VIRTUAL NETWORK TRANSPORT PROTOCOLS FOR MYRINET

Brent N. Chun

Alan M. Mainwaring

David E. Culler

University of California, Berkeley

Transport protocols using simple, low-cost mechanisms provide fast, robust, general-purpose cluster communication over protected virtual networks. Our design is evaluated on low- and high-contention microbenchmarks.

With microsecond switch latencies, gigabytes per second of scalable bandwidth, and low transmission error rates, cluster interconnection networks such as Myrinet[1] can provide substantially more performance than conventional local area networks. These properties stand in marked contrast to the network environments for which traditional network and internetwork protocols were designed. By exploiting these features, previous efforts in fast communication systems have produced a number of portable communication interfaces and implementations. For example, Generic Active Messages (GAMs),[2] Illinois Fast Messages (FMs),[3,4] the Real World Computing Partnerships's PM,[5] and BIP[6] provide fast communication layers. They constrain and specialize communication layers for an environment, for example, by supporting only single-program, multiple-data parallel (SPMD) programs or by assuming a perfect, reliable network. The systems achieve high performance, often times on a par with massively parallel processors.

Bringing this body of work into the mainstream requires more general-purpose and more robust communication protocols than those used to date.

Requirements

We needed a cluster protocol that would support

- multiprogramming,
- direct network access for all applications,
- protection from errant programs in the system,
- reliable message delivery in buffer overruns as well as dropped or corrupted packets, and
- mechanisms for automatically discovering the network's topology and distributing valid routes.

Multiprogramming is essential for clusters to become more than personal supercomputers. The communication system must provide protection between applications and isolate their respective traffic. Good performance requires direct network access and bypassing the operating system for all common case operations. The system should be resilient to transient network errors and faults—programmers ought not be bothered with transient problems that retransmission or other mechanisms can solve. However, catastrophic problems require handling at higher layers. Finally, the system should support automatic network management, including the periodic discovery of the network's topology and distribution of mutually deadlock-free routes between all pairs of functioning network interfaces.

Our protocol architecture makes a number of assumptions about the interconnect and the system. First, it assumes that the interconnect has network latencies on the order of a microsecond, a link bandwidth of a gigabit or more, and relatively error-free performance. It also assumes that the interconnect and host interfaces are homogeneous, and the problem of interest is communication within a single cluster network, not a cluster internet. System homogeneity eliminates a number of issues, such as the handling of different network maximum transmission units and packet formats, and probing for network operating parameters (for example, by TCP slow-start). Homogeneity also guarantees that the network fabric and the protocols used between its network interfaces are identical. This doesn't preclude use of heterogeneous hosts at the endpoints, such as hosts with different endian characteristics.

Lastly, the maximum number of nodes attached to the cluster interconnect is limited. This restriction enables memory resource

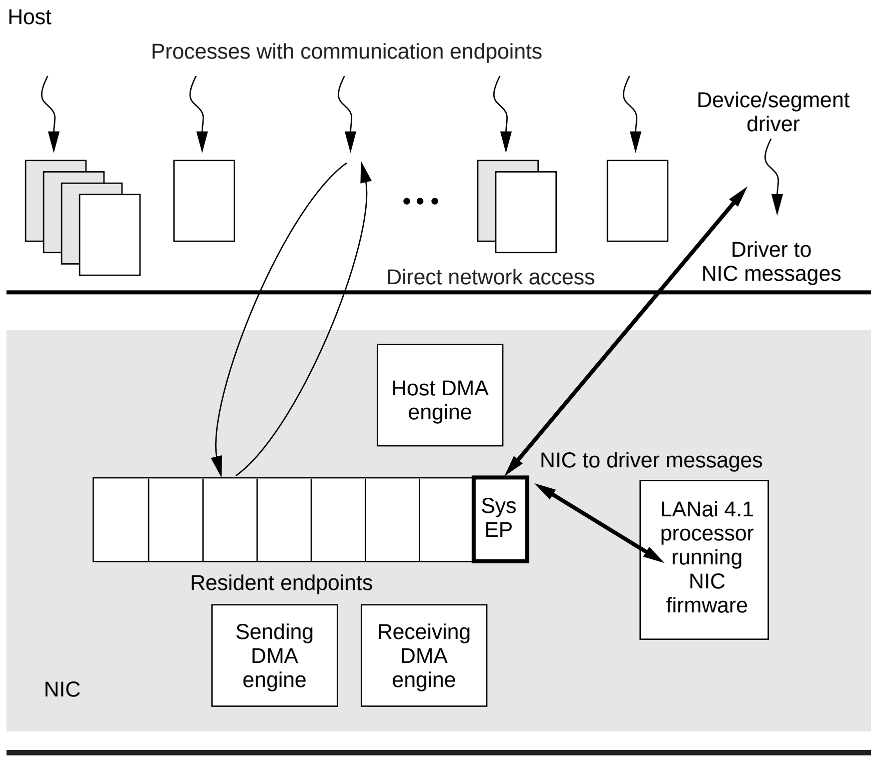


Figure 1. Processor/NIC node.

trading proportional to the number of network interfaces (NICs) in exchange for reduced computational costs on critical code paths. (Our system limits the maximum number of NICs to 256, though it would be straightforward to change the compile-time constants and to scale to a few thousand.)

Architecture

Our system has four layers:

- an active message applications programming interface,
- a virtual network system that abstracts network interfaces and communication resources,
- firmware executing on an embedded processor on the network interface, and
- processor and interconnection hardware.

AM-II API. Active Messages 2.0 (AM-II)[7] provides applications with the interface to the communication system. It allows an arbitrary number of applications to create multiple communication endpoints used to send and receive messages using a procedural interface to active messages primitives. AM-II supports three message types:

- short messages containing 4- to 8-word payloads,
- medium messages carrying a minimum of 256 bytes, and
- bulk messages providing large memory-to-memory transfers.

AM-II allows medium and bulk message data to be sent from anywhere in a sender's address space. The communication layer provides pageable storage for receiving medium messages. Upon receiving a medium message, its active message handler also receives a pointer to storage and can operate directly on the data. Bulk message data are deposited into per-endpoint virtual memory regions located anywhere in a receiver's address space. Receivers specify these regions with a base address and a message length. Applications can set and clear event masks to control whether semaphores associated with endpoints are posted when a message arrives into an empty receive queue in an endpoint. By setting the mask and waiting on the semaphore, multithreaded applications have the option of processing messages in an event-driven way. See Figure 1.

