21. Distributed Systems

The rest of the course is about distributed computing systems. In the next four lectures we will characterize distributed systems and study how to specify and code communication among the components of a distributed system. Later lectures consider higher-level system issues: distributed transactions, replication, security, management, and caching.

The lectures on communication are organized bottom-up. Here is the plan:

1. Overview.
2. Links. Broadcast networks.
3. Switching networks.
4. Reliable messages.
5. Remote procedure call and network objects.

Overview

An underlying theme in computer systems as a whole, and especially in distributed systems, is the tradeoff between performance and complexity. Consider the problem of carrying railroad traffic across a mountain range.[1] The minimal system involves a single track through the mountains. This solves the problem, and no smaller system can do so. Furthermore, trains can travel from East to West at the full bandwidth of the track. But there is one major drawback: if it takes 10 hours for a train to traverse the single track, then it takes 10 hours to switch from E-W traffic to W-E traffic, and during this 10 hours the track is idle. The scheme for switching can be quite simple: the last E–W train tells the W-E train that it can go. There is a costly failure mode: the East end forgets that it sent a 'last' E-W train and sends another one; the result is either a collision or a lot of backing up.

The simplest way to solve both problems is to put in a second track. Now traffic can flow at full bandwidth in both directions, and the two-track system is even simpler than the single-track system, since we can dedicate one track to each direction and don't have to keep track of which way traffic is running. However, the second track is quite expensive. If it has to be retrofitted, it may be as expensive as the first one.

A much cheaper solution is to add sidings: short sections of double track, at which trains can pass each other. But now the signaling system must be much more complex to ensure that traffic between sidings flows in only one direction at a time, and that no siding fills up with trains.

[1] This example is due to Mike Schroeder.

What makes a system distributed?

> *One man's constant is another man's variable.*
> Alan Perlis

> *A distributed system is a system where I can't get my work done because a computer has failed that I've never even heard of.*
> Leslie Lamport

There is no universally accepted definition of a distributed system. It's like pornography: you recognize one when you see it. And like everything in computing, it's in the eye of the beholder. In the current primitive state of the art, Lamport's definition has a lot of truth.

Nonetheless, there are some telltale signs that help us to recognize a distributed system:

It has *concurrency*, usually because there are multiple general-purpose computing elements. Distributed systems are closely related to multiprocessors.

Communication costs are an important part of the total cost of solving a problem on the system, and hence you try to minimize them. This is not the same as saying that the cost of communication is an important part of the system cost. In fact, it is more nearly the opposite: a system in which communication is good enough that the programmer doesn't have to worry about it (perhaps because the system builder spent a lot of money on communication) is less like a distributed system. Distributed systems are closely related to telephone systems; indeed, the telephone system is by far the largest example of a distributed system, though its functionality is much simpler than that of most systems in which computers play a more prominent role.

It *tolerates partial failures*. If some parts break, the rest of the system keeps doing useful work. We usually don't think of a system as distributed if every failure causes the entire system to go down.

It is *scaleable*: you can add more components to increase capacity without making any qualitative changes in the system or its clients.

It is *heterogeneous*. This means that you can add components that implement the system's internal interfaces in different ways: different telephone switches, different computers sending and receiving E-mail, different NFS clients and servers, or whatever. It also means that components may be *autonomous*, that is, owned by different organizations and managed according to different policies. It doesn't mean that you can add arbitrary components with arbitrary interfaces, because then what you have is chaos, not a system. Hence the useful reminder: "There's no such thing as a heterogeneous system."

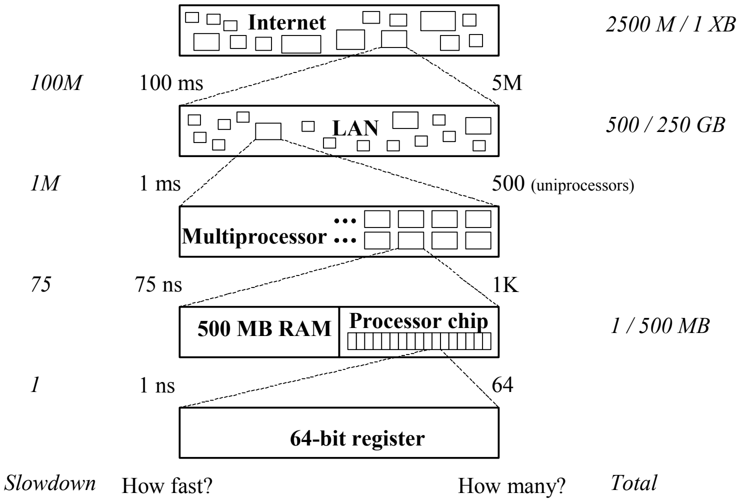


Fig. 1. Scales of interconnection. Relative speed and size are in italics.

Layers

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

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

You can look at a computer system at many different scales. At each scale you see the same basic components: computing, storage, and communications. The bigger system is made up of smaller ones. Figure 1 illustrates this idea over about 10 orders of magnitude (we have seen it before, in the handout on performance).

But Figure 1 is misleading, because it doesn't suggest that different levels of the system may have quite different interfaces. When this happens, we call the level a *layer*. Here is an example of different interfaces that transport bits or messages from a sender to a receiver. Each layer is motivated by different functionality or performance than the one below it. This stack is ten layers deep. Note that in most cases the motivation for separate layers is either compatibility or the fact that a layer has other clients or other code.

	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	function:	3 KHz low-latency signal
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

On top of TCP we can add four more layers, some of which have interfaces that are significantly different from simple transport.

	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, code for 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 (function: sharing)[2], 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 (function: compatibility).

[2] K. Li and P. Hudak: Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* 7, 321-359 (1989)

Another simple example is the layering of the various facsimile standards for transmitting images over the standard telephone voice channel and signaling. Recently, the same image encoding, though not of course the same analog encoding of the bits, has been layered on the internet or e-mail transmission protocols.

Addressing

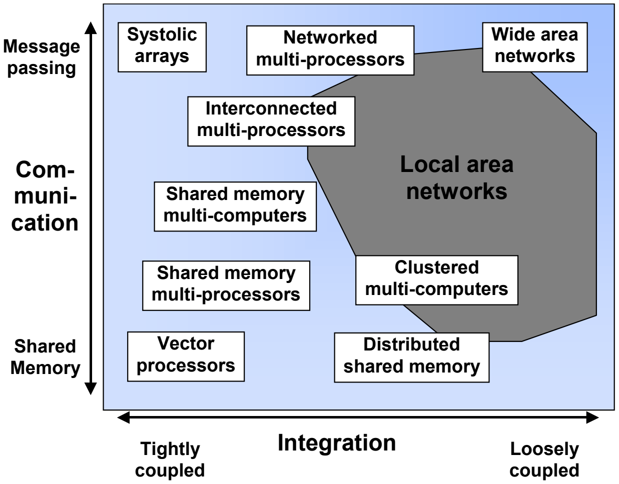
Another way to classify communication systems is in terms of the kind of interface they provide:

messages or storage,

the form of addresses,

the kind of data transported,

other properties of the transport.


Message passing
Com-muni-cation
Shared Memory
Systolic arrays
Networked multi-processors
Wide area networks
Interconnected multi-processors
Local area networks
Shared memory multi-computers
Shared memory multi-processors
Clustered multi-computers
Vector processors
Distributed shared memory
Tightly coupled
Integration
Loosely coupled

Here are a number of examples to bear in mind as we study communication. The first table is for messaging, the second for storage.

System	*Address*	*Sample address*	*Data value*	*Delivery*	
				Ordered	*Reliable*
J-machine[3]	source route	4 north, 2 east	32 bytes	yes	yes
IEEE 802 LAN	6 byte flat	`FF F3 6E 23 A1 92`	packet	no	no
IP	4 byte hierarchical	`16.12.3.134`	packet	no	no
TCP	IP + port	`16.12.3.134 / 3451`	byte stream	yes	yes
RPC	TCP + procedure	`16.12.3.134 / 3451 / Open`	arg. record	yes	yes
E-mail	host name + user	`blampson@microsoft.com`	`String`	no	yes

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

System	*Address*	*Sample address*	*Data value*
Main memory	32-bit flat	`04E72A39`	2^n bytes, $n \leq 4$
File system[4]	path name	`/udir/bwl/Mail/inbox/214`	0-4 Gbytes
World Wide Web	protocol + host name + path name	`http://research.microsoft.com/lampson/default.html`	typed, variable size

Layers in a communication system

The standard picture for a communication system is the OSI reference model, which shows peer-to-peer communication at each of seven layers (given here in the opposite order to the examples above):

physical (volts and photons),
data link,
network,
transport,
session,
presentation, and
application.

This model is often, and somewhat pejoratively, called the 'seven-layer cake'. The peer-to-peer aspect of the OSI model 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 (also known as request-response or remote procedure call) as soon as possible.

The examples we have seen should make it clear that real systems cannot be analyzed so neatly. Still, it is convenient to use the first few layers as tags for important ideas, which we will study in this order:

Data link layer:	framing and multiplexing.
Network layer:	addressing and routing (or switching) of packets.
Transport layer:	reliable messages.
Session layer:	naming and encoding of network objects.

We are not concerned with volts and photons, and the presentation and application layers are very poorly defined. Presentation is supposed to deal with how things look on the screen, but it's unclear, for example, which of the following it includes: the X display protocol, the Macintosh PICT format and the PostScript language for representing graphical objects, or the Microsoft RTF format for editable documents. In any event, all of these topics are beyond the scope of this course.

Figure 2 illustrates the structure of communication and code for a fragment of the Internet.

[4] M. Satyanarayanan: Distributed file systems. In S. Mullender (ed.) *Distributed Systems*, Addison-Wesley, 1993, pp 353-384

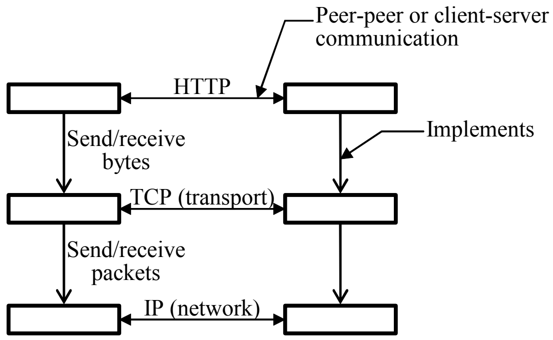


Fig. 2: Protocol stacks for peer-to-peer communication

Principles[5]

There are a few important ideas that show up again and again at the different levels of distributed systems: recursion, addresses, end-to-end reliability, broadcast vs. point-to-point, real time, and fault-tolerance.

Recursion

The 14-layer example of coding E-mail gives many examples of encapsulating a message and transmitting it over a lower-level channel. It also shows that it can be reasonable to code a channel using the same kind of channel several levels lower.

Another name for encapsulation is 'multiplexing'.

Addresses

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 code but flat at the interface, for instance an Internet address or a URL in the World Wide Web. The examples of addressing that we saw earlier should clarify these points; for more examples see handout 12 on naming.

People often make a distinction between names and addresses. What it usually boils down to is that an address is a name that is interpreted at a lower level of abstraction.

[5] My thanks to Alex Shvartsman for some of the figures in this section.

End-to-end reliability

A simple way to obtain reliable communication is to rely on the end points for every aspect of reliability, and to depend on the lower level communication system only to deliver bits with some reasonable probability. The end points check the transmission for correctness, and retry if the check fails.[6]

For example, an end-to-end file transfer system reads the file, sends it, and writes it on the disk in the usual way. Then the sender computes a strong checksum of the file contents and sends that. The receiver reads the file copy from its disk, computes a checksum using the same function, and compares it with the sender's checksum. If they don't agree, the check fails and the transmission must be retried.

In such an end-to-end system, the total cost to send a message is $1 + rp$, where r = cost of retry (if the cost to send a simple message is 1) and p = probability of retry. This is just like fast path (see handout 10 on performance). Note, however, that the retry itself may involve further retries; if $p << 1$ we can ignore this complication. For good performance (near to 1) rp must be small. Since usually $r > 1$, we need a small probability of failure: $p << 1/r < 1$. This means that the link, though it need not have any *guaranteed* properties, must transmit messages without error most of the time. To get this property, it may be necessary to do forward error correction on the link, or to do retry at a lower level where the cost of retry is less.

Note that p applies to the *entire* transmission that is retried. The TCP protocol, for example, retransmits a whole packet if it doesn't get a positive ack. If the packet travels over an ATM network, it is divided into small 'cells', and ATM may discard individual cells when it is overloaded. If it takes 100 cells to carry a packet, $p_{\text{packet}} = 100\, p_{\text{cell}}$. This is a big difference.

Of course r can be measured in different ways. Often the work that is done for a retry is about the same as the work that is done just to send, so if we count r as just the work it is about 1. However, the retry is often invoked by a timeout that may be long compared to the time to send. If latency is important, r should measure the time rather than the work done, and may thus be much greater than 1.

Broadcast vs. point-to-point transmission

It's usually much cheaper to broadcast the same information to n places than to send it individually to each of the n places. This is especially true when the physical communication medium is a broadcast medium. An extreme example is direct digital satellite broadcast, which can send a megabyte to everyone in the US for about $.05; compare this with about $.02 to send a megabyte to one place on a local ISDN telephone link. But even when the physical medium is point to point and switches are needed to connect n places, as is the case with telephony or ATM, it's still much cheaper to broadcast because the switches can be configured in a tree rooted at the source of the broadcast and the message needs to traverse each link only once, instead of once for each node that the link separates from the root. Figure 3 shows the number of times a

[6] J. Saltzer, D. Reed, and D. Clark: End-to-end arguments in system design. *ACM Transactions on Computer Systems* **2**, 277-288 (1984).

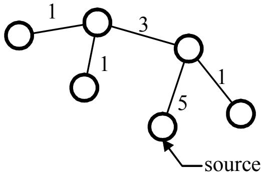


Fig. 3: The cost of doing broadcast with point-to-point communication

message from the root would traverse each link if it were sent individually to each node; in a broadcast it traverses each link just once.

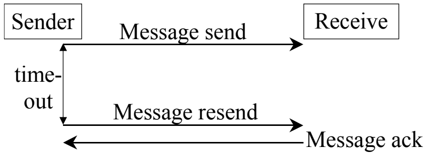
Historically, most LANs have done broadcast automatically, in the sense that every message reaches every node on the LAN, even if the underlying electrons or photons don't have this property; we will study broadcast networks in more detail later on. Switched LANs are increasingly popular, however, because they can dramatically increase the total bandwidth without changing the bandwidth of a single link, and they *don't* do broadcast automatically. Instead, the switches must organize themselves into a spanning tree that can deliver a message originating anywhere to every node.

Broadcast is a special case of 'multicast', where messages go to a subset of the nodes. As nodes enter and leave a multicast group, the shape of the tree that spans all the nodes may change. Note that once the tree is constructed, any node can be the root and send to all the others. There are clever algorithms for constructing and maintaining this tree that are fairly widely implemented in the Internet.[7]

Real time

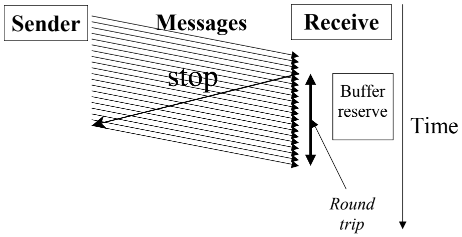
Although often ignored, real time plays an important role in distributed systems. It is used in three ways:

1. To decide when to retry a transmission if there is no response. This often happens when there is some kind of failure, for instance a lost Internet IP packet, as part of an end-to-end protocol. If the retransmission timeout is wrong, performance will suffer but the system will usually still work. When timeouts are used to control congestion, however, making them too short can cause the bandwidth to drop to 0.

