

User-Level Network Interface Protocols



User-level communication architectures remove the operating system from the critical communication path, but designing protocols for these architectures is not trivial. The authors survey six design issues using examples from 11 communication systems, all of which have been implemented on a Myrinet network.

Raoul A.F.
Bhoedjang
Tim Rühl
Henri E. Bal
Vrije
Universiteit
Amsterdam

Modern high-speed local area networks offer great potential for communication-intensive applications, but their performance is limited by the use of traditional communication protocols, such as TCP/IP. In most cases, these protocols require that all network access be through the operating system, which adds significant overhead to both the transmission path (typically a system call and data copy) and the receive path (typically an interrupt, a system call, and a data copy). To address this performance problem, several *user-level communication architectures* have been developed that remove the operating system from the critical communication path.^{1,2} In this article, we describe six important issues to consider in designing communication protocols for user-level architectures.

The issues we have selected focus on the performance and semantics of a communication system. The first, the *data transfer* mechanism a protocol uses, significantly affects latency and throughput. Direct memory access usually gives the best throughput in user-level communication, but to avoid unnecessary copying, sophisticated *address translation* mechanisms are needed, another important design issue. To make user-level protocols feasible in a multiprogramming environment, designers also need *protection* mechanisms that do not rely on operating system intervention. Network packets are received using some *control transfer* mechanism—either polling or interrupts, each of which has its advantages and drawbacks. The last issues we focus on are *reliability* and *multicast*, which in modern networks can be supported both on the host processor and the network interface.

To provide a basis for analyzing these issues, we present a simple network interface protocol for Myricom's Myrinet network.³ As the sidebar "The Myrinet Network" describes, Myrinet has a programmable network interface. Researchers can thus explore many protocol design options, and several groups have designed communication systems for Myrinet. In this article, we refer to 11 such systems. Table 1 lists the

characteristics of each. As the table shows, the systems differ significantly in how they resolve the design issues we describe, but all aim for high performance and provide a lean, low-level, and more or less generic communication facility.

A SIMPLE NETWORK INTERFACE PROTOCOL

Figure 1 shows the operation of our simple network interface protocol for Myrinet. Operation begins when a user process of the sending host—host A—invokes a simple `send` primitive to send a data packet. Because the protocol sends packets with a maximum payload of 256 bytes, users must fragment their data so that each fragment fits in a packet.

`Send` performs two actions. It first copies the user data to a packet buffer in a special staging area, the *DMA area*, in the host's memory (step 1). The host then writes a send request into a descriptor in network interface memory (step 2) and sets a flag to inform the network interface of this event. These descriptors, which are stored in a circular buffer, or *send ring*, contain the destination machine, the size of the packet's payload, and the packet's offset in the DMA area.

The network interface repeatedly polls the flag of the first empty descriptor in the send ring. As soon as the host sets this flag, the network interface reads the packet's offset in the descriptor and adds it to the address of the start of the DMA area, which yields the physical address of the packet (step 3). The network interface initiates a DMA transfer over the I/O bus to copy the packet's payload from host memory to its own memory (step 4). It then reads the destination machine in the descriptor and looks up the route for the packet in a routing table. Finally, it starts a second DMA to transmit the packet (step 5).

The data transfers in steps 1, 4, and 5 can be performed concurrently. For example, if the host sends a long, multipacket message, one packet's network DMA can be overlapped with the next packet's host-to-interface DMA. When the network interface detects that a given packet's network DMA has completed, it sets a flag to release the

