

Design Issues for User-Level Network Interface Protocols on Myrinet

Raoul Bhoedjang Tim Rühl Henri E. Bal
Dept. of Mathematics and Computer Science
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

This paper surveys the design issues for user-level network interface protocols for modern high-speed networks such as Myrinet. It first explains the principles of such protocols through a simple, unreliable protocol. Next, six important design issues are discussed in more detail: data transfers, address translation, protection, control transfers, reliability, and multicast. The design issues are illustrated by performance measurements on a Myrinet cluster and by representative examples taken from 11 communication systems for Myrinet.

Introduction

Modern high-speed local area networks offer great potential for communication-intensive applications. Traditional communication protocols (e.g., TCP/IP), however, are unable to realize this potential. In the common implementation of these protocols, all network access is through the operating system, which adds significant overheads to both the transmission path (typically a system call and a data copy) and the receive path (typically an interrupt and a data copy). In response to this performance problem, several *user-level communication architectures* have been developed that remove the operating system from the critical communication path [1, 2]. Our goal in this paper is to give insight into the design issues for communication protocols for these architectures. We concentrate on issues that determine the performance and semantics of a communication system: data transfer, address translation, protection, control transfer, reliability, and multicast.

To illustrate the design issues, we study communication protocols for Myrinet [3], a modern, wormhole-routed, Gigabit-per-second network technology for local and system area networks (see the sidebar on Myrinet). Our focus is on the hardware/software interface between a host and its network interface and on the data link layer communication protocol between communicating network interfaces. We shall refer to these two as *network interface protocols*; together they form the building block for higher-level communication layers (e.g., TCP and MPI).

Myrinet's programmable network interface (NI) makes it an interesting research vehicle. Its programmability has enabled the exploration of a significant part of the NI protocol design space and several research groups in the parallel

processing community have designed communication systems for Myrinet. In this paper, we use the following systems to illustrate the design issues: AM-II, BIP, FM, FM/MC, Hamlyn, LFC, PM, U-Net, VMMC, VMMC-2, and Trapeze (see the sidebar on Myrinet communication systems). All systems aim for high performance and all except Trapeze offer a user-level communication service. Interestingly, however, they differ significantly in how they resolve the design issues studied in this paper. This variety motivates our study; we aim to gain insight in the design tradeoffs for NI protocols.

In the rest of the paper we will first explain the basic principles of NI protocols by describing a simple, unreliable, user-level protocol. Next, we discuss six protocol design issues that determine system performance and semantics. We will illustrate these issues by giving performance data obtained on a Myrinet cluster. The nodes of this system are 200 MHz Pentium Pros and run the BSD/OS (Version 3.0) operating system. Although the performance figures clearly depend on the platform being used, they are useful to give a more quantitative analysis of several issues.

Sidebar on Myrinet

Myrinet is a switched, Gigabit-per-second local area network technology. It uses variable-length packets which are wormhole-routed through a network of highly reliable links and crossbar switches. Myrinet is relatively unique in that it employs some techniques (e.g., wormhole routing and hardware backpressure) that are normally found only in supercomputers.

Figure 1(a) illustrates the architecture of a node in a Myrinet cluster. Figure 1(b) gives details of the Myrinet configuration used for our measurements. Each machine (host) has an NI card that contains a processor and some memory, which is used to store the NI's control program and its data. The NI connects to the host's I/O bus, which is a typical organization for commodity hardware. More aggressive architectures (found in supercomputers and research machines) integrate the network interface more tightly with the host system by placing it on the memory bus or integrating it with the cache controller. These architectures allow for very low network access latencies and simpler protection schemes [2, 4, 5]. In this paper, however, we focus on architectures that use low-cost, off-the-shelf hardware and rely on advanced software techniques to achieve efficient user-level network access.

Myrinet requires that all packets be staged through NI memory, both at the sending and the receiving side. Myrinet uses fast but expensive SRAM, so the memory is relatively small. Other networks allow data to be transferred directly between host memory and the network (e.g., using memory-mapped FIFOs or DMA).

Both the host and the NI can use Direct Memory Access (DMA) to access data in each other's memory, but DMA transfers suffer from a startup overhead. In addition, the host can access the NI's memory using programmed I/O

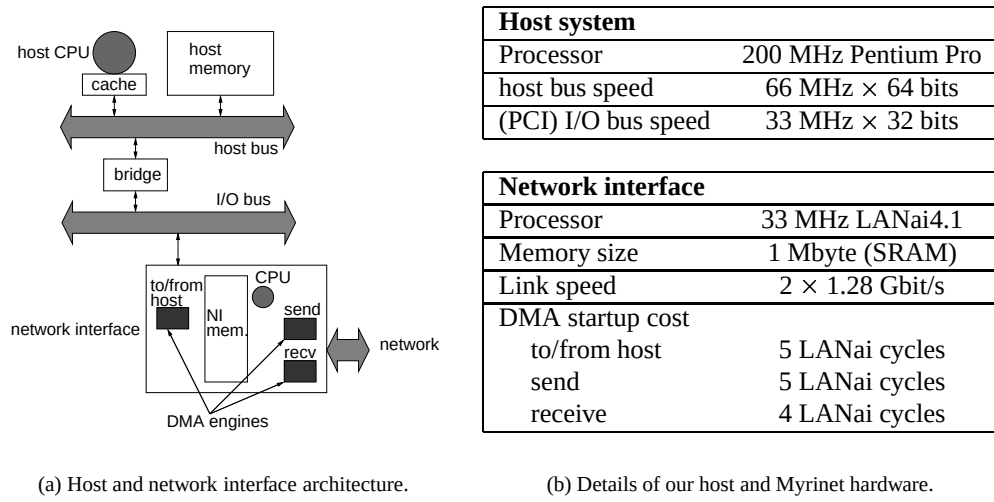


Figure 1: Myrinet hardware characteristics.

