High Performance User Level Sockets over Gigabit Ethernet[*]

Pavan Balaji[*] Piyush Shivam[*] Pete Wyckoff[†] Dhabaleswar Panda[*]

[*]Computer and Information Science
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210
{balaji, shivam, panda}@cis.ohio-state.edu

[†]Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
Phone: 614 247 7956
pw@osc.edu

Abstract

While a number of User-Level Protocols have been developed to reduce the gap between the performance capabilities of the physical network and the performance actually available, applications that have already been developed on kernel based protocols such as TCP have largely been ignored. There is a need to make these existing TCP applications take advantage of the modern user-level protocols such as EMP or VIA which feature both low-latency and high bandwidth. In this paper, we have designed, implemented and evaluated a scheme to support such applications written using the sockets API to run over EMP without any changes to the application itself. Using this scheme, we are able to achieve a latency of 28.5 μs for the Datagram sockets and 37 μs for Data Streaming sockets compared to a latency of 120 μs obtained by TCP for 4-byte messages. This scheme attains a peak bandwidth of around 840 Mbps. Both the latency and the throughput numbers are close to those achievable by EMP. The ftp application shows twice as much benefit on our sockets interface while the web server application shows up to six times performance enhancement as compared to TCP. To the best of our knowledge, this is the first such design and implementation for Gigabit Ethernet.

Keywords: *Gigabit Ethernet, Sockets, User-level protocol, Interprocessor Architecture*

1 Introduction

Networks of Workstations (NOWs) have been accepted as a viable alternative to mainstream supercomputing for a broad subset of computation intensive applications. Much of the success of these NOWs lies in the use of commodity based components, giving a high ratio of performance to cost to the end users. With the advent of modern high speed interconnects such as Myrinet [6], Gigabit Ethernet [10] and Quadrics [17], the bottleneck in the communication has shifted to the messaging software at the sending and the receiving side.

Earlier generation protocols relied upon the kernel for processing the messages. This caused multiple copies and many context switches [24]. Thus, the communication latency was high. Researchers have been looking at the alternatives by which one could increase the communication performance delivered by the NOWs in the form of low-latency and high-bandwidth user-level protocols such as FM [16] for Myrinet [6], U-Net [25] for ATM and Fast Ethernet, GM [8] for Myrinet, and others [26, 5, 27, 20, 18].

In the past few years, several industries have taken up the initiative to standardize high-performance user-level protocols such as the Virtual Interface Architecture (VIA) [11, 12, 7, 3, 19, 2, 4]. It has also led to the development of the InfiniBand Architecture (IBA) [1]. These developments are minimizing the gap between the performance capabilities of the physical network and that obtained by the end users.

One such physical network which is of particular interest is Gigabit Ethernet [10] as most of world's networks today use Ethernet. Gigabit Ethernet offers an excellent opportunity to build Gbps networks over the existing Ethernet installation base due to its backward compatibility with Ethernet. However, the user applications have not been able to take advantage of the high performance of Gigabit Ethernet because a vast majority of them still use the sockets interface, which has traditionally been implemented on kernel-based protocols like TCP and UDP.

[*] This research is supported by Sandia National Labs (contract number 12652 dated 31 Aug) and NSF grant # EIA-9986052

One way to get around this problem would be to develop a very low overhead user-level protocol similar to VIA over Gigabit Ethernet. This motivated us towards the development of Ethernet Message Passing (EMP) protocol [22, 23], using which the applications can take full advantage of the bandwidth offered by Gigabit Ethernet with minimum latency. While this approach is good for writing new applications, it might not be so beneficial for the already existing socket applications which were developed over a span of several years.

Sockets is a generalized library which can be implemented over numerous protocols. In this paper, we take on a challenge of developing a low overhead, user-level sockets interface on Gigabit Ethernet which uses EMP as the underlying protocol. There is no exact parallel between EMP and TCP or UDP. We analyze the semantic mismatches between the two protocols like connection management and unexpected message arrival to name a few. To capture these differences, we suggest various approaches for two commonly used options with sockets, namely data streaming and datagram. Finally, we suggest several performance enhancement techniques while providing these options and analyze each of them in detail.