Per-endpoint message tags specified by an application isolate message traffic for unrelated applications. Each outgoing message contains a message tag for its destination endpoint. Messages are delivered if the tag in the message matches the tag of the destination endpoint. AM-II provides an integrated return-to-sender error model for both application-level errors (for example, nonmatching tags) and catastrophic network failures (for example, losing connectivity with remote endpoints). Any message that cannot be delivered to its destination returns to its sender. Applications can register per-endpoint error handlers to process undeliverable messages and to implement recovery procedures if so desired. If the system returns a message to an application, simply retransmitting the message is highly unlikely to succeed.

Virtual networks. Virtual networks are collections of endpoints with mutual addressability and the requisite tags necessary for communication. While AM-II provides an abstract view of endpoints as virtualized network interfaces, virtual networks view collections of endpoints as virtualized interconnects. There is a one-to-one correspondence between AM-II endpoints and virtual network endpoints.

The virtual networks layer provides direct network access via endpoints, protection between unrelated applications, and on-demand binding of endpoints to physical communication resources. Figure 2 illustrates this idea. Applications create one or more communication endpoints using API functions that call the virtual network segment driver to create endpoint address space segments. Pages of network interface memory provide the backing store for active endpoints, whereas host memory acts as the backing store for less active endpoints from the on-NIC endpoint "cache." Endpoints are mapped into a process's address space, where they are directly accessed by both the application and the network interface, thus bypassing the operating system. Because endpoint management uses standard virtual memory mechanisms, the endpoints leverage the interprocess protection enforced between all processes running on a system.

Applications may create more endpoints than the NIC can accommodate in its local memory. Providing that applications exhibit bursty communication behavior, a small fraction of these endpoints may be active at any time. Our virtual network system takes advantage of this when "virtualizing" the physical interface resources. Specifically, our Myrinet system uses NIC memory as a cache of active endpoints, and pages endpoints on and off the NIC as demanded, much like virtual memory systems do with memory pages and frames.

Analogous to page faults, endpoint faults can occur when an application writes a message into a nonresident endpoint or when a message arrives for a nonresident endpoint. Endpoint faults also occur whenever messages (sent or received) reference host memory resources that are not pinned, or for which there are no current DMA mappings. Examples are the medium message staging area, arbitrary user-specified virtual memory regions for sending messages, or endpoint virtual memory segments for receiving messages. Network interface virtualization, including endpoint cache management and the paging of endpoints, is handled using a custom virtual network segment driver.

NIC firmware. The firmware implements a protocol that provides reliable and unduplicated message delivery between NICs. The protocols must address four core issues: the scheduling of outgoing traffic from a set of resident endpoints, NIC-to-NIC flow control mechanisms and policies, timer management to schedule and perform packet retransmissions, and detection and recovery from errors. Details on the NIC protocols are given later.

The protocols implemented in firmware determine an endpoint's

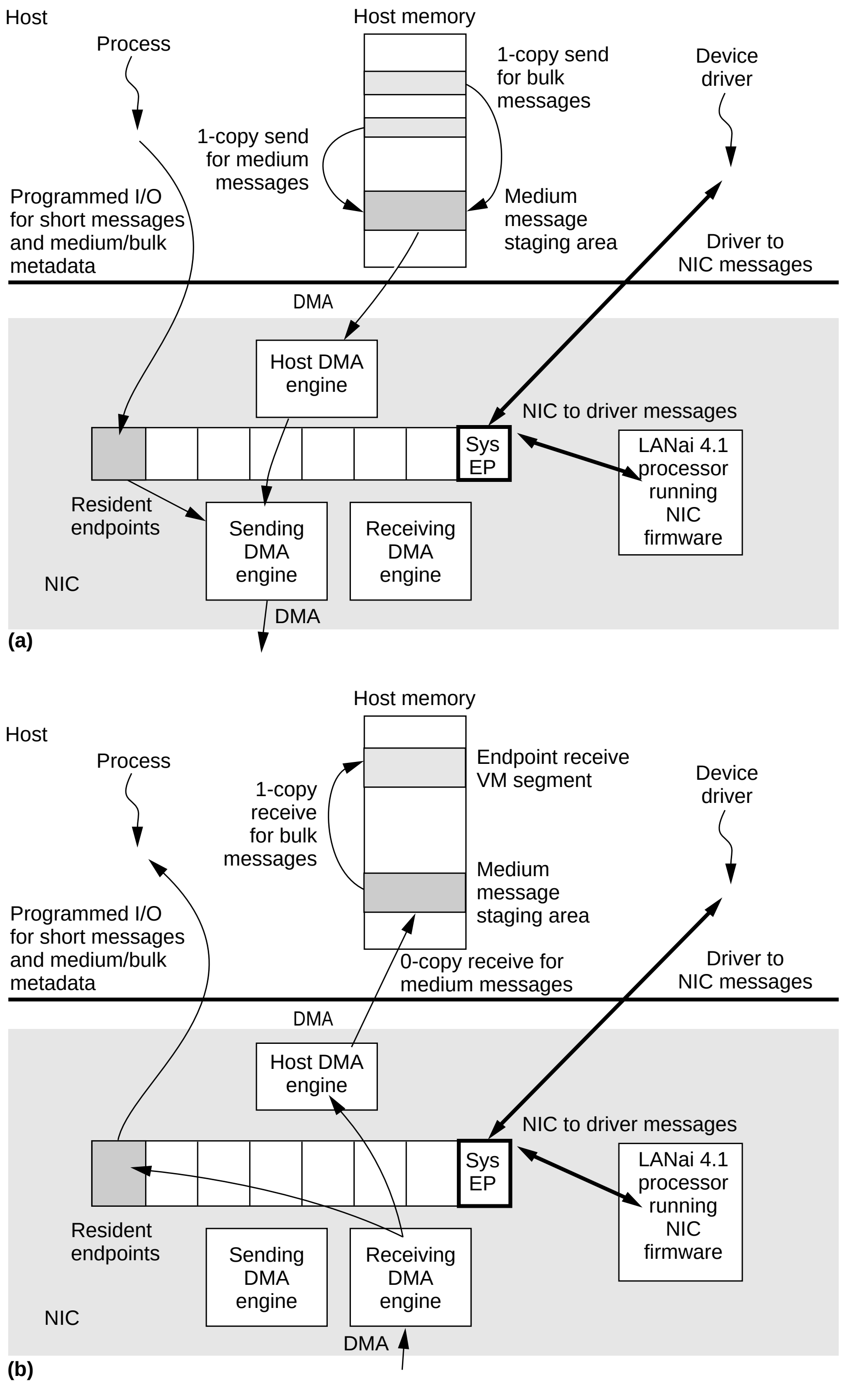

Host
Process
Host memory
1-copy send for bulk messages
Device driver
1-copy send for medium messages
Medium message staging area
Programmed I/O for short messages and medium/bulk metadata
Driver to NIC messages
DMA
Host DMA engine
NIC to driver messages
Sys EP
LANai 4.1 processor running NIC firmware
Resident endpoints
Sending DMA engine
Receiving DMA engine
NIC
DMA
(a)
Host memory
Host
Process
Endpoint receive VM segment
1-copy receive for bulk messages
Device driver
Medium message staging area
Programmed I/O for short messages and medium/bulk metadata
0-copy receive for medium messages
Driver to NIC messages
DMA
Host DMA engine
NIC to driver messages
Sys EP
LANai 4.1 processor running NIC firmware
Resident endpoints
Sending DMA engine
Receiving DMA engine
NIC
DMA
(b)