(PIO). PIO has no startup costs, but since the NI memory is accessed over the I/O bus the access times are higher than for host memory.

An interesting feature of Myrinet (and several other modern networks) is the availability of a programmable processor on the NI. This gives protocol designers much flexibility, since they can co-design software for the host and the NI. On the other hand, such NI processors invariably are much slower than the host CPU, so the NI's tasks should be kept simple. Myrinet's LANai4.x, for example, is a simple RISC processor which runs at 33 MHz, while the host processors in our clusters are superscalar, 200 MHz Pentium Pros. Adding only a few instructions to the critical path of a Myrinet control program results in noticeable increases in end-to-end latency.

A basic network interface protocol

The goal of this section is to explain the basics of NI protocols and to introduce the most important design issues. To structure our discussion, we describe the design of a simple, user-level, network interface protocol for Myrinet. The protocol ignores several important problems, which we address in subsequent sections.

To avoid the cost of kernel calls for each network access, the basic protocol maps all NI memory into user space. User processes write their send requests directly to NI memory, without operating system (OS) involvement. The basic protocol provides no protection, so the network device cannot be shared among multiple processes.

User processes invoke a simple send primitive to send a data packet. The basic protocol sends packets with a

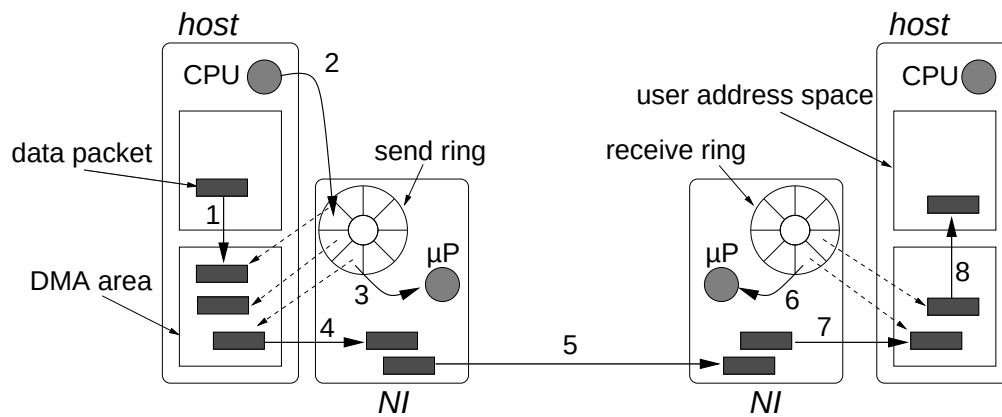


Figure 2: Operation of the basic protocol. (1) Host copies user data into DMA area. (2) Host writes packet descriptor to send ring. (3) NI processor reads packet descriptor. (4) NI DMA's packet to NI memory. (5) Network transfer. (6) NI reads receive ring to find empty buffer in DMA area. (7) NI DMA's packet to DMA area. (8) Optional copy to user buffer.

maximum payload of 256 bytes and requires that users fragment their data so that each fragment fits in a packet.

```
void send(int destination, void *data, unsigned size);
```

Send performs two actions (see Figure 2). First, it copies the user data to a packet buffer in a special staging area in host memory (step 1). The NI will later fetch the packet from this *DMA area* by means of a DMA transfer. (See the sidebar on Myrinet for a discussion of the various data transfer mechanisms supported by Myrinet.) Unlike normal user pages, pages in the DMA area are never swapped to disk by the OS. By only DMAing to and from pinned pages, the protocol avoids corruption of user memory and user messages due to paging activity of the OS.

Second, the host writes a send request into a descriptor in NI memory (step 2). These descriptors are stored in a circular buffer called the send ring. **Send** stores the destination machine, the size of the packet's payload, and the packet's offset in the DMA area into the next available descriptor in this send ring. **Send** also sets a **DescriptorReady** flag to inform the NI of this event. The descriptor is written using programmed I/O; since the descriptor is small, DMA would have a high overhead.

The NI repeatedly polls the **DescriptorReady** flag of the first descriptor in the send ring. As soon as this flag is set by the host, the NI reads the offset in the descriptor and adds it to the physical address of the start of the DMA area, resulting in the physical address of the packet (step 3). Next, the NI initiates a DMA transfer over the I/O bus to copy the packet's payload from host memory to NI memory (step 4). Subsequently, it reads the destination machine in the descriptor and looks up the route for the packet in a routing table.¹ Finally, the NI starts a second DMA

¹Routing is an important issue, but a discussion of routing algorithms is beyond the scope of this article.

to transmit the packet (step 5). Note that the data transfers in steps 1, 4, and 5 can all 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-NI DMA. When the NI detects that the network DMA for a given packet has completed, it sets a `DescriptorFree` flag to release the descriptor, and polls the next free descriptor in the ring. If the host wants to send a packet while no free descriptor is available, it busy-waits by polling the `DescriptorFree` flag of the descriptor at the tail of the ring.