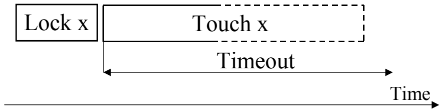


[7] S, Deering et al., An architecture for wide-area multicast routine, *ACM SigComm Computer Communication Review*, **24**, 4 (Oct. 1994), pp 126-135.

2. To ensure the stability of a load control system based on feedback. This requires knowing the round trip time for a control signal to propagate. For instance, if a network provides a 'stop' signal when it can't absorb more data, it should have enough buffering to absorb the additional data that may be sent while the 'stop' signal makes its way back to the sender. If the 'stop' comes from the receiver then the receiver should have enough buffering to cover a sender-receiver-sender round trip. If the assumed round-trip time is too short, data will be lost; if it's too long, bandwidth will suffer.



3. To code "bounded waiting" locks, which can be released by another party after a timeout. Such locks are called 'leases'; they work by requiring the holder of the lock to either fail or release it before anyone else times out.[8]. If the lease timeout is too short the system won't work. This means that all the processes must have clocks that run at roughly the same rate. Furthermore, to make use of a lease to protect some operation such as a read or write, a process needs an upper bound on how the operation can last, so that it can check that it will hold the lease until the end of that time. Leases are used in many real systems, for example, to control ownership of a dual-ported disk between two processors, and to provide coherent file caching in distributed file systems. See handout 18 on consensus for more about leases.



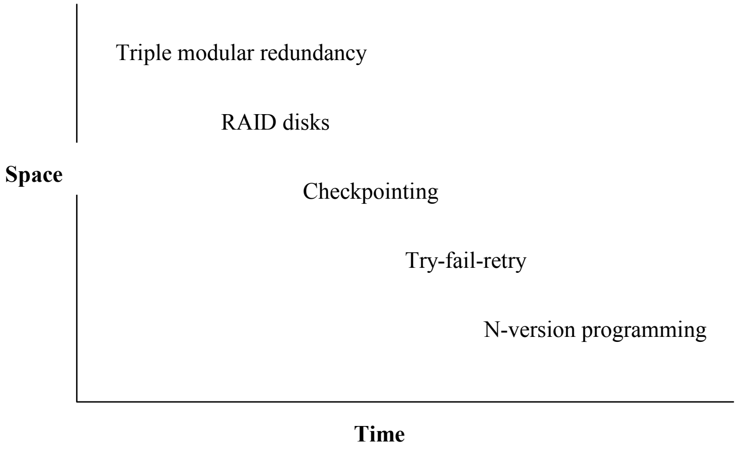
Fault tolerance

Fault tolerance is always based on redundancy. The simplest strategy for fault-tolerance is to get the redundancy by replicating fairly large components or actions. Here are three ways to do it:

1. Duplicate components, detect errors, and ignore bad components (replicate in *space*).
2. Detect errors and retry (replicate in *time*, hoping the error is transient).
3. Checkpoint, detect errors, crash, reconfigure without the bad components, and restart from the checkpoint (a more general way to replicate in time)

There is a space-time tradeoff illustrated in the following picture.

[8] C. Gray and D. Cheriton, Leases: An efficient fault-tolerant mechanism for distributed file cache consistency, *Proc. 12th Symposium on Operating Systems Principles*, Dec. 1989, pp 202-210.



Highly available systems use the first strategy. Others use the second and third, which are cheaper as long as errors are not too frequent, since they substitute duplication in time for duplication in space (or equipment). 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. The third strategy is difficult to program correctly without transactions, which are therefore an essential ingredient for complex fault tolerant systems.

Another way to look at the third approach is as *failover* 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, as we saw in the previous discussion of real time. When this knowledge is precise, we call the system 'synchronous'; 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.[9] 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.

[9] Hennessey and Patterson, section 8.3, pp 654-676.

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.

In a system that cannot be completely reset, it is important to have *self-stabilization*: the system can get from an arbitrary state (which it might land in because of a failure) to a good state.[10]

In any fault-tolerant system the algorithms must be 'wait-free' or 'non-blocking', which means that the failure of one process (or of certain sets of processes, if the system is supposed to tolerate multiple failures) cannot keep the system from making progress.[11] Unfortunately, simple locking is not wait-free. Locking with leases is wait-free, however. We will study some other wait-free algorithms that don't depend on real time. We said a little about this subject in handout 14 on practical concurrency.[12] Note that the Paxos algorithm is wait-free; see handout 18 on consensus.

Performance of communication

Communication has the same basic performance measures as anything else: latency and bandwidth.

- *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 into roughly equal-size parts if they are removed; it is a lower bound on the possible total rate of uniform 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' save bandwidth to a hot spot by combining several operations into one, several loads or increments for example.

[10] G. Varghese and M. Jayaram, The fault span of crash failures, *JACM*, to appear. Available here.

[11] These terms are not actually synonyms. In a wait-free system every process makes progress. In a non-blocking system some process is always making progress, but it's possible for a process to be starved indefinitely.

[12] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* **13**, 1 (Jan. 1991), pp 124-149.

Code shipping saves bandwidth by sending the code to the data.[13]

In addition, there are some other issues that are especially important for communication:

- *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[14] 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.

Specs for communication

Regardless of the type of message being transported, all the communication systems we will study implement one of a few specs. All of them are based on the idea of sending and receiving messages through a channel. The channel has state that is derived from the messages that have been sent. Ideally the state is the sequence of messages that have been sent and not yet delivered, but for weaker specs the state is different. In addition, a message may be acknowledged. This is interesting if the spec allows messages to be lost, because the sender needs to know whether to retransmit. It may also be interesting if the spec does not guarantee prompt delivery and the sender needs to know that the message has been delivered.

None of the specs allows for messages to be corrupted in transit. This is because it's easy to convert a corrupted message into a lost message, by attaching a sufficiently good checksum to each message and discarding any message with an incorrect checksum. It's important to realize that the definition of a 'sufficiently good' checksum depends on a model of what kind of errors can occur. To take an extreme example, if the errors are caused by a malicious adversary, then the checksum must involve some kind of secret, called a 'key'; such a checksum is called a 'message authentication code'. At the opposite extreme, if only single-bit errors are expected, (which is likely to be the case on a fiber optic link where the errors are caused by thermal noise) then a 32-bit CRC may be good; it is cheap to compute and it can detect three or fewer single-bit errors in a message of less than about 10 KB. In the middle is an unkeyed one-way function like MD5.[15]

These specs are for messages between a single sender and a single receiver. We allow for lots of sender-receiver pairs initially, and then suppress this detail in the interests of simplicity.

```
MODULE Channel[
     M,                                                  % Message
     A ] =                                               % Address

TYPE Q           =  SEQ M                                % Queue: channel state
     SR          =  [s: A, r: A]                         % Sender - Receiver
     K           =  ENUM[ok, lost]                       % acK

...

END Channel
```

Perfect channels

A perfect channel is just a FIFO queue. This one is unbounded. Note that `Get` blocks if the queue is empty.

```
VAR  q           := (SR -> Q){* -> {}}                   % all initially empty

APROC Put(sr, m)   = << q(sr) := q(sr) + {m} >>
APROC Get(sr) -> M = << VAR m | m = q(sr).head => q(sr) := q(sr).tail; RET m >>
```

Henceforth we suppress the `sr` argument and deal with only one channel, to reduce clutter in the specs.

Reliable channels

A reliable channel is like a perfect channel, but it can be down, in which case the channel is allowed to lose messages. Now it's interesting to have an acknowledgment. This spec gives the simplest kind of acknowledgment, for the last message transmitted. Note that `GetAck` blocks if `status` is `nil`; normally this is true iff `q` is non-empty. Also note that if the channel is down, `status` can become `lost` even when no message is lost.

```
VAR  q           := {}
     status      :  (K + Null) := ok
     down        := false

APROC Put(m)        = << q := q + {m}, status := nil >>

APROC Get() -> M    = << VAR m | m = q.head =>
     q := q.tail; IF q = {} => status := ok [*] SKIP FI; RET m >>

APROC GetAck() -> K = << VAR k | k = status => status := ok; RET k >>

APROC Crash()       = down := true
APROC Recover()     = down := false
```

[13] Thanks to Dawson Engler for this observation.

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

[15] B. Schneier, *Applied Cryptography*, Wiley, 1994, p 329.

```
THREAD Lose()       = DO                              % internal action
    << down =>
        IF  VAR q1, q2, m | q = q1 + {m} + q2 =>
            q := q1 + q2; IF q2 = {} => status := lost [*] SKIP FI
        [*] status := lost
        FI >>
    [*] SKIP OD
```

Unreliable channels

An unreliable channel is allowed to lose, duplicate, or reorder messages at any time. This is an interesting spec because it makes the minimum assumptions about the channel. Hence anything built on this spec can work on the widest variety of channels. The reason that duplication is important is that the way to recover from lost packets is to retransmit them, and this can lead to duplication unless a lot of care is taken, as we shall see in handout 26 on reliable messages. A variation (not given here) bounds the number of times a message can be duplicated.

```
VAR  q           := Q{}                               % as a multiset!

APROC  Put(m)     = << q := q \/ {m} >>
APROC  Get() -> M = << VAR m | m IN q => q := q - {m}; RET m >>

THREAD Lose()     = DO VAR m | << m IN q => q := q -  {m} >> [*] SKIP OD
THREAD Dup()      = DO VAR m | << m IN q => q := q \/ {m} >> [*] SKIP OD
```

An unreliable FIFO channel is a model of a point-to-point wire or of a broadcast LAN without bridging or switching. It preserves order and does not duplicate, but can lose messages at any time. This channel has `Put` and `Get` exactly like the ones from a perfect channel, and a `Lose` much like the unreliable channel's `Lose`.

```
VAR  q           := Q{}                               % all initially empty

APROC  Put(m)     = << q := q + {m} >>
APROC  Get() -> M = << VAR m | m = q.head => q := q.tail; RET m >>

THREAD Lose()     =
    DO << VAR q1, q2, m | q = q1 + {m} + q2 => q := q1 + q2 >> [*] SKIP OD
```

These specs can also be written in an 'early-decision' style that decides everything about duplication and loss in the `Put`. As usual, the early decision spec is shorter. It takes a prophecy variable (handout 8) to show that the code with `Lose` and `Dup` implements the early decision spec for the unreliable FIFO channel, and for the unordered channel it isn't true, because the early decision spec cannot deliver an unbounded number of copies of `m`. Prophecy variables can work for infinite traces, but there are complicated technical details that are beyond the scope of this course.

Here is the early decision spec for the unreliable channel:

```
VAR  q           := Q{}                               % as a multiset!

APROC  Put(m)     = << VAR i: Nat => q := q \/ {j :IN i.seq | | m} >>
APROC  Get() -> M = << VAR m | m IN q => q := q - {m}; RET m >>
```

and here is the one for the unreliable FIFO channel

```
VAR  q           := Q{}                               % all initially empty

APROC  Put(m)     = << q := q + {m} [] SKIP >>
APROC  Get() -> M = << VAR m | m = q.head => q := q.tail; RET m >>
```

22. Paper: Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links

The attached paper on Autonet by Michael Schroeder et al. appeared as report 59, Systems Research Center, Digital Equipment Corp., April 1990. Essentially the same version was published in *IEEE Journal on Selected Areas in Communications* **9***,* 8, (October 1991), pp1318-1335.

Read it as an adjunct to the lectures on distributed systems, links, and switching. It gives a fairly complete description of a working highly-available switched network providing daily service to about 100 hosts. The techniques used to obtain high reliability and fault-tolerance are characteristic of many distributed systems, not just of networks. The paper also makes clear the essential role of software in modern networks.

A second paper on Autonet describes the reconfiguration scheme in detail: Thomas Rodeheffer and Michael D. Schroeder. Automatic reconfiguration in Autonet. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 183-187, 1991.

Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links

Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, Charles P. Thacker

April 21, 1990

A slightly different version of this paper appeared in the *IEEE Journal on Selected Areas of Communications* **9**, 8, October 1991.

This version was converted from Acrobat PDF and may have errors.

A companion paper is Thomas L. Rodeheffer and Michael D. Schroeder, Automatic Reconfiguration in Autonet, *Proceedings of the 13th ACM Symposium on Operating System Principles*, 1991, pp 183-187.

Abstract

Autonet is a self-configuring local area network composed of switches interconnected by 100 Mbit/second, full-duplex, point-to-point links. The switches contain 12 ports that are internally connected by a full crossbar. Switches use cut-through to achieve a packet forwarding latency as low as 2 microseconds per switch. Any switch port can be cabled to any other switch port or to a host network controller.

A processor in each switch monitors the network's physical configuration. A distributed algorithm running on the switch processors computes the routes packets are to follow and fills in the packet forwarding table in each switch. This algorithm automatically recalculates the forwarding tables to incorporate repaired or new links and switches, and to bypass links and switches that have failed or been removed. Host network controllers have alternate ports to the network and fail over if the active port stops working.

With Autonet, distinct paths through the set of network links can carry packets in parallel. Thus, in a suitable physical configuration, many pairs of hosts can communicate simultaneously at full link bandwidth. The aggregate bandwidth of an Autonet can be increased by adding more links and switches. Each switch can handle up to 2 million packets/second. Coaxial links can span 100 meters and fiber links can span two kilometers.

A 30-switch network with more than 100 hosts is the service network for Digital's Systems Research Center.

1. Introduction

The Ethernet [10], with 10 Mbit/s host-to-host bandwidth and 10 Mbit/s aggregate bandwidth, has done well as the standard local area network (LAN) for high-performance workstations, but it is becoming a bottleneck in demanding applications. One modern workstation can use an Ethernet's entire data transfer capacity, and workstations are getting faster and more numerous. There is an increasing need for a faster, higher-capacity LAN.

This need is being addressed commercially by the FDDI [4, 5] token ring LAN. With ten times greater host-to-host and aggregate bandwidth, FDDI will provide considerable relief for the Ethernet bottleneck. Autonet is an alternative approach to a higher-speed, higher-capacity, general-purpose LAN that could replace Ethernet. The fundamental advantage of Autonet over FDDI is greater aggregate bandwidth from the same link bandwidth. With FDDI the aggregate network bandwidth is limited to the link bandwidth; with Autonet the aggregate bandwidth can be many times the link bandwidth. Other advantages of Autonet over FDDI include lower latency, a more flexible approach to high availability, and a higher operational limit on the number of host that can be attached to a single LAN. Also, Autonet appears to be simpler than FDDI. There is no intrinsic reason why an Autonet should cost more than an FDDI ring.

Any replacement for Ethernet must retain Ethernet's high availability and largely automatic operation, and be capable of efficiently supporting the protocols that work on Ethernet. Low latency is important in a new network because distributed computing makes request/response protocols such as RPC [9] as important as bulk-data transfer protocols. Because security will become increasingly important in the next decade, a new LAN must not hinder encrypted communication. Autonet addresses all these requirements.

