Virtual Network Transport Protocols for Myrinet

Brent N. Chun, Alan M. Mainwaring, and David E. Culler

University of California at Berkeley
Computer Science Division
{bnc,alanm,culler@cs.berkeley.edu}

Abstract

This paper describes a protocol for a general-purpose cluster communication system that supports multiprogramming with virtual networks, direct and protected network access, reliable message delivery using message timeouts and retransmissions, a powerful return-to-send error model for applications, and automatic network mapping. The protocols use simple, low-cost mechanisms that exploit properties of our interconnect without limiting flexibility, usability or robustness. We have implemented the protocols in an active message communication system that runs a network of 100^+ Sun UltraSPARC workstations interconnected with 40 Myrinet switches. A progression of microbenchmarks demonstrate good performance – 42 microsecond round-trip times and 31 MB/s node to node bandwidth – as well as scalability under heavy load and graceful performance degradation in the presence of high contention.

1 Introduction

With microsecond switch latencies, gigabytes per second of scalable bandwidth, and low transmission error rates, cluster interconnection networks such as Myrinet [BCF+95] 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 produced a number of portable communication interfaces and implementations. For example, Generic Active Messages (GAM) [CLM+95], Illinois Fast Messages (FM) [PKC97, PLC95], the Real World Computing Partnerships's PM [THI96], and BIP [PT97] provide fast communication layers. By constraining and specializing communication layers for an environment, for example by only supporting single-program multiple-data parallel programs or by assuming a perfect, reliable network, these systems achieved high-performance, oftentimes on 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. The communication interfaces should support client-server, parallel and distributed applications in a multi-threaded and multi-programmed environment. Implementations should use process scheduling as an optimization technique rather than as a requirement for correctness. In a time-shared system, implementations should provide protection and the direct application access to network resources that is critical for high-performance. Finally, the protocols that enable these systems should provide reliable message delivery, automatically handle non-catastrophic network errors, and support automatic network management tasks such as topology acquisition and route distribution.

Section 2 presents a core set of requirements for our cluster protocol and states our specific assumptions. Section 3 presents an overview of our system

This research is supported in part by ARPA grant F30602-95-C-0014, generous contributions from Sun Microsystems, Inc., the California State Micro Program, Professor David E. Culler's NSF Presidential Faculty Fellowship CCR-9253705, NSF Infrastructure Grant CDA-8722788, a NSF Graduate Research Fellowship, and a National Seminconductor Corportation Graduate Research Fellowship.

architecture and briefly describes the four layers of our communication system. Then in section 4, we examine the issues and design decisions for our protocols, realized in our system in network interface card (NIC) firmware. Section 5 analyses performance results for several challenging microbenchmarks. We finish with related work and conclusions.

2 Requirements

Our cluster protocol must support multiprogramming, direct access to the network for all applications, protection from errant programs in the system, reliable message delivery with respect to 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. 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 – but catastrophic problems require handling at the 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, link bandwidth of a gigabit or more and are relatively error-free. Second, 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, probing for network operating parameters (e.g., as by TCP slow-start), and guarantees that the network fabric and the protocols used between its network interfaces are identical. This does not preclude use of heterogeneous hosts at the endpoints, such as hosts with different endianness. Lastly, the maximum number of nodes attached to the cluster interconnect is limited. This enables trading memory resources proportional to the number of network interfaces (NICs) in exchange for reduced computational costs on critical code paths. (In our system, we limit 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.)

3 Architecture

Our system has four layers: (1) an active message applications programming interface, (2), a virtual network system that abstracts network interfaces and communication resources, (3), firmware executing on an embedded processor on the network interface, and (4), processor and interconnection hardware. This sections presents a brief overview of each layer and highlights important properties relevant for the NIC-to-NIC transport protocols described thoroughly in Section 4.

3.1 AM-II API

The Active Messages 2.0 (AM-II) [MC96] provides applications with the interface to the communications system. It allows an arbitrary number of applications to create multiple communications *endpoints* used to send and to receive messages using a procedural interface to active messages primitives. Three message types are supported: 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. Medium and bulk message data can 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 is passed a pointer to the storage and can operate directly on the data. Bulk message data are deposited into per-endpoint virtual memory regions. These regions can be located anywhere in a receiver's address space. Receivers identify these regions with a base address and length. Applications can set and clear event masks to control whether or not 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, multi-threaded applications have the option of processing messages in an event-driven way.

