# Process Communication on Clusters

Sultan Al-Muhammadi, Peter Petrov, Ju Wang, Bogdan Warinschi

November 21, 1998


## Abstract

*Clusters of computers promise to be the supercomputers of the future. Traditional mechanisms (in particular communication support) are not suitable anymore. Recent research was directed towards developing thin communication layers that exploit the new features of the hardware and that provides to the upper level up to 90% from the underlying hardware capability. Common mechanisms to different approaches include kernel bypassing and "0 copies" memory transfers. Several projects dealing with associated problems exists. We survey four of them.*


*Keywords: Cluster, communication, networking*

## 1 Introduction

A "Cluster" is a set of high speed PCs or workstations, interconnected with high-speed network. With the advance of low-cost computing technology, clusters of PCs and workstations become an attractive alternative to massively parallel processor (MPPs) architectures. Since the computing power of clusters is comparable to MPPs, while the cost of the system is much inexpensive, clusters are today's preferred implementation of distributed-memory MPPs. Such clusters can be expanded and upgraded incrementally as new technology becomes available.

However, a cluster is more than a collection of high performance computing nodes. The important issue here is the way its component parts are integrated. A challenge in building scalable systems using networks of commodity components is to achieve communication performance competitive with or better than custom-designed systems.

The network hardware itself is not the bottleneck anymore. The new high-speed Local Area Networks (LANs) available today (ATM, FDDI, Fibrechannel and Myrinet) offer comparable hardware latency and bandwidth to the proprietary interconnect found on MPPs. But these new hardware technologies are only part of the whole picture. Providing performance to applications requires communication software and operating system support capable of delivering the network performance. Fast network hardware alone is not sufficient to speed up communication. The software overhead becomes the dominant factor in communication, and simultaneously, an important factor of the overall performance of clusters.

Several current research projects are aimed at reducing the software overhead in communication, in order to deliver the high performance the hardware provides to user applications. In this paper, we will survey four leading projects in this area. They are Active Messages (AM) in UC Berkeley, Fast Messages in UIUC/UCSD, U-Net in Cornell University and VMMC in Princeton University.

The common approach is to move the communication layer into user-space in order to minimize the software overhead. However, different schemes to further reduce the communication overhead are used and deifferent approaches in resource management are employed. Comparing these approacehs could prove useful.

The structure of the paper is as follows. In Section 2 we present the key issues in process communication; the projects are introduced in

Section 3; in Section 4 we will attempt a discussion of the four approaches. The paper ends with a Conclusion(Section 5).

## 2 Issues in Communication

In the sequel we will give a short overview of what we consider to be some of the most important issues in process communication.

**Communication protocol** In "traditional" systems, the hardware does not provide reliable message delivery. This enforces the use of sophisticated protocols in order to ensure reliable communication.

Modern network hardware, such as the Myrinet[1], provide reliable data delivery comparable to the memory bus. Such network hardware also has very high bandwidth and extremely low transfer latency. Therefore it is unnecessary and wasteful to employ complicated transfer protocols such as TCP in the communication layer.

Recent research is interested in building a thin communication layer with a simple protocol to fully exploit the reliability and the high performance the hardware provides.

**Kernel involvement in communication** In traditional systems, process communication must go through the kernel. The kernel maintains the resources and provides full protection in communication. The communication protocol stack is also implemented in the kernel.

Even if going through the kernel provides a convenient way of process communication, using it is not efficient anymore. If in the past the overhead associated with the context switch was not significant when compared to the speed of the hardware and of the protocol, in our days a context-switch is a real bottleneck. This is true because if modern hardware can lower the communication overhead down to a few microseconds, a context switch will sometimes take more than 100 microseconds.

All the approaches we will discuss are trying to eliminate the kernel involvement in the critical path of communication. They are trying to build a user-level communication interface, which makes the applications to achieve communication latency and bandwidth very close to those of the available in hardware.

Usually, protection was ensured by the kernel. Trying to bypass the kernel raises new protection issues. We will discuss how some of the projects surveyed here manage this problem.

**Buffer management** Network buffer management is also an important issue in communication system.

When sending data through the network, usually the sender will try to take advantage of the DMA-engine (commonly found on modern network cards). However, the data transfered by the DMA-engine has to be memory resident before the DMA-engine is started.