Using our approach one will be able to transport the benefits of Gigabit Ethernet to the existing sockets application without necessitating changes in the user application itself. Our sockets interface is able to achieve a latency of 28.5 μs for the Datagram sockets and 37 μs for Data Streaming sockets compared to a latency of 120 μs obtained by TCP for 4-byte messages. We also attained a peak bandwidth of around 840 Mbps using our interface. In addition we tested our implementation on real applications like ftp and web server. For ftp we got almost twice the performance benefit as TCP while the web server application showed as much as six times performance enhancement.

The remaining part of the paper is organized is follows. Section 2 deals with the overview of EMP. In section 3, we discuss the current approaches to sockets over user-level protocols. Sections 4 and 5, discuss about the design issues and the various challenges faced during the implementation of our socket interface. Section 6 mentions the various techniques used to improve the performance delivered by our sockets interface. The experimental test-bed and the performance evaluation of the substrate are given in Section 7. Conclusions and future work are outlined in Section 8.

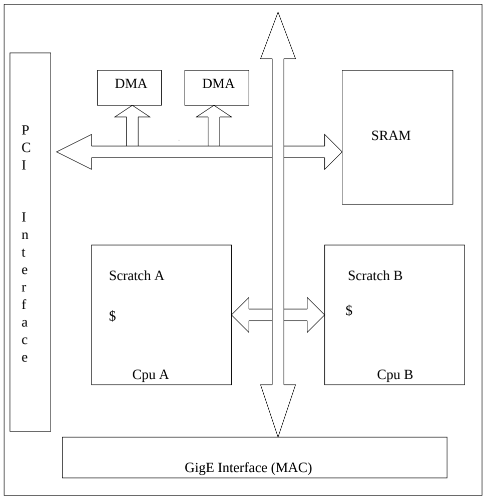

Figure 1: The Alteon NIC Architecture

2 Overview of EMP

In the past few years, a large number of user-level protocols have been developed to reduce the gap between the performance capabilities of the physical network and that achievable by an application programmer. The Ethernet Message Passing (EMP) protocol specifications [22, 23] have been developed at The Ohio Supercomputing Center and The Ohio State University to fully exploit the benefits of Gigabit Ethernet.

EMP is a complete zero-copy, OS-bypass, NIC-level messaging system for Gigabit Ethernet (Figure 2). This is the first protocol of its kind on Gigabit Ethernet. It has been implemented on a Gigabit Ethernet network interface chip-set based around a general purpose embedded microprocessor design called the Tigon2 [15] (produced by Alteon Web Systems, now owned by Nortel Networks). This is a fully programmable NIC, whose novelty lies in its two CPUs. Figure 1 provides an overview of the Alteon NIC architecture.

In EMP, message transmission follows a sequence of steps (Figure 2). First the host posts a transmit descriptor to the NIC (T1), which contains the location and length of the message in the host address space, destination node, and an MPI [9] specified tag. Once the NIC gets this information, it DMAs this message from the host (T5), one frame at a time, and sends the frames on the network. Message reception follows a similar sequence of steps with the difference that the target memory location in the host for incoming messages is determined by performing tag matching at the NIC (R4). Both the source index of the sender and an arbitrary user-provided 16-bit tag are used by the NIC to perform this matching, which allows EMP to make progress on all messages without host intervention.

EMP is a reliable protocol. This mandates that for each message being sent, a transmission record be maintained (T3). This record keeps track of the state of the message including the number of frames sent, a pointer to the host data, the sent frames, the acknowledged frames, the message recipient and so on.