Isolating message traffic for unrelated applications is done using per-endpoint message tags specified by the application. Each outgoing message

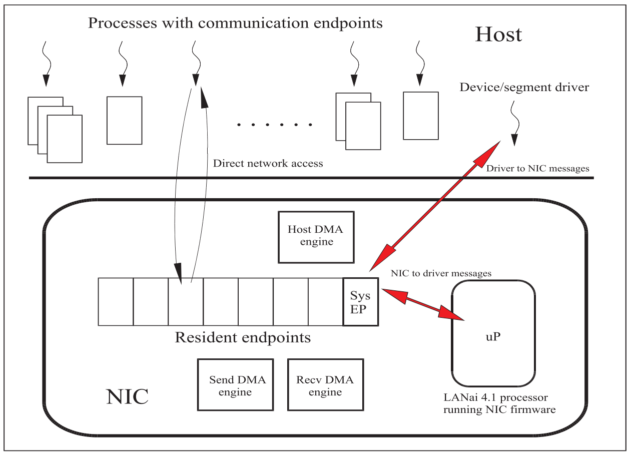
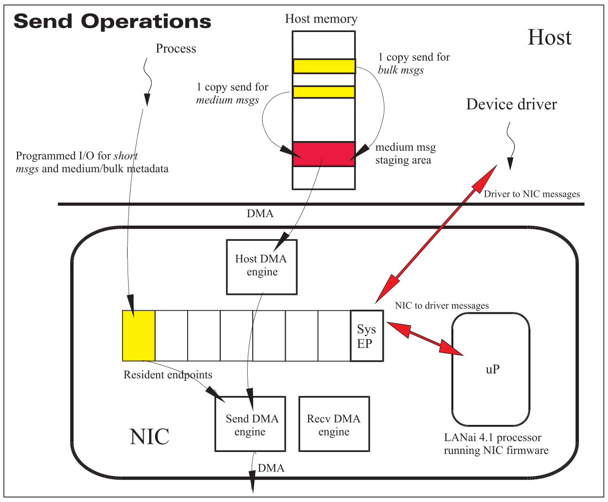


Figure 1: **Processor/NIC node.**



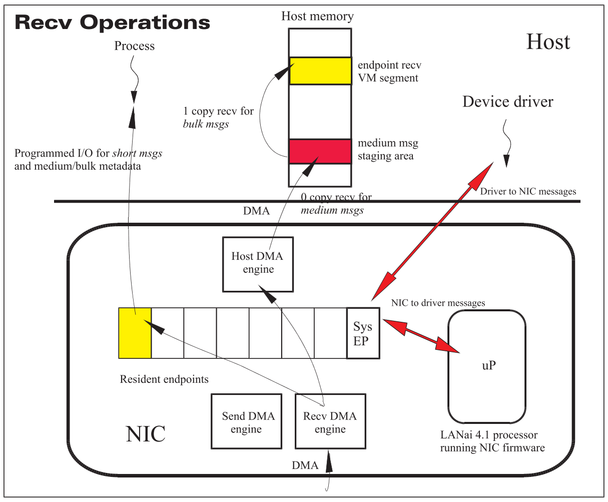


Figure 2: **Data paths for sending and receiving short, medium, and bulk active messages.** *Short messages are transferred using programmed I/O directly on endpoints in NIC memory. Medium messages are sent and received using per-endpoint medium message staging areas in the pageable kernel heap that 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.*

contains a message tag for its destination endpoint. Messages are delivered if the tag in the message matches the tag of the destination endpoint. The AM-II API provides an integrated return-to-sender error model for both application-level errors, such as non-matching tags, and for catastrophic network failures, such as losing connectivity with a remote endpoints. Any message that cannot be delivered to its destination is returned 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.