There are two ways of doing it: One is to allocate a system buffer at system boot time and make that buffer always memory resident. During transmission, user data are copied into that memory resident system buffer. This will introduce an additional copy but is easy to implement.

The alternative is to pin-down some user buffers to prevent them from being swapped out of the main memory. Then the data could be transferred directly from the user space. This will provide "zero-copy" on the sender side to remove the overhead of additional copy, but mechanisms that protect the pinned-down buffer and cooperate with the existing virtual memory subsystem have to be provided. A similar problem arises on the receiver's side.

Interesting solution are given in Princeton's Shrimp project and Cornell's U-Net/MM.

**Design of network interface** Previous-generation commodity network interfaces were

[1] Myrinet is a high speed LAN interconnect which uses byte-wide parallel copper links to achieve physical link bandwidth of 76.3 MB/s. Myrinet uses a network coprocessor (LANai) which controls the physical link and contains three DMA engines (incoming channel, outgoing channel and host) to move data efficiently

simple DMA engines on the I/O bus. With the increasing complexity of high-speed networks, network interfaces are becoming more sophisticated and routinely include an on-board co-processor, although such co-processors usually lag behind the host CPU in performance. This raises the question of how much functionality should be implemented in the network interface itself and how much should be relegated to the foster host.

In the projects we discuss, intermediate models are proposed which transfer just enough "inteligence" into the network interface in order to handle message buffers. This includes message multiplexing/demultiplexing and virtual address translation.

**Small vs. large messages handling** Some research on communication suggests that exchange of small messages is the dominant part in process communication. Therefore, most research focus on optimizing their communication system to get good performance for small messages.

# 3 Projects

In this section we will give overviews of some project which deal with on cluster process communication. For each project we will emphasize the most interesting feature.

## 3.1 Active Messages I

**Overview** The Active Message project has as main goal to provide low overhead communication.

It does so by employing the following mechanism: each message that is sent, contains in its header the address of a *handler*. This handler has as only purpose taking the message off the net and integrate it in the ongoing computation. The sender launches the message and then continues the computation. When the message is received, the receiver interupts the ongoing computation, runs the handler and resumes computation. Communication overlaps computation, since no blocking send/receive is executed. Asynchronous send/receive use a similar mechanism but complicated buffer management is used in order to keep the communication reliable.

The existance of handlers eliminate this drawback. The only requirement is that the receiver user program allocate the necessary space for the incomming data.

**Architecture** The architecture of Active Message I [12] is very simple. It assumes an environment of trusted processes and a SPMD[2] model of computation. Moreover, there is at most a process running on each processor. The key optimizaion is the elimination of buffers. Eliminating buffering on the receiving end is possible because either storage for arriving data is pre-allocated in the user program or the message holds a simple request to which the handler can imediately reply. If messages are small in size, then buffering in the network is sufficient for the sender's side.

Another optimization that was used concerned the polling mechanism. In the case that a process would not engage in a long computation then it would poll for incomming messages only when a send was executed.

A *request-reply* mechanism is used.

A primitive scheduling scheme is used. The handlers interupt the computation immediately after a messsage's arrival and execute to completion.

The AM were initially design to fit the available message passing architectures [3] As fast networking emerged, the paradigm has been extended as to encompass a broader range of supported applications.

## 3.2 Active Messages II

**Overview** The purpose of Active Message II [1] is to generalize the active message

[2]Single-Program Multiple-Data

[3]the implementation has been done on nCUBE and CM-5

techonolgy in order to be able to cope with a broader range of appliction and still preserve as much as possible from the underlaying hardware capabilities. In order to achieve this, the Active Message layer will present to the upper layer an uniform abstraction consisting of a set of *end-points*. A set of basic communication primitives are also provided, such that efficient implementation is possible. A protection model based on names and tags is used.

**Architecture** A process can create and manage several end-points. Each communication end-point has the following components: a send pool, a receive pool, a handler table (which is used to translate indices into functions), a virtual memory segment for memory transfers, a translation table that associates indices with global-endpoint names and tags, and finally, a tag for authenticating messages arriving at the endpoint. An active message is sent from an endpoint send pool to an endpoint receive pool. It carries an index into a handler table that selects the handler function for the message. Upon receiving, a request-handler is invoked. Similarly, when a reply message is sent, a reply-handler is run. The bulk data transfer functions copy memory from a sender's virtual address space to receiving endpoint's receive pool or a virtual memory segment and deliver an associated active message when the transfer is complete.