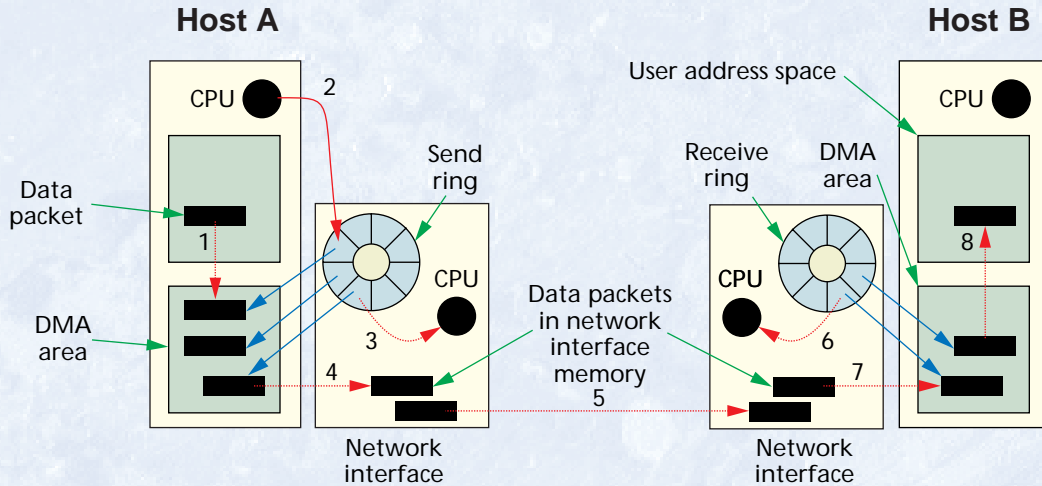


Figure 1. How the simple network interface protocol operates. Host A copies user data into the DMA area (step 1) and writes a packet descriptor to the send ring (step 2). The network interface reads the packet descriptor (step 3) and copies the packet through DMA to its memory (step 4). After the network transfer (step 5), the receiving network interface reads its receive ring to find an empty buffer in the DMA area of host B (step 6) and copies the packet using DMA to this area (step 7). Host B optionally copies data from the packet to a user buffer (step 8).

The Myrinet Network

Myrinet, developed by Myricom, is a switched, gigabit-per-second technology that uses variable-length packets. The packets are wormhole-routed through a network of highly reliable links and crossbar switches. Techniques like wormhole routing are normally found only in supercomputers.

Figure A illustrates the architecture of a node in a Myrinet cluster. Each machine (host) has a network interface card that contains a processor and some memory, which is used to store the interface's control program and data. The network interface connects to the host's I/O bus—a typical organization for commodity hardware.

Our host system has a 200-MHz Pentium Pro processor, a host bus speed of 66 MHz × 64 bits, and an I/O bus speed (PCI) of 33 MHz × 32 bits. All hosts run the BSD/OS (Version 3.0) operating system.

Myrinet requires that all packets be staged through network interface memory, both at the sending and the receiving side. It uses fast but expensive SRAM, so the memory is relatively small. Both the host and the network interface can use DMA to access data in each other's memory, but as the main article describes, DMA transfers suffer from a start-up overhead. The host can also access the network interface's memory using programmed I/O, which has no start-up costs, but results in high access times relative to host memory.

Like other modern networks, Myrinet has a programmable network interface processor. Although this gives protocol designers much flexibility, the network interface processor is much slower than the host CPU, so it can do only simple tasks. Adding just a few instructions to the critical path of a Myrinet control program noticeably increases end-to-end latency.

Our network interfaces have a 33-MHz LANai4.1 processor

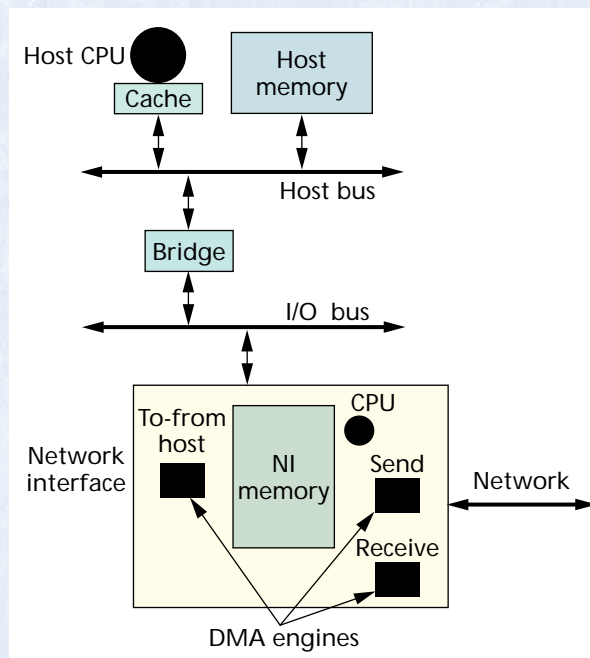


Figure A. The host and network interface architecture of Myricom's Myrinet.

(LANai is the processor's proprietary name), 1-Mbyte SRAM, a link speed of 2×1.28 Gbps, a to-from host DMA start-up cost of 5 LANai cycles, a send DMA start-up cost of 5 LANai cycles, and a receive DMA start-up cost of 4 LANai cycles.

