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, *Member, IEEE*, Thomas L. Rodeheffer, Edwin H. Satterthwaite, *Member, IEEE*, and Charles P. Thacker

Abstract—**Autonet is a self-configuring local area network composed of switches interconnected by 100 Mb/s, 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 ms/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 failover 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 2 km.

A 30-switch network with more than 100 hosts has been the service network for Digital's Systems Research Center since February 1990.

I. Introduction

Autonet is a self-configuring local area network composed of crossbar switches interconnected by 100 Mb/s, full-duplex, point-to-point links. It allows arbitrary interconnection; any switch port can be cabled to any other switch port or to a host network controller. Autonet attempts to provide a local area network (LAN) that presents a host interface similar to that of Ethernet [9] but offers higher performance. With 10 Mb/s host-to-host bandwidth and 10 Mb/s aggregate bandwidth, the Ethernet has served well as the standard LAN for high-performance workstations, but there is an increasing need for a faster, higher-capacity LAN.

Autonet and FDDI [3], [4] both use 100 Mb/s point-to-point links. The fundamental advantage of Autonet over a ring-based network such as FDDI is greater aggregate bandwidth from the same link bandwidth. With a ring, the aggregate network bandwidth is limited to the link bandwidth; with Autonet, the aggregate bandwidth can be many times greater. Other advantages of Autonet over a ring include lower latency, a more flexible approach to high availability, and a higher operational limit on the number of hosts that can be attached to a single LAN.

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 [6] as important as bulk-data transfer protocols. Autonet addresses all of these requirements.

The primary goal of the Autonet project was to build a useful local area network, rather than to do research into component technologies for computer networks. It focuses on the problems of composing switches into networks, not on those of building the switches themselves. 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.

Although Autonet is built from switches, it is not an ATM network. The unit of switching is the packet; packets are of variable length and may be several kilobytes long. Autonet does not attempt to solve many of the problems faced in telecommunications networks, despite their obvious importance. It is not designed to provide integrated voice or video service and does not offer guaranteed bandwidth to support real-time traffic. Our network load consists of bursty data traffic, which is tolerant of reordering but requires very low packet loss rates. We have assumed LAN traffic patterns and link latency in our design decisions. For example, Autonet uses simple designs for its switching, flow control, and link technology. We believe these to be adequate for a datagram LAN application.

The evolution of high-speed switched networks has blurred the distinction between LAN's and wide area networks (WAN's) on one hand and between LAN's and multiprocessor interconnects on the other. As a datagram-based network, Autonet is not directly comparable to a traditional WAN, but we believe that our basic design is scalable. Extension to a larger number of hosts is limited by the speed of the reconfiguration algorithm. The maxi-

Manuscript received January 8, 1991; revised May 9, 1991.

The authors are with the Systems Research Center, Digital Equipment Corporation, Palo Alto, CA 94301.

IEEE Log Number 9101970.

mum span of a single link is determined by link latency versus buffer size. Subsequent sections address these issues in detail. With current tradeoffs, a single Autonet can serve a campus spanning a few kilometers.

HPC [10], Nectar [5], and the Multiple Crossbar Network [11] are examples of hardware designs that are somewhat similar to Autonet but are intended to serve as "network backplanes" for multicomputer systems. We believe that Autonet is unique in addressing the issues involved in building a robust, self-configuring network that provides traditional LAN services.

The Autonet project has concentrated on automatic discovery of network topology, automatic failure isolation, deadlock-free packet routing without discarding packets, and compatibility with existing local area networks. The reason for this focus is that these seemed to be the most pressing problems that faced us in our environment. We believe that these problems are shared by other switch-based LAN's, and that the solutions may be applicable elsewhere.

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 distributed computing, Autonet's implementation uses distributed computing to perform its status monitoring and reconfiguration.

The development goal for Autonet was to produce 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 II of this paper contains a brief description of Autonet, to provide context for the rest of the paper. Section III describes the major design decisions that define the network. Section IV highlights the areas where Autonet appears to break new ground. Section V provides a more detailed description of the components of the network. Section VI describes the operation of these components. Finally, Section VII discusses our initial experience with Autonet and indicates directions for future work.

