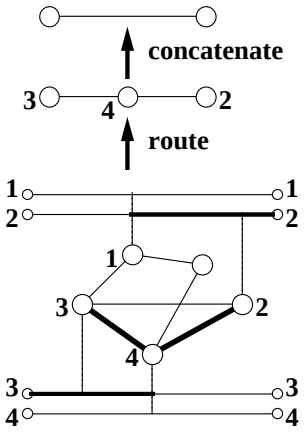


Fig. 5. Composing switches.

8.1 Concatenating links

First we observe that we can concatenate two links using a connector, as in the top half of Figure 5, to make a longer link. This structure is sometimes called a "pipeline". The only interesting thing about it is the rule for forwarding a single traffic unit: can the unit start to be forwarded before it is completely received ("wormholes") [15], and can parts of two units be intermixed on the same link ("interleaving")? As we shall see, wormholes give better performance when the time to send a unit is not small, and it is often not because a unit is often an entire packet. Furthermore, wormholes mean that a connector 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 on links. With no wormholes this term is the sum of the times the unit spends on each link (the size of the unit divided by the bandwidth of the link). With wormholes and interleaving, it is the time on the slowest link. With wormholes but without interleaving, if there are alternating slow and fast links $s_1\ f_1\ s_2\ f_2\ \ldots\ s_n\ f_n$ on the path (with f_n perhaps null), it is the total time on slow links minus the total time on fast links. A sequence of links with increasing times is equivalent to the slowest, and a sequence with decreasing times to the fastest. 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. 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 [22], with packets that take milliseconds to transmit, uses wormholes, as do multiprocessors like the J-machine [4] which have short messages but care about every microsecond of latency and every byte of network buffering. The same considerations apply to pipelines.

8.2 Routing

If we replace the connectors with switch nodes, we can assemble a mesh like the one at the bottom of Figure 5. The mesh can implement the bigger switch that surrounds it and is connected to it by dashed lines. The path from node 3 to node 4 is shown by the heavy lines in both the mesh and the switch. The pattern of links between switches is called the "topology" of the mesh.

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 implemented with a second map, from input addresses to output addresses.

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

— *Source* routing: the address is just the sequence of output links to use; each switch strips off the one it uses. The IBM token ring and several multiprocessors [4, 23] 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.

— *Hierarchical* routing: 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 and cascaded I/O busses use this.

— *Flat* routing: the address is flat, and each switch knows what links to use for every address. Broadcast networks like Ethernet and FDDI use this; the implementation 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 map is inadequate. The mechanism for routing 800 numbers is mainly flat.

8.3 Deadlock

Traffic traversing a composite link needs a sequence of resources (most often buffer space) to reach the end, and usually it acquires a resource while holding on to existing ones. This means that deadlock is possible. The left side of Figure 6 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,

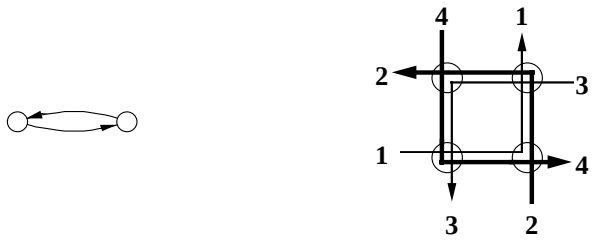


Fig. 6. Deadlock.

it is possible for all the messages to deadlock waiting for each other to release their buffers.

The simple rule for avoiding deadlock is well known: 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.

The right side of Figure 6 shows what can happen even on 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 a 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 [15]. 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 [22].

9 Layers

There are three rules for writing a novel.
Unfortunately, no one knows what they are.

Somerset Maugham

In the last section we studied systems composed by plugging together components with the same interface such as Internet routers, LAN bridges, and telephone switches and multiplexers. Here we look at systems in which the interface changes. When we implement an interface on top of a different one we call the implementation a "layer".