The request-reply scheme is inherited from AM I.

**Protection** End-points have globally unique name within the system. Since these names need not be easily manipulated or convinient represented, end-point translation tables are used in order to indirect these names. This way, the interface specification remains independent of external name servers, although implementations should make sure that an external agent(s) exists in order to manage mappings of global-endpoint names.

Tags are the second element that participate in the protection model. As with the end-point names, tags are associated with end-points. Implicitlly, tags can identify sets of communicating end-points. One benefit that comes with the use of tags is that applications can identify aggregates of end-points unambiguosly. Secondly, tags provide a simple means of authentification.

At any point the process that created an end-point can change its tag. This is equivalent with revoking capability, since messages can be sent to a process only if the right (processes' current) tag is attached to the message.

Managing end-points and tags is left to the implementer's convenience. In particular, passing tags can be done by any of the standard procedures for randevousz (like using a shared file system, or previously agreed-on file).

### 3.3 U-Net

**Overview** The U-Net [11] architecture provides low-latency and high-bandwidth communication over commodity networks of workstations and PCs. It achieves this by virtualizing the network interface such that every application can send and receive messages without operating system intervention. With U-Net, the operating system is removed from the critical path of communication. This allows communication protocols to be implemented at user-level where they can be integrated tightly with the application. In particular, the large buffering and copying costs found in typical in-kernel networking stacks can be avoided. Multiple applications can use U-Net at the same time without interfering.

Incorporating memory management into the user-level network interfaces to enable direct delivery of messages to and from user-space by the network interface while observing the traditional protection boundaries between processes is the main feature of U-Net [2].

**Architecture** The U-Net user-level network interface architecture virtualizes the interface in such a way that a combination of operating system and hardware mechanisms can provide every process the illusion of owning the network interface. Depending on the sophistication of the actual hardware, the U-Net components manipulated by a process may correspond to real hardware in the NI (network interface), to memory locations that are interpreted by the OS, or to a combination of the two. The role of U-Net is limited to multiplexing the actual NI among all processes accessing the network and enforcing protection boundaries as well as resource consumption limits. The main building blocks of U-Net architecture are endpoints. Endpoints serve as an application's handle into the network and contain three message queues, which hold descriptors for message buffers. Each endpoint is associated with a buffer area that might be *pinned* to contiguous physical memory and holds all buffers used with that endpoint.

**Memory Management Mechanisms** The pinned-down buffer in endpoints enables direct message transfer to or from user-space. The problem is that since the size of phisycal memory is limited the number of buffers pinned-down at the same time is also limited. In order to overcome this limitation, U-Net incorporates some memory management mechanisms into the network interface to page-in/out network buffers. This approach also increases the scalability in the number of concurrent network applications by allowing buffers belonging to inactive applications to be paged out.

In order to handle arbitrary user-space virtual addresses, the U-Net design incorporates a Translation Look-aside Buffer (TLB, which is not directly visible from user-space). The TLB is maintained by the network interface itself and maps (process ID, virtual address) pairs to physical page frames and read/write access rights. The network and the operating system cooperate in handling TLB misses and TLB coherency issues to ensure that the valid entries in the TLB correspond to the pages pinned-down in the network buffer.

The technical details about TLB management can be found in [2].

### 3.4 Fast Messages

**Overview** The Fast Messages (FM) is a communication library (layer). It has as primary goal delivery of the hardware performance to the application level. Another goal is to develop a library suitable for tightly coupled workstation clusters [9].

Two versions of the FM have been designed and implemented. FM 1.0 [10] has been based on the studies of essential communication guarantees (reliable, in-order communication with flow control). It was also optimized for realistic message size distribution(especially short messages).

The main drawback of this version was degraded performance when building high-level libraries (such as MPI[4]) on top of FM. The performance losses caused by the interface have been remarkable(network performance was limited to less than 10% from hardware's capability). The overhead originated from a number of memory-to-memory copies of the data, taking place at the interface between the libraries.

The second version of FM [8] eliminates this problem. It enables over 90% of hardware's performance to be delivered to the higer level.

Several new features were added.