The primary goal of the Autonet project was to build an useful local area network, rather than to do research into component technologies for computer networks. Except in a few aspects, Autonet is designed using ideas that have been tried in other systems in different combinations. But bringing together just the right pieces can be a challenge in itself, and can produce a result that advances the state of the art.

Building Autonet required combining expertise in networking, hardware design, computer security, system software, distributed systems, proof of algorithms, performance modeling, and simulation. While a primary purpose for Autonet was to support for distributed computing, Autonet's implementation uses distributed computing to perform its status monitoring and reconfiguration.

The development goal for Autonet was producing a network that would be put into service use. The prospect of service use forced us to develop practical solutions to both the big and the little problems encountered in the design process, and generated a strong preference for simplicity in the design. In early 1990 an Autonet replaced an Ethernet as the service LAN for our building, connecting over 100 computers. Service use is allowing the effectiveness of the design to be evaluated and the design to be improved based on operational experience.

Section 2 of this paper contains a brief description of Autonet, to provide context for the rest of the paper. Section 3 describes the major design decisions that define the network. Section 4 highlights the areas where Autonet appears to break new ground. Section 5 provides a more detailed description of the components of the network. Section 6 describes the operation of these components. Finally, section 7 discusses our early experience with Autonet and indicates directions for future work.

2. Overview

An Autonet, such as the one illustrated in Figure 1, consists of a number of switches and host controllers connected by 100 Mbit/s full-duplex links. As shown by the gray arrows, a packet generated by a source host travels through one or more switches to reach a destination host. Switches contain logic to forward packets from an input port to one or more output ports, as directed by the destination address in each packet's header. A non-blocking crossbar in each switch connects the input and output ports. Depending on the topology, the network can handle many packets at once. Packets even can flow simultaneously in opposite directions on a link.

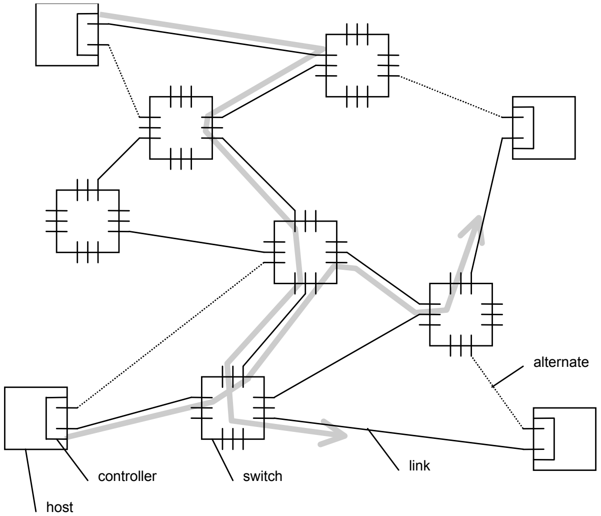


Figure 1: A portion of an Autonet installation

Switches can be interconnected in an arbitrary topology, and this topology will change with time as new switches and links are added to the network, or as switches and links fail. A processor in each switch monitors the state of the network. Whenever the topology changes, all switch processors execute a distributed reconfiguration algorithm. This algorithm determines the new topology and loads the forwarding tables of each switch to route packets using all operational switches and links. In normal operation the switch processor does not participate in the forwarding of packets.

Switches forward packets using a cut-through technique that minimizes switching latency. There is a small amount of buffering associated with each switch input port and a flow control mechanism that ensures these buffers do not overflow. Except during reconfiguration, Autonet never discards packets.

Hosts are connected to the Autonet via dual-ported controllers. For best network availability, a host is connected to two switches; the controller design allows only one of these connections to be used at a time. An Autonet ought to accommodate at least 1000 dual-connected hosts. Possible improvements to the reconfiguration algorithm would allow even larger Autonets.

3. Design decisions

This section summarizes the major decisions of the Autonet design.

3.1 Point-to-point links at 100 Mbit/s

Ethernet uses a broadcast physical medium. Each packet sent on an Ethernet segment is seen by all hosts attached to the segment. As described by Tobagi [20], the minimum size of an Ethernet packet is determined by the need to detect collisions between packets. Reliable collision detection requires that each packet last a minimum time. At high bit rates this time translates into unacceptably large minimum packet sizes. Most 100 Mbit/s and faster networks, including Autonet, use point-to-point links to get away from these limitations. Using point-to-point links also can produce a design that is relatively independent of the specific link technology. As long as a link technology has the needed length, bandwidth, and latency characteristics, then it can be incorporated into the network with appropriate interface electronics.

We settled on 100 Mbit/s for the link bandwidth in Autonet because that speed is much faster than Ethernet, but still well within the limits of standard signaling technology. We chose the AMD TAXI chip set [3] to drive the links, leaving the subtleties of phase-locked loops and data encoding on the link to others. The overall Autonet design should scale to ten times faster links.

We engineered Autonet to tolerate transmission delays sufficient for fiber optic links up to 2 km in length. The first link we have implemented uses 75 ohm coaxial cable, with full-duplex signaling on a single cable. Electrical considerations limit these coax links to a maximum length of 100 m. If both link types were implemented they could be mixed in a single installation: coaxial links might be used within a building because of their lower cost; fiber optic links might be used between buildings because of their longer length limit.

3.2 Unconstrained topology with pre-calculated packet routes

An Autonet is physically built from multi-port switches interconnected by point-to-point links in an arbitrary topology (although the network will work better when thought is given to the topology). Any switch port can be cabled to any other switch port, or to a port on a host controller. A packet is routed from switch to switch to its destination according to pre-calculated forwarding tables that are tailored to the current physical configuration.

A tree-shaped flooding network, like Hubnet [13], has an aggregate network bandwidth that is limited to the link bandwidth and has limited ability to configure around broken components. A ring topology like that used in FDDI has similar limitations. In addition, a ring has latency proportional to the number of hosts. A reasonably configured Autonet has latency proportional to the log of the number of switches. Autonet handles many packets simultaneously along different routes, has unconstrained topology, and allows a great deal of flexibility in establishing routes that avoid broken components.

3.3 Automatic operation

One of the virtues of Ethernet and FDDI is that in normal operation no management is required to route packets. Even when multiple networks are interconnected with bridges [14], a distributed algorithm executed by the bridges determines a forwarding pattern to interconnect all segments without introducing loops. The bridge algorithm also automatically reconfigures the forwarding pattern to include new equipment and to avoid broken segments and bridges.

Autonet also operates automatically. This function is provided by software executing on the control processor in each switch that monitors the physical installation. Whenever a switch or link fails, is repaired, is added, or is removed, this software triggers a distributed reconfiguration algorithm. The algorithm adjusts the packet routes to make use of all operational links and switches and to avoid all broken ones. Of course, human network management is still required to repair broken equipment and adjust the physical installation to reflect substantially changed loads.

3.4 Crossbar switches

An Autonet switch has 12 full-duplex ports that are internally interconnected by a crossbar. We chose a crossbar because its structure is simple and its performance is easy to understand, although a more sophisticated switch fabric could be used if it allowed a single input port to connect simultaneously to any set of output ports to support broadcast.

The small number of ports is a direct result of wanting to get the system into service quickly. All the Autonet hardware is built out of off-the-shelf components, and 12 ports was all that could be fit into a reasonably sized switch without using custom integrated circuits. The Autonet switch design would scale easily to 32 or 64 ports per switch by using higher levels of circuit integration. Such larger switches would be more cost-effective for all but the smallest installations, because fewer ports would be used for switch-to-switch links. A virtue of our small switch is that it generates a higher switch count, which in turn provides a more interesting test for the distributed reconfiguration algorithm.

3.5 Limited buffering with flow control

Autonet uses a FIFO buffer at each receiving switch port. A start/stop flow control scheme signals the transmitter to stop sending more bytes down the link when the receiving FIFO is more than half full. Packets are not discarded by the receiving switch in normal operation. With our flow control scheme a 1024-byte FIFO is sufficient to absorb the round-trip latency of a 2 km fiber optic link, although we actually use a 4096-byte FIFO to obtain deadlock-free routing for broadcast packets. The FIFO is only big enough to contain a few average-sized packets or less than one maximum-sized packet. Flow control is independent of packet boundaries so a single packet can be in several switches at once. A consequence of this scheme is that congestion can back up through the network, potentially delaying even packets that will not be routed over the congested link. Limited buffering also implies that a switch must be able to start forwarding a packet without having the entire packet in the local buffer. In fact, in Autonet such cut-through forwarding can begin after only 25 bytes have arrived.

An alternative buffering scheme would be to provide many packets of buffering at each receiving switch port, say using 1 Mbyte of memory, and to provide no flow control at this level. The port would have a higher capacity to absorb incoming traffic during periods of congestion, delaying the need to respond to the congestion and allowing time for congestion avoidance mechanisms to work. Also, longer links could be used because the absence of flow control eliminates the maximum link latency constraint. Eventually, though, a port would have to defend itself by discarding arriving packets.

We chose limited buffering with flow control because it uses less memory per switch port, making the switches simpler and smaller. In the absence of proven mechanisms for avoiding congestion, an additional advantage of our scheme may be that communication protocols will be more stable because the flow control scheme responds to link overload by backing up packets rather than by throwing them away.

3.6 Deadlock-free, multipath routing

Because Autonet uses flow controlled FIFOs for buffering and does not discard packets in normal operation, deadlock is possible if packets are routed along arbitrary paths. Deadlocks can be dealt with by detecting and breaking them, or by avoiding them. For Autonet we chose the latter approach. Detecting deadlocks reliably and quickly is hard, and discarding an individual packet to break a deadlock complicates the switch hardware. Our scheme uses deadlock-free routes while still allowing packet transmission on all working links. (See section 4.2.) The scheme has the property that it allows multiple paths between a particular source and destination, and takes advantage of links installed as parallel trunks.

3.7 Short addresses

The Autonet reconfiguration algorithm assigns a short address to each switch and host in the network. (A few short addresses are reserved for special purposes like broadcast.) Short addresses contain only enough bits (11 bits in the prototype) to name all switch ports in a maximal-sized Autonet. A forwarding table in each switch, indexed by a packet's destination short address (and incoming port number), allows the switch to quickly pick a suitable link for the next step in a route to the packet's destination. The forwarding table is constructed as part of the distributed configuration algorithm that runs whenever the physical installation changes, breaks, or is repaired. The short address of a switch or host can change when reconfiguration occurs, although it usually does not.

Autonet's addressing scheme lies between source routing, as used in Nectar [6] for example, and addressing by unique identifier (UID), as used in Ethernet. Of the three schemes, UID addressing is the most complex in a network that requires explicit routing, because the network must know a route to each UID-identified destination and do one or more UID-keyed lookups to forward a packet. Source routing removes from the network the responsibility for determining routes, placing it instead with the hosts in smart controllers or in system software. The network must contain mechanisms to report the physical configuration to the hosts and to alter packets as they are forwarded. Source routing eliminates the possibility of dynamic choice of alternative routes. In comparison, Autonet's use of short addresses results in relatively simple switch hardware without giving up dynamic multipath routing.

When considering alternative addressing schemes for LANs we must keep in mind that Ethernet has established UID addressing as the standard interface for datagrams. What the network hardware does not provide, the host software must. So the design question becomes one of splitting the work of providing UID addressing between network switches, host controllers, and host software. For Autonet, all host controllers and switches have 48-bit UIDs; host software implements UID addressing based on Autonet short addresses. (See section 3.11.)

3.8 Hardware-supported broadcast

Because Ethernet naturally supports broadcast, high-level protocols have come to depend upon low-latency broadcast within a LAN. Autonet switch hardware can transmit a packet on multiple output ports simultaneously. This capability is used to implement LAN-wide broadcast with low latency by flooding broadcast packets on a spanning tree of links. Since a broadcast packet must go everywhere in a network, the aggregate broadcast bandwidth is limited to the link bandwidth. As we found out, supporting broadcast complicates the problem of providing deadlock-free routing. (See section 6.6.6.) Having low-latency broadcast, however, simplifies the problem of mapping destination UIDs to short addresses.

3.9 Alternate host ports

In an Autonet, a host is directly connected to an active switch. In an Ethernet-based extended LAN, a host is directly connected to a passive cable. An active switch has a greater tendency to fail than a passive cable. The specific availability goal for Autonet is that no failure of a single network component will disconnect any host. Thus, Autonet allows each host to be connected to two different switches. The mechanism we chose for dual connection is to provide two ports on an Autonet host controller. The host chooses and uses one of the ports, switching to the alternate port after accumulating some evidence that the chosen port is not working.

Having alternate ports simplifies other areas of the design. For example, without alternate ports serious consideration would need to be given to providing "hot swap" for port cards in switches: otherwise, turning off a switch to change or add a port card would disable the network for all directly connected hosts. With alternate ports on host controllers, hot swap is not necessary: turning off a switch simply causes the connected hosts to adopt their alternate ports to the network. Port failover usually can be done without disrupting communication protocols. The obvious disadvantage of having alternate ports is the increased cost of more host-to-switch links and extra switches. For 100 Mbit/s links, however, the cost per link is quite low compared to the cost of the host that typically would be connected to such a network.

3.10 Integrated encryption

Security in most distributed systems must be based on encrypted communication. We wanted encrypted packets to be handled with the same latency and throughput as unencrypted ones -- secure communication is more likely to be used if there is no performance penalty. Therefore we have put a pipelined encryption chip in the host controller. This chip can encrypt and decrypt packets as they are sent or received with no increase in latency over unencrypted packets.

3.11 Generic LAN abstraction

Because of short addresses, Autonet presents a different interface to host software than does Ethernet. When faced with the job of integrating Autonet into our operating system, we quickly decided that this difference should be hidden at a low level in the host software. The interface "LocalNet" makes available to higher-level software multiple generic LANs that carry Ethernet datagrams addressed by UID. Machinery inside LocalNet notices whether an Ethernet or an Autonet is being used. For packets transmitted over Autonet, LocalNet supplies the Autonet packet header complete with destination and source short addresses. LocalNet learns the correspondence between UIDs and short addresses by inspecting arriving packets.

4. Innovations

In a few areas the Autonet design appears to break new ground. We highlight these areas here. Later sections describe these features in more detail.

4.1 Distributed spanning tree algorithm with termination detection

Deadlock-free routing and the flooding pattern for broadcast packets in Autonet are both based on identifying a spanning tree of operational links. The spanning tree is computed using a distributed algorithm similar to Perlman's [16]. That algorithm has the property that all nodes will eventually agree on a unique spanning tree, but no node can ever be sure that the computation has finished. For Autonet, indefinite termination is unacceptable, because an Autonet cannot carry host traffic while reconfiguration is in progress. To do so would invite deadlock caused by inconsistent forwarding tables in the various switches.

To eliminate this problem we extended Perlman's distributed spanning tree algorithm to notify the switch chosen as the root as soon as the tree has been determined. This prompt notice of termination allows the Autonet to open for business quickly after a reconfiguration and guarantees that all switch forwarding tables describe consistent deadlock-free routes.

4.2 Up/down* routing*

Deadlock-free routing in Autonet is based on a loop-free assignment of direction to the operational links. The basis of the assignment is the spanning tree described in the previous section, with "up" for each link being the end that is "closer" to the spanning tree root. The result of this assignment is that the directed links do not form loops. We define a legal route to be one that never uses a link in the "up" direction after it has used one in the "down" direction. This up*/down* routing guarantees the absence of deadlocks while still allowing all links to be used and all hosts to be reached.