3.2 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 1 illustrates this idea. Applications create one or more communications 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 or 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, they leverage the inter-process 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 on our Myrinet system, it uses NIC memory as a cache of active endpoints and pages endpoints on and off the NIC on-demand, much like virtual memory systems do with memory pages and frames. Analogous to pagefaults, endpoint faults can occur when either an application writes a message into a non-resident endpoint, or a message arrives for a non-resident endpoint. Endpoint faults also occur whenever messages (sent or received) reference host memory resources – medium message staging area, arbitrary user-specified virtual memory regions for sending messages, or endpoint virtual memory segment for receiving messages – that are not pinned, or for which there are no current DMA mappings. Network interface virtualization, including endpoint cache management and the paging of endpoints, is handled using a custom virtual network segment driver.

3.3 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 detecting and recovering from errors. Details on the NIC protocols are given in Section 4.

The protocols implemented in firmware determine the structure of an endpoint. 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 resident endpoints memory using programmed I/O. Medium and bulk messages use programmed I/O 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.)

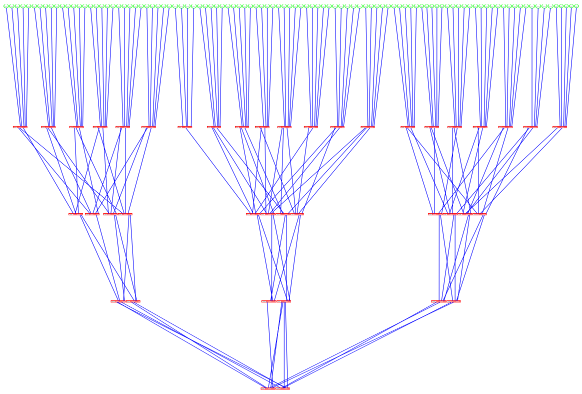


Figure 3: **Berkeley NOW network topology as discovered by the mapper.** *The network mapping daemons periodically explore and discover the network's current topology, in this case a fat tree-like network with 40 Myrinet switches. The three sub-clusters are currently connected using through two switches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

3.4 Hardware

The system hardware consists of 100^+ 167-Mhz Sun UltraSPARC workstations interconnected with Myrinet (Figure 3) [BCF+95], a high-speed local area network with cut-through routing and link-level back-pressure. The network uses 40 8-port crossbar switches with 160 MB/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 KB of SRAM, a single host SBus DMA engine but independent network send and receive DMA engines.

4 NIC Protocols

This section describes a set of lightweight NIC protocols that support the AM-II communications API and virtual networks and also provide basic transport protocol functionality. The protocols are implemented as firmware running on Myricom LANai4.1 cards.

4.1 Endpoint Scheduler

Because our system supports both direct network access and multiprogramming, the NIC has a new task of endpoint scheduling, i.e., sending messages from the current set of cached endpoints. This situation is different from that of traditional protocol stacks, such as TCP/IP, where 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.

The endpoint scheduling policy chooses 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 fair and is starvation free. If all endpoints always have messages waiting to send, this algorithm might be satisfactory. However, if application communication is bursty [LTW+94], spending equal time on each resident endpoint is not optimal. Better strategies exist which minimize the use of critical NIC resources examining empty queues.

The endpoint scheduling policy must balance optimizing the throughput and responsiveness of a particular endpoint 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 should the host enqueue additional messages. Loitering also allows firmware to cache state, such as packet headers and constants while sending messages from an endpoint, lowering per-packet overheads. While larger k's result in better performance during bursts, too large a k degrades system responsiveness with multiple active endpoints. Empirically, we have chosen a k of 8.

4.2 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 in order to fill the communication pipe between a sender and a receiver. On the other, it should 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 in order to send more data. Assuming the destination process is scheduled and attentive to the network, given a bandwidth B and a round trip time RTT, this requires allowing at least $B \cdot RTT$ bytes of outstanding data.

Our system addresses flow control at three levels: (1) user-level active message credits for each endpoint, (2) NIC-level stop-and-wait flow control over multiple, independent logical channels, and (3) network back-pressure.

4.2.1 User-level Credits

The user-level credits rely upon 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 K_{user} large enough, endpoint-to-endpoint communication proceeds at the maximum rate. To prevent receive buffer overflow, endpoint request receive queues are large enough to accommodate several senders transmitting at full speed. Because senders have at most a small number, K_{user}, 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 KB packets the bandwidth delay product is $31MB/s \cdot 349us = 11345$ bytes – less than two 8 KB messages. For short packets the bandwidth delay product is $62,578msgs/s \cdot 42us = 2.63$ 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.