**Architecture** FM differs from a pure message paradigm by not having explicit receiver. Instead, each message includes the name of a handler, which is a user-defined function that is invoked upon message arrival. FM provides buffering allowing senders to make progress while their corresponding receivers are computing and not servicing the network.

The FM interface is similar to the Active Messages(see Section 3.1) model from which it borrows the notion of message handlers. However a number of key differences exist: the FM API offers stronger guarantees (in particular in-order delivery), uniform handling of messages with respect to size, and it does not follow a rigid request-reply scheme. The FM's send calls do not normally process incoming messages (in con-

[4]Message Passing Interface is an industrial standard for writing "portable" message passing parallel programs

trast to Active Messages), enabling a program to control when the received data is processed.

Using Myrinet, FM provides MPP-like communication performance on workstation cluster. The addition of flow control and buffer management has been enough to provide reliable and in-order delivery.

Different investigations [7] showed that implementing strong guarantees built on top of a message library can increase communication overhead by over 200%.

To reduce these costs, the designers of FM have given full consideration of how to exploit the hardware features. The basic features are *gather/scatter, layer interleaving*, and *receiver flow control.*

*Stream abstraction*

This is the basic mechanism that is provided by the FM library. Messages are viewed as byte streams. Appropriate primitives to send and receive data, such as *FM_send_piece(.)* and *FM_receive_piece(.)* are provided.

Note that messages are interpreted as a byte stream, instead of a single contiguous region of memory.

*Gather/Scatter*

It is implemented using the stream abstraction by succesively calling FM_send_piece(.). The receiver has to use a sequence of FM_receive_piece(.).

These two features can be very efficiently implemented using programmable I/O[5].

*Layer Interleaving*

The second benefit from stream abstraction is the ability for controlled interleaving between FM's and application's threads. This makes it possible to eliminate staging buffers for exchanging the message data between communication layers, i.e. after getting the header of the message, the upper level can determine the destination message queue. Therefore it can provide the destination buffer (part of the message queue) to the FM which can directly transfer the data.

*Receiver Flow Control*

This allows the receiver to control the rate at which data is processed from the network. In many applications, the ability to intentionally delay the extraction of the message until a buffer becomes available simplifies the buffer management.

*Transparent Handler Multithreading*

The handler execution is not delayed until the entire message has arrived, rather, it is started as soon as the first packet is received. Since packets belonging to different messages can be received interleaved, the execution of several handlers can be pending at a given time. As it extracts each packet from the network, FM schedules the execution of the associated pending handler. By having the interleaved packet reception transparently drive the handler execution, a number of benefits are achieved. Combined with the stream abstraction this allows arbitrary sized data chunks to be composed/received, without any concern for packet boundaries. Handler multithreading plus packetization not only simplifies resource management, it can also increase performance by increasing effective pipelining.

## 3.5 VMMC

**Overview** Virtual memory-mapped communication (VMMC) is a communication model that allows user-to-user data transfers with latency and bandwidth close to the limits imposed by the underlying hardware. VMMC [5, 6] has been designed and implemented for the SHRIMP multicomputer with Myrinet. The basic idea is to allow data to be transmitted directly from a source virtual memory to a destination virtual memory. This aproach eliminates kernel involvement in data transfer.

It also provides full protection in multi-user environment. Other included features are user-level buffer management and zero-copy protocols. It minimizes host processor overhead when sending and there is no processor overhead when receiving. The model has been extended to include three mechanisms: user-managed TLB (UTLB) for address translation, transfer redirection, and reliable communication at the data link

[5]An input/output interface that can be programmed to do network transfer, using host CPU

layer.

**Architecture** On each host, there is a local VMMC daemon. User programs submit *export* and *import* requests to a local VMMC daemon. Daemons communicate with each other over Ethernet to match export and import requests and establish export-import mapping.

Protection is offered since it such a mapping can be established only when the receiver gives the sender the permission to send data. Data is sent to a designated area in the receiver's space.