Table 1. Characteristics of 11 communication systems built for Myrinet.

System	Data transfer (host-NI)	Translation	Protection	Control transfer	Reliability	Multicast support
AM-II ¹	PIO & DMA*	DMA areas	Yes	Polling + interrupts	Reliable, network interface: alternating bit, host: sliding window	No
FM ²	PIO	DMA area (recv)	No	Polling	Reliable, host-level credits	No
FM/MC ³	PIO	DMA area (recv)	No	Polling + interrupts	Reliable, unicast: host-level credits, multicast: network-interface-level credits	Yes (on network interface)
PM ⁴	DMA	Software TLB* on network interface	Yes (gang scheduling)	Polling	Reliable, ACK/NACK protocol on network interface	Yes (multiple sends)
VMMC ⁵	DMA	Software TLB on network interface	Yes	Polling + interrupts	Reliable, exploits hardware backpressure	No
VMMC-2 ⁶	DMA	UTLB* in kernel, cached on network interface	Yes	Polling + interrupts	Reliable	No
LFC ⁷	PIO	User translates	No	Polling + interrupts + watchdog	Reliable, unicast: network-interface-level credits, multicast: network-interface-level credits	Yes (on network interface)
Hamlyn ⁸	PIO & DMA	DMA areas	Yes	Polling + interrupts	Reliable, exploits hardware backpressure	No
Trapeze ⁹	DMA	DMA to page frames	No	Polling + interrupts	Unreliable	No
BIP ¹⁰	PIO & DMA	User translates	No	Polling	Reliable, rendezvous and backpressure	No
U-Net ¹¹	DMA	TLB on network interface (U-Net/MM)	Yes	Polling + interrupts	Unreliable	No

* PIO is programmed I/O, DMA is direct memory access, TLB is translation look-aside buffer, and UTLB is user-managed TLB.

1. B. Chun, A. Mainwaring, and D. Culler, "Virtual Network Transport Protocols for Myrinet," *IEEE Micro*, Jan. 1998, pp. 53-63.
2. S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proc. Supercomputing '95*, ACM Press, New York, 1995.
3. H.E. Bal et al., "Performance of a High-Level Parallel Language on a High-Speed Network," *J. Parallel and Distributed Computing*, Feb. 1997, pp. 49-64.
4. H. Tezuka et al., "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication," *Proc. Int'l Parallel Processing Symp.*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 308-314.
5. C. Dubnicki et al., "Design and Implementation of Virtual Memory-Mapped Communication on Myrinet," *Proc. Int'l Parallel Processing Symp.*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 388-396.

6. C. Dubnicki et al., "Myrinet Communication," *IEEE Micro*, Jan. 1998, pp. 50-52.
7. R.A.F. Bhoedjang, T. Rühl, and H.E. Bal, "Efficient Multicast on Myrinet Using Link-Level Flow Control," *Proc. Int'l Conf. Parallel Processing*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 381-390.
8. G. Buzzard et al., "An Implementation of the Hamlyn Sender-Managed Interface Architecture," *Proc. Symp. Operating Systems Design and Implementation*, Usenix Assoc., Berkeley, Calif., 1996, pp. 245-259.
9. K. Yocum et al., "Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging," *Proc. Int'l Symp. High Performance Distributed Computing*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 243-252.
10. L. Prylli and B. Tourancheau, "Protocol Design for High Performance Networking: A Myrinet Experience," Tech. Report 97-22, LIP-ENS Lyons, France, 1997.
11. T. von Eicken et al., "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," *Proc. Symp. Operating System Principles*, ACM Press, New York, Dec. 1995, pp. 303-316.

descriptor. The host may reuse a descriptor only after the network interface has released it.

The receiving host's—host B's—network interface contains a receive ring with descriptors that point to free buffers in its DMA area. Host B's network interface uses this information to determine where to store incoming packets (step 6). If no free host buffer is available, the network interface drops the packet; otherwise it starts a DMA to transfer the packet to host B's memory (step 7). Each host buffer contains a flag the network interface sets as part of the DMA transfer. The host checks if a packet is available by polling the flag of the next host's receive buffer. With each successful poll, a user function is invoked that handles

the packet. This function reads the packet and optionally copies its contents to a user buffer (step 8).