4.2.2 Stop-and-wait Over Logical Channels

To keep the communication pipe full in steady state, the NIC, which is responsible for timeout/retry, must allow at least K_{user} outstanding messages since 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 is necessary in order to prevent deadlock.

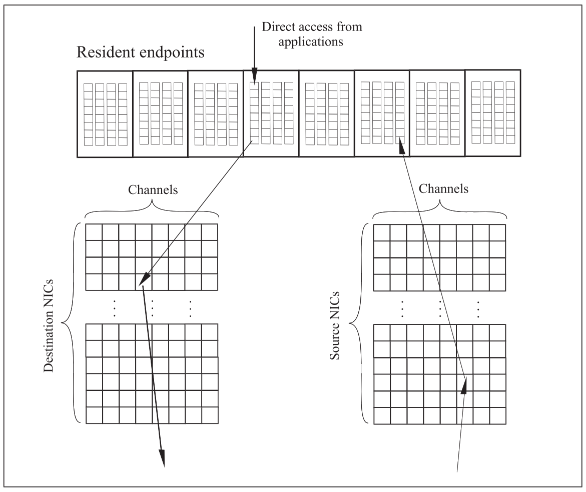


Figure 4: **NIC channel tables.** *The NIC channel tables provide easy-access to NIC flow control, timeout/retry, and error detection information. The NIC uses stop-and-wait flow control on each channel and manages communication state information in channel table entries. In the send table (left), each entry includes timeout/retry information (packet timestamp, 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 or not. 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 is reserved for replies. This is necessary in order to prevent deadlock.*

Two simple data structures manage NIC-to-NIC flow control information. These data structures also record timeout/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 $O(\#NICs \cdot \#channels)$ entries are sufficient.

4.2.3 Link-level Back-pressure

Link-level back-pressure ensures that under 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 utilization under load – allowing packets to block and to consume link and switch resources – for simplicity. Section 5 shows that this hybrid scheme performs very well.

4.2.4 Receiver Buffering

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

We provide request and response receive queues, each with 16 entries ($4 \cdot K_{user}$), for each endpoint. These are sufficient to absorb load from up to four senders transmitting at their maximum rates. When buffer overflow occurs, the protocol drops packets and NACKs senders. The firmware automatically retransmits such messages. An important consequence of sizing request and reply queues to be $4 \cdot K_{user}$ entries deep is that our virtual network segment driver can use a single virtual memory page per endpoint, simplifying its memory management activities.

4.3 Timeout and Retry

To guarantee at-most-once delivery semantics and address transient hardware errors (e.g. CRC errors and truncated packets), a communication system must perform timeout and retransmission of packets. The timeout/retry algorithm determines how packet retransmissions 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, and all send table entries are periodically scanned for packets to retransmit. The per-packet timeout/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 an acknowledgment reception to be negligible. Depending on the granularity of the timeout 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. Section 5 shows that this hybrid scheme has near zero amortized cost for workloads where packets are not retransmitted.

Our transport protocol implements timeout and retry with positive acknowledgments in the interface firmware. This provides efficient acknowledgements and minimizes expensive SBus transactions. (We currently do not perform the obvious piggybacking of ACKs and NACKs on active message reply messages). Channel management tables store timeout and transmission state. 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 id 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 which updates its sequencing information and frees the channel for use by a new packet.

By using a simple and easily-addressable data structures, each with $O(\#NICs \cdot \#channels)$ entries, scheduling and deleting packet retransmission events take constant time. For retransmissions, though, the NIC perform $O(\#NICs \cdot \#channels)$ 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 acknowledgement decrements the counter. These counts reduce retransmission overheads to be proportional to the total number of network interfaces.

4.3.1 Unavailable Resources

Virtual networks introduce new issues for at-most-once delivery semantics in the presence of hardware errors. Because endpoints may be non-resident or may not have DMA resources (e.g. 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 timeout/retry mechanism used to deal with hardware errors. Like hardware errors, resource unavailability is expected to be infrequent. Therefore, using the same timeout/retry mechanism, which may use coarse-grain timeouts, should add very little overhead.

4.4 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 the 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 timeouts and retransmissions for recovery. After 255 retransmission, for which no ACKs or NACKs were received, the protocol declares a message as undeliverable and returns it to the AM-II layer. (Timeout/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.