Figure 2. Data paths for sending (a) and receiving (b) active messages. Short messages are transferred using programmed I/O directly on endpoints (EP) in NIC memory. Medium messages are sent and received using per-endpoint medium message staging areas in the pageable kernel heap that are mapped into a process's address space. A medium message is a single-copy operation at the sending host and a zero-copy operation at the receiving host. Bulk memory transfers, currently built using medium messages, are single-copy operations on the sender and single-copy operations on the receiver.

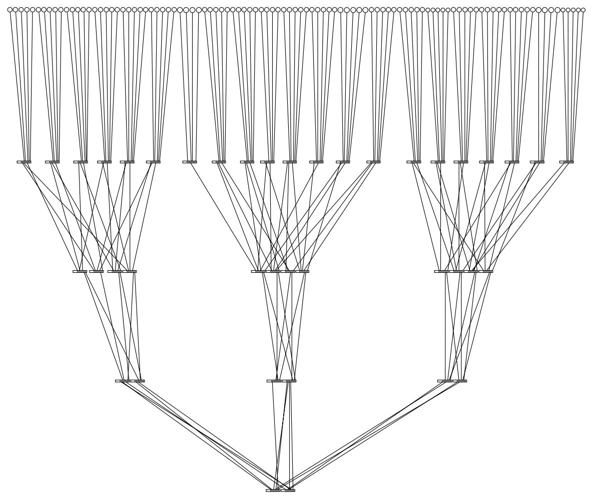


Figure 3. Berkeley NOW network topology as discovered by the mapper. The network mapping daemons periodically explore and discover the network's current topology, in this case a fat treelike network with 40 Myrinet switches. The three subclusters are currently connected through two switches using only 11 cables.

structure. Each endpoint has four message queues: request send, reply send, request receive, and reply receive. Each queue entry holds an active message. Short messages are transferred directly into a resident endpoint's memory using programmed I/O. Medium and bulk messages use programmed input and output for the active message portion and DMA for the associated bulk data transfer. Figure 2 illustrates the data flows for short, medium, and bulk messages through the interface. Medium messages require one copy on the sender and zero copies on the receiver. (Bulk messages, currently implemented using medium messages, require one copy on the sender and one copy on the receiver. The code for zero-copy bulk transfers exists but has not been sufficiently tested.)

Hardware. The system hardware consists of 100-plus 167-MHz Sun UltraSparc workstations interconnected with Myrinet (Figure 3),[8] a high-speed local area network with cut-through routing and link-level back pressure. The network uses 40 eight-port crossbar switches with 160-Mbyte/s full-duplex links. Each host contains a LANai 4.1 network interface card on the SBus. Each NIC contains a 37.5-MHz embedded processor, 256-Kbyte SRAM, and a single-host SBus DMA engine but independent network send and receive DMA engines.

NIC protocols

A set of lightweight NIC protocols that support the AM-II communications API and virtual networks also provide basic transport protocol functionality. The protocols are implemented as firmware running on Myricom LANai 4.1 cards.

Endpoint scheduler. Because our system supports both direct network access and multiprogramming, the NIC has a new task of endpoint scheduling: sending messages from the current set of cached endpoints. This situation differs from that of traditional protocol stacks, such as TCP/IP. There, messages from applications pass through layers of protocol processing and multiplexing before ever reaching the network interface, so the NIC services shared outbound (and inbound) message queues. With virtual networks, the queues are differentiated.

Endpoint scheduling policies choose how long to service any one endpoint and which endpoint to service next. A simple round-robin algorithm that gives each endpoint equal but minimal service time is both fair and starvation free. If all endpoints always have messages waiting to send, this algorithm might be satisfactory. However, if application communication is bursty,[9] spending equal time on each resident endpoint is not optimal. Better strategies exist that minimize the use of critical NIC resources examining empty queues.

The endpoint scheduling policy must balance optimizing the throughput of a particular endpoint's responsiveness against aggregate throughput and response time. Our current algorithm uses a weighted round-robin policy that focuses resources on active endpoints. Empty endpoints are skipped. For an endpoint with pending messages, the NIC makes 2^k attempts to send, for some parameter k. This holds even after the NIC empties a particular endpoint—it loiters in case the host enqueues additional messages. Loitering also allows a firmware-to-cache state, such as packet headers and constants while sending messages from an endpoint, lowering per-packet overheads. While a larger k results in better performance during bursts, too large a k degrades system responsiveness with multiple active endpoints. Empirically, we have chosen a k of 8.

Lightweight flow control. In our system, a flow control mechanism has two requirements. On one hand, it should allow an adequate number of unacknowledged messages to be in flight to fill the communication pipe between a sender and a receiver. It should also limit the number of outstanding messages and manage receiver buffering to make buffer overruns infrequent. In steady state, a sender should never wait for an acknowledgment before sending more data. Assuming the destination process is scheduled and attentive to the network, given bandwidth B and round-trip time RTT, this requires allowing at least $B \times RTT$ bytes of outstanding data.

Our system addresses flow control at three levels: user-level active message credits for each endpoint; NIC-level stop-and-wait flow control over multiple, independent logical channels; and network back-pressure.

User-level credits. These credits rely on the request-reply nature of AM-II, allowing each endpoint to have at most K_{user} outstanding requests waiting for responses. By choosing a large enough K_{user}, endpoint-to-endpoint communication proceeds at the maximum rate. To prevent receive buffer overflow, we designed endpoint request receive queues to be large enough to accommodate several senders transmitting at full speed. Because senders have at most a small K_{user} number of outstanding requests, setting the request receive queue to a small multiple of K_{user} is feasible. Additional mechanisms, discussed shortly, engage when overruns do occur.