4.3 Dynamic learning of short addresses

The LocalNet layer of host software, mentioned above, is given UID-addressed packets to transmit over the network. If a packet is to be delivered over an Autonet then LocalNet must provide the complete Autonet packet header, including the short addresses of the source and destination.

LocalNet uses a UID-addressed cache for recording the short addresses corresponding to various destination UIDs. The information in this UID cache comes from inspecting the source short-address and source UID in each packet that is received. When the specific short address of a destination is not known, a packet is transmitted using the broadcast short address; the destination UID in the packet allows the intended target host to accept the packet and all other hosts to reject it. The next response from the destination allows LocalNet to learn the correct short address. If responses are not forthcoming, LocalNet also can request the short address of another host by using Autonet broadcast to contact the LocalNet implementation at that host. This scheme allows a host to track the short addresses of various destinations without generating many extra packets and without bothering higher layers of software. The learning algorithm requires only 15 extra instructions per packet received.

4.4 Automatic reconfiguration

The Autonet reconfiguration mechanism is based on each switch monitoring the state of its ports. Hardware status indicators report illegal transmission codes, syntax errors, lack of progress, and other conditions for each port. As an end-to-end check, the switch control program verifies a good port by exchanging packets with the neighboring switch. The appearance or disappearance of a responding neighbor on some port will cause a switch to trigger a reconfiguration.

Building a stable, responsive mechanism for detecting faults and repairs has proved to be subtly difficult. The hard problems are determining error fingerprints for each commonly occurring fault, and designing hysteresis into the reconfiguration mechanism so that faults are responded to quickly but intermittent switches or links are ignored for progressively longer periods. Experience with an operational Autonet has allowed us to develop its fault and repair detection mechanisms to achieve both responsiveness and stability.

4.5 First-come, first-considered port scheduler

Packets arriving at an Autonet switch must in turn be forwarded to one or more output ports. (Packets destined for the control processor on the local switch are forwarded to a special internal port.) For packets to a single destination host, the switch determines a set of output ports by lookup in the forwarding table. Any port in the set can be used to send the packet. For broadcast packets the switch determines by lookup in the forwarding table the set of output ports that must forward the packet simultaneously. Scheduling the output ports to fulfill both sorts of requests must be done carefully to prevent starvation of particular input ports, which in turn could lead to performance anomalies including deadlocks.

An Autonet switch includes a strict first-come, first-considered scheduler that polls the availability of output ports and assigns them to the forwarding requests generated by the input ports. This scheduler, implemented in a single Xilinx programmable gate array [21], eliminates the problem of starvation and is a key element in achieving Autonet's best-case switch transit latency of 2 μs (achieved when the router queue is empty and a suitable output port is available).

5. Components

We begin a more detailed description of the Autonet design with an overview of the hardware and software components.

5.1 Switch hardware

Figure 2 presents a block diagram of the Autonet switch. The switching element is a 13 by 13 crossbar constructed from paired 8-to-1 multiplexer chips. Twelve of the crossbar inputs and outputs are connected to link units that can terminate external links. The 13th input and output are connected via a special link unit to the switch's control processor, so it can send and receive packets on the network. The crossbar provides a 9-bit data path from any input to any free output as well as a 1-bit path in the other direction. The former is used to forward packet data and the packet end marker; the latter to communicate a flow control signal. The crossbar also can connect a single input port to an arbitrary set of output ports.

The control processor is a Motorola 68000 [15] running on a 12.5 MHz clock. The processor uses 1 Mbyte of video RAM as both its main memory and its buffers for sending and receiving packets: the processor uses the random access ports to the memory while the crossbar uses the serial access ports. A 64-Kbyte ROM is available for booting the control processor at power-up. The processor has access to a timer that interrupts every 328 µs for calculating timeouts. Because of limited space on the board, however, no CRC or encryption hardware is provided. CRCs for packets to/from the control processor are checked/generated by software. Currently none of the packets sent or received by the control processor are encrypted. The control processor also has access to a ROM containing the switch's 48-bit UID, and to red and green LEDs on the switch front panel.

A link unit implements one switch port. It terminates both channels of a full-duplex coaxial link, receiving from one channel and transmitting to the other. The receive path uses the AMD TAXI receiver to convert from the 100 Mbit/s serial data stream on the link to a 9-bit parallel format. The 9th bit distinguishes the 256 data byte values from 16 command values used for packet framing and flow control. The arriving data bytes (and packet end marks) are buffered in a 4096 by 9 bit FIFO. Logic at the output of the FIFO captures the address bytes from the beginning of an arriving packet and presents them to the switch's router. Once the router has set up the crossbar to forward the packet, the link unit removes the packet bytes from the FIFO and presents them to the crossbar input. The flow control signal from the crossbar enables and disables the forwarding of packet bytes through the crossbar. As soon as a packet end command is removed from the FIFO and forwarded, the output port or ports become available for subsequent packets.

The transmit path in the link unit accepts parallel data from the crossbar and presents it to the AMD TAXI transmitter, which converts it to 100 Mbit/s serial form and sends it down the link. The receive and transmit portions of a single link unit are tied together so that the flow control state derived from the receiving FIFO can be transmitted back over the transmit channel on the same link. (See section 6.2.) A link unit does not include CRC hardware; an Autonet switch does not check or generate CRCs on forwarded packets.

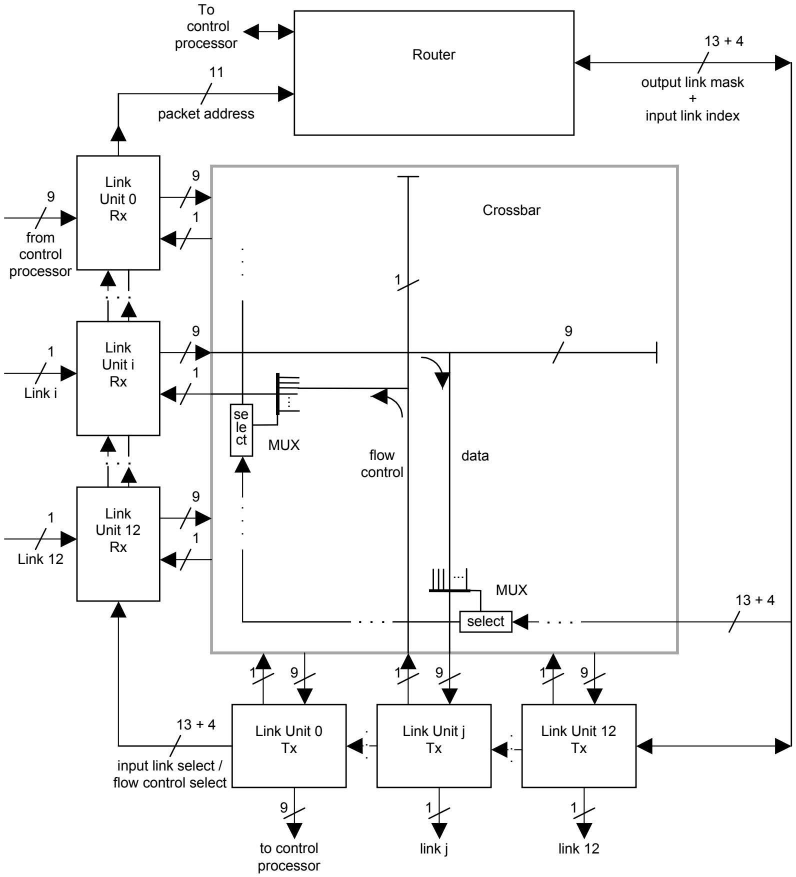


Figure 2: Structure of an Autonet switch

A link unit maintains a set of status bits that can be polled by the control processor. These status bits are a primary source of information for the algorithms that monitor the condition of the ports on a switch to decide when a network reconfiguration should occur. The control processor also has some control over the operation of an individual link unit. Via a control register each link unit can be instructed to illuminate LEDs on its front panel, to send special-purpose flow control directives, and to ignore received flow control.

The router contains 64 Kbytes of memory for the forwarding table and a routing engine that schedules the use of switch output ports. The forwarding tables are loaded by the control processor as part of a network reconfiguration. The routing engine is implemented in a single Xilinx 3090 programmable gate array.

Most of the switch runs on a single 80 ns clock. Link units can forward one byte of packet data into the crossbar on each clock cycle. The router can make a forwarding decision and set up a crossbar connection every 6 clock cycles, so the packet forwarding rate is about 2 million packets per second. The latency from receiving the first bit of a packet on an input link to forwarding the first bit on an output link is 26 to 32 clock cycles if the output link and router are not busy.

The Autonet switch is packaged on 5 card types in a 45 x 18 x 30 cm Eurocard enclosure. A completely populated switch contains 12 link units, 5 2-bit crossbar slices, 1 control processor, and 1 router, all implemented on 10 x 16 cm cards. The backplane, into which all other card types plug at right angles, is a 43 x 13 cm board. A switch draws about 160 w of power.

5.2 Controller hardware

The first host controller for Autonet, shown in Figure 3, attaches to the Digital Equipment Corporation Q-bus [11] that is used in our Firefly [19] multiprocessor computers. In general, we believe that a network controller should be both simple and fast, and play no role in the correct operation of the network fabric. Operating at the full 100 Mbit/s network bandwidth with low latency requires a completely pipelined structure and packet cut-through for transmit and receive. Simplicity requires no higher-level protocol processing in the controller. In the case of this first controller, however, the 14 Mbit/s bandwidth of the Firefly Q-bus allows use of a shared data bus within the controller and elimination of cut-through with little impact on controller latency or throughput.

The network ports are each implemented in a small cabinet kit designed to be mounted in the Firefly chassis. The cabinet kit includes the TAXI transmitter and receiver, and the circuit for driving the link. A signal on the ribbon cable to the controller card selects which cabinet kit is in use. Selection of which port to use is done by the host software.

The controller itself fills a 10.5 x 8.5 inch quad Q-bus card. The receive path is pipelined up to the point where arriving packets are stored in a 128-Kbyte receive buffer. The transmit path is pipelined outward from a 128-Kbyte transmit buffer. CRC checking and generation are done with a Xilinx 3020 [21]. Encryption is handled by an AMD 8068 encryption chip [2]. The connections between the transmit buffer, receive buffer, CRC chip, encryption chip, and Q-bus are via a 16-bit internal bus. The controller board includes a ROM containing a 48-bit UID that can be used as the host’s UID address.

The controller’s operation is under the direction of a microprogram executing on an AMD 29116 microprocessor [1]. The microcode initially comes from a 12-Kbyte boot ROM, but microcode can subsequently be downloaded from the host over the Q-bus. Microcode downloading has allowed us to experiment easily with the controller-to-host interface. This controller is able to use the full Q-bus bandwidth to send and receive packets. Encrypted packets can be sent and received with no performance penalty.

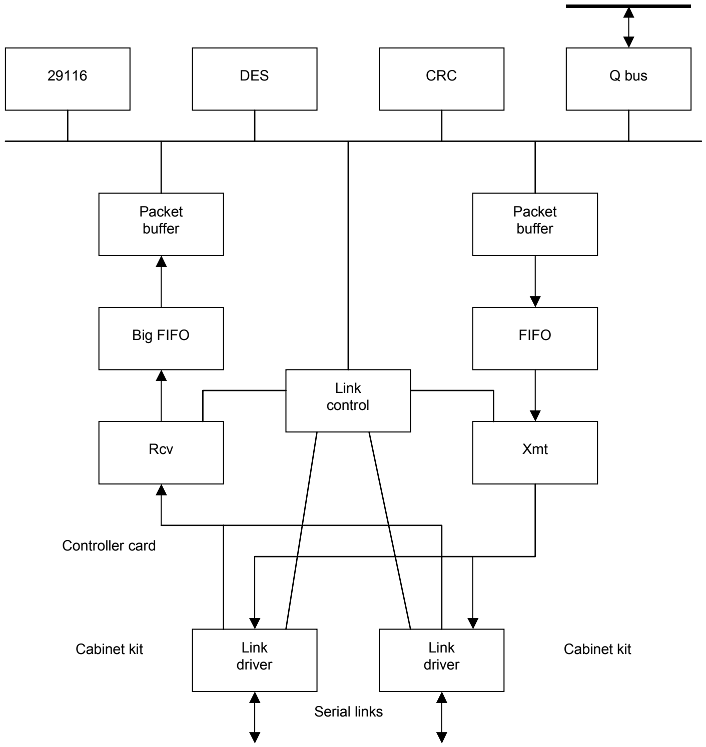


Figure 3: Structure of the Q-bus Autonet controller

5.3 Link hardware

The first links implemented for Autonet use 75 ohm coaxial cable. A hybrid circuit allows both channels of a full-duplex link to be carried on a single cable. This implementation has the consequence that signals transmitted on an Autonet port can be reflected and correctly received at the same port. Reflection occurs when no cable is attached, when an unterminated cable is attached, and when the attached cable terminates at an unpowered remote port. Thus, a host or switch must be prepared to receive its own packets.

The circuit driving the links includes a high-pass filter that prevents frequencies below about 10 MHz from being transmitted. This filter is needed because the data encoding scheme used by the TAXIs allows signals with low frequency components to be generated by sending certain legal sequences of bytes and commands. Without the filter, low frequency transitions can prevent the receiver from recovering the data correctly.

The service network in our building uses Belden 82108 low-loss cable and standard cable television “F” connectors. We accept cabinet kits and link unit cards for service if a packet-echoing protocol can send and receive 40,000 packets of 1,500 bytes each over a 100-meter link between the test host and test switch without a CRC error.

5.4 Switch control program

Autopilot, the software that executes on the control processor of each switch, is responsible for implementing Autonet’s automatic operation. Its major functions are propagating and rebooting new versions of itself, responding to monitoring and debugging packets, monitoring the physical network, answering short-address request packets from attached hosts, triggering reconfigurations when the physical network changes, and executing the distributed reconfiguration algorithm.

The Autopilot source code consists of about 20,000 lines written in C and 3500 lines written in assembler. This generates a 62,000-byte object program. A stable version of Autopilot is included in the switch boot ROMs and is automatically loaded when power is turned on or the switch is reset. Whenever a new version is ready for use, it is down loaded from the programming environment (a Firefly workstation) over the Autonet itself into the nearest switch. The version of Autopilot running there accepts the new version, boots it, and then propagates it to neighboring switches.

The structure of Autopilot is typical of small, real-time, control programs. Interrupt routines enqueue and dequeue buffers for packets sent and received by the control processor. Everything else runs at process level as tasks under the control of a non-preemptive scheduler. Tasks are structured as procedure calls that run to completion within a few milliseconds. The task scheduler manages a timer queue for tasks that need to be run after a timeout has expired. Current timeout resolution is 1.2 milliseconds. The major algorithms in Autopilot are described in later sections.

5.5 The SRC service LAN

The service Autonet for SRC contains 30 switches. The current topology uses four of the twelve ports on each switch for links to other switches and eight ports for links to hosts. With each host connected to two switches, this configuration has the capacity to attach 120 hosts. The Autonet is connected to the Ethernet in the building via a bridge. Thus the Autonet and Ethernet behave as a single extended LAN.

The hosts on Autonet are Firefly workstations and servers. A Firefly contains 4 CVax processors providing about 3 MIPS each and can have up to 128 Mbytes of memory. Typical workstations have 32 or 64 Mbytes of memory. All processors see the same memory via consistent caches. At least until the Autonet proves itself to be stable and reliable, and the more disruptive experiments stop, most Fireflies are connected to both the Autonet and the Ethernet. The choice of which network to use can be changed while the system is running. Switching from one network to the other can be done in the middle of an RPC call or an IP connection without disrupting higher-level software.