Although the protocol avoids all operating system overhead, keeps the network interface code very simple, and uses little network interface memory, it has several shortcomings:

- *All network transfers go through the DMA area.* As steps 1 and 7 in Figure 1 show, the data packet goes through the DMA areas of both hosts. Although this avoids the cost of kernel calls for each network access, it introduces extra memory copies for applications that send and receive from arbitrary locations.

The simple Myrinet protocol uses five data transfers to communicate a packet.

- *The protocol provides no protection.* As Figure 1 shows, network packets are staged through network interface memory. If the simple protocol lets multiple users access the network interface, the users can read and modify each other's data in the interface memory. They could even modify the interface's control program and use it to access any host memory location.
- *The receiving host uses polling to transfer control.* While simple, polling is not always the most efficient method to detect incoming packets (control transfer). If the host polls too frequently, it will have a high overhead; if it polls too late, it will not respond quickly enough to incoming packets. The alternative to polling, interrupts, is expensive on current operating systems.
- *The protocol is unreliable, even though the hardware is reliable.* If the senders send packets faster than the receiver can handle them, the receiving network interface will run out of buffer space and drop incoming packets.
- *The protocol supports only point-to-point messages.* Multicast is a fundamental component of collective communication operations, such as those supported by the MPI message-passing standard. Although you can implement multicast on top of unicast messages, it is usually inefficient.

The one-way latency of this simple protocol is 12 μ s and its throughput is 32 Mbytes/s. In contrast, on the same hardware, the highly optimized BIP system has a minimum latency of 4 μ s and a throughput of 126 Mbytes/s.

DATA TRANSFERS

As Figure 1 shows, the simple Myrinet protocol uses five data transfers to communicate a packet. Optimized protocols involve at least three transfers: the packet must be moved from the sender's memory to its network interface, a *host-to-interface* transfer; from this network interface to the receiver's network interface, an *interface-to-interface* transfer; and then to the receiving process's address space, an *interface-to-host* transfer. Some network technologies do not stage data through the network interface memory and require only a host-to-network and a network-to-host transfer. Although we describe only the Myrinet case, most design issues (use of programmed I/O versus DMA, pinning, and maximum packet size) also apply to the two-step case.

HOST TO INTERFACE

On Myrinet, this data transfer can use either DMA or *programmed I/O*.⁴ With programmed I/O, the host processor reads the data from host memory and writes

it into network interface memory, typically one or two words at a time, which results in many bus transactions. DMA uses a special DMA engine to transfer the entire packet in large bursts. Also, DMA transfers proceed in parallel with host computations.

On the surface, you might expect DMA to always outperform programmed I/O. However, much depends on the type of host CPU, the DMA engine, and the packet size. The Pentium Pro, for example, supports *write-combining buffers*, which boost the throughput of programmed I/O by combining multiple write commands over the I/O bus into a single bus transaction. Figure 2 shows the host-to-interface throughput using various data transfer mechanisms when copying data from a Pentium Pro to a Myrinet network interface card. As the figure shows, for buffers up to 1,024 bytes, programmed I/O with write combining is faster than DMA because of the DMA start-up cost.

With Myrinet, both a user process and the network interface can start a DMA transfer without any operating system involvement. Because DMA transfers are performed asynchronously, the operating system may decide to swap out the page that happens to be the source or destination of a running DMA transfer. If this happens, part of the transfer's destination will be corrupted. One solution is to let applications *pin* a limited number of pages in their address space, which the operating system can then never swap out. Unfortunately, pinning a page requires a system call, and the amount of memory that can be pinned is limited by the available physical memory and particular operating system policy.

Network interface protocols that use DMA often choose to copy the data into a reserved (and pinned) DMA area, which costs an extra memory copy and thus decreases throughput. As Figure 2 shows, DMA with copying is consistently slower than programmed I/O with write combining. The Shrimp system,² in contrast, provides special hardware that lets user processes start DMA transfers without pinning, but this mechanism works only for host-initiated transfers, not for transfers the network interface initiates, so pinning is still required on the receiving side.

With programmed I/O, pinning is not necessary. Even if the operating system swaps out the page during transfer, the next memory reference by the host will generate a page fault, causing the operating system to swap the page back in. As Table 1 shows, many protocols use DMA. FM, FM/MC, and LFC use programmed I/O for host-to-interface transfers; AM-II, Hamlyn, and BIP do so only for small messages.