In our protocol, with 8-Kbyte packets the bandwidth delay product is 31 Mbytes/s $\times$ 349 microseconds, or 11,345 bytes—less than two 8-Kbyte messages. For short packets the bandwidth delay product is 62,578 messages/s × 42 microseconds, or 2.63 messages. To provide slack at the receiver and to optimize arithmetic computations, K_{user} is rounded to 4. The NIC must provide at least this number of logical channels to accommodate this number of outstanding messages, as discussed next.

Stop-and-wait over logical channels. To keep a full communication pipe in steady state, the NIC, which is responsible for time-out and retry, must allow at least K_{user} outstanding messages. K_{user} is chosen to match the bandwidth delay product of the network. It accomplishes this by overlaying multiple, independent logical channels (K_{user} channels for requests and K_{user} channels for replies) over each physical route to each destination NIC. K_{user} logical channels for both requests and replies ensure that neither the sender nor the receiver is a bottleneck in steady state with respect to outstanding data. Having separate request and reply logical channels prevents deadlock.

Two simple data structures manage NIC-to-NIC flow control information. These data structures also record time-out/retry and error detection information. Each row of the send channel control table in Figure 4 holds the states of all channels to a particular destination interface. Each intersecting column holds the state for a particular logical channel. This implicit bound on the number of outstanding messages enables implementations to trade storage for reduced arithmetic and address computation. Two simple and easily addressable data structures with O(number of NICs × number of channels) entries are sufficient.

Link-level back pressure. This scheme ensures that under a heavy load, the network does not drop packets. Credit-based flow control in the AM-II library throttles individual senders but cannot prevent high contention for a common receiver. With link-level back pressure, end-to-end flow control remains effective and its overheads remain small. This trades network use under load—allowing packets to block and to consume link and switch resources—for simplicity. As shown later, this hybrid scheme performs very well.

Receiver buffering. Some fast communication layers prevent buffer overruns by dedicating enough receiver buffer space to accommodate all potential messages in flight. With P processors, credits for K outstanding messages, and a single endpoint per host, this requires $O(K \times P)$ storage. Small-scale systems with one endpoint made allocating $O(K \times P)$ storage practical. However, large-scale systems with a large number of communication endpoints require $O(K \times E)$ storage, where E is the number of endpoints in a virtual network. This has serious scaling and storage use problems that make preallocation approaches impractical, as the storage grows proportionally to virtualized resources, not physical ones. Furthermore, with negligible packet retransmission costs, alternative approaches involving modest preallocated buffers and packet retransmission become practical.

We provide request and response receive queues, each with 16 entries ($4 \times K_{\text{user}}$) for each endpoint. These are sufficient to absorb the load from up to four senders transmitting at their maximum rates. When buffer overflow occurs, the protocol drops packets and sends NACK messages to senders.

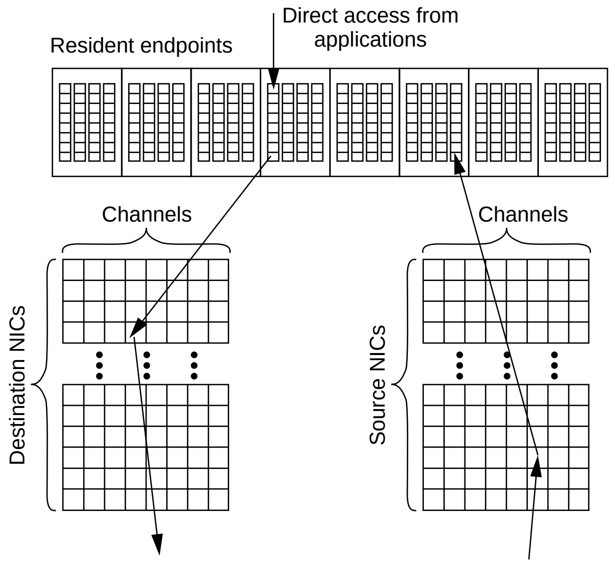


Figure 4. NIC channel tables provide easy access to NIC flow control, time-out/retry, and error detection information. The NIC uses stop-and-wait flow control on each channel and manages communication state information in channel table entries. In the send table (left), each entry includes time-out/retry information (packet time stamp, pointer to an unacknowledged packet, number of retries with no receiver feedback), sequencing information (next sequence number to use), and whether the entry is in use. In the receive table (right), each entry contains sequencing information for incoming packets (expected sequence number). Half of the channels are reserved for requests; the other half, for replies to prevent deadlock.

The firmware automatically retransmits such messages. An important consequence of sizing the request and reply queues to be $4 \times K_{\text{user}}$-entries deep is that our virtual network segment driver can use a single virtual memory page for each endpoint, simplifying its memory management activities.

Time-out and retry. To guarantee at-most-once delivery semantics and address transient hardware errors (CRC errors and truncated packets), a communication system must process packet time-outs and retransmissions. The time-out/retry algorithm determines how packet retransmission events are scheduled, how they are deleted, and how retransmission is performed. Sending a packet schedules a timer event; receiving an acknowledgment deletes the event. All send table entries are periodically scanned for packets to retransmit.

The per-packet time-out/retry costs must be small. This requires that the costs of scheduling a retransmission event on each send operation and deleting a retransmission event on reception of an acknowledgment be negligible. Depending on the granularity of the time-out quantum and the frequency of time-out events, different trade-offs exist that shift costs between per-packet operations and retransmissions. For example, we use a larger timer quantum and low per-packet costs at the price of more expensive retransmissions. The later section on performance shows that this hybrid scheme has near zero amortized cost for workloads in which packets are not retransmitted.

Our transport protocol implements time-out and retry with positive acknowledgments in the interface firmware. This provides efficient acknowledgments and minimizes expensive SBus transactions. (We currently do not perform the obvious piggybacking of ACKs and NACKs on active message replies.) Channel management tables store time-out and transmission states.

Sending a packet involves reading a sequence number from the appropriate entry in the send table indexed by the destination NIC and a free channel, saving a pointer to the packet for potential retransmissions, and recording the time the packet was sent. The receiving NIC then looks up sequencing information for the incoming packet in the appropriate receive table entry indexed with the sending NIC's identity and the channel on which the message was sent. If the sequencing information matches, the receiver sends an acknowledgment to the sender. Upon its receipt, the sender updates its sequencing information and frees the channel for use by a new packet.

By using a simple and easily addressable data structure, each with O(number of NICs $\times$ number of channels) entries, scheduling and deleting packet retransmission events take constant time. For retransmissions though, the NIC performs O(number of NICs $\times$ number of channels) work. Maintaining unacknowledged packet counts for each destination, NIC reduces this cost significantly. Sending a packet increments a counter to the packet's destination NIC and receiving the associated acknowledgment decrements the counter. These counts reduce retransmission overheads in proportion to the total number of network interfaces.