Similarly, on the receive side, the host pre-posts a receive descriptor at the NIC for the message which it expects to receive (R1). Here, the state information which is necessary for matching an incoming frame is stored (R4). Once the frame arrives (R3), it is first classified as a data, header, acknowledgment or a negative acknowledgment frame. Then it is matched to the pre-posted receive by going through all the pre-posted records (R4). If the frame does not match any pre-posted descriptor, it is dropped. Once the frame has been correctly identified the information in the frame header is stored in the receive data structures for reliability and other bookkeeping purposes (R4). For performance reasons, acknowledgments are sent for a certain window size of frames. In our current implementation, this was chosen to be four. Once the receive records are updated, the frame is scheduled for DMA to the host using the DMA engine of the NIC (R6).

EMP is a zero-copy protocol as there is no buffering of the message at either the NIC or the host, in both the send and receive operations. It is OS bypass in that the kernel is not involved in the bulk of the operations. However, to ensure correctness, each transmit or receive descriptor post must make a call to the operating system for two reasons. First, the NIC accesses host memory using physical addresses, unlike the virtual addresses which are used by application programs. Only the operating system can make this translation. Second, the pages to be accessed by the NIC must be pinned in physical memory to protect against the corruption that would occur if the NIC wrote to a physical address which no longer contained the application page due to kernel paging activity. We do both operations in a single system call (T2, R2), and subsequent operations on the same memory areas do not require another trip through the operating system since they are already pinned in memory. We use a translation cache which satisfies subsequent calls without invoking the operating system. One of the main features of this protocol is that it is a complete NIC based implementation. This gives maximum benefit to the host in terms of not just bandwidth and latency but also CPU utilization.

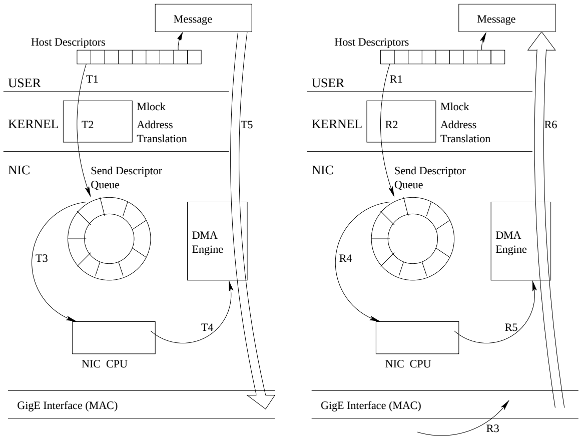

Figure 2: EMP protocol architecture showing operation for transmit (left), and receive (right).

3 Current Approaches to Sockets over User-Level Protocols

The traditional communication architecture involves just the application and the libraries in user space, while protocol implementations such as TCP/UDP, IP, etc reside in kernel space [13] (Figure 3). This approach not only entails multiple copies for each message, but also requires a context switch to the kernel for every communication step, thus adding a significant overhead. Most of the current NIC drivers, including the standard Acenic driver on Alteon NICs, use this style of architecture.

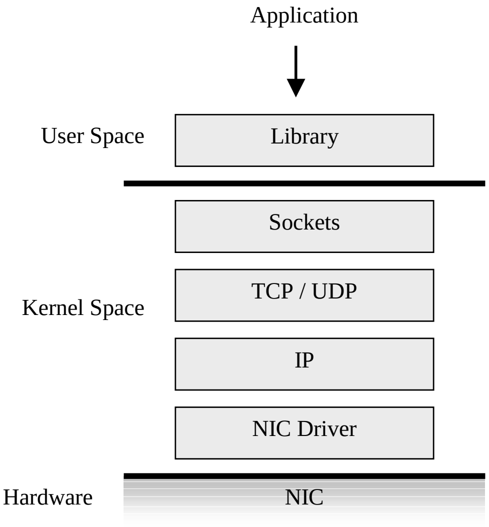

Figure 3: Traditional communication architecture.

User-level protocols like VIA were developed to solve the problem of multiple copies and many context switches. However, to run the existing sockets applications one would need to either rewrite the applications using VIA API or provide a sockets interface to VIA to run these applications without modifications. Since rewriting of these applications was cumbersome, researchers came out with many approaches for providing a sockets interface to VIA.