II. Overview

An Autonet, such as the one illustrated in Fig. 1, consists of a number of switches and host controllers connected by 100 Mb/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 nonblocking,

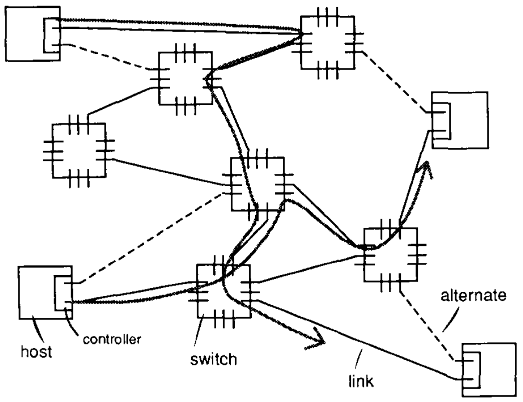

Fig. 1. A portion of an Autonet installation.

input-buffered crossbar in each switch connects the input and output ports. Depending on the topology, the network can handle many packets at once. Links are full-duplex.

Switches can be interconnected in an arbitrary topology, and this topology can 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 the best network availability, a host is connected to two switches. The current controller design allows only one of these connections to be used at a time. An Autonet should accommodate at least 1000 dual-connected hosts; possible improvements to the reconfiguration algorithm or faster switch processors would allow even larger Autonets.

III. Design Decisions

This section summarizes the major decisions that characterize the Autonet design.

A. *Point-to-Point Links at 100 Mb/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 *et al.* [25], 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 Mb/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, it can be incorporated into the network with appropriate interface electronics.

We settled on 100 Mb/s for the link bandwidth in Autonet because that speed represents a significant increase over Ethernet, while still being well within the limits of standard and affordable signaling technology. We chose the AMD TAXI chipset [1] 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 links that are ten times faster.

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 Ω 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, while fiber optic links might be used between buildings.

B. Unconstrained Topology with Precalculated Packet Routes

An Autonet is built from multiport 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 precalculated forwarding tables that are tailored to the current physical configuration.

A tree-shaped flooding network, like Hubnet [16], has an aggregate network bandwidth that is limited to the link bandwidth and has a 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.

C. 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 [12], a distributed algorithm executed by the bridges determines a forwarding pattern to interconnect all segments without introducing loops. The bridge algorithm 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 that executes on the control processor in each switch and 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.

D. 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, its latency is low, and its performance is easy to understand. A more sophisticated switch fabric could be used if it never discarded packets and if it could connect a single input port simultaneously to any set of output ports, to support broadcast. Some possible altneratives are surveyed by Ahmadi and Denzel [2], and their performance is analyzed by Oie *et al.* [18].

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 were all that could be fitted into a reasonably sized switch without using custom integrated circuits. The Autonet switch design would easily scale 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.

A disadvantage of crossbar fabrics is that the number of crosspoints grows with the square of the number of ports. We are interested in fairly small switches, which can be built economically as crossbars. For our purposes, a network of smaller switches is preferable to a single large switch. The latter accommodates incremental growth less gracefully and must be designed very carefully to avoid single points of failure.

E. 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 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 roundtrip 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 big enough to contain only 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.

Limited buffering 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 [15] can begin after 25 bytes have arrived.

Switches with FIFO input buffers suffer from head-of-line blocking. If a packet at the head of the queue is blocked because the required output port is busy, all packets behind it in the FIFO are blocked also. Karol *et al.* [14] show that, with certain assumptions, the maximum throughput of a nonblocking switch with such buffering is 58.6%. Related simulation results are reported by Newman [17]. Autonet switches depart from the assumptions of these studies in a number of details, but we expect them to saturate at similar levels. We consider this acceptable for our experiments. Datagram traffic patterns are likely to be bursty and nonuniform. The Autonet design allows us to add capacity incrementally, and we would expect to do so in a LAN well before reaching steady-state saturation.

Start–stop flow control is a link-level mechanism intended to prevent packet loss during short periods of overload. As discussed by Jain [13], such backpressure is not satisfactory for dealing with overloads of longer duration. Among other things, congestion can back up through the network, potentially delaying packets that will not even be routed over the congestion link. We expect backpressure to be supplemented by congestion control at the transport and network access levels.

An alternative buffering scheme would be to provide many packets of buffering at each receiving switch port 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 higher-level 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.

F. Deadlock-Free Multipath Routing

Because Autonet uses flow-controlled FIFO's 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 IV-B). 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.

G. 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 the result 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 [5] 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 a 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.

Short addresses differ from virtual circuit identifiers in that they are assigned to hosts rather than to logical streams of packets. Autonet carries only datagram traffic, switches do not distinguish streams of data, and the network does not guarantee in-order delivery. Therefore, we decided to use a simpler mechanism in which no circuit setup is required and the short address field in a packet is not changed as a packet traverses the network.