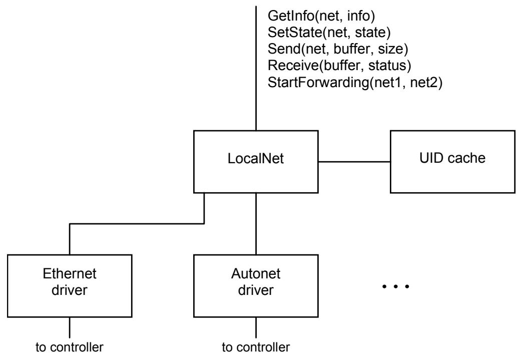


Figure 4: Structure of low-level LAN software for the Firefly

5.6 Host software

The Firefly host software for Autonet includes a driver for the controller, the `LocalNet` generic LAN with UID cache, and the Autonet-to-Ethernet bridging software. This software is written in Modula 2+ [18] and executes in VAX kernel mode. The Firefly scheduler provides multiple threads [7, 8] per address space (including the kernel), and the Autonet host software is written as concurrent programs that execute simultaneously on multiple processors.

Figure 4 illustrates the structure of the low-level LAN software for the Firefly. The `LocalNet` interface presents a set of generic, UID-addressed LANs that carry Ethernet datagrams. The `GetInfo` procedure allows clients to discover which generic nets correspond to physical networks. The `SetState` procedure allows clients to enable and disable these networks. An Ethernet datagram can be sent via a specific network with the `Send` procedure. The `Receive` procedure blocks the calling thread until a packet arrives from some network. The result of `Receive` indicates on which network the packet arrived. Usually many threads are blocked in `Receive`. Finally, the `StartForwarding` procedure causes the host to begin acting as a bridge between two networks.

For transmission on Autonet, the `LocalNet` UID cache provides the short address of a packet’s destination. This cache is kept up-to-date by observing the source UID and source short-address of all packets that arrive on the Autonet, and by occasionally requesting a short address from another `LocalNet` implementation using Autonet broadcast. (See section 6.8.1.) When a host is acting as an Autonet-to-Ethernet bridge, `LocalNet` observes the packets arriving on Ethernet as well, using the UID cache to record which hosts are reachable via the Ethernet. Thus, by looking up the destination UID of each packet that arrives on either network, `LocalNet` can determine whether the packet needs to be forwarded on the other network. (See section 6.8.2.)

6. Functions and algorithms

We now consider in more detail the major functions and algorithms of Autonet.

6.1 *Link syntax*

The TAXI transmitter and receiver are able to communicate 16 command values that are distinct from the 256 data byte values. We use these commands to communicate flow control directives and packet framing. When a TAXI transmitter has no other data or command values to send, it automatically sends a `sync` command to maintain synchronization between the transmitter and receiver. Thus, one can think of the serial channel between a TAXI transmitter and receiver as carrying a continuous sequence of slots that can either be filled with data bytes or commands, or be empty.

In Autonet, flow control prevents a sender from overflowing the FIFO in the receiving switch. Autonet communicates flow control information by time multiplexing the slots on a channel. Every 256th slot is a flow control slot. The remaining slots are data slots. Normally `start` or `stop` directives occupy each flow control slot, independent of what is being communicated in the data slots. To make it easy for a switch to tell whether a link comes from another switch or from a host, host controllers send a `host` directive instead of `start`. Because flow control directives are assigned unique command values, they can be recognized even when they appear unexpectedly in a data slot. Thus, the flow control system is self-synchronizing. Flow control is discussed in more detail in the next section.

Two special-purpose flow control directives, `idhy` and `panic`, may also be sent. `Idhy`, which stands for "I don't hear you", is sent on a switch-to-switch link when one switch determines that the link is defective, to make sure the other switch declares the link to be defective as well. `Panic` is intended to be sent to force the other switch to reset its link unit, clearing the receive FIFO and reinitializing the link control hardware so reconfiguration packets can get through. We have not yet implemented the panic facilities.

The data slots carry packets. A packet is framed with the commands `begin` and `end`. Data slots within packets are filled with `sync` commands when flow control stops packet data from being transmitted. Transmitters are required to keep up with the demand for data bytes, so neither controllers nor switches may send `sync` commands within packets when flow is allowed. Thus, a link is never wasted by idling unnecessarily within a packet, and a link unit can assume that in normal operation packet bytes are available to retrieve from the FIFO. Between packets all data slots are filled with `sync` commands.

6.2 *Flow control*

Figure 5 illustrates the Autonet flow control mechanism. The figure contains pieces of two switches and a link between them. The names "channel 1" and "channel 2" refer to the two unidirectional channels on the link. In the receiving link unit of channel 1, a status signal from the FIFO chip indicates whether the FIFO is more or less than half full. This information determines the flow control directives being sent on channel 2, the reverse channel of the same link. When a flow control slot occurs, a `start` command is sent if the receiving FIFO is less than half full; `stop` is sent if it is more than half full. Back at the receiving link unit of channel 2, the flow control directives generate a flow control signal for the crossbar. If the output port is forwarding a packet, then the flow control signal uses the 1-bit reverse path through the crossbar to open and close the throttle on the FIFO that is the source of the packet.

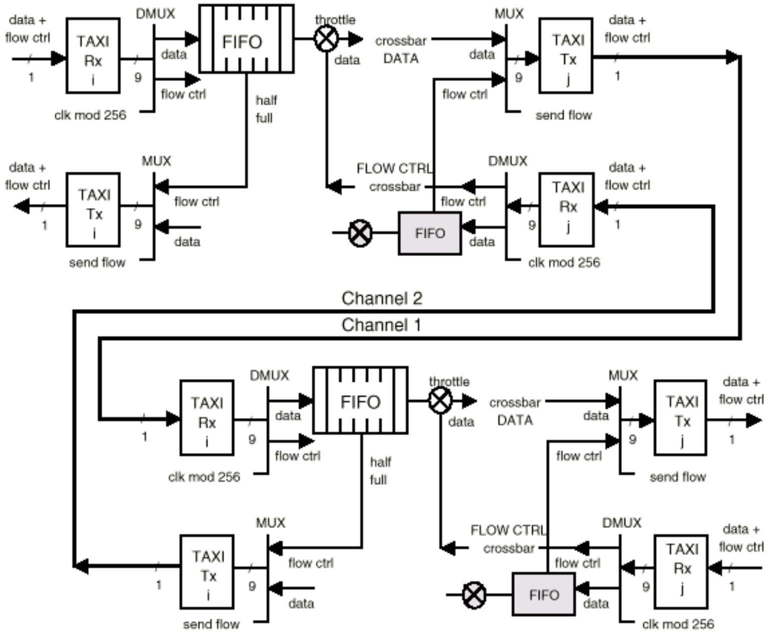


Figure 5: Switch-to-switch Flow Control Mechanism

An important special case is a port that is receiving no flow control commands. Because the host controller transmits only `sync` commands on its alternate link, receiving no flow control usually means that the other end of the link is connected to an alternate host port. Receiving no flow control commands should cause a link control unit to act as though `host` (or `start` if that directive has been received more recently than `host`) is being received, thus allowing packets to be forwarded on such a link, effectively discarding them. Due to an oversight in the design, however, link units that are receiving no flow control keep acting on the last flow control directive received. The last directive could have been `stop`; it is unpredictable following switch power up. Switch software detects and clears the backups that can result from such indefinite cessation of flow.

This flow control scheme can cause congestion to back up across several links. Consider a sequence of switches ABCD along the path of some packet. If the receiving FIFO in C issues `stop`, say because the CD link is not available at the moment, then the FIFO in B will stop emptying. Packet bytes arriving from A will start accumulating in B's FIFO and eventually B will have to issue `stop` to A. Thus congestion can back up through the network until the source controller is issued a `stop`. If the congestion persists long enough, then the network software on the host would stop sending packets; threads making calls to transmit packets would delay returning until more packets could be sent.

Autonet host controllers may not send `stop` commands. Thus, a slow or overloaded host cannot cause congestion to back up into the network. A slow host should have enough buffering in its controller to cover the bursts of packets that will be generated by the communication protocols being used. A controller will discard received packets when its buffers fill up.

We can now understand the relationship between FIFO length, the frequency of flow control slots, and link latency. Assume that the FIFO holds N bytes and that it issues `stop` whenever the FIFO contains more than $(1 - f)\,N$ bytes, where $0 < f \le 1$. A flow control command is sent every S slots. Assume that the link latency is W slot transmission times. In the worst case the receiving FIFO is not being emptied and the transmitter sends bytes continuously unless stopped. At the time the receiver causes a `stop` command to be sent, its FIFO may contain as many as $(1 - f)\,N + (S - 1)$ bytes. Another $2\,W$ bytes will arrive at the FIFO before the `stop` is effective, assuming the transmitter acts on the received `stop` with no delay. To prevent the FIFO from overflowing then, it must be that:

$$N \ \ge\ (1 - f)\,N + (S - 1) + 2\,W$$

From the speed of light, the velocity factor of fiber optic cable (which is a bit slower than coaxial cable), and a slot transmission time of 80 ns we can compute that $W = 64.1\,L$, where L is the cable length in kilometers. Thus:

$$N \ \ge\ (S - 1 + 128.2\,L)\,/\,f$$

For $S = 256$ slots, $f = 0.5$, and $L = 2$ km, we see that N must be 1024 bytes.

With these choices of S, f, and L, Autonet actually uses 4096-byte FIFOs. The larger FIFO is used to solve a deadlock problem that is associated with broadcast packets, as explained in section 6.6.6. The solution to the problem is to have a transmitter of a broadcast packet ignore `stop` commands until the end of the broadcast packet is reached, and make the receiver FIFO big enough to hold any complete broadcast packet whose transmission began under a `start` command. Thus, for broadcast packets flow control acts only between packets. For this case, we can calculate the maximum allowable broadcast packet length as the FIFO size minus the worst case count of bytes already in the FIFO when the first byte of the broadcast packet arrives. Thus:

$$B \ \le\ N - (1 - f)\,N - (S - 1) - 128.2\,L$$

So, taking B into account, the size needed for the FIFO becomes:

$$N \ \ge\ (B + S - 1 + 128.2\,L)\,/\,f$$

The minimum acceptable value for B is about 1550 bytes. This size allows Autonet to broadcast the maximum-sized Ethernet packet with an Autonet header prepended. The corresponding N is about 4096 bytes. This increase in FIFO size is one of the costs of supporting low-latency broadcast in Autonet.

6.3 Address interpretation

As indicated earlier, Autonet packets contain short addresses. In our implementation a short address is 11 bits, although increasing it to 16 bits would be a straightforward design change. The short address is contained in the first two bytes of a packet.

As shown in Figure 6, address interpretation starts as soon as the two address bytes have arrived at the head of the FIFO in a link unit. The short address is concatenated with the receiving port number and the result used to index the switch's forwarding table. Each 2-byte forwarding table entry contains a 13-bit port vector and a 1-bit *broadcast* flag. The bits of the port vector correspond to the switch's ports, with port 0 being the port to the control processor. When the *broadcast* flag is 0, the port vector indicates the set of alternative ports that could forward the packet. The switch will choose the first port that is free from this set. If several of the ports are free then the switch chooses the one with the lowest number. When the *broadcast* flag is 1, the port vector indicates the set of ports that must forward the packet simultaneously. Forwarding will not begin until all these ports are available. A broadcast entry with all 0's for the port vector tells the switch to discard the packet.

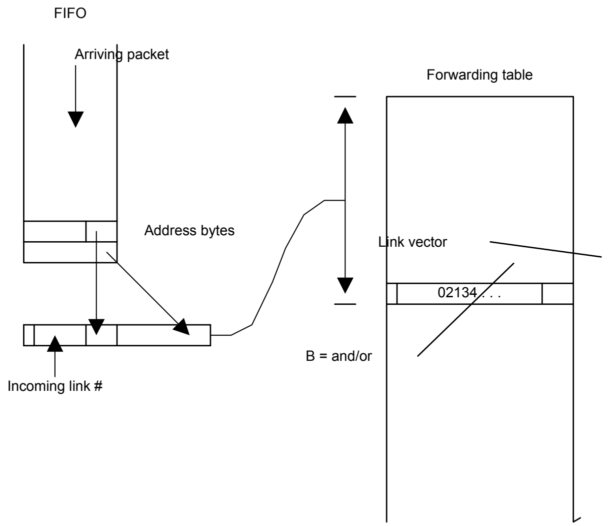


Figure 6: Interpretation of switch forwarding table

Because address interpretation in a switch requires just a lookup in an indexed table, it can be done quickly by simple hardware. Specification of alternative ports allows a simple form of dynamic multipath routing to a destination. For example, multiple links that interconnect a pair of switches can function as a trunk group. Including the receiving port number in the forwarding table index has several benefits; it provides a way to differentiate the two phases of flooding a broadcast packet (see section 6.6.6); it allows one-hop switch-to-switch packets to be addressed with the outbound port number; it provides a way to prevent packets with corrupted short addresses from taking routes that would generate deadlocks.

The mechanism for interpreting short addresses allows considerable latitude in the way short addresses are used. We have adopted the following assignments:

Short address	*Packet destination*
0000	From a host; the control processor of the switch attached to the active host port
0001 - 000F	From a switch; the switch or host attached to the addressed switch port
0010 - FFEF	Particular host or switch (packet discarded if address not in use)
FFF0 - FFFB	Packet discarded (reserved address values)
FFFC	From a host; loopback from switch attached to the active host port
FFFD	Every switch and every host
FFFE	Every switch
FFFF	Every host

Here each short address is expressed as 4 hexadecimal digits, but prototype switches interpret only the low order 11 bits of these values.

As part of the distributed reconfiguration algorithm performed by the switches, each useable port of each working switch in a physical installation is assigned one of the short addresses in the range "0010" through "FFEF". The assignment is made by partitioning a short address into a switch number and a port number, and assigning the switch numbers as part of reconfiguration. The forwarding tables are filled in to direct a packet (from any source) containing one of these destination short addresses to the switch control processor or host attached to the identified port. If the address is not in use, then the forwarding tables will at some point cause the packet to be discarded. The forwarding tables also discard packets that arrive at a switch port that is not on any legal route to the addressed destination; such misrouted packets may occur if bits in the destination short address are corrupted during transmission.

A host on the Autonet discovers its own short address by sending a packet to address "0000". This address directs the packet to the control processor of the local switch. The processor is told the port on which the packet arrived and knows its own switch number. Thus it can reply with a packet containing the host's short address.

The forwarding tables in every switch will reflect a packet addressed to "FFFC" back down the reverse channel of the link on which it was received. Thus, packets sent by a host to this address will be looped back to that host. This feature is used by a host to test its links to the network.

A packet addressed to "FFFF" from a host or switch will be delivered to all host ports in the network. (Section 6.6.6 describes the flooding pattern used.) The addresses "FFFD" and "FFFE" work in a similar way.

Finally, the addresses "0001" through "000F" are reserved for one-hop packets between switches. Each switch forwarding table directs a packet so addressed to be transmitted on the numbered local port if the packet is from port 0 (the control processor port); it directs transmission to port 0 if the packet is from any other port.