When the NI receives a network packet from a remote NI, it stores the packet in its memory using a receive DMA. The NI contains a receive ring with descriptors that point to free buffers in the host's DMA area. The NI uses this information to determine where to store incoming packets (step 6). These descriptors contain flag bits that are used in a similar way as for the send ring. If no free host buffer is available, the packet is simply dropped. Next, the NI starts a DMA to transfer the packet to host memory (step 7). Each host buffer also contains a flag that is set by the NI (as part of the DMA transfer) to inform the host that an incoming packet is available. The host can check if there is a packet available by polling the flag of the next unprocessed host receive buffer. Once the flag has been set, the receiving process can safely read the buffer and optionally copy its contents to a user buffer (step 8).

Since the delivery of network interrupts to user-level processes is expensive on current OSs, the basic protocol does not use interrupts, but requires users to poll for incoming messages. A successful poll results in the invocation of a user function (`handle_packet`) that handles an inbound packet:

```
void poll(void);  
void handle_packet(void *data, unsigned size);
```

Our (unoptimized) implementation of the basic protocol achieves a one-way latency of 12 μ sec and a throughput of 32 Mbyte/sec (using 256-byte packets). For comparison, on the same hardware the highly optimized BIP system achieves a minimum latency of 4 μ sec and can saturate the I/O bus (126 Mbyte/sec). Note that the basic protocol avoids all OS overhead, keeps the NI code very simple, and uses little NI memory. It is clear, however, that the protocol has several shortcomings:

- All inbound and outbound network transfers are staged through the DMA area. For applications that need to send and receive from arbitrary locations, this will introduce extra memory copies.
- The protocol provides no protection. If the basic protocol allowed multiple users to access the NI, these users could read and modify each other's data in NI memory. Users can even modify the NI's control program and use it to access any host memory location.

- The receiver-side control transfer mechanism, polling, is simple, but not always effective. For many applications it is difficult to determine a good polling rate. If the host polls too frequently, it will have a high overhead; if it polls too late, it will not reply quickly enough to incoming packets.
- The protocol is unreliable, even though the Myrinet hardware is highly reliable. If the senders send packets faster than the receiver can handle them, the receiving host will run out of buffer space and the NI will start to drop incoming packets.
- The protocol only supports point-to-point messages. Although multicast can be implemented on top of unicast messages, doing so may be inefficient. Multicast is an important service by itself, but is also a fundamental component of collective communication operations such as those supported by the message-passing standard MPI.

Below, we will discuss these problems in more detail and look at better design alternatives.

Data transfers

On Myrinet, at least three steps are needed to communicate a packet from one user process to another: the packet must be moved from the sender's memory to its NI (host-NI transfer), from this NI to the receiver's NI (NI-NI transfer), and then to the receiving process's address space (NI-host transfer). Network technologies that do not require data to be staged through NI memory use only two steps: host-to-network and network-to-host. Below, we discuss the Myrinet case, but most issues (PIO versus DMA, pinning, alignment, and maximum packet size) also apply to the two-step case.

The data transfers have a significant impact on the latency and throughput obtained by a protocol, so optimizing them is essential for obtaining high performance. As shown in Figure 2, the basic protocol uses five data transfers to communicate a packet, because it stages all packets through DMA areas. Below, we discuss the alternative designs for implementing the host-NI, NI-NI, and NI-host transfers.

Host – NI transfer. On Myrinet, this data transfer can use either DMA or Programmed I/O. A detailed description of both mechanisms is given in [6]. With PIO, the host processor reads the data from host memory and writes it into NI memory, typically one or two words at a time, which results in many bus transactions. DMA uses special hardware (a DMA engine) to transfer the entire packet in large bursts and asynchronously, so that the data transfer can proceed in parallel with host computations. One thus might expect DMA to always outperform PIO. The optimal choice, however, depends on the type of host CPU and on the packet size. The Pentium Pro, for example, supports

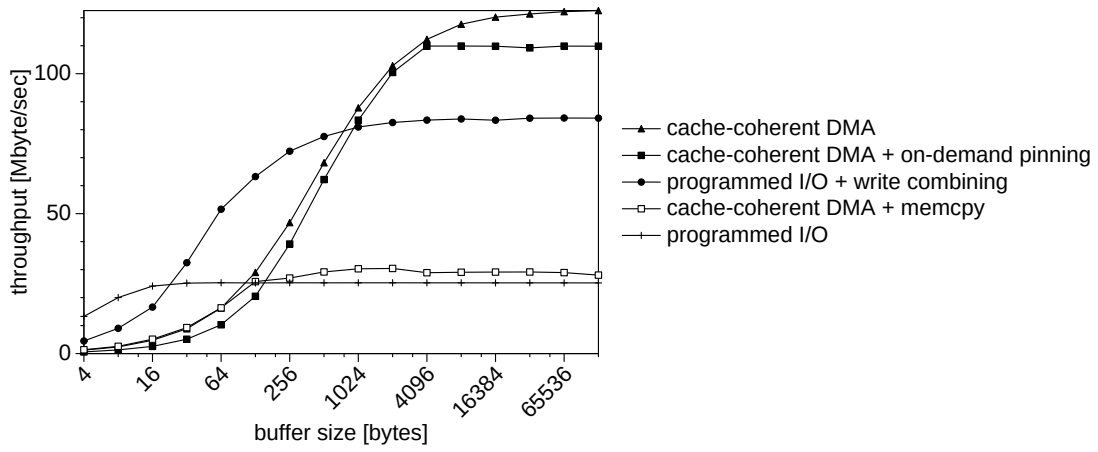


Figure 3: Host-NI throughput using different data transfer mechanisms.