When considering alternative addressing schemes for LAN's, 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. All Autonet host controllers and switches have 48-bit UID's. Host software implements UID addressing based on Autonet short addresses (see Section III-J).

H. 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. Supporting broadcast complicates the problem of providing deadlock-free routing (see Section VI-F). Having low-latency broadcast, however, simplifies the problem of mapping destination UID's to short addresses.

I. 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 altnerate 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 causes the connected hosts to adopt their alternate ports to the network. Port failover can usually be done without disrupting communication protocols. The obvious disadvantage of having alternate ports is the increased cost of more host-to-swtich links and extra switches. For 100 Mb/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.

J. 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 LAN's 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 UID's and short addresses by inspecting arriving packets. 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. Dynamic learning algorithms have been used in other networks. For example, AppleTalk ARP uses similar techniques to maintain a table translating protocol addresses to hardware addresses [23]. Implementing the LocalNet abstraction requires that learning be pervasive and efficient. In fact, the learning algorithm requires only 15 extra instructions per packet received.

IV. 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.

A. 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 alogrithm similar to Perlman's [19]. 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. An Autonet cannot carry host traffic while reconfiguration is in progress. Doing so would invite deadlock caused by inconsistent forwarding tables in the various switches. Since Autonet carries only datagram traffic, brief service outages during reconfiguration are tolerable, but such indefinite termination is unacceptable.

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.

B. 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 nearer to the spanning tree root. Ties are broken by comparing switch UID's. 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.

C. Automatic Reconfiguration

The Autonet reconfiguration mechanism is based on each switch's monitoring of the state of its ports. Hardware status indicators report illegal transmission codes, syntax errors, lack of progress, and other conditions for each port. As a definitive 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.

D. 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. If more than one port is allowed, out-of-order delivery is possible. 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. Backpressure through the network can make output progress on one port dependent upon input progress on another port of the same switch.

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 [26], 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).

V. Components

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

A. Switch Hardware

Fig. 2 presents a simplified block diagram of the Autonet switch showing the most important data paths. The switching element is a 13 $\times$ 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 thirteenth input and output

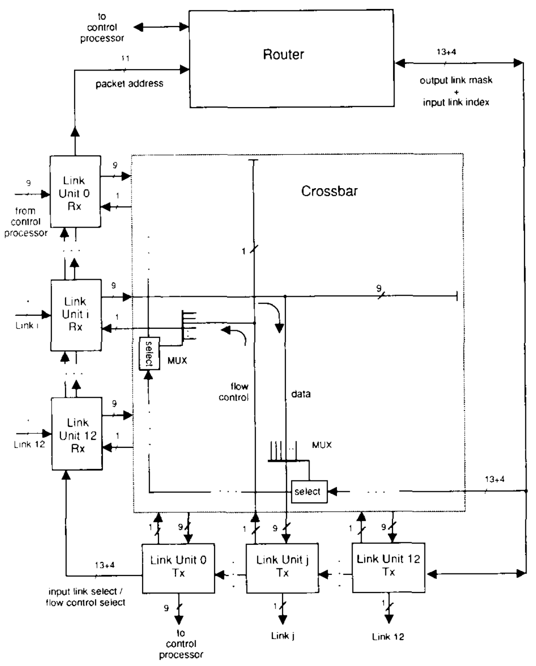

Fig. 2. Structure of an Autonet switch.

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 clocked at 12.5 MHz. The processor uses 1 Mbyte of video RAM as both its main memory and its buffers are 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 powerup. The processor subsystem also includes a timer that interrupts every 328 μs and a small ROM that contains the switch's 48 bit UID. Because of the limited space on the board, no CRC hardware is provided. CRC's for packets to/from the control processor are checked/generated by software.

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 Mb/s serial data stream on the link to a 9 bit parallel format. The ninth 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 $\times$ 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 Mb/s serial form and sends it over the link. The receive and transmit portions of a single link unit are tied together (by data paths not shown explicitly in Fig. 2) 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 VI-B). A link unit does not include CRC hardware; an Autonet switch does not check or generate CRC's on forwarded packets.

A link unit maintains a set of status bits that can be polled by the control processor. These 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. Via a control register, the processor can instruct each link unit to send special-purpose flow control directives to ignore received flow control and to illuminate status LED's on its front panel.