Another important design choice is the maximum packet size. Large packets typically yield better throughput, because per-packet overhead is incurred fewer times. The throughput of our simple protocol, for example, increases from 32 Mbytes/s to 43 Mbytes/s

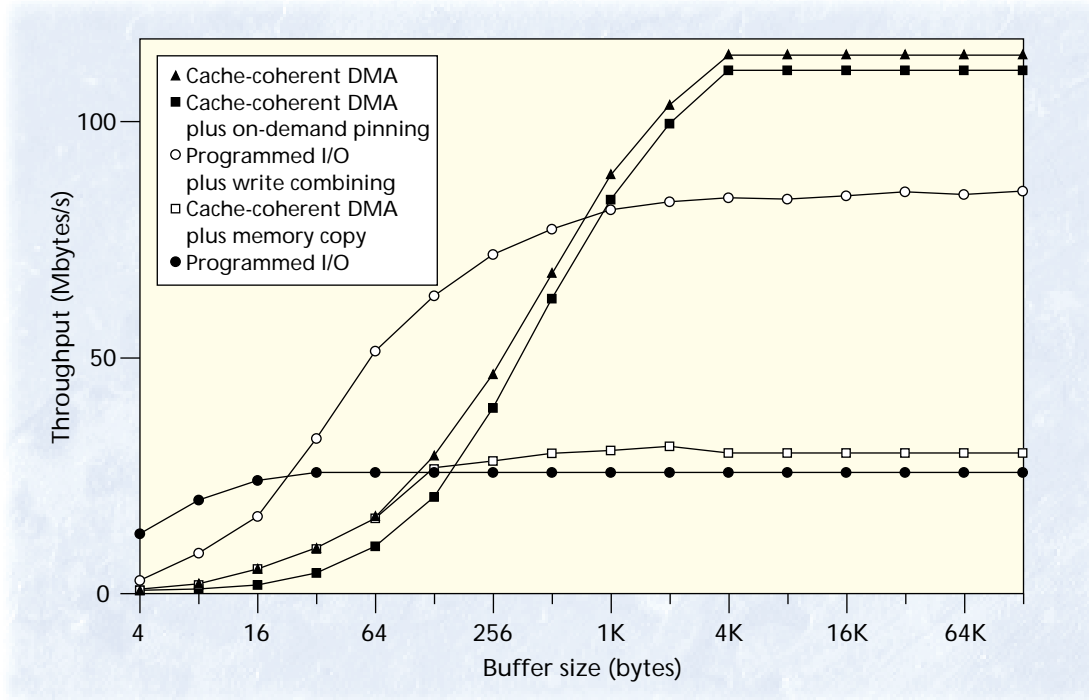


Figure 2. Throughput in a host-to-network-interface transfer using different transfer mechanisms. For buffers up to 1,024 bytes, programmed I/O with write combining outperforms DMA.

by using 1,024-byte instead of 256-byte packets. The choice of the maximum packet size is influenced by the system's page size, memory space considerations, and hardware restrictions.

INTERFACE TO INTERFACE

The interface-to-interface transfer goes through the Myrinet links and switches. All Myrinet protocols use the network interface's DMA engine to send and receive network data. In theory, they could use programmed I/O, but because the network interface processor is slow, DMA transfers are almost always faster.

On Myrinet, the hardware stalls the sending network interface if the receiver does not extract data. To prevent deadlock, however, there is a time limit on the stall. If the receiver does not drain the network within this time limit, the network will reset the network interface or truncate a blocked packet. Many Myrinet control programs deal with this real-time constraint by copying data fast enough to prevent resets. Other protocols use a software flow control scheme to avoid the problem, as we describe later.

INTERFACE TO HOST

The transfer from network interface to host on the receiving side can again use either DMA or programmed I/O. Because reads over the I/O bus are typically much slower than DMA transfers, most protocols use DMA. Some systems, such as AM-II, use programmed I/O on the host to receive small messages and use DMA for large messages.

ADDRESS TRANSLATION

The use of DMA transfers between host and network interface memory introduces two problems. First, most systems require that every host memory page involved in a DMA transfer be pinned (with a

system call) to prevent the operating system from replacing that page. Second, typically the network interface's DMA engine must know the physical addresses of each page it transfers data to or from. Operating systems, however, do not export virtual-to-physical mappings to users, so users normally cannot pass physical addresses to the network interface. Even if they could, the interface would have to check the addresses to ensure that users pass only addresses of pages they have access to.