write-combining buffers, which boost the throughput of PIO by combining multiple write commands over the I/O bus into a single bus transaction. Figure 3 shows the throughput obtained by PIO (with and without write combining) and cache-coherent DMA, for copying data from a Pentium Pro to a Myrinet NI card. For buffer sizes up to 1024 bytes, PIO with write combining is faster than DMA (which suffers from a startup cost).

In user-level communication systems, DMA transfers can be started either by a user process or by the network interface without any operating system involvement. Since 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 destination of the transfer will be corrupted. To avoid this, operating systems allow applications to *pin* a limited number pages in their address space. Pinned pages are never swapped out by the operating system. 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 by OS policies. The SHRIMP system [2] provides special hardware that allows user processes to start DMA transfers without pinning. This user-level DMA mechanism, however, only works for host-initiated transfers, not for NI-initiated transfers, so pinning is still required at the receiving side.

NI 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 may decrease the throughput. As can be seen in Figure 3, the extra memory copy decreases the throughput significantly, making DMA consistently slower than PIO with write combining. On processors that do not support cache-coherent DMAs, the DMA area needs to be allocated in uncached memory, which also decreases performance. (Most modern CPUs, including the Pentium Pro, support cache-coherent DMA, however.) With PIO, pinning is not necessary. Even if the OS would swap out the page during the transfer, the next memory reference by the host will generate a page fault, causing the OS to swap the page back in. In practice, many protocols use DMA; FM

and LFC use PIO; some protocols (AM-II, Hamlyn, BIP) use PIO for small messages and DMA for large messages.

A problem that arises with both DMA and PIO is that data transfers between unaligned data buffers can be significantly slower than between aligned buffers. The problem is aggravated when the network interface's DMA engines *require* that source and destination buffers are properly aligned. In this case, extra copying is required to align unaligned buffers.

Another important design choice is the maximum packet size. Large packets typically yield better throughput, because per-packet overheads are incurred fewer times than with small packets. The throughput of our basic protocol, for example, increases from 32 Mbyte/sec to 43 Mbyte/sec by using 1024-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.

NI – NI transfer. The second data transfer is between the sending and receiving NI, so this transfer goes through the Myrinet links and switches. All Myrinet protocols use the NI's DMA engine to send and receive network data. In theory, PIO could be used, but since the NI processor is slow, DMA transfers are always faster.

To prevent network congestion, the receiving NI should extract incoming data from the network fast enough. On Myrinet, the hardware uses *backpressure* to stall the sending NI if the receiver does not extract data fast enough. To prevent deadlock, however, there is a time limit on the backpressure mechanism. If the receiver does not drain the network within a certain time period, the network will reset the NI or truncate a blocked packet. Many Myrinet control programs deal with this real-time constraint by just copying data fast enough to prevent resets. Other protocols avoid this problem by using a software flow control scheme, as we will discuss later.

NI – host transfer. The transfer from NI to host at the receiving side can again use either DMA or PIO. On Myrinet, however, only the host (not the NI) can use PIO, making DMA the method of choice for most protocols. Some systems (e.g., AM-II) use PIO on the host to receive small messages. For large messages, all protocols use DMA, because reads over the I/O bus are typically much slower than DMA transfers.

Table 1 gives an overview of the three data transfers, including the available hardware (peak) bandwidth, the measured throughput for large messages (on the Pentium Pro/Myrinet cluster), and an indication of which system component limits the throughput. Note that the I/O (PCI) bus is the bottleneck. The table also shows the memory-to-memory copy throughput on a Pentium Pro. An interesting observation is that a local memory copy on a single Pentium Pro obtains a lower throughput than a remote memory copy over Myrinet. Although this problem is partly due to the inferior memory copy performance of the Pentium Pro architecture, it is also a general hardware trend that network bandwidth is becoming much less a bottleneck than memory bandwidth. For comparison, recall that the basic

Transfer	Physical bandwidth	Measured copy throughput	Bottleneck
Host - NI	126	123 (DMA), 84 (PIO)	PIO or DMA speed over PCI bus
NI - NI	153	127	NI processor or link speed
NI - host	126	123	DMA speed over PCI bus
Host - host	176	57	Memory bus

Table 1: Hardware and measured throughputs (in Mbyte/sec) on a Pentium Pro/Myrinet cluster.

protocol achieves a throughput of only 32 Mbyte/sec. The reason is that the basic protocol uses fairly small packets (256 bytes) and performs memory copies to and from DMA areas, which interferes with DMA transfers. Several of the systems listed in the sidebar on Myrinet communication systems can saturate the I/O bus: the key issue is to avoid the copying to and from DMA areas. The next section describes techniques to achieve this.