6.4 Scheduling switch ports

Once the appropriate entry has been read from a switch's forwarding table, the next step in delivering a packet is scheduling a suitable transmission port. Scheduling needs to be done in a way that avoids long-term starvation of a particular request. The availability of the Xilinx programmable gate array allowed this problem to be solved by the simple strategy of implementing a strict first-come, first-considered scheduler.

Figure 7 illustrates the scheduling engine that contains a queue of forwarding requests. The queue slots are the columns in the figure. Only 13 slots are required because with head-of-line blocking, each port can request scheduling for at most one packet at a time; only the packet at the head of the FIFO is considered. Each queue slot can remember the result of a forwarding table lookup along with the number of the receive port that is requesting service.

When a request arrives at the scheduling engine, the request shifts to the right-most queue slot that is free. Periodically a vector representing the free transmit ports enters the scheduling engine from the right. This vector is matched with occupied queue slots proceeding from right to left, in the arrival order of the requests. Each forwarding request in turn has the opportunity to capture useful free ports.

If a request is for alternative ports (*broadcast* = 0), then it will capture any free transmit port that matches with the requested port vector. If multiple matches occur, then the free port with the lowest number port is chosen. For alternative ports, a single match allows the satisfied request to be removed from the queue and newer requests to be moved to the right. The satisfied request is output from the scheduling engine and is used to set up the crossbar, allowing packet transmission to begin.

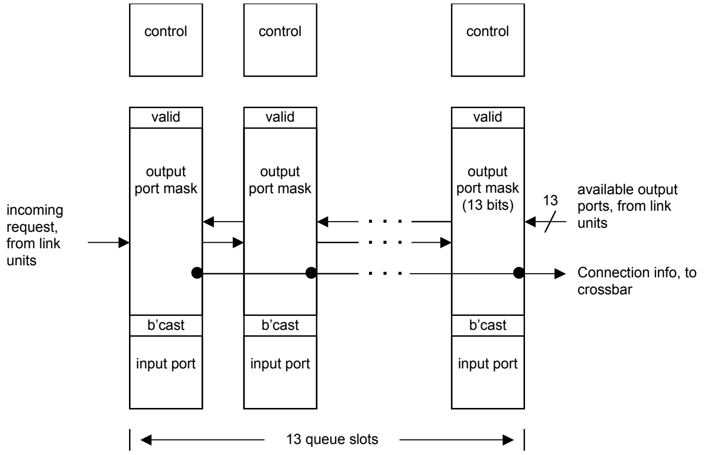


Figure 7: Scheduling engine for switch output ports

If a request is for simultaneous ports (*broadcast* = 1), then it will accumulate all free transmit ports that match the requested port vector. In the case that some requested ports still remain unmatched the vector of free ports proceeds on to newer requests, minus the ports previously captured. If the matches complete the needed transmit port set, then the satisfied broadcast request is removed from the queue, as above. The crossbar is set up to forward from the receive port to all requested transmit ports, and packet transmission is started.

The scheduling engine can accept and schedule one request every 480 ns and thus is able to process up to 2 million requests per second.

Notice that the scheduling engine allows requests to be serviced out-of-order when useful free ports are not suitable for older requests. Queue jumping allows some requests to be scheduled faster than they would be with a first-come, first-served discipline. Also notice that a broadcast request will effectively get higher and higher priority until it is at the head of the queue. Once there, the request has first choice on free transmit ports; each time a needed port becomes free, the broadcast request reserves it. Thus, the broadcast request is guaranteed to be scheduled eventually, independent of the requests being presented by the other receive ports.

6.5 Port state monitoring

Our goal of automatic operation requires that the network itself keep track of the set of links and switches that are plugged together and working, and determine how to route packets using the available equipment. Further, the network should notice when the set of links and switches changes, and adjust the routing accordingly. Changes might mean that equipment has been added or removed by the maintenance staff. Most often changes will mean that some link or switch has failed.

Autopilot, the switch control program, monitors the physical condition of the network. The Autopilot instance on each switch keeps watch on the state of each external port. By periodically inspecting status indicators in the hardware, and by exchanging packets with neighboring switches, Autopilot classifies the health and use of each port. When it detects certain changes in the state of a port, it triggers the distributed reconfiguration algorithm to compute new forwarding tables for all switches.

The mechanism for monitoring port states has several layers. The lowest layer is hardware in each link unit that reports hardware status to the control processor of the switch. The next layer is a status sampler implemented in software that evaluates the hardware status of all ports. The third layer is a connectivity monitor, also implemented in software, that uses packet exchange to determine the health and identity of neighboring switches. Stabilizing hysteresis is provided by two skeptic algorithms. We now explain these mechanisms in more detail.

6.5.1 Port states

The port state monitoring mechanism dynamically classifies each port on an Autonet switch into one of following six states:

Port state	*Definition*
s.dead	The port does not work well enough to use.
s.checking	The port is being monitored to determine if it is attached to a host or to a switch.
s.host	The port is attached to a host.
s.switch.who	The port is being probed to determine the identity of the attached switch.
s.switch.loop	The port is attached to another port on the same switch, or is reflecting signals.
s.switch.good	The port is attached to a responsive neighbor switch.

Figure 8 illustrates these port states and shows the actions associated with the state transitions. As will be explained in more detail in the next two sections, the state transitions shown as black arrows are the responsibility of the status sampler; those shown as gray arrows are the responsibility of the connectivity monitor. The actions triggered by a transition are indicated by the attached action descriptions.

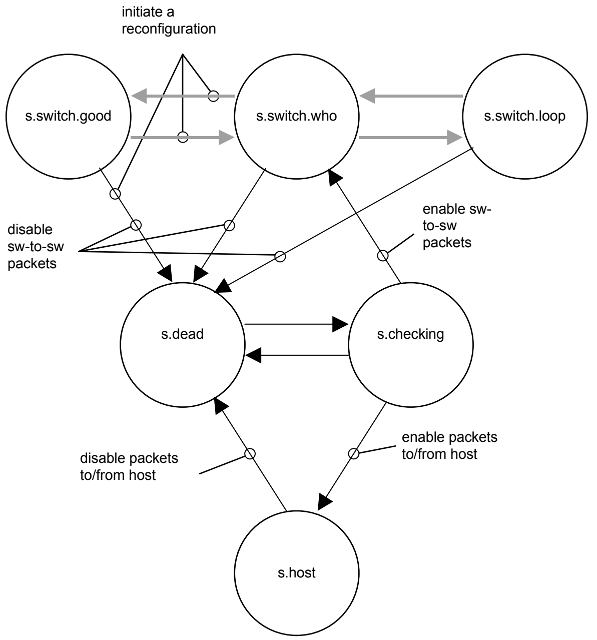


Figure 8: Switch port states and transitions

6.5.2 Hardware port status indicators

Each link unit reports status bits that help Autopilot note changes in the state of the port. These status bits can be read by the control processor of the switch. Some status bits indicate the current condition of a port:

Status bit	*Current port condition represented*
IsHost	last flow control received on link indicates a host is attached
XmitOK	last flow control received on link allows transmission
InPacket	transmitter is in the middle of a packet

Other status bits indicate that one or more occurrences of a condition have occurred since the bit was last read by the control processor:

Status bit	*Accumulated port condition represented*
BadCode	TAXI receiver reported violation
BadSyntax	out-of-place flow control directive, unused command value received, improper packet framing
Overflow	FIFO overflow occurred
Underflow	FIFO underflow occurred inside a packet
IdhySeen	`idhy` flow control directive received
PanicSeen	`panic` flow control directive received
ProgressSeen	FIFO forwarded some bytes or has seen no packets
StartSeen	`start` or `host` flow control directive received

There is considerable design latitude in choosing exactly which conditions to report in hardware status bits. As we will see below, all switch-to-switch links are verified periodically by packet exchange. The hardware status bits provide a more prompt hint that something might have changed. If most changes of interest reflect themselves in the hardware status bits, however, then port status changes will be noticed more quickly; Autopilot can use the hardware status change to trigger an immediate verification by packet exchange.

6.5.3 Status sampler

The next layer of port state monitoring is the status sampler. This code, which runs continuously, periodically reads the link unit status bits. A counter corresponding to each status bit from each port is incremented for each sampling interval in which the bit was found to be set. The status sampler also counts CRC errors on packets received by the local control processor (such as the connectivity test or reply packets described in the next section), even though CRC errors are actually detected by software. Based on the status counts accumulated over certain periods, each port is dynamically classified into one of the four states *s.dead*, *s.checking*, *s.host*, and *s.switch.who*.

When a switch boots, all ports are initially classified as *s.dead*. This state represents ports that are to be evaluated, but not used. While classified as *s.dead*, a switch port is forced to send `idhy` in place of normal flow control to guarantee that the remote port will be classified by the neighboring switch as no better than *s.checking*. Receiving `idhy` is not counted as an error when a port is classified as *s.dead*. When a port has exhibited no bad status for the appropriate period, it moves from *s.dead* to *s.checking*. The length of the error-free period required is determined by the status skeptic described in section 6.5.5. A port is directed to send normal flow control when it enters *s.checking*. A port that has no bad status counts except for receiving `idhy` stays classified as *s.checking*.

Once a port is in *s.checking*, the status sampler waits for `idhy` flow control to cease, and then tries to determine whether the port is cabled to a switch or to a host. The *IsHost* bit is used to distinguish the cases. Reflecting ports, and ports cabled to another port on the same switch, will be classified as *s.switch.who*, because such ports receive the `start` flow control directives sent from the local switch, causing *IsHost* to be FALSE. Alternate host ports will send continuous `sync` commands, but no flow control directives. This pattern generates *BadSyntax* and makes the *IsHost* bit useless, so a port showing constant *BadSyntax* status, but no other errors, is classified as *s.host*.

When a port's state is changed to *s.host*, the local forwarding table is updated to permit communication over the port. The port's entries in the forwarding table are set to forward all suitably addressed packets to the port and to allow packets received from the port to be forwarded to any destination in the network. Because both active and alternate host ports are classified as *s.host*, switching to the alternate by a host will cause no forwarding table changes, assuming that the alternate port does not then start producing bad status counts.

When a port is changed from *s.checking* to *s.switch.who,* the forwarding table is set to allow the control processor to exchange one-hop packets with the possible neighboring switch. This forwarding table change allows the connectivity monitor to probe the neighboring switch in order to distinguish between the states *s.switch.who*, *s.switch.loop*, and *s.switch.good*.

A port moves back to *s.dead* from other states if certain limits are exceeded on the bad status counts accumulated over a time period. As indicated in Figure 8, transitions back to *s.dead* will cause the local forwarding table to be changed to stop packet communication through the port.

A side effect of status sampler operation is the removal of long-term blockages to packet flow. By reading the *StartSeen* bit, the status sampler counts intervals during which only `stop` flow control directives are received at each port. When such intervals occur too frequently, the port is classified as *s.dead*. The associated changes to the forwarding table cause all packets addressed to the port to be discarded, preventing the port from causing congestion to back up into the network. The *ProgressSeen* status bit allows the status sampler to count intervals during which a packet has been available in a FIFO to be forwarded, but made no progress. From this count the status sampler can classify a port as *s.dead* and remove it from service when it is stuck due to local hardware failure.

6.5.4 Connectivity monitor

A transition from *s.checking* to *s.switch.who* means that the status sampler approves the port for switch-to-switch communication. A port thus approved is always being scrutinized by the top layer of port state monitoring, the connectivity monitor. The state *s.switch.who* means that Autopilot does not know the identity of the connected switch.

The connectivity monitor tries to determine the UID and remote port number for the connected switch. The connectivity monitor periodically transmits a connectivity test packet on the port and watches for a proper reply. As long as no proper reply is received, the port remains classified as *s.switch.who*. Thus, a non-responsive remote switch will cause the port to remain in this state indefinitely. To be accepted, a reply must match the sequence information in the test packet and echo the UID and port number of the test packet originator. The connectivity monitor looks at the source UID of an accepted reply packet to distinguish a looped or reflecting link from a link to a different switch. In the former case, the connectivity monitor relegates the port to *s.switch.loop*; such ports are of no use in the active configuration. In the latter case, the connectivity monitor sets the state to *s.switch.good* and initiates a reconfiguration of the entire network. The reconfiguration causes all switches to compute new forwarding tables that take into account the existence of the new switch-to-switch link (and possibly a new switch).

The connectivity monitor continuously probes all ports in the three *s.switch* states. At any time it may cause the transitions to and from *s.switch.who* shown by gray arrows in Figure 8. In the case of a transition from *s.switch.good* to *s.switch.who*, a network-wide reconfiguration is initiated to remove the link from the active configuration. Note from Figure 8 also that a network-wide reconfiguration is initiated when the status sampler, described in the previous section, removes its approval of a port in *s.switch.good* by reclassifying it as *s.dead*.

6.5.5 The skeptics

Two algorithms in Autopilot prevent links that exhibit intermittent errors from causing reconfigurations too frequently. They are the status skeptic and the connectivity skeptic.

The status skeptic controls the length of the error-free holding period required before a port can change from *s.dead* to *s.checking*. The length of the holding period for a particular port depends on the recent history of transitions to *s.dead*: transitions to *s.dead* lengthen the holding period; intervals in *s.host* or any of the *s.switch* states shorten the next holding period.

The connectivity skeptic operates in a similar manner to increase the period over which good connectivity responses must be received before a port is changed from *s.switch.who* to *s.switch.good*. This skeptic therefore limits the rate at which an unstable neighboring switch can trigger reconfigurations. The sequences of delays introduced by the skeptic algorithms are still being adjusted.

6.6 Reconfiguration and routing

We are now ready to describe how Autopilot calculates the packet routes for a particular physical configuration and how it fills in the forwarding tables in a consistent manner. The goals for routing are to make sure all hosts and switches can be reached, to make sure no deadlocks can occur, to use all correctly operating links, and to obtain good throughput for the entire network. The distributed reconfiguration algorithm achieves these goals by developing a set of loop-free routes based on link directions that are determined from a spanning tree of the network.

Reconfiguration involves all operational network switches in a five step process:

1. Each switch reloads its forwarding table to forward only one-hop, switch-to-switch packets and exchanges tree-position packets with its neighbors to determine its position in a spanning tree of the topology.

2. A description of the available physical topology and the spanning tree accumulates while propagating up the tree to the root switch.
3. The root assigns short addresses to all hosts and switches.
4. The complete topology, spanning tree, and assignments of short addresses are sent down the spanning tree to all switches.
5. Each switch computes and loads its own forwarding table, based on the information received in step 4, and starts accepting host-to-host traffic.

Because host packets will be discarded during the reconfiguration process, it is important that the entire process occur quickly, certainly in less that a second. Note that the reconfiguration process will configure physically separated partitions as disconnected operational networks.

As described in the previous section, reconfiguration starts at one or more switches that have noticed relevant port state changes. In step 1 these initiating switches clear their forwarding tables and send the first tree-position packets to their neighbors. Other switches join the reconfiguration process when they receive tree-position packets and they, in turn, send such packets to their neighbors. In this way the reconfiguration algorithm starts running on all connected switches.

The reloading of the forwarding tables in step 1 has two purposes. First, it eliminates possible interference from host traffic, allowing the reconfiguration to occur more quickly. Second, it guarantees that no old forwarding tables will still exist when the new tables are put into service at step 6: coexistence could lead to deadlock and packets being routed in loops.