Virtual networks introduce new issues for at-most-once delivery semantics in the presence of hardware errors. Because endpoints may be nonresident or may not have DMA resources (medium-message staging areas) set up, a packet may need to be retried because of unavailable resources.

Our system sends NACKs to senders when destination endpoint resources are unavailable. Upon receiving such a NACK, a sender notes that the receiving interface is still reachable. The packet will be retried by the same time-out/retry mechanism used to deal with hardware errors. Like hardware errors, resource unavailability should be infrequent. Therefore, using the same time-out/retry mechanism, which may use coarse-grain time-outs, should add very little overhead.

Error handling. Our system addresses packet delivery problems at three levels: NIC-to-NIC transport protocols, the AM-II API return-to-sender error model, and user-level network management daemons. The transport protocols are the building blocks on which the higher level API error models and the network management daemons depend. The transport protocols handle transient network errors by detecting and dropping each erroneous packet and relying upon time-outs and retransmissions for recovery. After 255 retransmissions for which no ACKs or NACKs were received, the protocol declares a message as undeliverable and returns it to the AM-II layer. (Time-out/retransmission mechanisms require that sending interfaces have a copy of each unacknowledged message anyway.) The AM-II library invokes a per-endpoint error handler function so that applications may take appropriate recovery actions.

Transient errors. Positive acknowledgment with time-out and retransmission ensures the delivery of packets with valid routes. Data packets can be dropped or corrupted as well as protocol control messages. To ensure that data packets are never delivered more than once to a destination despite retransmissions, they are tagged with sequence numbers and time stamps. With a maximum of $2k$ outstanding messages, detecting duplicates requires $2k + 1$ sequence numbers. For our alternating-bit protocol on independent logical channels, k equals 0.

Return-to-sender. The NIC determines that destination endpoints are unreachable by relying upon its time-out and retransmission mechanisms. If after 255 retries (several seconds) the NIC receives no ACKs or NACKs from the receiver, the protocol deems the destination endpoint as unreachable. When this happens, the protocols mark the sequence number of the channel as uninitialized and return the original message to the user level via the endpoint's reply receive queue. The application handles undeliverable messages as it would any other active message, with a user-specifiable handler function. Should no route to a destination NIC exist, all of its endpoints are trivially unreachable.

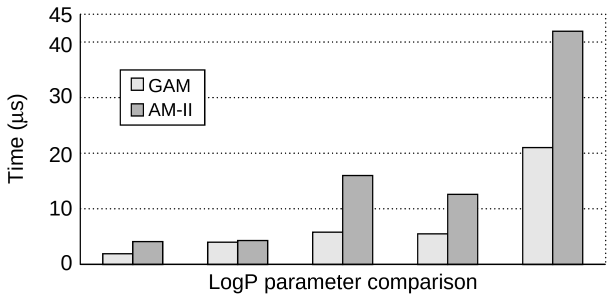
Network management errors. The system uses privileged mapper daemons, one for each interface on each system node, to probe and discover the current network topology. Given the current topology, the daemons elect a leader that derives and distributes a set of mutually deadlock-free routes to all NICs in the system.[10] Discovering the topology of a source-routed, cut-through network with anonymous switches, like Myrinet, requires use of network probe packets that may potentially deadlock on themselves or on other messages in the network. Hence, on-line mapping daemons can cause truncated and corrupted packets to be received by interfaces (as a result of switch hardware detecting and breaking deadlocks), even when the hardware is working perfectly. From the transport protocol's perspective, mapper daemons perform two specialized functions:

- sending and receiving probe packets with application-specified source-based routes to discover links, switches, and hosts, and
- reading and writing entries in NIC routing tables.

These special functions can be performed using privileged endpoints available to privileged processes.

Performance results

The first microbenchmarks for performance characterize the system using the LogP communication model. They lead to a comparison with a previous generation of an active message system and to an understanding of the costs of the added functionality. The next benchmarks examine performance between hosts under varying degrees of destination



	O_s	O_r	g	L	RTT
GAM	1.90	4.00	5.80	5.50	21.00
AM-II	4.09	4.28	15.98	12.60	41.94

Figure 5. Performance characterization using the LogP model. LogP parameters for two active message systems on identical hardware: AM-II, our general-purpose active message system with virtual networks and the return-to-sender error model, and GAM, an earlier active message system for SPMD parallel programs without virtual networks, an error models, and other features.

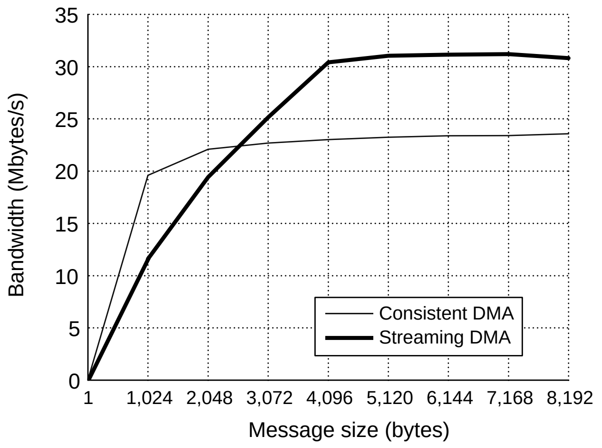


Figure 6. Sending bandwidth as a function of message size in bytes. Consistent host-to-NIC DMA operations across the SBus have higher performance for small transfers. Streaming transfers achieve higher performance once the data transfer times swamp the cost of flushing a hardware stream buffer in the SBus bridge chip.

endpoint contention. They conclude with an examination of system performance as the number of active virtual networks increases. We ran all programs on the Berkeley Network of Workstations (NOW) system in a stand-alone environment. We disabled topology acquisition and routing daemons, eliminating background communication activities normally present.

LogP characterization. This model uses four parameters to characterize the performance of communication layers. This parameterization enables the fair comparison of different communication layers. Certain microbenchmarks[2] automatically derive the model parameters of latency *L*, overhead *O*, and gap *g*. Each parameter has a simple interpretation. The number of processors *P* is given. *O* has two components, sending (O_s) and receiving (O_r). These measure the host processor time spent writing a message to and reading a message from an endpoint. *g* measures the time through the system's rate-limiting stage; *L* is the remaining time unaccounted for by the overheads.