Address translation

The use of DMA transfers between host and NI memory introduces two problems. First, most systems require that every host memory page that is involved in a DMA transfer be pinned to prevent the operating system from replacing that page. Pinning a page requires an expensive system call that should be kept off the critical path. The second problem is that on most architectures the network interface's DMA engine needs to know the physical addresses of each page that 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 NI. Even if they could, the NI would have to check those physical addresses, to ensure that users only pass addresses of pages that they have access to.

We consider three approaches to solve these problems. The first approach is to avoid all DMA transfers by using programmed I/O. Due to the high cost of I/O bus reads, however, this is only a realistic solution at the sending side.

The second approach, used by the basic protocol, requires that users copy their data into and out of special DMA areas (see Figure 2). This way, only the DMA areas need to be pinned. This is done once, when the application opens the device, and not during send and receive operations. The address translation problem is then solved as follows. The operating system allocates for each DMA area a contiguous chunk of physical memory and passes the area's (physical) base address to the NI. Users specify send and receive buffers by means of an offset in their DMA area; the NI adds this offset to the area's base address to find the buffer's physical address. The NI now has to maintain only a small amount of state per area to be able to translate (and check) virtual addresses in this area: the area's physical base address and its size. Several systems (e.g., AM-II, Hamlyn) use this approach, taking the extra copying costs for

granted. As shown in Figure 3, however, the extra copy significantly affects throughput.

In the third approach, the copying to and from DMA areas is eliminated by dynamically pinning (and unpinning) user pages such that DMA transfers can be performed directly to those pages. Systems that use this approach (e.g., VMMC-2, PM, and BIP) can track the 'cache-coherent DMA' curve in Figure 3. The main implementation problem is that the NI needs to know the current virtual-to-physical mappings of individual pages. Since NIs are usually equipped with only a small amount of memory and since their processors are relatively slow, they do not store information for every single virtual page. Some systems (BIP, 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 disadvantage of this 'user-translates' approach is that the NI cannot check if the physical addresses it receives are valid and if they refer to pinned pages.

An alternative approach is to let the kernel and NI cooperate such that the NI can keep track of valid address translations (either in hardware or in software). Systems like VMMC-2 and U-Net/MM [7] (an extension of U-Net) let the NI cache a limited number of valid address translations which refer to pinned pages (this invariant must be maintained cooperatively by the NI and the operating system). This caching will work well for all applications that exhibit locality in the pages they use for sending and receiving data. When the translation of a user-specified address is found in the cache, the NI can access that address using a DMA transfer. In the case of a miss, special action must be taken to translate the address and to pin the page.

In U-Net/MM, the NI 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 NI. In VMMC-2, address translations for user buffers are managed by a library. This user-level library maps users' virtual addresses to *references* to address translations which users can pass to the NI. The library creates these references by invoking a kernel module. This module translates virtual addresses, pins the corresponding pages, and stores the translations in a *User-managed TLB* (UTLB) in kernel memory. To avoid invoking the operating system every time a reference is needed, the library maintains a user-level lookup data structure that keeps track of the addresses for which a valid UTLB entry exists. The library only invokes the UTLB kernel module when it cannot find the address in its lookup data structure. When the NI receives a reference, it can find the translation using a DMA transfer to the kernel module's data structure. To avoid such DMA transfers on the critical path, the NI maintains its own cache of references. The 'on-demand pinning' curve in Figure 3 shows the throughput obtained by a benchmark that imitates the miss behavior of a UTLB. For each page transferred, the benchmark invokes a system call to pin a page (simulating a miss in the host lookup structure); in addition, the NI fetches a single word from host memory before it fetches the data that is to be transferred (thus

simulating a miss in the NI cache).

Protection

Since user-level architectures give users direct access to the NI, they cannot rely on the OS to check network transfers. In the basic protocol, for example, users directly write to NI memory to initialize send descriptors. If multiple users shared the NI, one user could corrupt another user's send descriptors. The basic protocol avoids this problem by providing user-level network access to at most one user at a time, but this limitation is clearly undesirable in a multi-user and multiprogramming environment. Also, even with a single user, this user can modify the NI control program and use it to read or write any location in host memory.

A straightforward solution to both problems is to use the virtual-memory system to give each user access to a different part of NI memory [1]. Mapping this part into the user's address space is done by the operating system when the user opens the device. Once the mapping has been established, all user accesses outside the mapped area will be trapped by the virtual-memory hardware. Users write their commands (and possibly their network data) to their own pages in NI memory. It is the responsibility of the NI to check each user's page for new requests and to process only legal requests.

Since NI memory is typically small, only a limited number of processes can be given direct access to the NI this way. To solve this problem, AM-II virtualizes network endpoints in the following way. Part of NI memory acts as a *cache* for active communication endpoints; inactive endpoints are stored in host memory. When an NI receives a message for an inactive endpoint or when a process tries to send a message via an inactive endpoint, the NI and the operating system cooperate to activate the endpoint. Activation consists of moving the endpoint's state (send and receive buffers, protocol status) to NI memory, possibly replacing another endpoint which is then swapped out to host memory.