Most of the these approaches were designed to provide a simple mapping from the traditional sockets interface to the user-level protocol. This provided compatibility without taking into account the performance issues. One such approach was used by GigaNet Incorporation (now known as Emulex) to develop their LAN Emulator (LANE) [11] driver to support the TCP stack over their VIA-aware cLAN cards.

The LANE driver supplied by GigaNet for its cLAN adapters used a simple approach. They provide a IP-to-VI layer which maps IP communications onto VI NICs (Figure 4). However, TCP is still required for reliable communications, multiple copies are necessary, and the entire setup is in the kernel as with the traditional architecture outlined in Figure 3. Although this approach gives us the compatibility, it does not give any performance improvement.

Some other socket implementations over VIA [13, 21] take good advantage of the user-level protocol but the motivation for our work is to provide a high performance sockets layer over Gigabit Ethernet given the advantages associated with Gigabit Ethernet. M-VIA [14], while providing a VIA interface over Gigabit Ethernet, is a kernel-based protocol and hence the current sockets interface over VIA will not be able to exploit the benefits of Gigabit Ethernet. To the best of our knowledge EMP is the only complete OS-bypass, zero-copy and NIC-driven protocol over Gigabit Ethernet. Thus, we focus our research on the EMP protocol.

To be able to take advantage of the high performance offered by EMP, two important changes are required from the traditional sockets implementation. First, the TCP and IP layers must be removed to avoid message copies, which requires implementing a sockets library directly on EMP. Second, the entire interface library must exist in user space, to avoid the additional context switch to the kernel for every communication, in essence removing the kernel from the critical path.

The solution proposed in this paper creates an intermediate layer which maps the sockets library onto EMP. This layer ensures that no change is required to the application itself. This intermediate layer will be referred to as the "EMP Substrate". Figure 5 provides an overview of the proposed Sockets-over-EMP architecture.

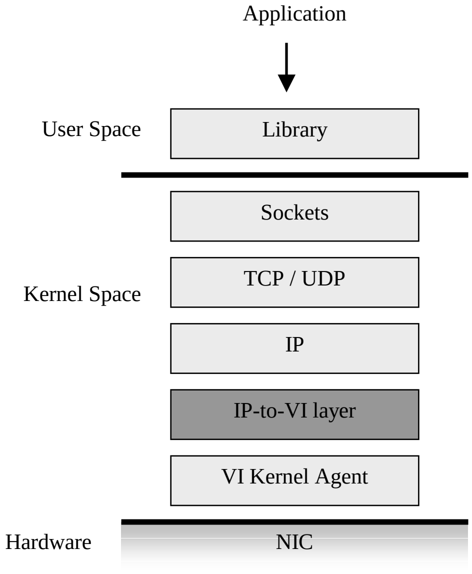

Figure 4: LANE IP-to-VI Architecture

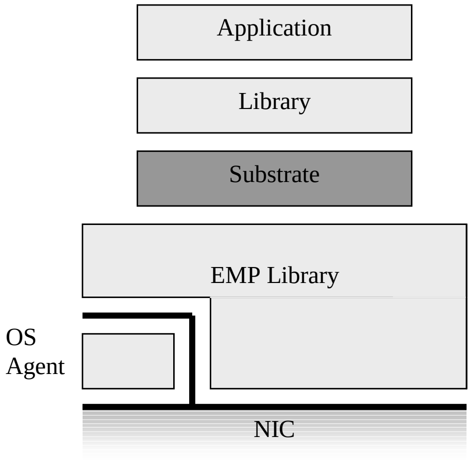

Figure 5: Sockets-over-EMP architecture

4 Design Challenges

While implementing the substrate to support sockets applications on EMP, we faced a number of challenges. In this section, we mention a few of them. In the next section, we discuss the possible alternatives, the pros and cons of each of the alternatives and the justifications behind the solutions.

4.1 API Mismatches

The mismatches between reliable sockets (namely TCP) and EMP are not limited to the syntax and semantics alone. The motivation for developing TCP was to obtain a reliable, secure and fault tolerant protocol. However, EMP was developed to obtain a low-overhead protocol to support high performance applications on Gigabit Ethernet.