4.4.1 Transient Errors

Positive acknowledgement with timeout and retransmission ensures the delivery of packets with valid routes. Not only can data packets be dropped or corrupted but protocol control messages can be as well. To ensure that data packets are never delivered more than once to a destination despite retransmissions, they are tagged with sequence numbers and timestamps. With a maximum of 2^k outstanding messages, detecting duplicates requires 2^{k+1} sequence numbers. For our alternating-bit protocol on independent logical channels, $k = 0$.

4.4.2 Return-to-Sender

The NIC determines that destination endpoints are unreachable by relying upon its timeout and retransmission mechanisms. If after 255 retries (i.e., 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 marks the sequence number of the channel as uninitialized and returns the original message back to user-level via the endpoint's reply receive queue. The application handles undeliverable message 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.

4.4.3 Network Management Errors

The system uses privileged mapper daemons, one for each interface on each node of the system, to probe and to 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 [MCS+97]. 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 online 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: (1) sending and receiving probe packets with application-specified source-based routes to discover links, switches, and hosts and (2) reading and writing entries in NIC routing tables. These special functions can be performed using privileged endpoints available to privileged processes.

5 Performance Results

This section presents performance measurements and analysis. The first microbenchmarks characterize the system using the LogP communication model and 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 endpoint contention. It concludes with an examination of system performance as the number of active virtual networks increases. All programs were run on the Berkeley Network of Workstations system in a stand-alone environment. Topology acquisition and routing daemons were disabled, eliminating background communication activity normally present.

5.1 LogP Characterization

The LogP communications model uses four parameters to characterize the performance of communication layers. This parameterization enables the fair comparison of different communication layers. The microbenchmarks of [CLM+95] automatically derive the model parameters of L (latency), overhead (o) and gap (g). Each parameter has a simple interpretation. The number of processors (P) is given. The overhead has two components, the sending overhead (O_s) and the receiving overhead (O_r). These measure the host processor time spent writing a message to and reading a message from an endpoint, respectively. The gap measures the time through the rate-limiting stage in the system and the latency is the remaining time unaccounted for by the overheads.

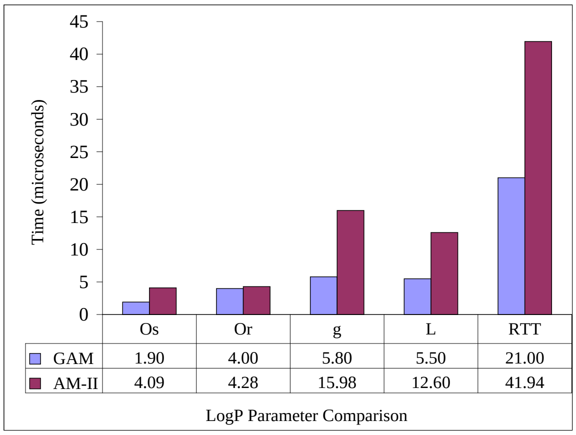


Figure 5: **Performance characterization using the LogP model.** *LogP parameters for two active message systems on identical hardware: AM-II, our new general-purpose active message system with virtual networks and the return-to-sender error model, and GAM, an earlier active message system for SPMD parallel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

Figure 5 shows the LogP parameters for AM-II and GAM. It shows the measured AM-II round-trip time of 41.94 microseconds as compared with GAM's round-trip time of 21 microseconds. Of the 17.37 microsecond one-way time, the system spends 5 microseconds writing the message into the sender's endpoint and 3.3 microseconds reading the messages from the receiver's endpoint. The two network interfaces spend 13.82 microseconds transmitting the data message as well as transmitting its acknowledgement. For AM-II, the gap 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 microbenchmark 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.