Figure 5 shows the LogP parameters for AM-II and GAM. It compares the measured AM-II round-trip time of 41.94 microseconds with GAM's time of 21 µs. Of the 17.37-µs one-way time, the system spends 5 µs writing the message into the sender's endpoint and 3.3 µs reading the messages from the receiver's endpoint. The two network interfaces spend 13.82 µs transmitting the data message as well as transmitting its acknowledgment. For AM-II, *g* is larger than (O_s + O_r), because the network interface firmware limits the message rate. For GAM, the gap is smaller than its ($O_s + O_r$), indicating that the active message library code executing on the host processors limits the message rate.

Although in both cases the microbenchmarks use active messages with 4-word payloads, the AM-II send overhead is larger because additional information such as a tag is stored to the network interface across the SBus. The AM-II gap is also larger because the firmware constructs a private header for each message, untouchable by any application, that is sent using a separate DMA operation. This requires additional firmware instructions and memory accesses.

Table 1. Aggregate bandwidth (BW) and average round-trip times (*RTT*) for 92 nodes with different message permutations. (More recent work has considerably improved upon this performance. See http://now.cs.berkeley.edu.)

Permutation	Average BW (Mbytes/s)	Aggregate BW (Gbytes/s)	Average *RTT* (µs)
Cshift	25.42	2.33	67.8
Neighbor	30.97	2.85	47.5
Bisection	5.65	0.52	50.8

Contention-free performance. Figure 6 shows the endpoint-to-endpoint bandwidth between two machines. Because the NIC can only send DMA messages between the network and its local memory, a store-and-forward delay for large messages moves data between host memory and the interface. Although the current network interface firmware does not pipeline bulk data transfers to eliminate this delay, streaming transfers nevertheless reach 31 Mbytes/s with 4-Kbyte messages. (With GAM, pipelining DMA operations to receive messages from the network with DMA operations to move buffers to host memory increased bulk transfer performance to 38 Mbytes/s.)

In Table 1's cshift permutation, each node sends requests to its right neighbor and replies to requests received from its left neighbor. With neighbors, adjacent nodes perform pairwise exchanges. In bisection, pairs of nodes separated by the network bisection perform pairwise exchanges. Bandwidth measurements use medium messages, whereas RTT

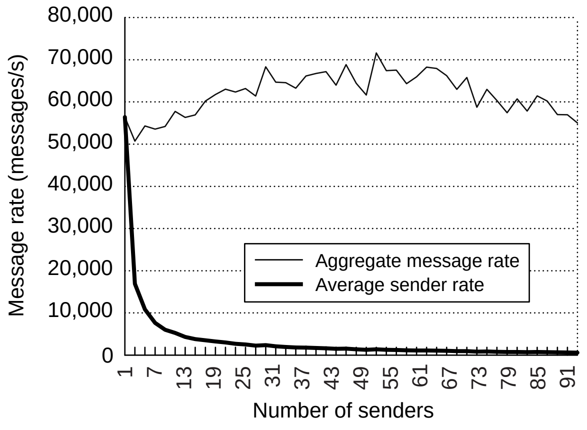


Figure 7. Active message rates with destination endpoint contention within a single virtual network.

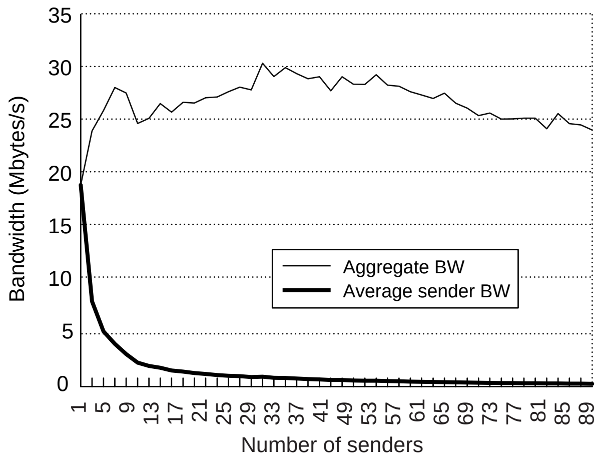


Figure 8. Delivered bandwidths (BWs) with destination endpoint contention within a single virtual network.

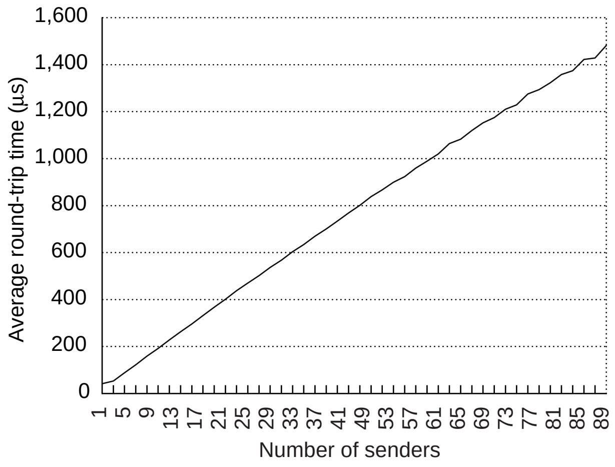


Figure 9. Round-trip times with destination endpoint contention within a single virtual network.

measurements use 4-word active messages.

Table 1 presents three permutations and their resulting average per-host sending bandwidths, aggregate sending bandwidths, and per-message round-trip times when run on 92 NOW machines. Each column shows that the bandwidth scales as the system reaches a nontrivial size. The first two permutations, circular shift and neighbor exchange, are communication patterns with substantial network locality. As expected, these cases perform well, with bandwidths near their peaks and per-message round-trip times within a factor of 2 of optimal. The bisection exchange pattern shows that a large number of machines can saturate the network bisection bandwidth. See Figure 3 again for the network topology and the small number of bisection cables when this study was performed.

Single virtual network. The next three figures show the performance of the communication subsystem in the presence of contention, specifically when all hosts send to a common destination host. All traffic destined for the common host is also destined for the same endpoint. For reasons that will become clear, we refer to the host with the common destination as the server, and all other hosts as the clients.

Figure 7 shows the aggregate message rate of the server (top line) as the number of clients sending it 4-word requests and receiving 4-word response messages increases. Additionally it shows the average per-client message rate (bottom line) as the number of clients increases to 92. Figure 8 presents similar results, showing the sustained bandwidth with bulk transfers to the server as the number of clients sending 1-Kbyte messages to it and receiving 4-word replies. The average per-client bandwidth gracefully and fairly degrades. We conjecture that the fluctuation in the server's aggregate message rates and bandwidths arises from acknowledgments for reply messages encountering congestion (namely, other requests also destined for the server). The variation in per-sender rates and bandwidths is too small to be observable on the printed page. Figure 9 shows the average per-client round-trip time as the number of clients grows to 92 hosts. The slope of the line is exactly the gap measured in the LogP microbenchmarks.