6.6.1 Spanning tree formation

The distributed algorithm used to build the spanning tree is based on one described by Perlman [16]. Each node maintains its current tree position as four local variables: the root UID, the tree level at this switch (0 is the root), the parent UID, and the port number to the parent. Initially, each switch assumes it is the root. A switch reports this initial tree position and each new position to each neighboring switch by sending tree-position packets, retransmitting them periodically until an acknowledgment is received.

Upon reception of a tree-position packet from a neighbor over some port, a switch decides if it would achieve a better tree position by adopting that port as its parent link. The port is a better parent link if it leads to a root with a smaller UID than the current position, if it leads to a root with the same UID as the current position but via a shorter tree path, if it leads to the same root via the same length path but through a parent with a smaller UID, or if it leads to the current parent but via a lower port number.

If each switch sends tree-position packets to all neighbors each time it adopts a new position, then eventually all switches will learn their final position in the same spanning tree. Unfortunately, no switch will ever be certain that the tree formation process has completed, so the switches will not be able to decide when to move on to step 2 of the reconfiguration algorithm. To eliminate this problem we extend Perlman’s algorithm. We say that a switch S is *stable* if all neighbors have acknowledged S’s current position and all neighbors that claim S as their parent say they are stable. While transitions from unstable to stable and back can occur many times at most switches, a transition from unstable to stable will occur exactly once at the switch which is the root of the spanning tree. Thus, when some switch becomes stable while believing itself to be the root of the spanning tree, then the spanning tree algorithm has terminated and all switches are stable.

Conceptually, implementing stability just requires augmenting the acknowledgment to a tree-position packet with a "this is now my parent link" bit. A neighbor acknowledges with this bit set TRUE when it determines that its tree position would improve by becoming a child of the sender of the tree-position packet. Thus a switch will know which neighbors have decided to become children, and can wait for each of them to send a subsequent "I am stable" message. When all children are stable then a switch in turn sends an "I am stable" message to its parent.

Step 2 of the reconfiguration process has the topology and spanning tree description accumulate while propagating up the spanning tree to the root switch. This accumulation is implemented by expanding the "I am stable" messages into topology reports that include the topology and spanning tree of the stable subtree. As stability moves up the forming spanning tree towards the root, the topology and spanning tree description grows. When the switch thinking itself to be the root receives reports from all its children, then it is certain that spanning tree construction has terminated, and it will know the complete topology and spanning tree for the network. A non-root switch will know that spanning tree formation has terminated when it receives the complete topology report that is handed down the new tree from the root in step 4. Each switch can then calculate and load its local forwarding table from complete knowledge of the current physical topology of the network. The upward and downward topology reports are all sent reliably with acknowledgments and periodic retransmissions.

6.6.2 Epochs

To prevent multiple, unsynchronized changes of port state from confusing the reconfiguration process, Autopilot tags all reconfiguration messages with an *epoch number*. Each switch contains the local epoch number as a 64-bit integer variable, which is initialized to zero when the switch is powered on. When a switch initiates a reconfiguration, it increments its local epoch number and includes the new value in all packets associated with the reconfiguration. Other switches will join the reconfiguration process for any epoch that is greater than the current local epoch, and reset the local epoch number variable to match.

Once a particular epoch starts at each switch, then any change in the set of useable switch-to-switch links visible from that switch (that is, port state changes in or out of *s.switch.good)* will cause Autopilot to add one to its local epoch and initiate another reconfiguration. Such changes can be caused by the status sampler and the connectivity monitor, which continue to operate during a reconfiguration. Thus, the reconfiguration algorithm always operates on a fixed set of switch-to-switch links during a particular epoch.

If a switch sees a higher epoch number in a reconfiguration packet while still involved in an earlier reconfiguration, it forgets the tree position and other state of the earlier epoch and joins the new one. If changes in port state stop occurring for long enough, then the highest numbered epoch eventually will be adopted by all switches, and the reconfiguration process for that epoch will complete. Completion is guaranteed eventually because the status and connectivity skeptics reject ports for increasingly long periods.

6.6.3 Assigning short addresses

Short addresses are derived from switch numbers that are assigned during the reconfiguration process. Each switch remembers the number it had during the previous epoch, and proposes it to the root in the topology report that moves up the tree. A switch that has just been powered-on proposes number 1. The root will assign the proposed number to each switch unless there is a conflicting request. In resolving conflicts the root satisfies the switch with the smallest UID and then assigns unrequested low numbers to the losers.

A short address is formed by concatenating a switch number and a port number. (The port number occupies the least significant bits.) For a host, then, the short address is determined by the switch port where it attaches to the network. A host's alternate link thus has a distinct short address. For a switch's control processor, the port number 0 is used. Because switches propose to reuse their switch numbers from the previous epochs, short addresses tend to remain the same from one epoch to the next.

6.6.4 Computing packet routes

To complete step 5 of the reconfiguration process, each switch must fill in its local forwarding table based on the topology and spanning tree information that is received from the root. Autonet computes the packet routes based on a direction imposed by the spanning tree on each link. In particular, the "up" end of each link is defined as:

1. The end whose switch is closer to the root in the spanning tree.
2. The end whose switch has the lower UID, if both ends are at switches with the same tree level.

The "up" end of a host-to-switch link is the switch end. Links looped back to the same switch are omitted from a configuration. The result of this assignment is that the directed links do not form loops.

To eliminate deadlocks while still allowing all links to be used, we introduce the up*/down* rule: a legal route must traverse zero or more links in the "up" direction followed by zero or more links in the down direction. Put in the negative, a packet may never traverse a link in the "up" direction after having traversed one in the "down" direction.

Because of the ordering imposed by the spanning tree, packets following the up*/down* rule can never deadlock, for no deadlock-producing loops are possible. Because the spanning tree includes all switches, and a legal route is up the tree to the root and then down the tree to any desired switch, each switch and host can send a packet to every switch or host via a legal route. Because the up*/down* rule excludes only looped-back links, all useful links of the physical configuration can carry packets.

While it is possible to fill in the forwarding tables to allow all legal routes, it is not necessary. The current version of Autopilot allows only the legal routes with the minimum hop count. Allowing longer than minimum length routes, however, may be quite reasonable, because the latency added at each switch is so small. When multiple routes lead from a source to a destination, then the forwarding table entries for the destination short address in switches at branch points of the routes show alternative forwarding ports. The choice of which branch to take for a particular packet depends on which links are free when the packet arrives at that switch. Use of multiple routes allows out-of-order packet arrivals.

Note that the up*/down* rule can be enforced locally at each switch. Recall that Autonet forwarding tables are indexed by the incoming port number concatenated with the short address of the packet destination. If this short address were corrupted during transmission, then it might cause the next switch to forward the packet in violation of the up*/down* rule. To prevent this possibility, the forwarding table entries at a switch that correspond to forwarding from a "down" link to an "up" link are set to discard packets.

6.6.5 Performance of reconfiguration

With the first implementation of Autopilot, reconfiguration took about 5 seconds in our 30-switch service network. The 30 switches are arranged as an approximate 4 x 8 torus, with a maximum switch-to-switch distance of 6 links. The reconfiguration time is measured from the moment when the first tree-position packet of the new epoch is sent until the last switch has loaded its new forwarding table. This initial implementation was coded to be easy to understand and debug. As confidence in its correctness has grown, we have begun to improve the performance. The current version reconfigures in about 0.5 seconds. We believe we can achieve a reconfiguration time of under 0.2 seconds for this network.[1] We do not yet understand fully how reconfiguration times vary with network size and topology, but it should be a function of the maximum switch-to-switch distance.

6.6.6 Broadcast routing and broadcast deadlock

A packet with a broadcast short address is forwarded up the spanning tree to the root switch and then flooded down the spanning tree to all destinations. This is a case where the incoming port number is a necessary component of the forwarding table index. Here, the incoming port differentiates the up phase from the down phase of broadcast routing. With the Autonet flow control scheme described earlier, however, broadcast packets can generate deadlocks.

Figure 9 illustrates the problem. Here we see part of a network including five switches V, W, X, Y, Z, and three hosts A, B, and C. The solid links are in the spanning tree and the arrow heads indicate the "up" end of each link. Host B is sending a packet to host C via the legal route BWYZC. This packet is stopped at switch Z by the unavailability of the link ZC. It is a long packet, however, and parts of it still reside in switches Y and W. As a result, the link WY is not available. At the same time, a broadcast packet from host A is being flooded down the spanning tree. It has reached switch V and is being forwarded simultaneously on links VW and VX, the two spanning tree links from V. The broadcast packet flows unimpeded through X and Z, and is starting to arrive at host C, where its arrival is blocking the delivery of the packet from B to C. At switch W the broadcast packet needs to be forwarded simultaneously on links WB and WY. Because WY is occupied, however, the broadcast packet is stopped at W, where it starts to fill the FIFO of the input port. As long as the FIFO continues to accept bytes of the packet, it can continue to flow out of switch V down both spanning tree links. But when the FIFO gets half full, flow control from W will tell V to stop sending. As a result, sending also will stop down the VXZC path. At this point we have a deadlock.

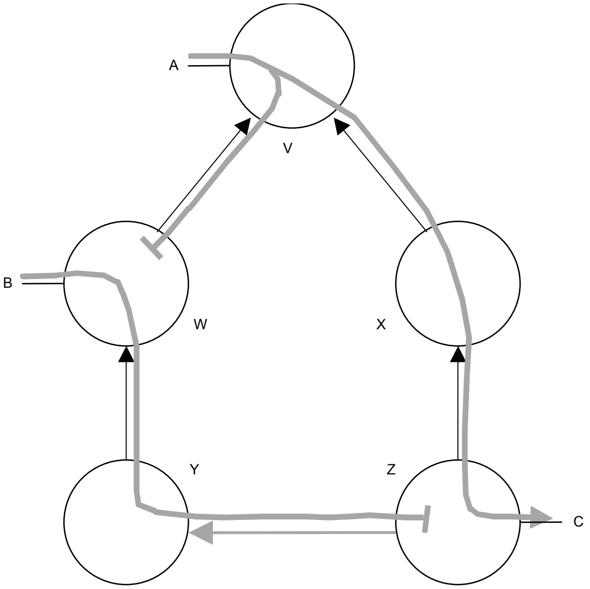


Figure 9: Broadcast deadlock

The solution to this broadcast deadlock problem was discussed in section 6.2. The transmitter of a broadcast packet ignores `stop` flow control commands until the end of the broadcast packet is reached, and the receiver FIFO is made big enough to hold any complete broadcast packet whose transmission began under a `start` command. In our example, switch V will ignore the `stop` from W and complete sending the broadcast packet. Thus, the broadcast packet will finish arriving at C and link ZC will become free to break the deadlock.

6.7 Debugging and monitoring

The main tool underlying Autonet's debugging and monitoring facilities is a source-routed protocol (SRP) that allows a host attached to Autonet to send packets to and receive packets from any switch. The source route is a sequence of outbound switch port numbers that constitute a switch-by-switch path from packet source to packet destination. The source route is embedded in the data part of the SRP packet. At each stage along this path the packet is received, interpreted, and forwarded by the switch control processor. Each forwarding step is done using the destination short address that delivers the packet to the control processor of the switch next in the source route. Delivery of SRP packets depends only on the constant part of a switch's forwarding table that permits one-hop communication with neighbor switches. Thus, SRP packets are likely

[1] Later work has yielded a 170 ms reconfiguration time.

to get through even when routing for other packets is inoperative. In particular, the SRP packets continue to work during reconfiguration.

Based on SRP, we are developing a set of tools for debugging and monitoring Autonet. For example, Autopilot keeps in memory a circular log of events associated with reconfiguration. The log entries are timestamped with local clock values. An SRP protocol allows an Autonet host to retrieve this log. By normalizing the timestamps and merging the logs for all switches, a complete history of a reconfiguration can be displayed. The merged log is a powerful tool for discovering functional and performance anomalies. Another protocol layered on SRP allows most switch state variables to be retrieved, including the forwarding table. A protocol to recover the physical network topology and the current spanning tree has also been built.

Tracking down a difficult bug usually requires adding statements to Autopilot to enter extra entries in the log, downloading this new version of Autopilot, waiting for all switches to boot the new version, triggering the problem, retrieving all the logs, and inspecting them. This debugging method is just a more cumbersome version of adding print statements to a program!

6.8 A generic LAN

The `LocalNet` generic LAN interface in the host software hides most differences between Autonet and Ethernet from client software. To simplify implementing `LocalNet`, we have defined client Autonet packets to consist of a 32-byte Autonet header followed by an encapsulated Ethernet packet. Two differences, however, are not hidden from the clients. First, Autonet packets may contain more data than Ethernet packets. Second, Autonet packets may be encrypted. When either of these differences are exploited, `LocalNet` clients must be aware that an Autonet is being used.

The format of an Autonet packet is:

Bytes	*Field use*
2	Destination short address
2	Source short address
2	Autonet type (type = 1 is shown)
26	Encryption information
6	Destination UID
6	Source UID
2	Ethernet type
0 - 64K	Data (1500-byte limit for broadcast & Ethernet bridging)
8	CRC

The destination short address field is the only part of the packet examined by the switches as the packet traverses the network. It contains the short address of the host (or switch control processor) to which this packet is directed, or some special-purpose address such as the broadcast address. The source short address is used by the receiving host (or switch) to learn the short address of the packet sender. The type field identifies the format of the packet. The format described here is the one used for encapsulated Ethernet packets. Reconfiguration, SRP, and special switch diagnostic protocols use different Autonet type values.

A large fraction of the header consists of encryption information. The encryption header, whose details we omit here, is used by the receiving controller to decide whether to decrypt this packet, which part of the packet to decrypt, which key to use, and where in memory to place the packet after decryption. The encryption facilities are based on Herbison’s master key encryption scheme [12]. A complete description awaits experience in using these facilities to provide secure communication.

The destination UID, source UID, and Ethernet type fields form the header of an Ethernet packet that has been encapsulated within an Autonet packet. The data field may be up to 64K bytes in length for normal Autonet packets; broadcast packets and packets to be bridged to an Ethernet are constrained to the 1500-byte Ethernet limit. The CRC field is generated and checked by the controller.

Occasionally hosts will misaddress packets by placing the wrong short address in the header. This might happen when, for example, a short address changes after a network reconfiguration. The receiving host is responsible for checking the destination UID in the packet and discarding misaddressed packets. The receiving host also does filtering on multi-cast UIDs. These function are done by the Autonet driver software for the Firefly, but they could be done by the controller if it were deemed necessary to avoid overloading a host.

6.8.1 Learning short addresses

In order to hide the differences in addressing between the Autonet and the Ethernet, `LocalNet` maintains a cache of mappings from 48-bit Ethernet UIDs to short addresses. The Autonet driver updates the UID cache by observing the correspondence between the source short address and source UID fields of arriving packets, and, if necessary, by sending Address Resolution Protocol (ARP) requests [17]. An ARP reply sent on Autonet will contain the correct source short address in the Autonet header. When transmitting a packet to an Autonet, `LocalNet` obtains the destination short address using a cache lookup keyed with the destination UID.