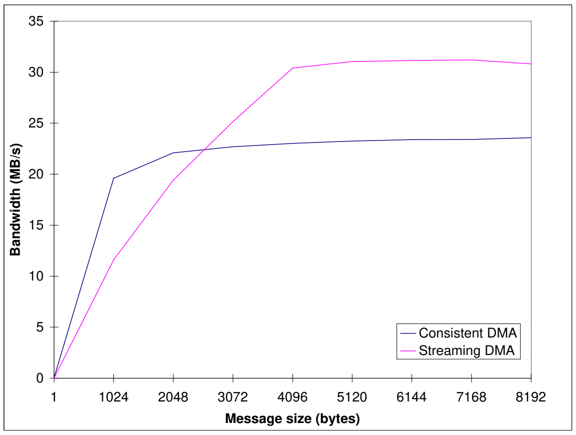


Figure 6: **Sending bandwidth as a function of message size in bytes**. *Consistent host-to-NIC DMA operations across the SBus have higher performance for small transfers. Streaming transfers obtain higher performance once the data transfer times swamp the cost of flushing a hardware stream buffer in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

5.2 Contention-Free Performance

Figure 6 shows the endpoint-to-endpoint bandwidth between two machines. Because the NIC can only DMA messages between the network and its local memory, a store-and-forward delay is introduced for large messages moving 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 MB/s with 4KB messages. (With GAM, pipelining of DMA operations to receive messages from the network with DMA operations to move buffers to host memory increased bulk transfer performance to 38 MB/s.)

Perm	Avg BW	Agg BW	Avg RTT
Cshift	25.42 MB/s	2.33 GB/s	67.8 us
Neighbor	30.97 MB/s	2.85 GB/s	47.5 us
Bisection	5.65 MB/s	0.52 GB/s	50.8 us

Table 1: This table shows aggregate bandwidth and average round trip times for 92 nodes with different message permutations. In the cshift permutation, each node sends requests to its right neighbor and replies to requests received from its left neighbor. With neighbor, adjacent nodes perform pairwise exchanges. In bisection, pairs of nodes separated by the network bisection perform pairwise exchanges. Bandwidth measurements used medium messages, whereas RTT measurements used 4-word active messages. (More recent work has considerably improved upon this performance. See `http://now.cs.berkeley.edu`)

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 machines of the NOW. Each column shows that the bandwidth scales as the system reaches a non-trivial 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. Refer to figure 3 to see the network topology and the small number of bisection cables.

5.3 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, the host with the common destination is referred to as the server and all other hosts are referred to as the clients.

Figure 7 shows the aggregate message rate of the server (top line) as the number of clients sending 4-word requests to it and receiving 4-word response messages increases. Additionally it shows

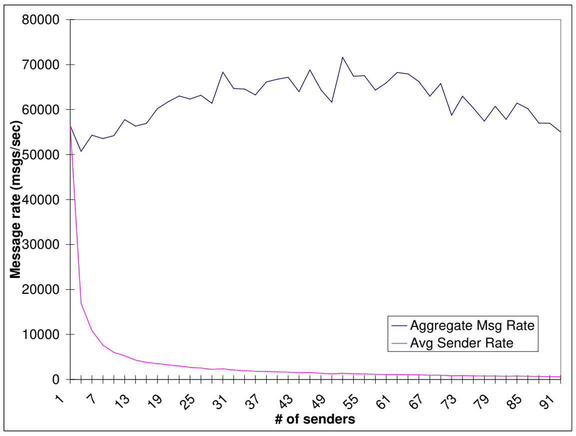


Figure 7: Active Message rates with destination endpoint contention within a single virtual network.

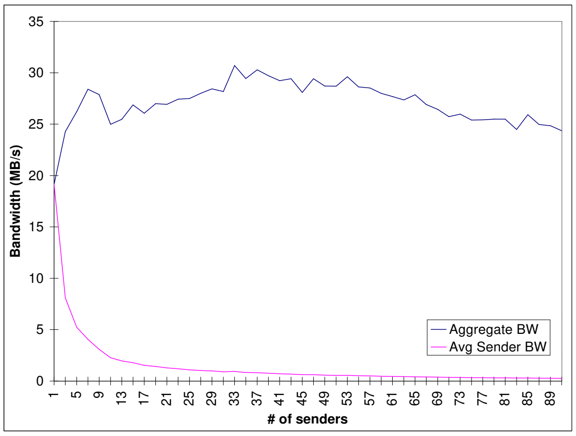


Figure 8: Delivered bandwidths with destination endpoint contention within a single virtual network.

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 1KB 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 acknowledgements for reply messages encountering congestion (namely other requests also destined for the server). The variation in per-sender rates and bandwidths are 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.