The *router* contains 64 Kbytes of memory for the *forwarding table* and a *scheduling 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 scheduling engine is implemented in a single Xilinx 3090 programmable gate array.

Most of the switch operates 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 six clock cycles. The maximum packet forwarding rate is about 2 million packets/second, which is adequate to dispatch continuous streams of 80 byte packets arriving on all 13 inputs. The latency from receiving the first bit of a packet on an input link to forwarding the first bit on an output link is 26–32 clock cycles if the output link and router are not busy.

The Autonet switch is packaged on five card types in a single-height Eurocard enclosure. A completely populated switch contains 12 link units, 5 two-bit crossbar slices, 1 control processor, and 1 router, all implemented on 100 $\times$ 160 mm cards. The backplane, into which all other card types plug at right angles, is a 430 $\times$ 130 mm board. A switch draws about 160 W of power.

B. Controller Hardware

The first host controller implemented for Autonet attaches to the Digital Equipment Corporation Q-bus [7] that is used in our Firefly [24] multiprocessor computers. The controller hardware supports optional DES encryption. Two cabinet kits, each containing link interface circuitry, implement the dual network ports. Host software determines which port to use. We describe this controller in more detail elsewhere [22]. The sustainable throughput is limited to the 14 Mb/s bandwidth of the Firefly Q-bus. Encrypted packets can be sent and received with no performance penalty.

In general, we believe that a network controller should be both simple and fast, and should play no role in the correct operation of the network fabric. Operating at the full 100 Mb/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. Using these principles, we are currently developing a higher-speed Autonet controller for the Digital Equipment Corporation TurboChannel [8]. It will support simultaneous transmission and reception on both attached links for an aggregate bandwidth of 4 $\times$ 100 Mb/s.

C. Link Hardware

The first links implemented for Autonet use 75 Ω coaxial cable. A hybrid circuit allows both channels of a full-duplex link to be carried on a single cable. Such links operate with adequate reliability when link lengths are restricted to 100 m. In practice, we never see errors on our best links, while a ''good'' link will have 1 or 2 detected errors per week (BER approaching 10^{-14}). The poorest links that we tolerate have occasional periods during which errors or short error bursts occur once an hour on average.

Our 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.

D. 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. A stable version of Autopilot is included in the switch boot ROM's and is automatically loaded when power is turned on or the switch is reset. Whenever a new version is ready for use, it is downloaded from a 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 major algorithms in Autopilot are described in later sections.

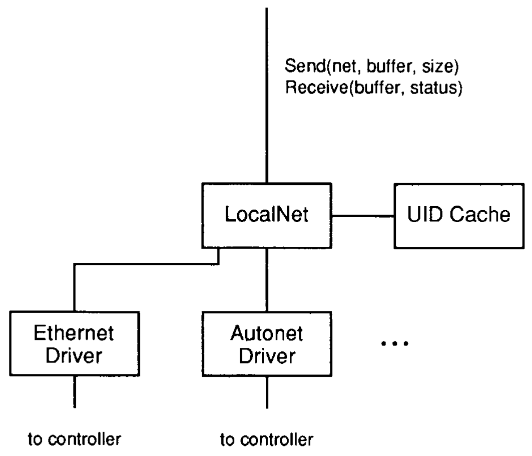

Fig. 3. Structure of low-level LAN software for the Firefly.

E. Host Software

Fig. 3 illustrates the structure of the *LocalNet* implementation, the low-level LAN software for the Firefly. 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 VI-H).

VI. Functions and Algorithms

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

A. Link Syntax

The TAXI transmitter and receiver are able to communicate 16 command values that are distinct from the 256 data byte values. **Sync** commands sent on an otherwise idle channel 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 be filled with either data bytes or commands, or can 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.

A special-purpose flow control directive, **idhy**, may also be sent. **Idhy**, which stands for "I do not 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.

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.

B. Flow Control

Fig. 4 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 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 at least 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 is synchronized by the transmitter and sent on the 1 bit reverse path through the crossbar to open and close the throttle on the FIFO that is the source of the packet.

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 a **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 a **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, the network software on the host would stop sending packets. Threads of control making calls to transmit packets would delay returning until more packets could be sent.