There are several ways to address these problems. One is to avoid all DMA transfers by using programmed I/O. However, the high cost of I/O bus reads means this solution is realistic only on the sending side.

Another approach—and one the simple Myrinet protocol uses—is to require users to copy their data into and out of special DMA areas. These areas need to be pinned only once, when the application opens the device, not during send and receive operations. Address translation is easy because, for each DMA area, the operating system allocates a contiguous chunk of physical memory and passes the area's physical address to the network interface. Users then specify send and receive buffers using an offset in their DMA area, as described earlier. The network interface maintains only the area's physical address and size. AM-II and Hamlyn use this approach, taking the extra copying costs for granted. As Figure 2 shows, however, this copying significantly affects throughput.

In the third approach, the application or a library dynamically pins and unpins the pages that contain send and receive buffers. The network interface then performs DMA transfers directly to those pages. VMHC-2, PM, and BIP use this approach, which follows the cache-coherent DMA curve in Figure 2. However, the network interface must know the current virtual-to-physical mappings of individual pages,

Users write to network interface memory to initialize send descriptors.

but cannot store information for every single virtual page.

BIP and LFC provide a simple kernel module that translates virtual addresses to physical addresses. Users are responsible for pinning their pages and obtaining the physical addresses of these pages from the kernel module. The drawback here is that the network interface cannot check if the physical addresses it receives are valid and if they refer to pinned pages.

An alternative is to let the kernel and network interface cooperate such that the interface can track valid address translations in either hardware or software. Systems like VMMC-2 and U-Net/MM⁵ (an extension of U-Net) let the network interface *cache* a limited number of valid address translations that refer to pinned pages. This caching works well for applications that exhibit locality in the pages they use for sending and receiving data. When the network interface finds a translation of a user-specified address in the cache, it can access that address using a DMA transfer. If there is a cache miss, the network interface must interact with the kernel to pin the page and obtain its translation.

In U-Net/MM, the network interface generates an interrupt when it cannot translate an address. The kernel receives the interrupt, looks up the address in its page table, pins the page, and passes the translation to the network interface. In VMMC-2, a library manages address translations for user buffers. It invokes a kernel module to create *references* to address translations and then maps users' virtual addresses to these references. When the network interface receives an unknown reference, it finds the corresponding translation using a DMA transfer to the kernel module's data structure. To avoid such DMA transfers on the critical path, the network interface maintains its own cache of references. The "cache-coherent DMA + on-demand pinning" curve in Figure 2 approximates throughput with this approach.

PROTECTION

As Figure 1 shows, users write to network interface memory to initialize send descriptors. If multiple users share the network interface, one could corrupt another's send descriptors. The simple protocol avoids this problem by providing user-level network access to, at most, one user at a time—a limitation that is undesirable in many cases.

A straightforward solution to the protection problem is to use the virtual-memory system to give each user access to a different part of the network interface memory.¹ The operating system maps this part into the user's address space when the user opens the device. The virtual-memory hardware then traps all user accesses outside the mapped area. Users write

their commands, and possibly their network data, to their own pages in network interface memory. The network interface must then check each user's page for new requests and process only legal requests.

Because network interface memory is typically small, only a limited number of processes can be given direct access to the network interface in this way. AM-II applies a technique similar to paging to solve this problem. It uses part of its network interface memory to cache active communication end points, and stores inactive end points in host memory. When its network interface receives a message for an inactive end point or when a process tries to send a message via an inactive end point, the interface and operating system cooperate to activate the end point by moving its state to network interface memory, possibly replacing another end point.

Likewise, to maintain protection in the DMA areas, users need their own area. VMMC-2 and U-Net/MM eliminate the need for a fixed DMA area by dynamically pinning and unpinning user pages and by caching translations in network interface memory. BIP also eliminates the DMA area, but does not maintain protection.

CONTROL TRANSFER

Interrupts are generally expensive (delivery time of 10 μ s on our hardware), so all user-level architectures support some form of polling. The goal of polling is to give the host a fast mechanism to check if a message has arrived. This check must be inexpensive, because it may be executed very often. A simple approach is to let the network interface set a flag in its memory and to let the host check this flag. This approach is inefficient, however, because every poll results in an I/O bus transfer. In addition, polling traffic will slow down other I/O traffic, including network packet transfers between network interface and host memory.

