# Implementing Protected Multi-User Communication for Myrinet*

Shailabh Nagar, Dale Seed, and Anand Sivasubramaniam

Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802.
Phone: (814) 865-1406
{nagar,seed,anand}@cse.psu.edu

**Abstract.** A Network of Workstations (NOW) is emerging as a cost-effective solution to high performance computing. However, we need to lower the cost of communicating between the workstations to make this platform viable. With the advent of high-performance networks such as ATM and Myrinet, the physical network is no longer the communication bottleneck. Rather, the major overhead can now be attributed to software. This overhead is a direct result of the cost that a message incurs as it travels through different protection domains. We can alleviate this problem by allowing protected user-level access directly to the network, thereby eliminating the kernel from the critical path. This paper presents a description of the design, implementation and performance of a protected user-level messaging system over Myrinet, called MU-Net, that can handle multiple application processes concurrently. MU-Net has been implemented on the SUN Solaris 2.5 operating system.

## 1 Introduction

Despite the concerted effort from both industry and academia to design and manufacture multiprocessors, their success in the commercial arena has been limited [7,9]. Most multiprocessor designs include custom-made components which not only increase the costs, but also the time to market. Recently, there has been a trend to use a Network of Workstations (NOW) as a more cost-effective solution to high performance computing [1]. Rather than relying on custom-made components, NOWs can be constructed from commercial off-the-shelf hardware. This not only decreases the cost, but also increases the availability of high performance computing platforms. Such designs also make it easier to upgrade the hardware with anticipated improvements in workstation and networking technology. Further, the workstations themselves, can be used as general purpose computing engines.

* This research is supported in part by a NSF Career Award MIP-9701475, and equipment grants from NSF and IBM.

D. K. Panda and C. B. Stunkel (Eds.): Network-Based Parallel Computing, CANPC 98
LNCS 1362, pp. 30–44, 1998. 

The major limitation of NOWs is the high communication overhead in exchanging messages between nodes. Unlike a multiprocessor that has tightly coupled nodes and a custom low-latency/high bandwidth network, the nodes in a NOW are loosely connected by a Local Area Network (LAN) with a lower bandwidth. The associated software costs for communication further limit the scalability of the applications executing on the NOW. Therefore, to make this platform more attractive, we need to address three important issues related to communication costs. First, the network connecting the workstations should be fast. Second, the interface to the network should provide an efficient way of transferring data between the memory on the workstation and the network. Finally, the software messaging layers should add minimal overhead to the cost of moving data between two application processes running on two different workstations. Recent research has been addressing these issues. High speed networks such as ATM [4] and Myrinet [2] can potentially deliver point-to-point hardware bandwidths that are comparable to the link bandwidths of the interconnection networks in multiprocessors. To address the second issue, newer network interface designs have been proposed with hardware enhancements to overlap communication with computation and to facilitate direct user-level interaction with the network interface [10]. Finally, low latency software messaging layers have been proposed [11,6,3,8,5,12] making use of these network and network interface innovations.

Of all the costs discussed above, the software costs have traditionally been the most dominant for NOW environments. In particular, the involvement of the kernel in the critical send/receive path results in a significant degradation in performance. This is due to the cost of crossing protection boundaries, and multiple levels of copying that must occur between the kernel, user processes, and the network device. A recent research trend has focused on removing the kernel from the critical path of sending and receiving messages to cut down these costs. This has resulted in several user-level messaging platforms such as Cornell's U-Net [10], Myricom's API and GM[2,5], Illinois' Fast Messages [6], HP's Hamlyn [3], PM [8], and Trapeze [12]. The common goal of these designs is to minimize the end-to-end communication latency of a message by off-loading communication responsibilities from the kernel to both the user and the *smart* network interface. Since the kernel is responsible for providing protection between user processes, alternative communication mechanisms must also enforce this to allow multiple processes to access the network concurrently. If this protection is not provided, then one process may potentially receive or corrupt another process's messages. In NOW environments, which can have multiple application processes executing on each node and sharing the network concurrently, providing protected communication is extremely important.