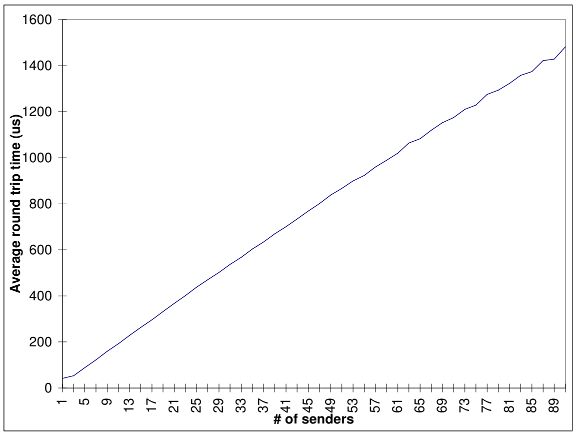


Figure 9: Round-trip times with destination endpoint contention within a single virtual network.

5.4 Multiple Virtual Networks

We can extend the previous benchmark to stress virtual networks. First, by increasing the number of server endpoints up to the maximum of 7 that can be cached in the interface memory, and then continuing 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.

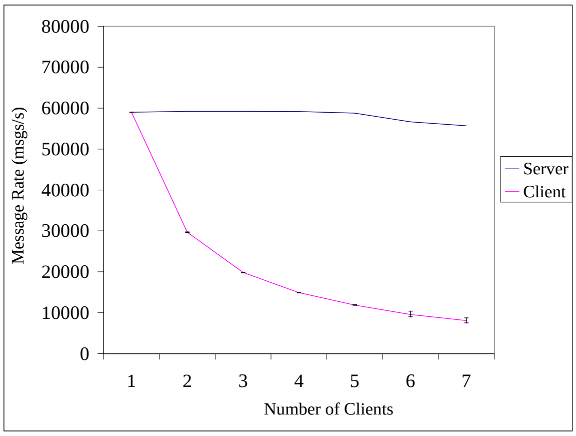


Figure 10: Aggregate server and per-client message rates with small numbers of virtual networks.

Figure 10 shows the average server message rate and per-client message rates (with error bars) over a five minute interval. The number of clients continuously making requests of the 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 microseconds. The per-client message rates with within 16% of their ideal fair share of $1/N$th of the server's throughput. Steady server performance and the graceful response of the system to increasing load demonstrate the effective operation of the flow-control, endpoint scheduling, and multiplexing mechanisms throughout the system.

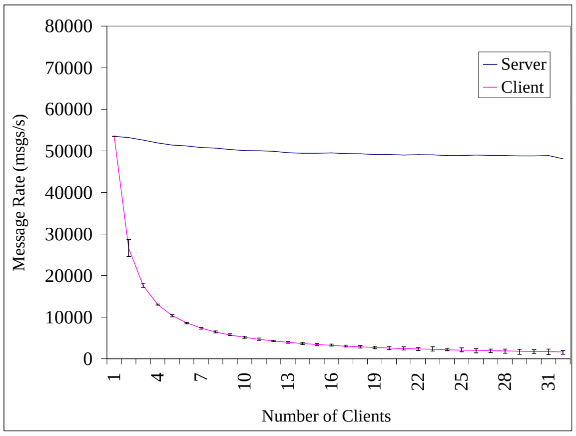


Figure 11: Aggregate server and per-client message rates with large numbers of virtual networks.

Figure 11 extends the scenario shown in Figure 10. Previously, the server host was a single-threaded process, polling its endpoints in a 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 in an on-demand fashion. When the server attempts to write a reply message into a non-resident endpoint (or when a request arrives for a non-resident endpoint), a pagefault occurs and the virtual network driver moves the backing storage and re-mapped the endpoint pages as necessary. However, during this time the server process is suspended and thus it neither sends nor receives additional messages. Messages arriving for non-resident endpoints and for endpoints being relocated are NACKed. This would result in a significant performance drop when interface endpoint 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 non-empty. 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 pagefaults while 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 five minute sampling interval remains small, the variation in message rates between clients increases with load, with some clients rates 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 resident server endpoints burst messages at rates as shown in Figure 10 while others wait until both their endpoints become resident and the appropriate server thread is scheduled.

6 Related Work

Recent communication systems can be categorized by their support for 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 and portability to MPPs. 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 non-scientific applications.