In the simple Myrinet protocol, the network interface writes a flag in cached host memory (using DMA) when a message is available. The host polls by reading its local memory; because polls are executed frequently, the flag usually resides in the data cache, so failed polls do not generate memory or I/O traffic. When the network interface writes the flag, the host incurs a cache miss and reads the flag from its memory. On a 200-MHz Pentium Pro, this scheme costs 5 ns for a failed poll (a cache hit) and 125 ns for a successful poll (a cache miss). In contrast, each poll over the I/O bus costs 500 ns.

Even if the polling mechanism is efficient, inserting polls manually is tedious and error-prone. Several systems use a compiler or a binary rewriting tool to insert polls in loops and functions. Finding the right polling frequency remains arduous, however. Multiprocessor systems can solve this problem by dedicating a second processor to polling and message handling.

AM-II, FM/MC, LFC, Hamlyn, Trapeze, U-Net,

VMMC, and VMMC-2 support both interrupts and polling. The receiver usually enables or disables interrupts; sometimes the sender also sets a flag in each packet that determines if an interrupt is to be generated when the packet arrives. To reduce interrupts, LFC implements a polling watchdog⁶ on the network interface. This mechanism starts a timer when a message arrives and generates an interrupt only if the host does not issue a poll before the timer expires.

RELIABILITY

Existing Myrinet protocols differ widely in the way they address reliability. The most important choice is whether or not to assume that the network is reliable. Myrinet has a very low error rate: The risk of a packet getting lost or corrupted is small enough to consider it a fatal event. If necessary, such events can be handled using higher level software (checkpointing, for example).

Many Myrinet protocols assume the hardware is reliable. The advantage of this approach is efficiency, because no retransmission protocol or time-out mechanism is needed. Even if the network is fully reliable, however, the software protocol may still drop packets because of insufficient buffer space—a common cause of packet loss. Each protocol needs communication buffers on both the host and network interface and both are a scarce resource. This problem, which occurs in the simple protocol, can be solved by recovering from buffer overflow or preventing it.

Several other protocols do not assume the network is reliable and so either present an unreliable programming interface, as in U-Net and Trapeze, or implement a retransmission protocol on either the host or network interface. The cost of setting timers and processing acknowledgments is typically no more than a few microseconds.

RECOVERING FROM OVERFLOW

In PM, which assumes the hardware is reliable, the receiver simply discards incoming data packets if it has no room. It returns an acknowledgment, ACK, to the sender if it accepts the packet and a negative acknowledgment, NACK, if it does not. When an acknowledgment arrives, the receiver processes it completely before it accepts a new packet from the network. A NACK indicates a packet was discarded and needs to be retransmitted. Retransmission continues until an ACK is received. Because PM never drops acknowledgments and assumes that the network hardware is reliable, it need not use time-outs. This protocol is fairly simple; the main disadvantages are the extra acknowledgment messages and the increased network load when retransmissions occur.

PREVENTING OVERFLOW

The other approach when assuming the hardware

is reliable is to avoid overflow. Some systems use a flow control scheme that blocks the sender if the receiver is running out of buffer space. For large messages, BIP requires the application to use rendezvous-style communication to deal with flow control. The receiver posts a receive request and provides a buffer before a message is sent. Thus, the send of a large message never completes before a receive has been posted. FM and FM/MC implement flow control using a host-level credit scheme. Before a host can send a packet, it needs to have a credit for the receiver, which represents a packet buffer in the receiver's memory. Credits can be handed out in advance (by preallocating buffers for specific senders), but if a sender runs out of credits it must block until it gets new credits.

A host-level credit scheme does not prevent the overflow of network interface buffers, however. With some protocols, the network interface temporarily stops receiving messages if the network interface buffers overflow. Such protocols rely on Myrinet's hardware, link-level, flow-control mechanism (back-pressure) to stall the sender. LFC takes another approach and applies a flow-control scheme to network interface buffers that is similar to the host-level credit scheme of FM and FM/MC. To avoid host buffer overflow, LFC implements additional flow control between a receiving network interface and its host to ensure that the network interface does not release a packet buffer before it has copied the packet's contents to a free host buffer. Flow control at the network interface level also makes it easier to implement multicasting on the network interface. LFC, for example, uses a single flow-control scheme for reliable point-to-point and multicast communication.