This paper presents the design, implementation and performance of MU-Net, a user-level messaging platform for Myrinet that allows protected multi-user access to the network. Our design of MU-Net has drawn from ideas of other user-level platforms [10,6] but there are some key differences. Unlike the original Fast Messages implementation [6], MU-Net allows multiple applications on a node to

concurrently use the network in a protected manner. Recently, an implementation of Fast Messages supporting multiple processes on PCs running Windows NT or Linux has been announced, but such a version for SUN Solaris platforms is not yet available. Further, MU-Net provides additional mechanisms for transferring longer messages in cases where the host CPU has other useful work to do, while Fast Messages always employs the host CPU to packetize/reassemble longer messages. Our performance results on a SUN Ultra Enterprise 1 platform running Solaris 2.5 show that despite being able to support multiple application processes, MU-Net performs comparably to Fast Messages 2.0 (the version which supports only one application process per node). Our results also give a detailed breakup of the costs incurred in the various stages of a message as it propagates from the sender to the receiver. The differences between MU-Net and U-Net [10] are in the implementation details specific to the Myrinet platform which will become clearer in the following sections.

The rest of this paper is organized as follows. Section 2 gives a hardware description of the Myrinet platform. The design of MU-Net and a description of its operations is discussed in detail in Section 3 and the performance results are given in Section 4. Finally, we summarize the related work in Section 5 and present concluding remarks in Section 6.

## 2 Myrinet

Myrinet [2] is a high-speed switch-based network which allows variable length packets. A typical Myrinet network consists of point-to-point links that connect hosts and switches. A network link can deliver 1.28 Gbits/sec full duplex bandwidth. Source routing is used for packets on the Myrinet network. The sender appends a series of routing bytes onto the head of each packet. When a packet arrives at a switch, the leading byte is stripped and used to determine the outgoing port. Myrinet does not impose any restrictions on packet sizes and leaves the choice to the software. Myrinet does not guarantee reliable delivery, however cyclic-redundancy-checking (CRC) hardware is provided in both the network interfaces and switches. This allows for error detection, but reliable delivery is left for higher level software layers to implement. However, the network itself guarantees in-order delivery of messages.

The Myrinet network interface card (which sits on the workstation's I/O bus such as the SBUS on the SPARCstations) consists of a 256KB SRAM and a 37.5 MHz 32-bit custom-built processor called the LANai 4. In addition, there are 3 DMA engines on the card. Two of these engines are responsible for sending(receiving) packets to(from) the external network from(to) SRAM. The third is responsible for moving data between SRAM and host (workstation) memory. All three DMA engines can operate independently. Note that the LANai processor cannot communicate directly with host memory, and therefore must rely on DMA operations for reading and writing host memory. The LANai processor executes a MU-Net Control Program (MCP) that manages and coordinates activities on the interface card and interfaces with programs running on the

host CPU. Though it shares the same initials as the Myrinet Control Program supplied by Myricom, it differs significantly in functionality.

## 3 MU-Net

Our goal is to develop an efficient communication substrate for parallel applications on a Network of Workstations. We have used Myrinet [2] for the network hardware since it has a high raw bandwidth, and puts no restriction on packet sizes making it convenient to tailor the messaging system to application characteristics. In developing this substrate, our design draws ideas from the implementation of U-Net [10] on ATM, and Fast Messages [6] on Myrinet. Hence, the name *MU-Net* standing for Myrinet U-Net. The design goals for MU-net are summarized below:

- provide a low-latency, high bandwidth user-level messaging layer (to avoid costs of crossing protection boundaries),
- support multiple processes running on a workstation to concurrently use the network without compromising on protection and without significantly degrading communication performance,
- implement optimizations for shorter messages, while lowering the cost of packetization/reassembly for larger messages,
- allow alternate send mechanisms that can be used to overlap useful CPU computation with communication wherever needed,
- provide in-order delivery with flow control.

The inclusion of these features in the MU-Net design requires detailed development of user level libraries (API), kernel level drivers, and software for the LANai. We have implemented these components for Solaris 2.5 running on a range of SPARCstations. The following subsections describe the design and responsibilities of each of these components.