A similar problem exists for the DMA area. To maintain protection, each user needs its own DMA area. Since the use of a DMA area introduces an extra copy, some systems eliminate it and store address translations on the NI. VMMC-2 and U-Net/MM do this in a protected way, either by letting the kernel write the translations to the NI or by letting the NI fetch the translations from kernel memory. BIP, on the other hand, eliminates the DMA area, but does not maintain protection.

Control transfers

The **control transfer** mechanism determines how a receiving host is notified of message arrivals. The options are to use interrupts, polling, or a combination of these.

Interrupts are notoriously expensive. With most current OSs, the time to deliver an interrupt (as a signal) to a user process even exceeds the network latency. On the Pentium Pro, dispatching an interrupt to a kernel interrupt handler costs approximately 10 μ sec. Dispatching to a user-level signal handler costs even more, approximately 24 μ sec for BSD/OS. Note that this exceeds the latency of our basic protocol (12 μ sec).

Given the high costs of interrupts, 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 would be to let the NI set a flag in its memory and to let the host check this flag. This approach, however, is inefficient, since every poll now results in an I/O bus transfer (see Figure 1). In addition, this polling traffic will slow down other I/O traffic, including network packet transfers between NI and host memory.

A very efficient solution is to use a special device register that is shared between the NI and the host [8]. Current hardware, however, does not provide these shared registers.

On architectures with cache-coherent DMA, a practical solution is to let the NI write a flag in cached host memory (using DMA) when a message is available. This approach is used by our basic protocol. The host polls by reading its local memory; since polls are executed frequently, the flag will usually reside in the data cache, so failed polls are cheap and do not generate memory or I/O traffic. When the NI writes the flag, the host will incur a cache miss and read the flag from its memory. On a 200 MHz Pentium Pro, the scheme just described costs 5 nanoseconds for a failed poll (i.e., a cache hit) and 125 nanoseconds for a successful poll (i.e., a cache miss). For comparison, each poll in the simple scheme (i.e., an I/O bus transfer) costs 500 nanoseconds.

Even if the polling mechanism is efficient, polling is a mixed blessing. Inserting polls manually is tedious and error-prone. Several systems therefore use a compiler or a binary rewriting tool to insert polls in loops and functions. The problem of finding the right polling frequency remains however. In multiprocessor architectures this problem can be solved by dedicating a second processor to polling and message handling.

Several systems (AM-II, FM/MC, LFC, Hamlyn, Trapeze, U-Net, VMMC, VMMC-2) support both interrupts and polling. Interrupts usually can be enabled or disabled by the receiver; sometimes the sender can also set a flag in each packet that determines whether an interrupt is to be generated when the packet arrives. To reduce the number of interrupts, LFC has implemented a *polling watchdog* [9] on the NI. This mechanism starts a timer when a message

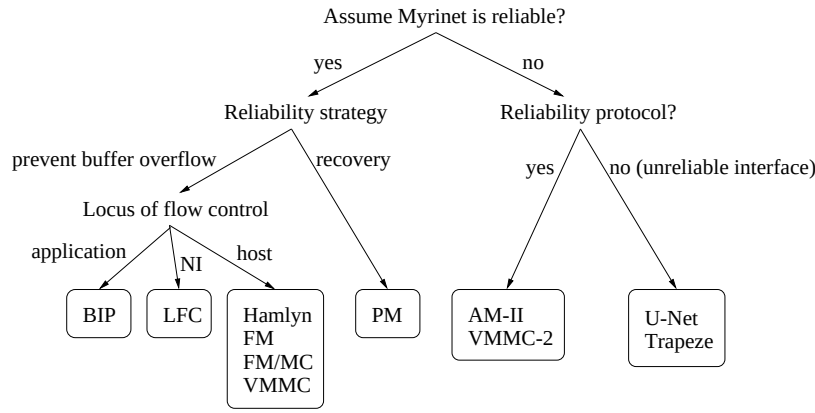


Figure 4: Design decisions for reliability.

arrives and generates an interrupt only if no poll is issued by the host before the timer expires.

Reliability

Existing Myrinet protocols differ widely in the way they address reliability. In Figure 4 we show the choices made by various systems. 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 as a “fatal event” (much like a memory parity error or an OS crash). Such events can be handled by higher-level software (e.g., using checkpointing), or else cause the application to crash.

Many Myrinet protocols indeed assume that the hardware is reliable, so let us look at these protocols first. 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 due to lack of buffer space. In fact, this is the most common cause of packet loss. Each protocol needs communication buffers on both the host and the NI, and both are a scarce resource. This problem occurred in the basic protocol described earlier. It can be solved in one of two ways: either *recover* from buffer overflow or *prevent* overflow to happen.

The first idea (recovery) is used in PM. The receiver simply discards incoming packets if it has no room for them. It returns an acknowledgement (ACK or NACK) to the sender to indicate whether or not it accepted the packet. The protocol never drops acknowledgements: Myrinet is reliable (by assumption) and when an acknowledgement arrives, the receiver processes it completely before it accepts a new packet from the network, so at most one acknowledgement needs to be buffered. Myrinet’s hardware flow control ensures that pending network packets are not dropped. A NACK

indicates a packet was discarded; the sender will later retransmit that packet. This process continues until an ACK is received, in which case the sender can release the buffer space for the message. This protocol is fairly simple; the main disadvantages are the extra acknowledgement messages and the increased network load when retransmissions occur.