Autonet host controllers must obey **stop** commands but may not send them. 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 state the relationship between FIFO length, the frequency of flow control slots, and link latency. As-

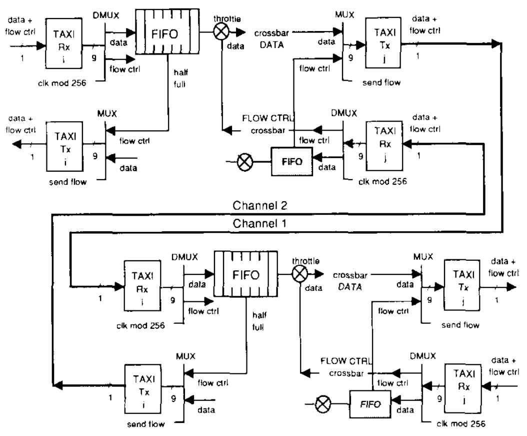

Fig. 4. Switch-to-switch flow control mechanism.

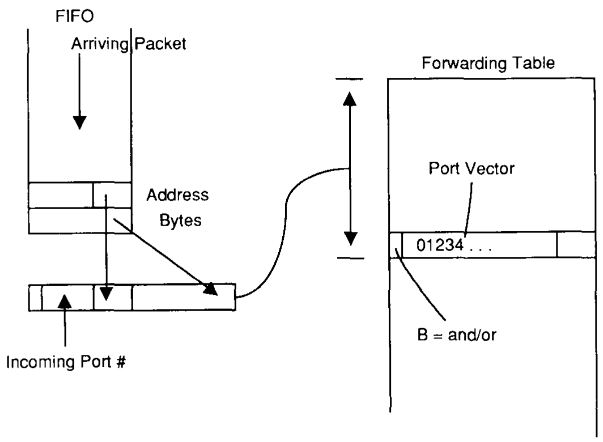

Fig. 5. Interpretation of switch forwarding table.

sume that the FIFO holds N bytes and that it issues a **stop** whenever the FIFO is more than half full. A flow control command is sent every S slots. For fiber optic cable (which is slightly slower than coaxial cable) and a slot transmission time of 80 ns, the link latency is 64.1 L slot transmission times, where L is the cable length in kilometers. To prevent the FIFO from overflowing then, it must be that:

$$N \geq 2(S - 1 + 128.2\ L).$$

For $S = 256$ slots and $L = 2$ km, we see that N must be 1024 bytes. A more general analysis appears elsewhere [22].

With these choices of S and L, Autonet actually uses 4096 byte FIFO's. The larger FIFO is used to solve a deadlock problem that is associated with broadcast packets. As explained in Section VI-F, the solution requires a transmitter of a broadcast packet to ignore **stop** commands until the end of the packet is reached. Thus, for broadcast packets flow control acts only between packets. The maximum allowable broadcast packet length B is the FIFO size minus the worst-case count of bytes already in the FIFO when the first byte of the broadcast packet arrives. Taking B into account, the size needed for the FIFO becomes:

$$N \geq 2(B + S - 1 + 128.2\ L).$$

The minimum acceptable value for B is about 1550 bytes. This size allows broadcast of 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.

C. Address Interpretation

As indicated earlier, Autonet packets contain *short addresses*. In our implementation, a short address is 11 bits, but 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 Fig. 5, 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, 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 of these ports are available. A broadcast entry with all 0's for the port vector tells the switch to discard the packet.

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: 1) it provides a way to differentiate between the two phases of flooding a broadcast packet (see Section VI-F); 2) it allows one-hop switch-to-switch packets to be addressed with the outbound port number; and 3) 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 they are used. As part of the distributed reconfiguration algorithm performed by the switches, each usable port of each working switch 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 a reserved address that directs the packet to the control processor of the local switch. The processor is told (via a register not shown in Fig. 2) 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.

A packet sent to the broadcast short address will be delivered to every host port in the network. (Section VI-F describes the flooding pattern used.) Additional addresses specify similar delivery to every switch or to every switch and host. Another reserved short address specifies packet loopback. A host uses this feature to test its links to the network.

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. These addresses are used during the initial stages of reconfiguration and also allow source-routed packet exchange among switches.