### 3.1 User-level Library

As in [10], MU-Net supports the notion of an endpoint that is intended to give user processes a handle into the network. The endpoint is visible to the user as a structure which contains state information about pending messages (to be sent or received) on that endpoint. A user process can only access those endpoints that it creates. In essence, an endpoint virtualizes the network for each user process, and uses the traditional virtual memory system to enforce protection between these processes.

The MU-Net API provides user applications with an interface for creating an endpoint in the user's address space, destroying an endpoint, sending to and receiving messages from a destination endpoint. Creation and destruction of endpoints involve the kernel driver and are expensive. These are done only in the initialization and termination phases and hence do not impact the latency of the critical path.

There are two kinds of send operations both of which are nonblocking. The first is meant for processes which want to minimize latency at the cost of host CPU cycles. In this kind of send, the host CPU is used to transfer data to the network interface card instead of the DMA engine, similar to the send mechanism in Fast Messages [6]. However, for long messages, the time spent in the send call (proportional to the size of the message) may become significant especially if the CPU has other work to do. Hence, in our API, we provide a second mechanism called `send_DMA()`, which uses the DMA for large data transfers to the card. If the length of the data to be transferred is smaller than 128 bytes, send_DMA defaults to the normal send. We expect applications to use this send when they have work that can be overlapped with the operation, and when they can tolerate a slightly higher latency. If the applications cannot proceed with useful computation, and the message to be sent is larger than 128 bytes, then they can use the normal `send()` mechanism which uses the host CPU to packetize the data and transfer it to the network interface.

There is only one receive call which is used to receive both kinds of messages.

Currently, we are implementing a credit-based flow control strategy for MU-Net similar to Fast Messages [6], that throttles the sender whenever it uses up its credits for a particular receiver. Each endpoint statically partitions its receive buffer area amongst the maximum number of endpoints which could send to it. Flow control is thus implemented fully on the host without taxing the LANai.

### 3.2 MU-Net Kernel Drivers

MU-Net uses a 2 driver design in the operating system kernel similar to [2]. The MU-Net *myri* driver is used to attach to the Myrinet device, and map the LANai registers and SRAM into kernel memory. The second MU-Net driver, called the *mlanai*, is a *pseudo* driver, and therefore does not actually drive any physical device. Rather, the *mlanai* driver provides ioctls for creating and destroying endpoints. It also maps a communication segment into user space. The details of the communication segment are described later.

### 3.3 MU-Net MCP

The MU-Net Control Program (MCP) is the program that runs on the LANai processor which is downloaded into the card SRAM by the host during initialization.

The MCP does not directly interact with the host. It detects send requests by polling the SRAM. It multiplexes these send requests and sends them out into the network. Incoming messages are first buffered on the SRAM, demultiplexed and then delivered to the destination endpoint. An examination of an incoming message is required to determine the destination endpoint and a DMA operation is used to perform the delivery.

The maximum number of endpoints supported by the card is statically determined. However, the number of active endpoints is dynamic and communicated

to the MCP by the host driver. This ensures that the MCP does not spend time polling for send requests from inactive endpoints.

### 3.4 Details of MU-Net Operations

The MU-Net operations are described in the same logical order they would most likely occur in a typical user application. The implementation of these operations is also discussed along with other design alternatives.