When an Autonet host first boots, it knows only two short addresses: address “FFFF”, which reaches all hosts on the Autonet, and address “0000”, which reaches the local switch. The host contacts the local switch to obtain its own short address, which it then inserts in the source short-address field of all packets that it transmits. Thereafter, the host uses the following algorithm for transmitting and receiving packets:

> **Receiving**: The source short address is entered in the cache entry for the source UID, and a timestamp is updated in the cache entry. If the packet was sent to the broadcast short address, but was addressed to the UID of the receiving host (rather than to the broadcast UID), then the sending host no longer knows the receiver’s short address and an ARP response is immediately sent to the sending host in order to update its cache entry.
>
> **Transmitting**: The cache entry for the destination UID is found, and the short address in the entry is copied into the packet before it is transmitted. If necessary, a new cache entry is created giving the short address for this UID as “FFFF”, the broadcast short address. If the cache entry was updated within the two seconds prior to its use, or if it is updated in the two seconds following its use, no further action is taken. Otherwise, an ARP request is sent to the

> short address given in the cache entry. If no response is received within two seconds, the short address in the cache entry is set to the broadcast short address, which action is equivalent to removing the entry from the cache. If a packet to be transmitted is larger than the maximal broadcast packet, and the short address of the destination is unknown, the packet is discarded and an ARP request is sent in its place.

This algorithm does not attempt to maintain cache entries that are not being used by the host, so no ARP packets are sent unless a host has recently failed to respond to some other packet. Moreover, ARP packets are usually directed to the last known address of the destination, rather than being broadcast. Packets are sent to the broadcast short address only when the real short address of the destination is unknown. This is typically the case for the first packet sent between a pair of hosts, and for the packets sent to a host that has recently crashed, or changed its short address. Fortunately, higher-level protocols seldom transmit large numbers of packets to hosts that do not respond, so the total number of packets sent to the broadcast short address is quite small. It might be necessary to review this algorithm if higher-level protocols that do not behave in this way were to become commonplace.

This algorithm generates few additional packets, but can take several seconds to update a cache after a short address has changed. In order to minimize the delays seen by higher-level protocols, hosts broadcast an ARP response packet when their short address changes, so other hosts can update their caches immediately. Short addresses change quite infrequently, so this does not lead to a large number of broadcasts. If the number of broadcasts of this type were to become excessive, an alternative approach is to send packets to hosts whose short address cache entries have recently been updated. This has the effect of updating the caches of hosts that were recently using the changed short address.

The current techniques for managing short addresses are good enough that hosts can change short addresses without causing protocol timeouts, yet generate little additional load on the network or the hosts. The code for accessing the short address cache adds 15 VAX instructions to both the transmit path and the receive path.

6.8.2 Bridging

A bridge is a device that sits between two networks and forwards packets from one to the other. It differs from a gateway in that a bridge is usually transparent to protocols above the data link layer. It differs from a repeater in that not all packets need appear on both sides of a bridge. Existing Ethernet bridges [14] forward packets from one Ethernet to another only if it appears likely that a host on the other network might wish to receive a packet. They do this by observing the traffic on both networks and learning which side each host is on. When the destination is on the other network, or when the location of the destination is unknown, they forward the packet.

We have implemented software that enables a Firefly to function as an Ethernet bridge, an Autonet bridge, and an Autonet-to-Ethernet bridge. Although we normally use only the last variation, it is easier to understand its operation by first considering a bridge between two Autonets. An Autonet bridge is slightly more complicated than an Ethernet bridge because a short address is not useful outside a single Autonet. When an Autonet bridge forwards a packet, it must modify the short addresses in the header. The destination short address is found using the techniques described in the previous section; the source short address is simply the short address of the bridge on the destination network. Unlike an Ethernet bridge, which receives all packets on the attached Ethernets, an Autonet bridge receives only broadcast packets and packets sent to its short address. Thus an Autonet bridge receives only a fraction of the packets on the attached networks and forwards most of the packets it receives.

As well as forwarding packets, an Autonet bridge also responds to ARP packets for hosts known to be on its other network. If the bridge is unsure of the location of a host, it does not respond to ARP requests immediately, but sends its own ARP requests on the other network; it responds to the original ARP request only if the destination responds. To hosts on the bridged Autonets, an Autonet bridge behaves like a large number of hosts sharing the same short address.

An Autonet-to-Ethernet bridge, the variation we normally use, has a few extra complications. It refuses to forward encrypted packets or packets longer than the maximum Ethernet size, though such forwarding could be arranged with a special encapsulation protocol. The bridge marks the header of all packets from the Ethernet to indicate to Autonet hosts that they should not attempt to use either encrypted communication or long packets when talking to the source host. This bridge adds or removes Autonet headers as packets are forwarded between the two networks. ARP packets from the Autonet are dealt with as previously described, except that they are never forwarded to the Ethernet. Instead, the location of Ethernet hosts is deduced from the client packets they send, in the same way as it is by Ethernet bridges.

In our Autonet-to-Ethernet bridge built on a Firefly, two of the four processors are devoted to forwarding packets: one executes the Ethernet driver thread and another executes the Autonet driver thread. In one second, the bridge can discard about 5000 small packets (66 bytes each), or forward over 1000 small packets, or forward 200-300 maximum-size Ethernet packets. The bridge is limited by its CPU when dealing with small packets, and by the speed of its I/O bus when dealing with large packets. The latency of the bridge is about a millisecond for a small packet. The bridge uses the LocalNet UID cache to remember which hosts are on which network as well as to map UIDs to short addresses for Autonet hosts. Using a single cache requires that a given UID be on one network or the other, never both.

6.8.3 Managing alternate links

Each host is connected to the Autonet via two links, but only one is in use at any given time. The Autonet driver is responsible for deciding which link to use, and for switching to the alternate link if the active link fails.

In normal operation, the driver sends a packet to the local switch every few seconds, both to confirm the host’s short address, and to verify that the link works. If the controller reports a link error, or if the switch fails to respond promptly, the driver tries to contact the local switch more vigorously. If the local switch has still not responded within three seconds, the driver switches links. After switching links, the driver forgets its short address, and tries to contact the local switch attached to the new link. If the switch responds, the host advertises its new short address and continues. If there is no response, the driver switches back to the first link after ten seconds. If neither link is operational, a host will switch between them once every ten seconds until it can contact a local switch.

The driver interface lets a client program switch the active link on demand and gather error rate statistics. Thus the alternate link can be tested, and if necessary replaced, before it is needed.

The current timeouts for link failover are quite long, and we expect to reduce them significantly in order to meet client failover requirements. At present, the mechanism is sufficient to allow a switch to fail without disrupting higher-level protocols. An enhancement to the protocol used between the switch and host would allow the driver to choose between two working links connected to different Autonet partitions by selecting the larger of the two partitions. Experience so far indicates that partition is extremely unlikely in a well connected Autonet, and so this improvement is likely to be of only marginal benefit.

7. Conclusions and future work

We are beginning to accumulate operational experience with Autonet. Our initial experience confirms that the goal of largely automatic operation of a network using arbitrary topology and active switches is realistic. Autonet is now the service network for most of the workstations at SRC. A new distributed file system is coming online with its servers only on Autonet. Once reconfiguration time was reduced below 1 second we ceased receiving complaints from users about the new network. Before that, with reconfigurations taking more than four seconds, users complained of dropped connections and RPC call failures. These symptoms were especially noticeable when the release of a new version of Autopilot caused 30 or more reconfigurations in quick succession. We now limit the disruption caused by the release of new Autopilot versions by making compatible versions propagate more slowly. Now users find Autonet indistinguishable from Ethernet. So far Autonet's higher bandwidth is largely masked by the Fireflies.

Even though Autonet has been in service for only a limited time, we have already learned some useful lessons. We would make several improvements to the switch hardware on the next iteration. The most significant change would be to allow the control processor to update the forwarding table without first resetting the switch. Resetting destroys all packets in the switch. Coupling resetting with reloading causes the initial forwarding table reload of a reconfiguration to destroy some tree-position packets, thus making reconfiguration take longer. Also, incremental reloads of the forwarding table to isolate problematic host links during normal operation are fairly disruptive with the present design.

One amusing surprise was caused by the fact that an unterminated link reflects signals. Such an unterminated link will occur, for example, when a host on the network is turned off. A packet addressed to the particular host would be reflected and retransmitted repeatedly, although for such unicast packets this would not be disruptive. Broadcast packets, however, are another matter. A reflected broadcast packet looks like a new broadcast packet, and is forwarded up the spanning tree to the root switch and then flooded down the spanning tree to all hosts where, of course, it is reflected again by the reflecting link. A "broadcast storm" results, with all hosts on the network receiving thousands of broadcast packets per second. Fortunately, the transition from terminated to unterminated almost always causes enough *BadCode* status to be counted at the link unit to cause the status sampler to classify the link broken and remove it from the forwarding table. We believe that a better solution to this problem is to make packets traveling in the "up" direction over a link look different than those traveling in the "down" direction. For example, different `start` flow control commands could be used. The link unit could then automatically discard packets headed in the wrong direction.

Another hardware change would be to make host controllers transmit the `host` flow control directive on the alternate port. This change would make it simpler for Autopilot to detect switch posts that are connected to alternate host ports.

Some lessons are quite mundane. The female F-connectors on host cabinet kits and switch link units have flats on their threaded barrel to allow a wrench to be used when mounting them. These flats make screwing on a cable very difficult, because it's hard to get the threads started correctly. The connectors without flats on the threads would be much better.

Autopilot has provided a series of interesting lessons. As a distributed program it has demonstrated a series of instructive bugs which we plan to document in another report. We have been reminded how hard such bugs are to find when packet traffic between switches cannot be observed directly and limited debugger facilities are available. Merging the logs of all switches is a very powerful technique for function and performance debugging, but synchronizing the timestamps from the individual logs must be done with high precision for the merged log to be useful.

Getting the status sampler, connectivity monitor, hardware skeptic, and connectivity skeptic algorithms structured and tuned for smooth operation also has been hard. Achieving both responsiveness and stability has required several iterations of the design. Further iterations probably will occur.

We expect that continued service use of the network will provide more lessons and expose areas where improvements in performance and reliability can be made.

Future work planned with Autonet includes building higher-speed controllers; developing network monitoring and management tools; improving the performance of reconfiguration; understanding how reconfiguration time varies with network size and topology; using the encryption facilities to support secure, authenticated communication; and applying the Autonet architecture to much faster links. We are interested in exploring modified algorithms that can perform local reconfigurations quickly when global reconfigurations are not required; finding ways to partition large installations into separately reconfigurable regions; and understanding the performance characteristics of different topologies and different routing algorithms.

We also would like to learn how to write an Autonet installation guide. For a network like Autonet to be widely employed, simple recipes must be developed for designing the topology of the physical configuration. The number of switches and the pattern of the switch-to-switch and host-to-switch links determine network capacity, reliability, and cost. Site personnel will need detailed guidance on determining a reasonable pattern to follow when installing the network and when growing it to meet increased load.

Acknowledgements

Autonet grew out of conversations between Andrew Birrell, Butler Lampson, Chuck Thacker, and Michael Schroeder in the summer of 1987. Roger Needham explored many overall

architectural options. Manolis Katevenis worked out a preliminary switch design. Michael Burrows was primarily responsible for the host and bridge software, with help from Michael Schroeder. Hal Murray, with help from Chuck Thacker, designed and implemented the Q-bus controller and also was responsible for wiring the building. Tom Rodeheffer was responsible for the switch control program and many switch diagnostics. Tom Rodeheffer and Michael Schroeder have worked on improving the performance of reconfiguration. Ed Satterthwaite designed and implemented the switch, with help from John Dillon and Chuck Thacker. Chuck Thacker worked out the scheme for full-duplex signaling on a single cable and designed the first-come, first-considered router. Tom Rodeheffer and Leslie Lamport invented the spanning tree algorithm. Michael Burrows and Andrew Birrell developed the short-address learning scheme. Bill Ramirez did the mechanical assembly of the switches. Herb Yeary checked out all switches and controllers, and diagnosed and repaired those that did not work. Michael Schroeder was technical project leader.

References

1. Advanced Micro Devices. 16-bit CMOS microprocessors (preliminary). AM29C116/116-1/116A. Publication 07697, Sunnyvale, CA, March 1988.
2. Advanced Micro Devices. Data ciphering processor. AmZ8086/Am9518. Publication 00618B, July 1984.
3. Advanced Micro Devices. TAXIchip integrated circuits (preliminary). AM7968/AM7969. Publication 07370, Sunnyvale, CA, May 1987.
4. American National Standard for Information Systems. Fiber distributed data interface (FDDI). Token ring media access control (MAC). ANSI Standard X3.139. American National Standards Institute, Inc., 1987.
5. American National Standard for Information Systems. Fiber distributed data interface (FDDI). Token ring physical layer protocol (PHY). ANSI Standard X3.148. American National Standards Institute, Inc., 1988.
6. Arnould, E.A., Bitz, F.J., Cooper, E.C., Kung, H.T., Sansom, R.D., and Steenkiste, P. A. The design of Nectar: a network backplane for heterogeneous multicomputers. In Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, (Boston, MA, April 3-6, 1989) ACM, New York, 1989, 205-216.
7. Birrell, A.D. An introduction to programming with threads. Research Report 35, DEC Systems Research Center, Palo Alto, CA, 1989.
8. Birrell, A.D., Guttag, J.V., Horning, J.J., and Levin, R. Synchronization primitives for a multiprocessor: a formal specification. In Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, (Austin, Texas, November 8-11, 1987), published as Operating Systems Review 21, 5, 94-102.
9. Birrell, A.D. and Nelson, B.J. Implementing remote procedure calls. ACM Transactions on Computer Systems 2, 1 (February 1984), 39-59.
10. The Ethernet local network: three reports. Tech. Rep. CSL-80-2, Xerox Palo Alto Research Center, Palo Alto, CA, 1980.
11. Digital Equipment Corp. Microsystems handbook, Appendix A: Q-bus. EB-26085-41/85. West Concord, MA, 1985.
12. Herbison, B.J. Low cost outboard cryptographic support for SILS and SP4. Submitted to Thirteenth National Computer Security Conference, Oakland, CA, 1990.
13. Ikeman, H., Lee, E.S, and Boulton, P.I.P. High-speed network uses fiber optics. Electronics Week 57, 28 (October 1984), 95-100.
14. Institute of Electrical and Electronic Engineers. Draft IEEE Standard 802.1. New, internetworking and systems management, Part D (MAC Bridge Standard), 1988. Available from Global Engineering Documents, Irvine, CA.
15. Motorola, Inc. *M68000 8-/16-/32-bit Microprocessors User's Manual*. Prentice-Hall, 1989.
16. Perlman, R. An algorithm for distributed computation of a spanning tree in an extended LAN. In Proceedings of the Ninth Data Communications Symposium, (Whistler Mountain, British Columbia, September 10-13, 1985), ACM, New York, 1985, 44-53.
17. Plummer, D.C, An Ethernet address resolution protocol -or- converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware. Network Information Center RFC826, SRI International, Menlo Park, CA, 1982.
18. Rovner, P.R. Extending Modula-2 to Build Large, Integrated Systems. IEEE Software 3, 6 (November 1986), 46-57.
19. Thacker, C.P., Stewart, L.C., and Satterthwaite, E.H. Jr. Firefly: a multiprocessor workstation. IEEE Transactions on Computers 37, 8 (August 1988), 909-920.
20. Tobagi, F.A., Borgonovo, F., and Fratta, L. Expressnet: a high-performance integrated-services local area network. IEEE Journal on Selected Areas in Communications SAC-1, 5 (November1983), 898-913.
21. Xilinx: the programmable gate array data book. Xilinx, Inc., San Jose, CA, 1989.