Mutual virtual networks. We can extend the previous benchmark to stress virtual networks. We increase the number of server endpoints to the maximum of seven that can be cached in the interface memory. Then we continue to incrementally add endpoints to increasingly overcommit the resources. Thus, rather than clients sharing a common destination endpoint, each client endpoint now has its own dedicated server endpoint. With *N* clients, the server process has *N* different endpoints, where each one is paired with a different client, resulting in *N* different virtual networks. This contains client messages within their virtual network and guarantees that messages in other virtual networks make forward progress.

Figure 10 shows the aggregate server message rate and per-client message rates (with error bars) over a 5-minute interval. The number of clients continuously making requests of the

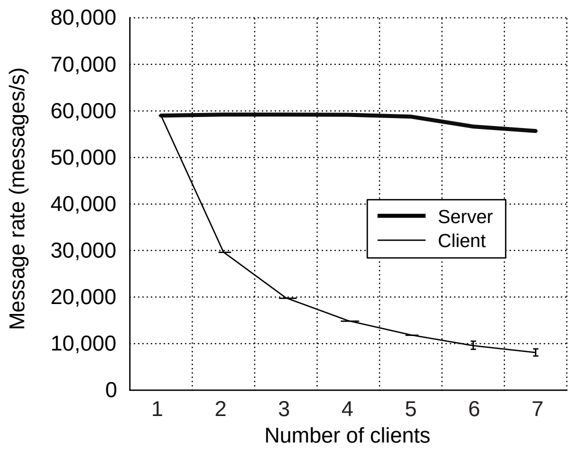


Figure 10. Aggregate server and per-client message rates with small numbers of virtual networks.

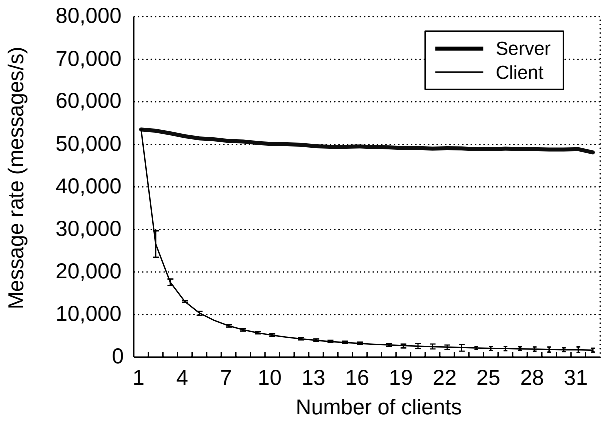


Figure 11. Aggregate server and per-client message rates with large numbers of virtual networks.

server varies from one to seven. In this range, the network interface's seven endpoint frames can accommodate all server endpoints. This scenario stresses both the scheduling of outgoing replies and the multiplexing of incoming requests on the server. The results show server message rates within 11% of their theoretical peak of 62,578 messages per second, given the measured LogP gap of 15.98 μs. The per-client message rates are within 16% of their ideal fair share of $1/N$th of the server's throughput. Steady server performance and the graceful system response to increasing the load demonstrate the effective operation of the flow-control, endpoint scheduling, and multiplexing mechanisms throughout the system.

Figure 11 extends the scenario shown in Figure 10. Previously, the server host was a single-threaded process, polling its endpoints in round-robin fashion. When the number of busy endpoints exceeds the network interface capacity, the virtual network system actively loads and unloads endpoints into and out of interface memory on demand. When the server attempts to write a reply message into a nonresident endpoint (or when a request arrives for a nonresident endpoint), a page fault occurs. The virtual network driver moves the backing storage and remaps the endpoint pages as necessary. However, during this time the server process is suspended, neither sending nor receiving additional messages. Messages arriving for nonresident endpoints and for endpoints being relocated are NACKed. This would result in a significant performance drop when interface end-

Related work

We can categorize recent communication systems by their support for the virtualization of network interfaces and communication resources and their positions on multiprogramming and error handling.

GAM, PM, and FM use message-based APIs with little to no support for multiprogramming. GAM is the canonical fast active message layer. PM and FM add support for gang-scheduling of parallel programs. These systems are driven primarily by the needs of SPMD parallel computing, such as support for MPI (message-passing interface, a portable interface for parallel applications) and portability to massively parallel processors. FM handles receive buffer overruns but ignores other types of network errors. None of these systems have explicit error models, which hinders the implementation of highly available and nonscientific applications.

Shrimp, U-Net, and Hamlyn are closer to our system. These systems provide direct, protected access to network interfaces using techniques similar to those found in application device channels.[11] The Shrimp project, which uses a virtual memory-mapped communication model, has run multiple applications and has preliminary multiprogramming results. U-Net and U-Net/MM can support multiprogramming. Hamlyn presented a vision of sender-based communication that should have been able to support multiprogramming, but demonstrated results using only ping-pong type benchmarks.

The most important distinction between previous work and our own lies in the virtualization of network interfaces and communication resources. In Shrimp, the level of indirection used to couple virtual memory to communication effectively virtualizes the network. U-Net provides virtualized interfaces but leaves routing, buffer management, reliable message delivery, and other protocol issues to higher layers. Hamlyn allows a process to map contiguous regions of NIC-addressable host memory into its address space. These "messages areas" afford a level of indirection that allows the system to virtualize the network interface. The position taken on virtualization has direct impact on the error model. In the event of an error, Shrimp and Hamlyn deliver signals to processes. U-Net delegates responsibility for providing adequate buffer resources and conditioning traffic to higher level protocols, and drops packets when resources are unavailable.

point frames become overcommitted.

To extend this scenario and to avoid the pitfalls of blocking, the server spawns a separate thread (and Solaris LWP) per client endpoint. Each server thread waits on a binary semaphore posted by the communication subsystem upon a message arrival. That causes an endpoint receive queue to become nonempty, that is, change from empty to containing messages. Additional messages may be delivered to the endpoint while the server thread is scheduled. Once running, the server thread disables further message arrival events and processes a batch of requests before re-enabling arrival events and again waiting on the semaphore. Apart from being a natural way to write the server, this approach allows a large number of server threads to be suspended, pending resolution of their endpoint page faults. Server threads with resident endpoints remain runnable and actively send and receive messages.