While developing the EMP substrate to support applications written using the sockets interface (on TCP and UDP), it must be kept in mind that the application was designed around the semantics of TCP. We have identified the following significant mismatches in these two protocols and given solutions so as to maintain the semantics for each of the mismatches with regard to TCP. More importantly, this has been done without compensating much on the performance given by EMP.

4.1.1 Connection Management

TCP is a connection based protocol, unlike EMP. At first sight, this does not appear to be too much of a problem as the connection can always be assumed to be present. However, by doing so, we overlook certain essential features of the connection requests. Together with requesting a remote node for a connection, the connection request in TCP contains additional information such as the address of the requesting client. This feature would be lost if we assume that a connection is already present.

We have outlined the solution alternatives possible for this problem in more detail in section 5.1

4.1.2 Unexpected message arrivals

Like most other user-level protocols, EMP has a constraint that before a message arrives, a descriptor must have been posted so that the NIC knows where to DMA the arriving message. However, EMP is a reliable protocol. So, when a message arrives, if a descriptor is not posted, the message is dropped by the receiver and eventually retransmitted by the sender. This facility relaxes the descriptor posting constraint to some extent. Posting a descriptor before the message arrives is not essential for the functionality, but is crucial for performance issues. For this problem, we have proposed two feasible solutions, giving the application user a choice to choose either of them.

One of the high points of TCP is its data-streaming option. In this option, the message boundaries supplied by the transmitter are not enforced at the receiver. When a message arrives, the receiving node has the option of reading any number of bytes at any time. In EMP, when a message arrives, the data is directly transferred to the user space, and thus this option is disabled.

In our substrate, a temporary buffer is optionally used where the data is stored as soon as it arrives. These solutions are discussed in more detail in section 5.2

4.2 Resource Management

EMP does not have a garbage collection mechanism – every descriptor is required to be either used for a message or explicitly unposted by the application. Our implementation must carefully account for this resource to avoid leaking NIC resources on socket open() and close() operations. This has been discussed in more detail in Section 5.3.

4.3 Overloading function name-space

Applications built using the sockets interface use a number of standard UNIX system calls including specialized ones such as listen(), accept() and connect(), and generic overloaded calls such as open(), read() and write(). The generic functions are used for a variety of external communication operations including local files, named pipes and other devices. Mapping these calls by directly overriding them does not work due to their multiple interpretations.

In section 5.4, we have discussed a number of solutions for doing these mappings in order to maintain the compatibility with the UNIX sockets.

5 Alternative Solutions for the Substrate

Having seen the mismatches between TCP and EMP, we have come up with solutions that the substrate can use to patch these mismatches without making any changes to the application itself. The following few subsections state the solution options for some of the mismatches and the final solution adopted.

5.1 Connection Management

As mentioned earlier, TCP is a connection based protocol, unlike EMP. We consider two approaches to solve the connection management problem.

Null Functions: In this approach, we return success without performing any communication since the connections are already present. In essence, the connect(), accept(), etc., calls are just null functions. However, this approach has a number of flaws. In TCP, when a connection request is sent to the server, it contains information about the client requesting the connection. In this approach, this information is not available since there's no explicit message for the connection.

Data Message Exchange: In this approach, the client sends an explicit message to the server containing information about the client requesting the connection. However, this puts an additional requirement on the substrate to post descriptors for the connection management messages too. When the application calls the listen() call, the substrate posts a number of descriptors equal to the usual sockets parameter of a backlog which limits the number of connections that can be simultaneously waiting for an acceptance. When the application calls accept(), the substrate blocks on the completion of the descriptor at the head of the backlog queue. In this approach, we need to distinguish connection messages from data messages, for which we used the tag matching facility provided by EMP. This approach has been adopted in our substrate.

5.2 Unexpected Message Arrivals

As mentioned previously, since EMP is a reliable protocol, handling unexpected messages is not essential for functionality. However, allowing the nodes to retransmit packets indefinitely might congest the network and harm performance. To avoid this, we instead explicitly handle unexpected messages at the substrate, and avoid these retransmissions. We examined three separate mechanisms to deal with this.