The followings summarize the basic steps in communication:

1. The receiver *export* a portion of its address space as *receive-buffers* to accept incoming data.

2. The sender *import* remote receive-buffers as destination for transferred data.

3. Sender can transfer data from its virtual memory space to the imported buffer.

4. Security of the receiver's address space is enforced by VMMC.

5. The receiver may restrict possible-importers of a buffer.

6. This restriction is enforced by VMMC when an import is attempted.

There are two data transfer modes supported by VMMC: deliberate update and automatic update. Deliberate update is an explicit request to transfer data blocks from anywhere in the caller's space to an imported buffer. While automatic update is implicitly done on each write to local memory the automatic update is performed as follows: the sender creates automatic-update mapping(a region of the sender's vertual machine is mapped to a remote receive-buffer). All sender's writes to its automatic-update region are automatically transferred to the remote buffer. The VMMC subsystem infers the destination address from the local address specified by the sender (in the write operation). Note that the automatic-update destination associated with a given local address must be unique.

Received messages are transferred directly into the memory of the receiving process without interrupting the receiver's CPU; i.e. no explicit receive operation in VMMC.

To establish an import-export mapping, the receiving process exports a part of its address space as receive-buffers. The sender import the remote receive-buffers as destination for transfer data. The receive-buffer is mapped into the sender's destination proxy space.

Once it is established, the import-export mapping can be destroyed by either the receiver or the sender. A sender calls *unimport* if it no longer wants to transfer data to a given receive-buffer. Then the range of the sender's destination proxy space become free and available for another imported buffer. Alternatively, a receiver calls *unexport* to close a given receive buffer. This call destroys all import mappings to this buffer.

### 3.6 The VMMC-2 Model

The VMMC-2 [4] extends the basic VMMC model with three more mechanisms: User-managed TLB (UTLB) for address translation, a transfer redirection mechanism which avoids copying on the receiver's side and a reliable communication protocol at the data link layer which avoids copying on the sender's side.

The UTLB is used for address translation which enables user libraries to dynamically manage the amount of pinned space and requires only driver support from many operating systems. The UTLB consists of an array for every process holding physical address of pages belonging to this process virtual memory portions that are pinned in the host physical memory.

Transfer redirection mechanism is used to avoid copying on the receiver's side. The idea is to use a default redirectable receive-buffer addresses. The sender always sends to this default buffer. Then the redirection mechanism checks for a redirection address given by the receiver, and puts the message into the user buffer directly from the network without any copying.

Reliable communication at the data link layer is provided by VMMC-2 to deal with transient network failures, such as: CRC errors, corrupted packets, and all network fabric errors. If a transient network failure becomes permanent, the remote node is declared unreachable, imported buffers from that node are invalidated, the user is notified, and all packets that cannot be delivered are dropped. The user needs to reestablish the mappings in order to resume the communication with the remote node.

The performance is achieved by doing communication via the mapped receive-buffer. After mapping the receive buffer, the sender can transfer data without having to check the availability of the receive buffer space.

## 4 Discussion

We now discuss these different projects in the light of their design goals, decisions, and assumptions.

**Small vs. large message** It appears desirable for a communication layer to provide efficient massage passing for both short and long messages. This is not necessarily true. Some research shows that small messages are predominant in the network traffic. Therefore some project focus on optimizing the communication performance for short messages. Active Message I is a specialized library (it is targeted at scientific parallel computing). This allows it to achieve high efficiency when transferring small messages. It is not optimized for large messages. On the other hand, Fast Messages approach is tailored as to cope with small data. However, the stream abstraction facilitates efficient large message handling.

**Buffers and copy overhead** In Active Message I buffering is not provided. However, upon the arrival of a message, it is either incorporated in the ongoing computation or it is stored in preallocated user space. Fast Messages do provide buffering. The presence of efficient flow control allows reasonable small buffer size while decreasing the cost of buffer management.

U-Net messaging layer provides buffer management, demultiplexing in hardware but no flow control, and thus data can be lost due to overflow. Therefore, retransmission mechanisms should be employed thus increasing the complexity of the protocol.

On the other hand VMMC provides direct delivery of messages from and to user space buffers. As a consequence, no network buffering is needed.

Secondly, avoiding additional copy during data transfer is an important way of reducing communication overhead. There are several ways of doing it. Modern network interface, such as Myrinet, has two ways of data transfer. One is to use DMA engine. The DMA engine can only access memory resident area, but it can achieve high bandwidth and does not require involvement of host CPU. The problem is that you have to make sure the network buffer is memory resident when the transfer begins.

VMMC and U-Net use this approach. In VMMC, the communication buffers are pinned-down by user's explicit request. After the data path is setup, data can be transferred directly to/from user-space buffers without additional copy. However, the total size of pinned-down pages could not be very large, since the size of physical memory is limited. This implies that the scalability of this approach is not satisfying.