D. 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.

Fig. 6 illustrates the scheduling engine which 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 rightmost 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

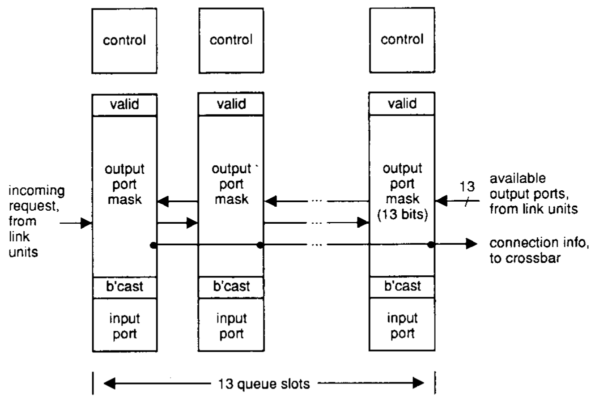

Fig. 6. Scheduling engine for switch output ports.

requested port vector. If multiple matches occur, then the free port with the lowest number 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.

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.

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 fster 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 the first choice of 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.

E. 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.

1) Port States: The port state monitoring mechanism dynamically classifies each port on an Autonet switch into one of the 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.

Fig. 7 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 and 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.

2) Hardware Port Status Indicators: Each link unit reports status bits that help the Autopilot to 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, while others indicate that one or more instances of a condition have occurred since the bit was last read by the control processor.

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. If most changes of interest reflect themselves in the hardware status bits, however, port status changes will be noticed more

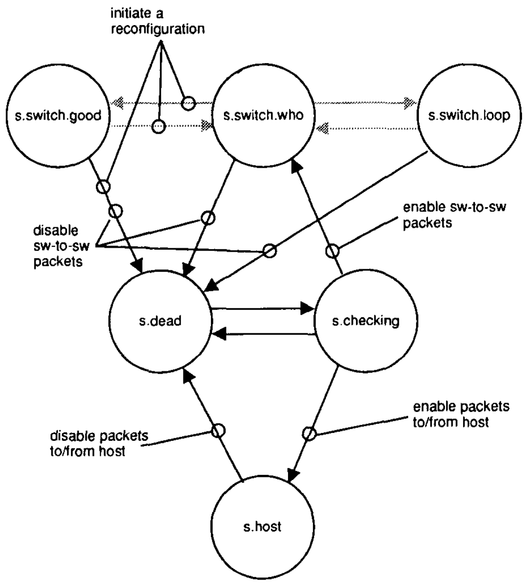

Fig. 7. Switch port states and transitions.

quickly. Autopilot can use the hardware status change to trigger an immediate verification by packet exchange.

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). 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 later in this section. 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. This bit records whether the most recently received flow control command was **host** (see Section VI-A). 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.

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.

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 Fig. 7, 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 a status bit set by receipt of **start** or **host**, 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. Another 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. A link attached to a switch or controller failing to obey flow control will cause repeated FIFO overflows and will be similarly classified.

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 nonresponsive 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 Fig. 7. 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 also from Fig. 7 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*.

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, while 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.

F. 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. 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 5—coexistence could lead to deadlock and packets being routed in loops.

1) Spanning Tree Formation: The distributed algorithm used to build the spanning tree is based on one described by Perlman [19]. 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 been 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 again can occur many times at most switches, a switch that believes itself to be the root of the spanning tree never becomes stable unless it really is the root and all other switches are already stable. Thus, the real root detects termination of the spanning tree algorithm.

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, it is then certain that spanning tree construction has terminated and it will know the complete topology and spanning tree for the network. A nonroot switch will know that the 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.

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, any change in the set of usable 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, 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 intermittent ports for increasingly longer periods.

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, 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.

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 because 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 small. When multiple routes lead from a source to a destination, 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.

5) Performance of Reconfiguration: In our service network, the 30 switches are arranged as an approximate 4×8 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 to the moment the last switch has loaded its new forwarding table. With the current version of Autopilot, reconfiguration of our network takes about 165 ms. Earlier versions were designed to facilitate debugging and were much slower. In trials using different subsets of our network, we found that the formula

$$58 + 3.34D + 1.36N + 0.315DN$$

where D is the network diameter and N is the number of switches, gives a reasonable approximation of the reconfiguration time in milliseconds. Analysis of the reconfiguration algorithm suggests that its performance is limited by the speed of the switch control processors—faster processors would allow larger networks.

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.