Separate Communication Thread: In the first approach, we post a number of descriptors on the receiver side and have a separate communication thread which watches for descriptors being used and reposts them. This approach was evaluated and found to be too costly. With both threads polling the synchronization cost of the threads themselves comes to about 20 μs. Also, the effective percentage of CPU cycles the main thread can utilize would go down to about 50%, assuming equal priority threads. In case of a blocking thread, the Operating System scheduling granularity makes the response time too coarse (order of milliseconds) for any performance benefit.

Rendezvous Approach: The second approach was through rendezvous communication with the receiver as shown in Figure 6. Initially, the receive side posts a descriptor for a request message, not for a data message. Once the sender sends the request, it blocks until it receives an acknowledgment. The receiver on the other hand, checks for the request when it encounters a read() call, and posts two descriptors – one for the expected data message and the other for the next request, and sends back an acknowledgment to the sender. The sender then sends the data message.

Effectively, the sender is blocked till the receiver has synchronized and once this is done, it is allowed to send the actual data message. This adds an additional synchronization cost in the latency. Note that the acknowledgment message used in this approach (and the next approach), is separate from the acknowledgment message used by the EMP protocol for reliability. The acknowledgment used by EMP is generated and consumed by the NICs and never seen by the host. The acknowledgments described in this approach are for flow-control and are generated by the application library.

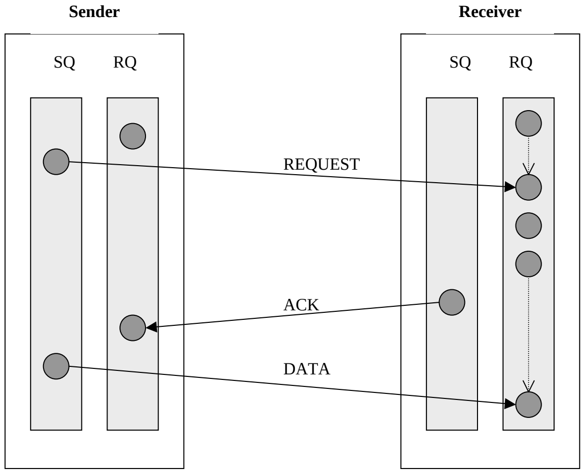

Figure 6: Rendezvous approach

Though this approach is straight forward, it has a few disadvantages. TCP supports data-streaming. For example, if the sender sends 10 bytes of data, TCP allows the user to read it as two sets of 5 bytes each, potentially into different user buffers, which would not be possible in this approach.

Another disadvantage is a possibility of deadlocks. When the sender wants to send a data message, it first sends a request and waits for the acknowledgment, which is sent only when the receiver encounters a read() call. Consider the case when both the nodes want to send data to each other. Both might call write() and then read(), in that order. Sockets over a normal TCP implementation allows this to some extent as the system provides temporary buffer space, generally 32 Kbytes, which can be used to satisfy the write operations and allow the matching reads to proceed. Using this rendezvous approach, however, both nodes would block waiting for acknowledgment and deadlock (Figure 7). However, rendezvous is a standard technique used by most message passing layers (including MPI), which do not give any guarantees about deadlocks, putting the onus of keeping the application deadlock free on the end user.