The second approach is to avoid buffer overflow to occur by using 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 deal with flow control by means of a rendezvous mechanism: the receiver must post a receive request and provide a buffer before a message may be sent. That is, a large-message send 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 pre-allocating buffers for specific senders), but if a sender runs out of credits it must block until it succeeds in getting new credits.

A host-level credit scheme prevents overflow of host buffers, but not of NI buffers, which are usually even scarcer (because NI memory is smaller than host memory). With some protocols, the NI temporarily stops receiving messages if the NI buffers overflow. Such protocols rely on Myrinet's hardware, link-level, flow control mechanism (backpressure) to stall the sender in such a case. LFC takes another approach and applies an NI-level credit-based flow control scheme. This scheme is similar to the credit scheme of FM and FM/MC described above, but it is applied to NI buffers and therefore prevents NI buffer overflow. To avoid host buffer overflow, LFC implements additional flow control between a receiving NI and its host to ensure that the NI does not release a packet buffer before it has copied the packet's contents to a free host buffer. An advantage of NI-level flow control is that it simplifies the implementation of an NI-level multicast scheme (described later).

The protocols described so far thus implement a reliable interface by depending on the reliability of the hardware. Several other protocols do not assume the network to be reliable, and either present an unreliable programming interface or implement a retransmission protocol. U-Net and Trapeze present an unreliable interface (like UDP). Instead, higher software layers (e.g., MPI, TCP) are supposed to retransmit lost messages. Other systems do provide a reliable interface, by implementing a timeout-retransmission mechanism, either on the host or the NI. The cost of setting timers and processing acknowledgement is modest, typically no more than a few microseconds.

Multicast

Multicast is an important communication service, and hardware support for multicast in switched, wormhole-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. This solution is inefficient, because the

point-to-point startup cost is incurred for every multicast destination; this cost includes the data copy to NI memory (possibly preceded by a copy to a DMA area). With some NI support, the repeated copying can be avoided by passing all multicast destinations to the NI, which then repeatedly transmits the same packet to each destination. Such a 'multisend' primitive is provided by PM.

Although more efficient than a repeated send on the host, a multisend still leaves the network interface as a serial bottleneck. Most proposals for NI-supported multicasting are therefore based on spanning tree protocols that allow multicast packets to be transmitted in parallel (with logarithmic rather than linear complexity). Tree-based protocols can be implemented efficiently by performing the forwarding of multicast packets on the network interface instead of on the host. On our cluster, NI-level forwarding [10] improves the latency of a broadcast to 64 nodes by 25% (from 136 μ sec to 101 μ sec). The performance gain becomes even larger when packets are received by interrupts, because this increases host-level forwarding latencies. NI-level forwarding of multicast packets is implemented by LFC and FM/MC.

An important issue in the design of a multicast protocol is flow control. Multicast flow control is more complex than point-to-point flow control, because the sender of a multicast packet needs to obtain buffer space on all receiving NIs and hosts. FM/MC uses a central buffer manager (implemented on one of the NIs) to keep track of the available buffer space at all receivers. This manager handles all requests for multicast buffers and allows senders to prefetch buffer space for future multicasts. An advantage of this scheme is that it avoids deadlock by acquiring buffer space in advance; a disadvantage is that it employs a central buffer manager. An alternative scheme is to acquire buffers on the fly: the sender of a multicast packet is responsible only for acquiring buffer space at its children in the multicast tree. This scheme is used by LFC. LFC uses a single, NI-level, flow control scheme for point-to-point and multicast traffic. This flow control scheme scales better than the central buffer manager, but care must be taken to avoid buffer deadlocks.

Discussion and conclusions

The paper has discussed several design issues and tradeoffs for network interface protocols, in particular:

- how to transfer data between hosts and NIs (DMA or programmed I/O);
- how to avoid copying to and from DMA areas by means of address translation techniques;
- how to achieve protected, user-level network access in a multi-user environment;

- how to transfer control (interrupts and/or polling);
- how and where to implement reliability (application, hosts, network interfaces);
- whether to implement additional functionality on the NIs, such as multicast.

Much research has been done on these issues, both for Myrinet and other network technologies. The sidebar on Myrinet communication layers summarizes how 11 existing communication systems address each issue. An interesting observation is that many novel techniques exploit the programmable NI processor. The latter capability gives NIs much flexibility, which often compensates for the lack of hardware support present in the more advanced interfaces used by Massively Parallel Processors (MPPs). We have seen several examples in the paper:

- The polling watchdog device [9] can easily be implemented in the NI control program, as shown by LFC.
- Address translation can be implemented in software on the NI and the host (OS), giving comparable functionality as address translation hardware of machines like the Meiko CS2.
- Programmable NIs enable efficient software multicast implementations and thus compensate for the lack of hardware support.

Eventually, hardware implementations may be more efficient, but the availability of a programmable NI has enabled fast and relatively easy experimentation 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.