SHRIMP, U-Net [vEBB+95] 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 [DPD94]. The SHRIMP project, which uses 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 style 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.

7 Conclusions

Bringing direct, protected communication into mainstream computing requires a general-purpose and robust communication protocol. The AM-II API and virtual networks abstraction extends traditional active messages with reliable message delivery, a simple yet powerful error model and supports use in arbitrary sequential and parallel programs. We have presented the design of the NIC-to-NIC transport protocols required by this more general system. For our Myrinet implementation, we have measured the costs of the generality relative to GAM, a minimal active message layer, on the same hardware. In particular, we have explored the costs associated with endpoint scheduling, flow control, timer management, reliable message delivery and error handling.

The implementation achieves end-to-end latencies of 21 microseconds for short active messages with a peak bandwidth of 31 MB/s. These numbers represent twice the end to end latency and 77% of the bandwidth provided by GAM. The cost of reliable message delivery makes the most significant contribution above basic communication costs. Using additional benchmarks, we have demonstrated that the protocols provide robust performance and graceful degradation for the virtual networks abstraction, even when physical network interface resources are overcommitted by factors of 12 or more. These benchmarks demonstrate the feasibility of truly virtualizing network interfaces and their resources and show the importance of supporting multi-threaded applications.

The NIC-to-NIC protocols discussed in this paper perform well, and, enable a diverse set of timely research efforts. Other researchers at Berkeley are actively using this system to investigate explicit and implicit techniques for the co-scheduling of communicating processes [DAC96], an essential part of high-performance communication in multiprogrammed clusters of uni and multiprocessor servers. Related work on clusters of SMPs [LMC97] investigates the use of multiple network interfaces and multiprotocol active message layers. The impact of packet switched networks, such as gigabit ethernet, on cluster interconnect protocols is an open question. We are eager to examine the extent to which our existing protocol mechanisms and policies apply in this new regime.

8 Acknowledgments

We thank Rich Martin and Remzi Arpaci-Dusseau for providing valuable feedback on earlier versions of this paper. 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 port of the software to Pentium-based platforms, and Eric Anderson for discussions on specializations, and improvements to the API error model. We especially thank Andrea Arpaci-Dusseau for comments and suggestions for improving this paper.

References

[BDF+95] M. Blumrich, C. Dubnicki, E. Felton, and K. Li. Virtual-Memory Mapped Network Interfaces. In *IEEE Micro Magazine*, Feb. 1995, pp. 21-28.

[BCF+95] N. Boden, D. Cohen, R. Felderman, A. Kulawik. C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit per Second Local Area Network. In *IEEE Micro Magazine*, February 1995, pp. 29-36.

[CLM+95] D. Culler, L. Liu, R. Martin, and C. Yoshikawa. Assessing Fast Network Interfaces. In *IEEE Micro Magazine*, Feb. 1996, pp. 35-43.

[DAC96] A. Dusseau, R. Arpaci, D. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, 1996.

[DPD94] P. Druschel, L. Peterson, and B. Davie. Experiences with a High-speed Network Adaptor: A Software Perspective. In *Proceedings of ACM SIGCOMM '94 Symposium*, August, 1994, pp. 2-13.

[LMC97] S. Lumetta, A. Mainwaring, D. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of SC97*, November 1997.

[LTW+94] W. Leland, M. Taqqu. W. Willinger, and D. Wilson. On the Self-Similar Nature of Ethernet Traffic (Extended Version). IEEE/ACM Transactions on Networking, 2(1):1-15, Feb. 1994.

[MC96] A. Mainwaring and D. Culler. Active Message Application Programming Interface and Communication Subsystem Organization. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[MCS+97] A. Mainwaring, B. Chun, S. Schleimer, and D. Wilkerson. System Area Network Mapping. *Proceedings of 9th ACM Symposium on Parallel Algorithms and Architectures*, June 1997, pp. 116-126.

[PKC97] 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.

[PLC95] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings on Supercomputing '95*, December 1995.

[PT97] L. Prylli and Bernard 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.

[THI96] H. Tezuka, A. Hori, and Y. Ishikawa, PM: A High-Performance Communication Library for Multi-user Parallel Environments, RWC, Tsukuba, Japan, Technical Report TR-96015, 1996,

[vEBB+95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.