U-Net incorporates a TLB into the network interface to track the pinned-down pages in the virtual memory. That makes it possible to dynamically pinned-down user-space pages. Although the TLB mechanism in this approach is not sophisticated enough for achieving real dynamically pinned-down buffers, this scheme gives an efficient way of pinning-down pages and the scalability of the system is greatly enhanced.

In fact, most recent version of VMMC [3] also incorporate some more sophisticated TLB (UTLB) into their system and further realized "dynamically pinned-down pages".

The other choice is to use programmable I/O operation. The host CPU can directly write data into the network registers and transfer them directly without additional copy. However, it means that host CPU has to be involved in sending data, and the bandwidth can not be as good as using DMA engine. The advantage of using programmable I/O is that it can achieve relatively low latency for small messages(the cost of starting the DMA engine is relatively high).

The FM layer uses this approach 3.4. This is not benefitial to the large size message transmission but it reduces short message communication overhead.

FM uses some stream and layer-interleaving mechanisms to enhance the performance of large messages. This is a trade-off between host CPU usage and short-message latency.

**Kernel involvement and protection** As described above, these projects have similar goals - they all try to build a thin communication layer on top of the network hardware, in order to fully exploit the performance the hardware provides. Since the cost of crossing the kernel/user boundary is relatively high, all of them are moving the communication layer into the user-space in order to avoid the OS kernel involvement on the critical path of communication. However, the protection boundary that used to be maintained by the operating system should now be well preserved. Therefore, in each approach, there is some scheme of doing it. In Active Messsage I, a trusted environment is assumed(associated with single process per processor), therefore no protection scheme is used. AM II and U-Net visualize the network resource as endpoints, which maintain the protection boundaries among user processes. In Active Message II the protection is also ensured by using an external mechanism to manage naming schemes and tags. In VMMC, the protection is ensured by the OS and the network interface. Checks of buffer boundary and access rights provide the basic protection scheme.

While in FM, there is no full protection among multiple processes. In the recent version of FM-II, the network resources are divided into two parts and make the two running processes invisible of each other's resource. Therefore, only two communication processes are allowed on the same host at the same time.

# 5 Conclusion

In this paper we presented several approaches for achieving efficient interprocess communication inside clusters. The four projects we surveyed: Active Messages, Fast Messages, U-Net and VMMC take different approaches to deliver the performance the hardware provides to the user applications. We considered that buffers and copy overhead, size of the messages and kernel involvement are the most influential factors in building an efficient user space communication layer. In the discussion part we tried to list pros and cons for using each of these. Each of these projects have advantages and disadvantages. The optimal approach might incorporate hybrid techniques.

# References


[1] A.M.Mainwaring and D.E.Culler. Active message applications programming interface and communication subsystem organization. Technical report, U.C. Berkeley, 1996.

[2] Anindya Basu, Matt Welsh, and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. In *Hot Interconnects V*, 1997.

[3] Yuqun Chen, Angelos Bilas, Stefanos Damianakis, Cezary Dubnicki, and Kai Li. Utlb: A mechanism for address translation on network interfaces. Technical Report 580-98, Princeton University, 1998.

[4] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefano Damianakis, and Kai Li. Vmmc-2: Efficient support for reliable, connection-oriented communication. In *Hot Interconnects V, August 1997*, 1997.

[5] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philibin. Design and implementation of virtual memory mapped communication on myrinet. In *Proceedings of 11th International Parallel Processing Symposium*, 1997.

[6] Cezary Dubnicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software support for virtual memory-mapped communication. In *10th International Parallel Processing Symposium*, 1996.

[7] V. Karamcheti and A. Chien. Software overhead in messagin layers: Where does the time go? In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Opperating Systems*, 1994.

[8] Mario Lauria, Scott Pakin, and Andrew Chien. Efficient layering for high speed communication: Fast messages 2.x. In *High Performance Distributed Computation*, 1998.

[9] Scott Pakin and Andrew Chien. Fast messages (fm): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurency*, 5(1), 1997.

[10] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Ilinois fast messages (fm) for myrinet. In *Supercomputing'95*, 1995.

[11] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[12] Thorsten von Eicken, David Culler, Seth Goldstein, and Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.