Acknowledgements

We thank Cerial Jacobs, Aske Plaat, Kees Verstoep, and the anonymous reviewers for reviewing a draft of this paper and providing valuable feedback. We thank Andrew Chien and Scott Pakin for making the FM software available to us. Mario Lauria and Matt Buchanan suggested the idea of using write combining on the Pentium Pro. This research is supported in part by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

References

- [1] P. Druschel, L.L. Peterson, and B.S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proc. of the 1994 Conf. on Communications Architectures, Protocols, and Applications (SIGCOMM)*, pages 2–13, London, UK, September 1994.

- [2] M.A. Blumrich, R.D. Alpert, Y. Chen, D.W. Clark, S.N. Damianakis, E.W. Felten, L. Iftode, K. Li, M. Martonosi, and R.A. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *Proc. of the 25th Int. Conf. on Computer Architecture*, pages 330–341, Barcelona, Spain, June 1998.
- [3] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.
- [4] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Int. Conf. on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.
- [5] S.S. Mukherjee and M.D. Hill. The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. In *The 4th Int. Symp. on High-Performance Computer Architecture*, pages 207–218, Las Vegas, NV, January 1998.
- [6] P.A. Steenkiste. A Systematic Approach to Host Interface Design for High-Speed Networks. *IEEE Computer*, 27(3):47–57, March 1994.
- [7] A. Basu, M. Welsh, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Hot Interconnects'97*, Stanford, CA, April 1997.
- [8] S.S. Mukherjee, B. Falsafi, M.D. Hill, and D.A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proc. of the 23rd Int. Conf. on Computer Architecture*, pages 247–258, Philadelphia, PA, May 1996.
- [9] O. Maquelin, G.R. Gao, H.H.J. Hum, K.B. Theobald, and X. Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proc. of the 23rd Int. Conf. on Computer Architecture*, pages 179–188, Philadelphia, PA, May 1996.
- [10] K. Verstoep, K.G. Langendoen, and H.E. Bal. Efficient Reliable Multicast on Myrinet. In *Proc. of the 1996 Int. Conf. on Parallel Processing*, volume III, pages 156–165, Bloomington, IL, August 1996.

Sidebar on Myrinet Communication Systems

The table below classifies 11 user-level communication systems and shows how each system deals with the design issues discussed in the paper. All 11 systems aim for high performance and provide a lean, low-level, and more or less generic communication facility. All systems except VMMC and U-Net were developed first on a Myrinet platform.

Most systems implement a straightforward message-passing model. These systems differ mainly in their reliability and protection guarantees and their support for multicast. Several of these systems use *active messages*. With active messages, the sender of a message specifies not only the message’s destination, but also a *handler function*, which is invoked at the destination when the message arrives.

VMMC, VMMC-2, and Hamlyn provide *virtual memory-mapped* and sender-based communication instead of message passing. In both models the sender of a data packet specifies where in the receiver’s address space the packet must be deposited. This model sometimes allows data to be moved directly to its destination without any unnecessary copying.

In addition to the research projects listed above, industry has recently created a draft standard for user-level communication in cluster environments. Implementations of this *Virtual Interface (VI)* architecture are being constructed by Intel and UC Berkeley. A description of the VI architecture is available at <http://www.viarch.org/>.

System	Data transfer (host-NI)	Translation	Protection	Control transfer	Reliability	Multicast support
AM-II [1]	PIO & DMA	DMA areas	yes	polling + interrupts	Reliable. NI: 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. Ucast: host-level credits. Mcast: NI-level credits.	yes (on NI)
PM [4]	DMA	software TLB on NI	yes (gang scheduling)	polling	Reliable. ACK/NACK protocol on NI.	multiple sends
VMMC [5]	DMA	software TLB on NI	yes	polling + interrupts	Reliable. Exploits hardware backpressure.	no
VMMC-2 [6]	DMA	UTLB in kernel, cached on NI	yes	polling + interrupts	Reliable.	no
LFC [7]	PIO	user translates	no	polling + interrupts + watchdog	Reliable. Ucast: NI-level credits. Mcast: NI-level credits.	yes (on NI)
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 NI (U-Net/MM)	yes	polling + interrupts	Unreliable.	no

References

- [1] B. Chun, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnects'97*, Stanford, CA, April 1997.
- [2] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, December 1995.
- [3] H.E. Bal, R.A.F. Bhoedjang, R. Hofman, C. Jacobs, K.G. Langendoen, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. *Journal of Parallel and Distributed Computing*, 40(1):49–64, February 1997.
- [4] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *12th Int. Parallel Processing Symposium*, pages 308–314, Orlando, FL, March 1998.
- [5] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *11th Int. Parallel Processing Symposium*, pages 388–396, Geneva, Switzerland, April 1997.
- [6] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects'97*, Stanford, CA, April 1997.
- [7] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Int. Conf. on Parallel Processing*, Minneapolis, MN, August 1998.

- [8] G. Buzzard, D. Jacobson, M. MacKey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *The 2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 245–259, Seattle, WA, October 1996.
- [9] K. Yocum, J. Chase, A. Gallatin, and A. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging. In *The 6th Int. Symp. on High Performance Distributed Computing*, Portland, OR, August 1997.
- [10] L. Prylli and B. Tourancheau. Protocol Design for High Performance Networking: A Myrinet Experience. Technical Report 97-22, LIP-ENS Lyon, July 1997.
- [11] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th Symp. on Operating System Principles*, pages 303–316, Copper Mountain, CO, December 1995.