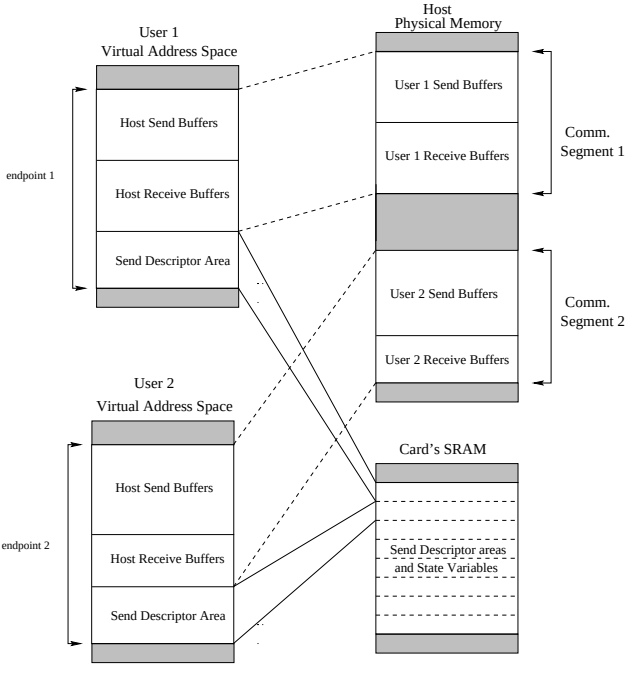


**Fig. 1.** Mapping of Endpoint into User Address Space

**Creating an Endpoint** Messages are exchanged between endpoints and not processes. One process could create multiple endpoints though it is not necessary since endpoint communication is connectionless and the same endpoint can be used for sending/receiving from multiple remote nodes. An endpoint consists of three regions, two of which reside on the host memory and the third on the card SRAM. All three areas are mapped into the user processes' address space. The first two regions are the Host Send Buffer and the Host Receive Buffer. The Host Send Buffer is used to store long messages in host memory for a subsequent LANai-initiated DMA to SRAM (send_DMA() operation). The Host Receive Buffer is the target of LANai-initiated DMA's for incoming short

and long messages. The third region is the Send Descriptor area located on the SRAM. It is written by the host CPU during a send call. Figure 1 shows two user processes' endpoints and their mappings. The two host buffers together constitute the *communication segment* of the endpoint.

MU-Net allows a user program to dynamically create endpoints. To create an endpoint, the user program calls the user-level MU-Net API and provides the desired sizes of the Host Send Buffer and Host Receive Buffer. The size of the Send Descriptor area cannot be specified by individual user processes. The communication segment needs to be allocated and pinned in physical memory and mapped into the DMA space of the DMA engine located on the card. Since this memory mapping can only be done in kernel mode, the MU-Net API makes an ioctl call to the MU-Net *mlanai* driver to perform these operations.

The communication segment memory is provided by the kernel and is not shared amongst endpoints. The card SRAM, which is a shared resource, is also protected by the selective mapping of only one Send Descriptor area into user space. Moreover, since the MU-Net API library can only use virtual addresses, it can only access those portions of the SRAM mapped in by the driver. Thus, MU-Net uses the operating system's virtual memory system to implement protected user-level communication similar to [10]. The driver, being part of the kernel, is assumed to be secure while the user can potentially use a different library with the same API.

The major design issue relevant to endpoint creation is the partitioning of the communication segment. In MU-Net, the partitioning of the communication segment into Send and Receive areas is done at initialization and cannot be changed in the middle. If the number/size of one type of message (send or receive) is much lesser than the other, the corresponding area is underutilized and may even affect the latency/bandwidth of the other type (when the latter operates at peak buffer capacity). An alternative would be to divide up the entire communication segment into fixed size buffers, which could be dynamically assigned for a send or a receive (by maintaining pointers to these buffers on the host and on the card). This approach, used in [10], has the disadvantage of limiting the size of a single data transfer between the host and the card (in either direction). The fixed size of the buffers would have to be large to avoid multiple data transfers in the average case. However larger sized buffers would also lead to greater internal fragmentation. Hence the rigid partitioning approach was chosen. If the user process has a priori knowledge of traffic patterns, it can choose to have one area larger than the other. With the communication segment being pinned in physical memory, a process needs to exercise restraint in the sizes requested since it can impact overall host memory performance. This restraint can also be enforced by the driver on creation of an endpoint.

**Sending a Message** From the user's viewpoint, once an endpoint has been created, the process of sending a message is straightforward. The user calls the MU-Net API with a pointer to message data, length of message and destination endpoint.