The simplest kind of layering is "encapsulation", in which we stick converters on the ends of a link that implements one interface to get a link that implements a different one; see Figure 7. Examples are transporting Internet packets over an 802 LAN, or over DECnet, or the reverse encapsulations of 802 or DECnet packets over the Internet. Another way to think of this is as multiplexing several protocols over a single link. As usual, multiplexing needs an address field in each message, so it is prudent and cus-

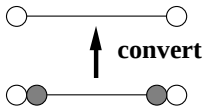


Fig. 7. Encapsulation.

tomary to provide a “protocol type” field in every link interface. A “version number” field plays a similar role on a smaller scale.

Here is encapsulation in the large. We can build

	What	*Why*
a)	a TCP reliable transport link	function: reliable stream
b)	on an Internet packet link	function: routing
c)	on the PPP header compression protocol	performance: space
d)	on the HDLC data link protocol	function: packet framing
e)	on a 14.4 Kbit/sec modem line	function: byte stream
f)	on an analog voice-grade telephone line	compatibility
g)	on a 64 Kbit/sec digital line multiplexed	function: bit stream
h)	on a T1 line multiplexed	performance: aggregation
i)	on a T3 line multiplexed	performance: aggregation
j)	on an OC-48 fiber.	performance: aggregation

This stack is ten layers deep. Each one serves some purpose, tabulated in the right column and classified as function, performance, or compatibility. Note that compatibility caused us to degrade a 64 Kbit/sec stream to a 14.4 Kbit/sec stream in layers (f) and (g) at considerable cost; the great achievement of ISDN is to get rid of those layers.

On top of TCP we can add four more layers, some of which don’t look so much like encapsulation:

	What	*Why*
w)	mail folders	function: organization
x)	on a mail spooler	function: storage
y)	on SMTP mail transport	function: routing
z)	on FTP file transport	function: reliable char arrays

Now we have 14 layers with two kinds of routing, two kinds of reliable transport, three kinds of stream, and three kinds of aggregation. Each serves some purpose that isn’t served by other, similar layers. Of course many other structures could underlie the filing of mail messages in folders.

Here is an entirely different example, an implementation of a machine’s load instruction:

	What	*Why*
a)	load from cache	function: data access
b)	miss to second level cache	performance: space
c)	miss to RAM	performance: space
d)	page fault to disk	performance: space

Layer (d) could be replaced by a page fault to other machines on a LAN that are sharing the memory [12] (function: sharing), or layer (c) by access to a distributed cache over a multiprocessor’s network (function: sharing). Layer (b) could be replaced by

access to a PCI I/O bus (function: device access) which at layer (c) is bridged to an ISA bus (compatibility).

The standard picture for a communication system is the OSI reference model, which shows peer-to-peer communication at each of seven layers: physical, data link, network, transport, session, presentation, and application. The peer-to-peer aspect of this picture is not as useful as you might think, because peer-to-peer communication means that you are writing a concurrent program, something to be avoided if at all possible. At any layer peer-to-peer communication is usually replaced with client-server communication as soon as possible.

It should be clear from these examples that there is nothing magic about any particular arrangement of layers. The same load/store function is provided for the file data type by NFS and other distributed file systems [20], and for an assortment of viewable data types by the World Wide Web. What is underneath is both similar and totally different. Furthermore, it is possible to collapse layers in an implementation as well as add them; this improves efficiency at the expense of compatibility.

We have seen several communication interfaces and many designs for implementing them. The basic principle seems to be that any interface and any design can be useful anywhere, regardless of how lower layers are done. Something gets better each time we pile on another abstraction, but it's hard to predict the pattern beforehand.

10 Fault-tolerance

The simplest strategies for fault-tolerance are

Duplicate components, detect errors, and ignore bad components.

Detect errors and retry.

Checkpoint, detect errors, crash, reconfigure without the bad components, and restart from the checkpoint.

Highly available systems use the first strategy, others use the second and third. The second strategy works very well for communications, since there is no permanent state to restore, retry is just resend, and many errors are transient.

A more complex approach is to fail over to an alternate component and retry; this requires a failover mechanism, which for communications takes the simple form of changes in the routing database. An often overlooked point is that unless the alternate component is only used as a spare, it carries more load after the failure than it did before, and hence the performance of the system will decrease.