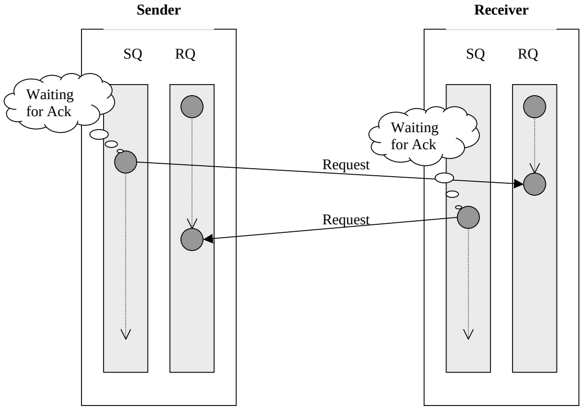

Figure 7: Deadlock in the rendezvous approach

Eager with Flow Control: This approach is similar to the rendezvous approach. The receiver initially posts a descriptor. When the sender wants to send a data message, it goes ahead and sends the message. However, for the next data message, it waits for an acknowledgment from the receiver confirming that another descriptor has been posted. Once this acknowledgment has been received, the sender can send the next message. On the receiver side, when a data message comes in, it uses up the pre-posted descriptor. Since this descriptor was posted without synchronization with the read() call in the application, the descriptor does not point to the user buffer address, but to some temporary memory location. Once the receiver calls the read() call, the data is copied into the user buffer, another descriptor is posted and an acknowledgment is sent back to the sender. This involves an extra copy on the receiver side. Figure 8 illustrates the eager approach with flow control.

Note that even this approach is not completely free from deadlocks. However, we have proposed an extension to this idea in section 6.1 which reduces the possibilities of one.

The first solution, using a separate communication thread, was not found to give any significant benefit in performance. However, the second and third approaches, namely the rendezvous and eager with flow control respectively, were found to give significant benefit in latency and bandwidth. Both these approaches have been implemented in the substrate, giving the user an option of choosing either one of them.

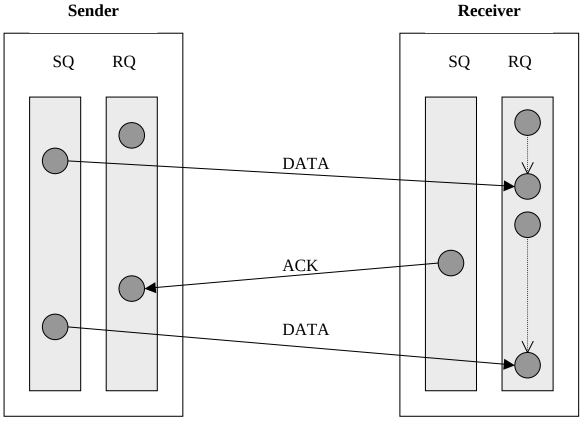

Figure 8: Eager with Flow Control

5.3 Resource management

Applications using sockets require closing of sockets after they have been used to avoid exhaustion of resources. Similarly, application built on EMP also require to clean up the unused resources. However, this is not taken care by the applications. So, the onus of the resource management falls on the substrate.

To deal with this problem, the substrate maintains a static table containing information about all the active sockets. We define a socket active if it is involved in communication, but a socket being used to listen on a port is not considered an active socket. On closing a socket, the substrate cleans up all the associated descriptors and sends back a closed message to the connected node. This takes care of cleaning up the descriptors local to each connection.

For connection management, global descriptors are used, which are not associated to any particular connection. Each new application resets the EMP state, thus there is no need to explicitly close these on termination.

5.4 Overloading function name-space

UNIX sockets provide an interface similar to TCP sockets abstracting the difference between a local file and a remote file from the user. In the substrate, these calls were mapped to the corresponding EMP calls (sets of calls). This mapping can be done in a number of ways.

Function Overriding: In this approach, the TCP function calls are directly mapped to the corresponding EMP function calls by overriding them. This approach works for calls such as listen(), which have just one interpretation. But, for calls such as read() and write(), this approach does not work, as the read can be on a socket or on a file. Overriding cannot distinguish between these two interpretations.

Application changes: In this approach, minor changes are made to the application by adding a parameter which allows the substrate to distinguish between a call to the EMP library and one to the libc library. This approach gives the flexibility of using both sockets over EMP as well as over TCP. However, since the aim of the substrate was to avoid any changes to the application, this approach was not used.

File descriptor tracking: In the approach used in our substrate, no changes are made to the application. We cause our functions to be loaded into the application before the standard C library, and monitor library calls which change the state of file descriptors, including open(), close() and socket(). In this way, on a read() or write(), for instance, our functions can decide whether to call into the EMP substrate or to pass the parameters on to the standard system function of the same name.