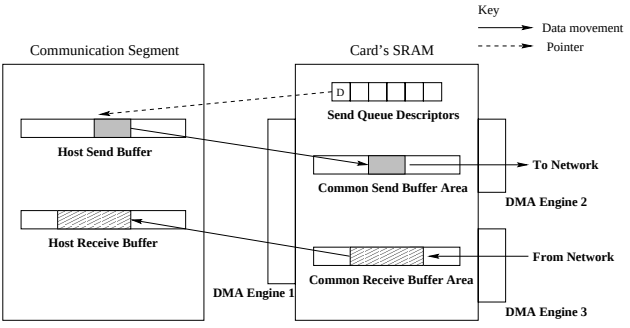


**Fig. 2.** Anatomy of a MU-Net Operation

For a message longer than 128 bytes (henceforth called a long message), the API copies the message data into the Host Send Buffer, creates a Send Descriptor and appends it to the Send Descriptor queue of the endpoint in the SRAM. Within the Descriptor is a pointer to the data in the Host Send Buffer (which is part of the DMA space of the card). The MCP polls the Send Descriptor queue to detect messages to be sent. Upon detection of the newly added descriptor, it uses the pointer within it to initiate a DMA of the message data into a Common Send Buffer area in SRAM (Figure 2). The MCP also creates a message header that includes routing information and a 2-byte tag (1 byte for the message type and the other for the destination endpoint number). After the DMA from host completes, the packet (header + data) is DMA'ed from the SRAM onto the network. Figure 3 shows the layout of a MU-Net packet.

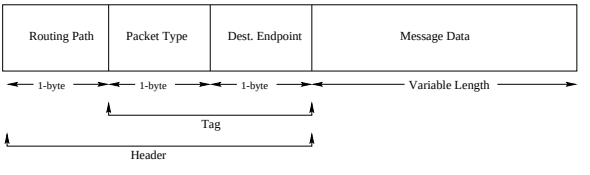


**Fig. 3.** Structure of a MU-Net Packet

The above send_DMA mechanism works well for situations where the application can tolerate slightly higher latencies because it has other work to do. As observed in [6], it is more expensive to transfer data into the SRAM using DMA than having the host CPU directly write it. So for the normal send mechanism, we have an optimization wherein the descriptor entry itself contains the data (and not a pointer to it). A 128 byte size buffer is part of the Send Descriptor. For messages up to 128 bytes, the API copies the data into the Send Descriptor created. The Host Send Buffer is not accessed. More importantly, the MCP does not have to perform a DMA to get the data from the host and can proceed with

the other steps of a message send directly. When the application wants to send messages longer than 128 bytes, the API packetizes the message and places them as separate entries in the descriptor queue. This packetization is transparent to the MCP.

We have found that the performance of the send operation depends heavily on the implementation details. The following are some key issues :

- **Descriptor detection**: Instead of polling head and tail pointers to the descriptor queue (which we have found to be highly inefficient), the MCP can poll a designated location within the next expected descriptor entry to find out if data is ready to be sent. This entry needs to be the last field in the descriptor filled by the host to avoid a race condition.
- **Queue management**: Any queue that is accessed by both the host CPU and the MCP needs to be managed carefully for correctness and efficiency. In MU-Net , there is one producer and one consumer for items in a queue and these are on different sides of the I/O bus. For the Send Descriptor queue, the host CPU is the producer and the MCP the consumer. On the receive side, the situation is reversed (as we shall see later). The host CPU can access the SRAM directly (through memory mapped I/O) but the MCP can only access host memory through DMA operations. DMA is not suitable for frequent accesses to individual variables. So all queue maintenance information is kept on the SRAM in the Send Descriptor Area.
  All queues are circular and both producer and consumer have to wrap around when they reach the end of a queue. Since wrap around occurs relatively infrequently, the cost for checking it need not be incurred on each queue access. Another cost saving measure is to reduce the number of updates required to queue variables by the producer and consumer. Typically the producer increments the tail after putting an item on the queue and the consumer increments the head after consuming an item. Queue full and empty checks require both variables to be read by the producer and consumer respectively. In MU-Net, the user process and the MCP are decoupled as far as possible for purposes of queue maintenance. The user process keeps a local copy of the Send Descriptor head and tail. The portion between the head and tail represent messages queued on the SRAM but not yet sent on the network. Additions to the queue (after a send), result in an update of the local tail alone. Updating the head kept on the SRAM is done by the MCP after it consumes a new message in the queue. The local head is important only when the local tail becomes equal to it (signifying a queue full condition). It is only at that point that the user needs to find out the real value of head (kept on SRAM). As long as both MCP and user maintain FIFO order of the queue, this decoupling improves performance without compromising correctness.
- **Overlap of steps**: The MCP tasks for a send operation need to be carefully examined to maximize overlap of operations without affecting correctness. After initiating a DMA from the host for long messages, the MCP has to wait for it to complete before going on with the next step of the send oper-