The results show that event mechanisms and thread overheads degrade peak server message rates by 15%, to 53,488 messages per second. While variation in average per-client message rates across the 5-minute sampling interval remains small, the variation in message rates between clients increases with load. Some clients' rates become 40% higher than average, while others are 36% lower than average. A finer-grain time series analysis (not shown) of client communication rates reveals the expected behavior: clients with burst messages from resident server endpoints at the rates shown in Figure 10. Others wait until both their endpoints become resident and the appropriate server thread is scheduled.

BRINGING DIRECT AND PROTECTED NETWORK multiprogramming into mainstream cluster computing requires innovations in three key areas: application programming interfaces, network virtualization systems, and lightweight communication protocols for high-speed interconnects.

The AM-II API extends traditional active messages with support for client-server computing and facilitates the construction of parallel clients and distributed servers. Our virtual network segment driver enables a large number of arbitrary sequential and parallel applications to access network interface resources directly in a concurrent but fully protected manner. The NIC-to-NIC communication protocols provide reliable and at-most-once message delivery between communication endpoints. The NIC-to-NIC protocols perform well as the number of endpoints and the number of hosts in the cluster are scaled.

The flexibility afforded by the underlying protocols enables a diverse set of timely research efforts. Other Berkeley researchers are actively using this system to investigate implicit techniques for the coscheduling of communicating processes,[12] an essential part of high-performance communications in multiprogrammed clusters of uni- and multiprocessor servers. Other researchers are extending the active message protocols described here for clusters of symmetric multiprocessors,[13] using so-called multiprotocol techniques and multiple network interfaces per machine. The impact of packet-switched networks with more buffering and without the link-to-link flow control of Myrinet, such as gigabit Ethernets, on cluster communication protocols is an open question. We are eager to examine the extent to which our existing protocol mechanisms and policies apply in these new regimes.

Acknowledgments

ARPA grant F30602-95-C-0014, Sun Microsystems, the California State Micro Program, David E. Culler's NSF Presidential Faculty Fellowship CCR-9253705, NSF Infrastructure grant CDA-8722788, an NSF Graduate Research Fellowship, and a National Semiconductor Corporation Graduate Research Fellowship supported this research. We thank Rich Martin and Remzi Arpaci-Dusseau for providing valuable feedback on earlier versions of this article and Eric Anderson for discussions on specialization and improvements to the API error model. Bob Felderman provided valuable information on Myrinet at all hours of the night. We thank Alec Woo for his debugging efforts, as well as his part of the software to Pentium-based platforms. We especially thank Andrea Arpaci-Dusseau for comments and suggestions for improving the article.

References

1. M. Blumrich et al., "Virtual-Memory Mapped Network Interfaces," *IEEE Micro,* Feb. 1995, pp. 21-28.
2. D. Culler et al., "Assessing Fast Network Interfaces," *IEEE Micro*, Feb. 1996, pp. 35-43.
3. S. Pakin, Karacheti, and A. Chien, "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors," *IEEE Parallel and Distributed Technology,* Vol. 5, No. 2, Apr.-June 1997.
4. S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proc. Supercomputing 95,* IEEE Computer Society, Los Alamitos, Calif., Dec. 1995.
5. H. Tezuka, A. Hori, and Y. Ishikawa*, PM: A High-Performance Communication Library for Multi-User Parallel Environments,* RWC, Tsukuba, Japan, Tech. Report TR-96015, 1996.
6. L. Prylli and B. Tourancheau, *Protocol Design for High Performance Networking: A Myrinet Experience,* Laboratoire de l'Informatique du Parallelisme, Lyon, France, Research Report No. 97-22, July 1997.
7. A. Mainwaring and D. Culler, *Active Message Application Programming Interface and Communication Subsystem Organization,* Tech. Report CSD-96-918, Univ. of California, Berkeley, Oct. 1996.
8. N. Boden et al., "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro,* Feb. 1995, pp. 29-36.
9. W. Leland et al., "On the Self-Similar Nature of Ethernet Traffic (Extended Version)," *IEEE/ ACM Trans. Networking,* Vol. 2, No. 1, Feb. 1994, pp. 1-15.
10. A. Mainwaring et al., "System Area Network Mapping," *Proc. Ninth ACM Symp. Parallel Algorithms and Architectures,* June 1997, pp. 116-126.
11. P. Druschel, L. Peterson, and B. Davie, "Experiences with a High-Speed Network Adaptor: A Software Perspective," *Proc. ACM SIGCOMM94 Symp.,* Aug. 1994, pp. 2-13.

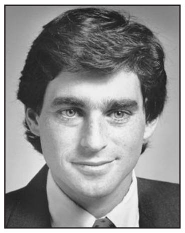
12. A. Dusseau, R. Arpaci, and D. Culler, "Effective Distributed Scheduling of Parallel Workloads," *Proc. 1996 ACM Sigmetrics Int'l Conf. Measurement and Modeling of Computer Systems*, Assoc. of Computing Machinery, N.Y., May 1996.
13. S. Lumetta, A. Mainwaring, and D. Culler, "Multi-Protocol Active Messages on a Cluster of SMPs," *Proc. Supercomputing 97,* IEEE CS Press, 1997.

Brent N. Chun is a PhD student in computer science at the University of California at Berkeley. His research interests include computer networks and computer-supported collaborative work.

Chun received his MS degree in computer science from the University of California at Berkeley and his BS degree in electrical engineering from the University of Southern California in Los Angeles. He is a member of the ACM and Usenix.

Alan M. Mainwaring is a PhD candidate in computer science at the University of California at Berkeley. His research interests include software and abstractions for high-performance cluster interconnection networks, the construction of communication protocols for parallel and distributed applications, and the integration of high-performance communications with virtual memory and process scheduling systems. He previously spent five years as a member of technical staff at Thinking Machines Corporation, where he worked in the Advanced Programming Models Group, as well as with the runtime systems and languages groups.

Mainwaring received his BA degree in applied mathematics from the University of Rochester and his MS degree in computer science from the University of California at Berkeley.



David E. Culler is a professor of computer science at the University of California at Berkeley. His research addresses parallel computer architecture, parallel programming languages, and high-performance communication structures. He is known for his work on Networks of Workstations (NOW), Active Messages, Split-C, the Threaded Abstract Machine (TAM), and dataflow systems.

Culler received his PhD degree from the Massachusetts Institute of Technology. He received the NSF Presidential Young Investigator award in 1990 and the Presidential Faculty Fellowship in 1992.

Direct comments about this article to Brent N. Chun, University of California at Berkeley, Computer Science Division, Berkeley, CA 94720-1776; bnc@cs.berkeley.edu.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 168 Medium 169 High 170