MULTICAST

Hardware support for multicast in switched, worm-hole-routed network technologies like Myrinet is an active research area. The simplest way to implement a multicast in software is to let the sender send a point-to-point message to each multicast destination. However, the point-to-point start-up cost, which is incurred for every multicast destination, includes the data copy to network interface memory, possibly preceded by a copy to a DMA area. PM avoids this repeated copying by passing all multicast destinations to the network interface, which then repeatedly transmits the same packet to each destination.

Although more efficient than a repeated send on the host, this multisend still means the network interface is a serial bottleneck. Most proposals for network-interface-supported multicasting are therefore based on *spanning tree protocols*, which allow multicast packets to be transmitted in parallel (with logarithmic rather

Existing Myrinet protocols differ widely in the way they address reliability. The most important choice is whether or not to assume that the network is reliable.

than linear complexity). The system can implement tree-based protocols efficiently by forwarding multicast packets on the network interface instead of on the host. This is particularly efficient when combined with interrupt-driven packet delivery because it removes the interrupt handling costs from the forwarding path. LFC and FM/MC implement the forwarding of multicast packets at the network interface level.⁷

Much research has been done on all these design issues for both Myrinet and other network technologies. In addition to the 11 systems in Table 1, which originate mostly from academic environments, industry has recently developed the Virtual Interface Architecture (<http://www.viarch.org>), a draft standard for user-level communication in cluster environments.

Programmable network interfaces have much flexibility, often compensating for the lack of hardware support in the more advanced interfaces used by massively parallel processors. The polling watchdog device, for example, is easily implemented in the network interface control program, as in LFC. Address translation can be implemented in software on the network interface and the operating system, yielding functionality comparable to the address translation hardware of machines like Meiko's CS2.

Eventually, hardware implementations may be more efficient, but Myrinet's programmable network interface has let us and other researchers quickly and easily experiment with different protocols for commodity network interfaces. As a result, the performance of these protocols has substantially increased. In combination with the economic advantages of commodity networks, this makes such networks a key technology for parallel cluster computing. ❖

.....
Acknowledgments

We thank Cerial Jacobs, Rutger Hofman, Aske Plaat, Kees Verstoep, and the anonymous *Computer* reviewers for their valuable feedback. We thank Andrew Chien and Scott Pakin for making the FM software available and Mario Lauria and Matt Buchanan for their suggestion to use write combining on the Pentium Pro.

This research is supported in part by a Pionier grant from the Netherlands Organization for Scientific Research.

.....
References

1. P. Druschel, L.L. Peterson, and B.S. Davie, "Experiences with a High-Speed Network Adaptor: A Software Perspective," *Proc. Conf. Communications Architectures,*

Protocols, and Applications, ACM Press, New York, 1994, pp. 2-13.

2. M.A. Blumrich et al., "Design Choices in the SHRIMP System: An Empirical Study," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 330-341.
3. N.J. Boden et al., "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, Jan./Feb. 1995, pp. 29-36.
4. P.A. Steenkiste, "A Systematic Approach to Host Interface Design for High-Speed Networks," *Computer*, Mar. 1994, pp. 47-57.
5. M. Welsh et al., "Memory Management for User-Level Network Interfaces," *IEEE Micro*, Mar./Apr. 1998, pp. 77-82.
6. O. Maquelin et al., "Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 179-188.
7. K. Verstoep, K.G. Langendoen, and H.E. Bal, "Efficient Reliable Multicast on Myrinet," *Proc. Int'l Conf. Parallel Processing*, Vol. III, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 156-165.

Raoul A.F. Bhoedjang is a researcher in computer science at the *Vrije Universiteit*, where his research interests are in parallel programming and the structure of communication architectures. He received an MSc in computer science from the *Vrije Universiteit*. He is a member of the ACM.

Tim Rühl is a researcher in computer science at the *Vrije Universiteit* and a software engineer at *Data Distilleries B.V.* His research interests are advanced parallel programming models and their efficient implementation. He received an MSc in computer science from the *Vrije Universiteit*. He is a member of the ACM and the IEEE.

Henri E. Bal is a professor in the Faculty of Sciences at the *Vrije Universiteit*, where he heads research groups on parallel programming and physics-applied computer science. His research interests include parallel and distributed programming, network computing, applications, and programming languages. He received an MSc in mathematics from Delft University of Technology and a PhD in computer science from the *Vrije Universiteit*.

Contact the authors at the Dept. of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands, {raoul, tim, bal}@cs.vu.nl.