Fig. 8 illustrates the problem. Here we see part of a network including five switches V, W, X, Y, Z, and three

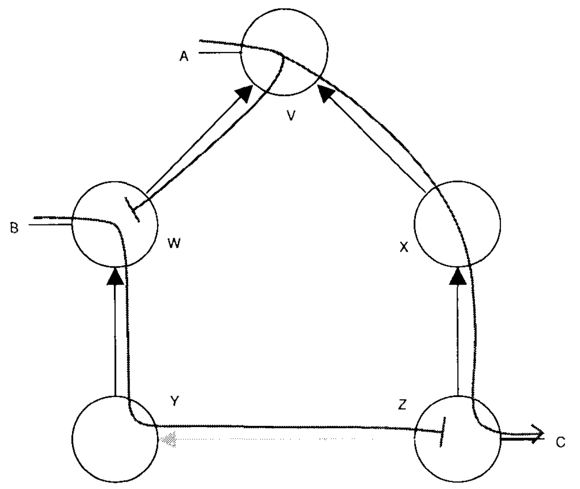

Fig. 8. Broadcast deadlock.

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.

The solution to this broadcast deadlock problem was mentioned in Section VI-B. 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.

G. 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 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 obtained. The merged log is a powerful tool for discovering functional and performance anomalies.

H. A Generic LAN

The LocalNet generic LAN interface in the host software hides most of the 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 is 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 field or 802.3 length field
0–64 K	Data (1500-byte limit for broadcast and 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. 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 details of which we omit here.

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 64 K 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 multicast UID's.

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 UID's 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 [20].

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.

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 and receive paths.

2) 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 0.7 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 every 0.1 seconds. If the local switch has still not responded within one second, 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.

VII. Conclusions and Future Work

We are accumulating 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 has been the service network for most of the workstations at SRC since February 1990. The servers in a new distributed file system are connected only to Autonet. Once reconfiguration time was reduced below one 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.

We have learned some useful lessons from operating a service network. We would make several improvements to the details of the switch hardware on the next iteration. For example, updating the forwarding table currently requires resetting the switch. This destroys all packets in the switch, so that isolating failed host links is disruptive and reconfiguration times are increased.

Autopilot has provided a series of interesting lessons. As a distributed program, it has demonstrated a series of instructive bugs. 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 difficult. Achieving both responsiveness and stability has required several iterations of the design. We are documenting our experiences to date in a separate report [21].

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.

Current work on Autonet includes building higher-speed controllers, developing network monitoring and management tools, and understanding how reconfiguration time varies with network size and topology. We are interested in exploring modified algorithms that can perform reconfigurations without disrupting service, finding ways to partition large installations into separately reconfigurable regions, understanding the performance characteristics of different topologies and different routing algorithms, and applying the Autonet architecture to faster links.

References

[1] Advanced Micro Devices. TAXIchip integrated circuits (preliminary). AM7968/AM7969. Publication 07370, Sunnyvale, CA, May 1987.

[2] H. Ahmadi and W. G. Denzel, "A survey of modern high-performance switching techniques," *IEEE J. Select. Areas Commun.*, vol. 7, no. 7, pp. 1091-1103, Sept. 1980.

[3] American National Standard for Information Systems, "Fiber distributed data interface (FDDI). Token ring media access control (MAC)," ANSI X3.139, 1987.

[4] ——, "Fiber distributed data interface (FDDI). Token ring physical layer protocol (PHY)," ANSI X3.148, 1988.

[5] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste, "The design of Nectar: A network backplane for heterogeneous multicomputers," in *Proc. Third Int. Conf. Architect. Supp. Program. Lang. Operat. Syst.*, Boston, MA, Apr. 3-6, 1988, pp. 205-216.

[6] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39-59, Feb. 1984.

[7] Digital Equipment Corp., *Microsystems Handbook, Appendix A: Q-bus.* 1985.

[8] Digital Equipment Corp., *Turbochannel Data Sheet EF-A0811-50.* TRI/ADD Program, 1990.

[9] "The Ethernet local network: Three reports," Tech. Rep. CSL-80-2, Xerox Palo Alto Research Center, Palo Alto, CA, 1980.

[10] R. D. Gagliano, B. S., Robinson, T. L. Lindstrom, and E. E. Sampieri, "HPC/VORX: A local area multicomputer system," in *Proc. Ninth Int. Conf. Distrib. Comput.*, Newport Beach, CA, June 5-9, 1989, pp. 542-549.

[11] R. Hoebelheinrich and R. Thomsen, "Multiple crossbar network: Integrated supercomputing framework," in *Proc. Supercomput. '89*, Reno, NV., Nov. 13-17, 1989, pp. 713-720.

[12] Draft IEEE Standard 802.1, "New, internetworking and systems management, Part D (MAC Bridge Standard)," 1988.

[13] R. Jain, "Myths about congestion management in high-speed networks," DEC-TR-724, 1990.