ation. There is potential for doing useful work here (unrelated to the send) instead of busy waiting which is possible because of the separation of all three DMA engine operations. Trapeze [12] exploits the decoupling of these DMA operations through a mechanism called cut-through delivery.
– **Flow control** : Before starting to compose a message, the API checks to see if the sending endpoint has enough credits for the receiving endpoint. This translates to a guarantee that the receiver has enough buffer space to accommodate the message about to be sent. If the credits are insufficient, the sender checks the received message queue for any credits that might have come in since the last receive was performed. If processing of the piggybacked credit information on newly arrived messages is not enough, the sender busy waits for incoming messages. Running out of credits for sending is not the common case so the impact of the flow control code on minimum latency is limited to the cost of checking for credits before sending. Updating the credit information after a send is not in the critical path and does not impact the latency.

**Receiving a Message** The MCP detects incoming packets from the network by checking flag bits in a LANai Interrupt Status Register (ISR). The MCP then initiates a DMA from the network into a Common Receive Buffer Area on the SRAM. After this DMA completes, the length of the message and the destination endpoint are read off the message header. A DMA to that endpoint's Host Receive Buffer is initiated. Upon completion of the DMA, the appropriate queue variables are updated for that endpoint. We would like to point out that there is an optimization possible here which we are currently investigating. Instead of DMAing off the net and then examining the header for the length and the endpoint, there is a way to examine just the first bytes of a message and program the host DMA with these parameters in parallel with the DMA of the packet onto the SRAM. Note that the host DMA cannot actually begin before the entire packet is in the SRAM.

The user process calls the MU-Net API to receive a message. The API checks for a pending message by polling the queue state variables. It then copies the message data to the location provided by the the user process and updates the queue state variables to reflect consumption of the message. Some details for the receive operation are discussed below:

– **Short message optimization**: Unlike the send operation, no distinction is made between long and short messages and consequently no optimization is possible for short messages.
In [10], the receive operation is done differently for long and short messages. For long messages, the message data is first DMA'ed to the host. This could require multiple DMA's to be initiated since the host buffer sizes are fixed. After it completes, a *receive descriptor* is DMA'ed to a queue residing in host memory. Besides message length, the receive descriptor contains pointers to the host buffers used as destination for the first DMA. The user detects a

message by polling on the receive descriptor. For shorter messages, the data is copied to the receive descriptor buffer and only one DMA is done.
We are not using this strategy because we feel that the LANai processor, which is already quite slow [6], would be taxed more than needed. Also, we are not using the pool of fixed buffers strategy. Instead, we allow the Host Receive Buffer to accommodate variable sized messages that are placed adjacent to each other in the order received. Hence all messages, short and long, need only one DMA to the host.

- **Message detection**: The only way the MCP can communicate with the host is via the DMA. But, we would like to limit the number of DMA operations to one (for transferring the data to the host memory and to inform the host of the arrival of the message). To ensure correctness, notification should occur only after the whole message data is in host memory. The MCP can determine that this has occurred by checking for completion of the DMA it initiates. The user can determine it by polling for the arrival of the last byte or word of an expected message. Notification is automatic on completion of DMA of the data. The MCP does not need to wait for completion of the DMA it initiates. However, polling for the last word of an expected message has its own difficulties. Instead we use an intelligent probe of the state variables on the SRAM by the host to solve these problems (the details of which are not discussed here due to space limitations).
- **Flow control** : After the received message contents are copied to the application buffer, a part of the Host Receive Buffer is free to receive more messages. This free buffer space is credited to the sender of the message just consumed. The API checks to see if the credits accumulated by the sender has crossed a high water mark. If so, it sends a zero length control message back to the sender containing only the credit information (piggybacked in the usual way). These control message exchanges for flow control are not the common case and have a minimal impact on latency. However the check for accumulated credits and updating the credit information is in the critical path of a message.