In general fault tolerance requires timeouts, since otherwise you wait indefinitely for a response from a faulty component. Timeouts in turn require knowledge of how long things should take. When this knowledge is precise timeouts can be short and failure detection rapid, conditions that are usually met at low levels in a system. It's common to design a snoopy cache, for instance, on the assumption that every processor will respond in the same cycle so that the responses can be combined with an or gate. Higher up there is a need for compatibility with several implementations, and each lower level with caching adds uncertainty to the timing. It becomes more difficult to set timeouts appropriately; often this is the biggest problem in building a fault-tolerant system. Perhaps we should specify the real-time performance of systems more

carefully, and give up the use of caches such as virtual memory that can cause large variations in response time.

All these methods have been used at every level from processor chips to distributed systems. In general, however, below the level of the LAN most systems are synchronous and not very fault-tolerant: any permanent failure causes a crash and restart. Above that level most systems make few assumptions about timing and are designed to keep working in spite of several failures. From this difference in requirements follow many differences in design.

11 Conclusion

We have seen that both interfaces and implementations for interconnecting computing elements are quite uniform at many different scales. The storage and message interfaces work both inside processor chips and in the World Wide Web. Links, converters, and switches can be composed by concatenation, routing, and layering to build communication systems over the same range. Bandwidth, latency, and connectivity are always the important performance parameters, and issues of congestion, flow control, and buffering arise again and again.

This uniformity arises partly because the ideas are powerful ones. The rapid improvement in silicon and fiber optics technology, which double in cost/performance every two years, also plays a major role in making similar designs appropriate across the board. Computer scientists and engineers should be grateful, because a little knowledge will go a long way.

References

1. R. Barton: A new approach to the functional design of a digital computer. *Proc . Western Joint Computer Conference* (1961)
2. A. Birrell and B. Nelson: Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2, 39-59 (1984)
3. D. Culler et al.: Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. *4th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, 164-175 (1991)
4. W. Dally: A universal parallel computer architecture. *New Generation Computing* 11, 227-249 (1993)
5. P. Green: The future of fiber-optic computer networks. *IEEE Computer* 24, 78-87 (1991)
6. J. Hennessy and N. Jouppi: Computer technology and architecture: An evolving interaction. *IEEE Computer* 24, 18-29 (1991)
7. C. Hoare: Communicating sequential processes. *Communications of the ACM* 21, 666-677 (1978)
8. V. Jacobsen: Congestion avoidance and control. *ACM SigComm Conference*, 1988, 314-329
9. L. Lamport: How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* C-28, 241-248 (1979)
10. B. Lampson: Reliable messages and connection establishment. In S. Mullender (ed.) *Distributed Systems*, Addison-Wesley, 1993, 251-282
11. D. Lenkosi et al.: The Stanford Dash multiprocessor. *IEEE Computer* 25, 63-79 (1992)
12. K. Li and P. Hudak: Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* 7, 321-359 (1989)
13. C. Mead and L. Conway: *Introduction to VLSI Systems*. Addison-Wesley, 1980

14. R. Metcalfe and D. Boggs: Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM* 19, 395-404 (1976)
15. L. Ni and P. McKinley: A survey of wormhole routing techniques in direct networks. *IEEE Computer* 26, 62-76 (1993)
16. R. Pike et al.: The use of name spaces in Plan 9. *ACM Operating Systems Review* 27, 72-76 (1993)
17. R. Rettberg et al.: The Monarch parallel processor hardware design. *IEEE Computer* 23, 18-30 (1990)
18. F. Ross: An overview of FDDI: The fiber distributed data interface. *IEEE Journal on Selected Areas in Communication* 7 (1989)
19. J. Saltzer, D. Reed, and D. Clark: End-to-end arguments in system design. *ACM Transactions on Computer Systems* 2, 277-288 (1984)
20. M. Satyanarayanan: Distributed file systems. In S. Mullender (ed.) *Distributed Systems*, Addison-Wesley, 1993, 353-384
21. M. Schroeder and M. Burrows: Performance of Firefly RPC. *ACM Transactions on Computer Systems* 8, 1-17 (1990)
22. 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, 1318-1335 (1991)
23. C. Seitz: The cosmic cube. *Communications of the ACM* 28, 22-33 (1985)