[14] M. J. Karol, M. G. Hluchyj, and S. P. Morgan, "Input versus output queueing on a space-division packet switch," *IEEE Trans. Commun.*, vol. 35, pp. 1347-1356, Dec. 1987.

[15] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Comput. Net.*, vol. 3, pp. 267-286, 1979.

[16] E. S. Lee and P. I. P. Boulton, "The principles and performance of Hubnet: A 50 Mb/s glass fiber local area network," *IEEE J. Select. Areas Commun.*, vol. SAC-1, no. 5, pp. 711-720, Nov. 1983.

[17] P. Newman, "A fast packet switch for the integrated services backbone network," *IEEE J. Select. Areas Commun.*, vol. 6, no. 6, pp. 1468-1479, Dec. 1988.

[18] Y. Oie, T. Suda, M. Murata, D. Kolson, and H. Miyahara, "Survey of switching techniques in high-speed networks and their performance," in *Proc. IEEE InfoCom '90*, 1990, pp. 1242-1251.

[19] R. Perlman, "An algorithm for distributed computation of a spanning tree in an extended LAN," in *Proc. Ninth Data Commun. Symp.*, Whistler Mountain, British Columbia, Sept. 10-13, 1985, pp. 44-53.

[20] D. C. Plummer, "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.

[21] T. L. Rodeheffer and M. D. Schroeder, "Automatic reconfiguration in Autonet," Res. Rep. 77, DEC Systems Research Center, Palo Alto, CA, 1991.

[22] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker, "Autonet: A high-speed, self-configuring, local area network using point-to-point links," Res. Rep. 59, DEC Systems Research Center, Palo Alto, CA, 1990.

[23] G. S. Sidhu, R. F. Andrews, and A. B. Oppenheimer, *Inside AppleTalk, Second Ed.* Reading, MA: Addison-Wesley, 1990, pp. 2.7-2.11.

[24] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, "Firefly: A multiprocessor workstation," *IEEE Trans. Comput.*, vol. 37, no. 8, pp. 909-920, Aug. 1988.

[25] F. A. Tobagi, F. Borgonovo, and L. Fratta, "Expressnet: A high-performance integrated-services local area network," *IEEE J. Select. Areas Commun.*, vol. SAC-1, no. 5, pp. 898-913, Nov. 1983.

[26] *Xilinx: The Programmable Gate Array Data Book.* San Jose, CA: Xilinx, 1989.

Michael D. Schroeder received the Ph.D. degree from the Massachusetts Institute of Technology, Cambridge, in 1972.

He is currently a member of the research staff at Digital Equipment Corporation's Systems Research Center in Palo Alto, CA. He has worked on computer security, authentication protocols, computer mail systems, naming in large distributed systems, remote procedure call performance, distributed file systems, and high-speed local area networks.

Andrew D. Birrell received the Ph.D. degree from the University of Cambridge, Cambridge, England, in 1978.

He has worked on a wide range of distributed systems problems. Projects include: "Grapevine" (a distributed naming and electronic mail service), the Cedar RPC system, DEC's "Global Name Service," and the Echo distributed file system. His current interests include fault tolerance and large-scale distributed systems.

Michael Burrows works at Digital Equipment Corporation's Systems Research Center. He dislikes biographies.

Hal Murray received the B.S. degree in electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1965.

He is currently employed at Digital Equipment Corporation's Systems Research Center. His main interests are high-speed networks and their connections to computers.

Roger M. Needham (M'84) received the Ph.D. degree from Cambridge University, Cambridge, England, in 1961.

He is Professor of Computer Systems and Head of the Computer Laboratory in the University of Cambridge, and was elected to the Royal Society of London in 1985.

Thomas L. Rodeheffer received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1985.

He is currently a member of the research staff at Digital Equipment Corporation's Systems Research Center. His main interests include invisible low-level support systems such as microcode, operating system kernels, and computer networks.

Edwin H. Satterthwaite (M'85) received the Ph.D. degree in computer science from Stanford University, Stanford, CA, in 1975.

He is a member of the research staff at Digital Equipment Corporation's Systems Research Center. His current interests include computer networks and data communications.

Charles P. Thacker received the A.B. degree in physics from the University of California, Berkeley, in 1967.

He is a Corporate Consultant Engineer at the Digital Systems Research Center in Palo Alto, CA. His research interests include computer architecture, computer networking, and computer-aided design.