**Destroying an Endpoint** Once a user application no longer needs to access the network, a call to the MU-Net API can be made to *teardown* an endpoint. Destroying an endpoint involves deallocating the associated host memory. We also need to inform the MCP that it need not spend time polling for messages sent using this endpoint. The MU-Net *mlanai* driver must be called to complete this process since these tasks cannot be done at the user level.

## 4 Performance Results

To evaluate the performance of our MU-Net implementation, we have exercised this software over a couple of SUN Ultra 1 Enterprise servers connected by Myrinet (through an 8-port switch) and present preliminary performance results

| | Message Size (in bytes) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **8** | **64** | **128** | **256** | **512** | **1024** | **2048** | **4096** |
| MU-Net | 40 | 50 | 57 | 71 | 104 | 162 | 272 | 495 |
| Fast Messages | 47 | 53 | 60 | 80 | 118 | 196 | 334 | 598 |
| Myricom | 211 | 216 | 227 | 243 | 271 | 329 | 447 | 681 |

**Table 1.** Comparison of Roundtrip Latencies (in $\mu$s)

here. We have used a simple microbenchmark that ping-pongs packets between two SUN Ultra Enterprise 1 Model 170 workstations.

Table 1 shows the roundtrip latencies for messages using this microbenchmark as a function of the message size for MU-Net, Fast Messages (FM 2.0) and Myricom's API for the same hardware platform. The reader should note that the public distribution of FM 2.0 and the Myricom API were actually run on the Ultra Enterprise machines in our laboratory to obtain these results.

For a fair comparison, we run MU-Net using only 1 endpoint though the code for multiple endpoints is in place. The Send_DMA() call is used on the send side which as we mentioned defaults to the normal send (the CPU explicitly copies the entire message to the descriptor on the interface card) when the message size is less than or equal to 128 bytes, and uses the DMA for data transfer to the card (the CPU is free to do useful work) when the message size is larger than 128 bytes.

The results indicate that MU-Net compares favorably with Fast Messages 2.0 for both short and long messages. The Myricom API latencies are considerably higher primarily because the API is a general purpose messaging layer supporting TCP/IP and automatic network remapping. The cost of the additional code is especially noticeable for small messages. For larger messages, the differences between the three layers diminish as data transfer dominates the critical path.

| | **8 bytes** | | | | **1024 bytes** | **4096 bytes** |
|---|---|---|---|---|---|---|
| | 1 endpt | 2 endpts | 4 endpts | 8 endpts | 1 endpt | 1 endpt |
| Detected in send queue | 6.5 | 7.0 | 7.5 | 9.0 | 6.5 | 13.0 |
| Header sent on network | 7.0 | 7.5 | 8.0 | 9.5 | 12.5 | 45.0 |
| Data sent on network | 8.0 | 8.0 | 9.0 | 10.0 | 23.5 | 73.0 |
| Data in SRAM | 9.5 | 10.0 | 11.5 | 14.0 | 47.0 | 152.0 |
| DMA to host | 18.0 | 18.5 | 19.0 | 23.5 | 70.5 | 219.5 |
| Message received by host | 19.5 | 20.5 | 22.5 | 23.5 | 81.0 | 247.0 |

**Table 2.** Anatomy of a Message in $\mu$s (Effect of message size and multiple endpoints)

Table 2 shows the anatomy of the different operations performed for short and long messages in MU-Net with different number of endpoints (though the remaining endpoints are not exercised, the LANai still has to multiplex/demultiplex messages between these endpoints). The times given in each row are cumulative for the operations that occur from the time the user makes an API send call. The first row gives the time taken by the MCP to detect a message. As expected, this time grows with the number of endpoints being polled by the MCP (even though only one of them is exercised). For messages longer than 128 bytes, there is an additional memory copy within the send call. The cost of this copy, however, shows up only beyond 1K bytes.

The second row gives the time for the MCP to send the header out over the network. The difference with the previous row is insignificant for the small message since it is already on the card SRAM. For the longer messages, the difference is a measure of the cost of a DMA to get the data from host memory.

The third row marks the time for the message to be completely sent out over the network. The difference with the previous row accounts for the cost of network DMA setup and transfer. Again, this is significant only for long messages.

The remaining rows denote operations at the receive end. The fourth row is cumulative until the time the receiver detects the incoming message and DMAs it into the SRAM completely. The reader should note that the receiver LANai is not just waiting for an incoming message, but is also polling its own endpoints for outgoing messages. As a result, an increase in the number of endpoints affects the performance of this operation slightly.

The fifth row shows the time taken for completion of DMA of message data to host memory. Since both short and long messages require a DMA at this step, both show a significant increase from the previous row.

The final row indicates the time when the host has completely received the message in the application. This includes the time to detect the message, copy it to the program specified buffer and update the state variables. The differences with the previous row are almost the same for multiple endpoints since this part of the receive operation is unaffected by the number of active endpoints. For larger messages, the difference with the previous row grows with message size due to the cost of the memory copy within the receive call.

We are currently in the process of evaluating MU-Net with benchmarks to quantify the effect of multiple active endpoints.

## 5 Related Work

Besides U-Net [10] and FM [6], there are other high-performance messaging layers using Myrinet. Since they run on a variety of workstation platforms and have different objectives, it is difficult to use their performance numbers alone as a means of evaluating their design choices. However, it is instructive to examine their implementation experiences to understand the tradeoffs in design alternatives.

The PM messaging library [8] aims at providing multiple users direct access to the network interface hardware. However, the library is used along with a daemon doing gang scheduling of the user processes. As a result, the network interface is used by only two processes at any time, namely the the daemon process and the user process scheduled by it. Though, they have to deal with the issues of these two processes sharing the Myrinet card, protection is not a serious concern since the daemon process is assumed to be a trusted agent. An interesting idea, called Immediate Sending, to overlap network and host side DMA operations is used by PM to enhance performance. Overlapping (pipelining) DMA operations is also used in Trapeze [12] at both the sender and receiver and is referred to as cut-through delivery.

Hamlyn's [3] design objective is to provide varying levels of protection between user processes. Protection is done more rigorously by comparing keys on each message. As in MU-Net, Hamlyn has two versions of send, one using DMA and the other using a memory copy done by the host. Other notable features of the messaging layer are buffer management by the sender, use of zero copy protocols and provision for out of order delivery.

Myricom also provides a low latency, high bandwidth messaging layer called GM [5] in addition to the API discussed in the preceding sections. GM provides multiple user-level accesses to the network interface simultaneously. It also provides automatic mapping of the Myrinet network and provision for two levels of packet priority. Flow control is an important issue and is made visible to the GM API.

## 6 Concluding Remarks and Future Work

This paper has summarized our experiences in implementing MU-Net, a user-level messaging platform for Myrinet that allows protected multi-user access to the network. Our design of MU-Net has drawn from ideas of other user-level platforms [10,6]. It uses the idea of virtualizing the network from U-Net [10] to provide protected multi-user access. However, unlike U-Net, it does not fragment the communication segment into fixed-size buffers, thus being able to avoid multiple DMA transfers for longer messages. The implementation of MU-Net has also benefited from the experiences of [6] in working with the Myrinet hardware. In addition to efficient transfers for short messages as in [6], MU-Net provides a mechanism for transferring longer messages in cases where the host CPU has other work to do.

MU-Net has been implemented on the SUN Solaris 2.5 operating system, and performance results on a Ultra Enterprise 1 platform show that despite being able to support multiple application processes, MU-Net's performance is comparable to other messaging substrates which do not allow protected multi-user access.

## References


1. T. Anderson et al. A case for networks of workstations. *IEEE Micro*, pages 54–64, February 1995.
2. N. J. Boden et al. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
3. G. D. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 245–259, October 1996.
4. Martin de Prycker. *Asynchronous Transfer Mode: solution for broadband ISDN.* Ellis Horwood, West Sussex, England, 1992.
5. Myricom Inc. GM Documentation and Software, 1997. http://www.myri.com/GM/index.html.
6. S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, December 1995.
7. Kendall Square Research. Technical summary, 1992.
8. H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High Performance Communication Library. In *Lecture Notes in Computer Science*, volume 1225, pages 708–717. Springer-Verlag, April 1997. From Proceedings of High-Performance Computing and Networking '97.
9. Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, October 1991.
10. 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.
11. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.
12. K. Yocum, J. Chase, A. Gallain, and A. R. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, August 1997.