6 Techniques for Performance Enhancement

While implementing the substrate, the functionality of the calls was taken into account so that the application does not have to suffer due to the changes. However, these adjustments do affect the performance the substrate is able to deliver. In order to improve the performance given by the substrate, we have come up with some new techniques, which are described below.

6.1 Credit-based flow control

As mentioned earlier, the scheme we have chosen for handling unexpected messages is not free from the possibility of a deadlock. However, an extension of the idea is possible to reduce the possibility of its occurrence. The sender is given a certain number of credits (tokens). It loses a token for every message sent and gains a token for every acknowledgment received. If the sender is given N credits, the substrate has to make sure that there are enough descriptors and buffers pre-posted for N unexpected message arrivals on the receiver side. In this way, the substrate can tolerate up to N outstanding write() calls before the corresponding read() for the first write() is called, without the occurrence of a deadlock (Figure 9).

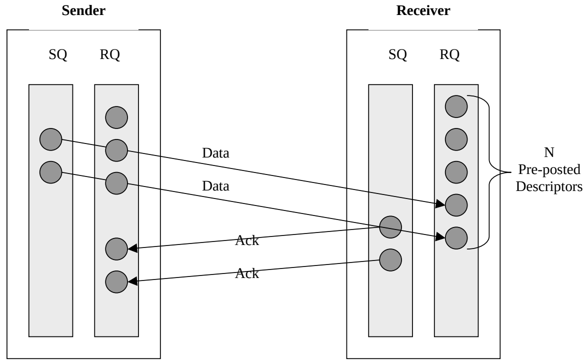

Figure 9: The Credit Based Approach

One problem with applying this algorithm directly is that the acknowledgment messages also use up a descriptor and there is no way the receiver would know when it is reposted, unless the sender sends back another acknowledgment, thus forming a cycle. To avoid this problem, we have proposed the following solutions:

Blocking the send: In this approach, the write() call is blocked until an acknowledgment is received from the receiver, which would increase the time taken for a send to a round-trip latency. Further, this scheme is unable to exploit the benefits obtained with respect to avoiding deadlock occurrences.

Piggy-back acknowledgment: In this approach, the acknowledgment is sent along with the next data message from the receiver node to the sender node. This approach again requires synchronization between both the nodes. Though this approach is used in the substrate when a message is available to be sent, we cannot always rely on this approach and need an explicit acknowledgment mechanism too.

Post more descriptors: In this approach, $2N$ number of descriptors are posted where N is the number of credits given. In this approach, it can be proved that at any point of time, the number of unattended data and acknowledgment messages will not exceed $2N$. On the basis of the same, this approach was used in the substrate.

A minor problem with this approach is the number of buffers registered. Since the arrival of a data message or the acknowledgment cannot be predicted, a buffer has to be allocated corresponding to each of the descriptors, effectively wasting half the buffer space. In our substrate, this limitation was overcome by using the NIC based tag-matching feature supported by EMP. Each descriptor can be associated with a tag, and the incoming message is matched with the tags to find the appropriate descriptor. We used this feature in our implementation to avoid this problem.

6.2 Disabling Data Streaming (Datagram)

As mentioned earlier, TCP supports the data streaming option, which allows the user to read any number of bytes from the socket at any time (assuming that at-least so many bytes have been sent). To support this option, we use a temporary buffer to contain the message as soon as it arrives and copy it into the user buffer as and when the read() call is called. Thus, there would be an additional memory copy in this case.

However, there are a number of applications which do not need this option. To improve the performance of these applications, we have provided an option in the substrate which allows the user to disable this option. In this case, we can avoid the memory copy for larger message sizes by switching to the rendezvous approach to synchronize with the receiver and DMA the message directly to the user buffer space. In this approach, the responsibility to avoid a deadlock lies on the user.

6.3 Delayed Acknowledgments

To improve performance, we delay the acknowledgments so that an acknowledgment message is sent only after half the credits have been used up, rather than after every message (Figure 10). This reduces the overhead per byte transmitted and improves the overall throughput.

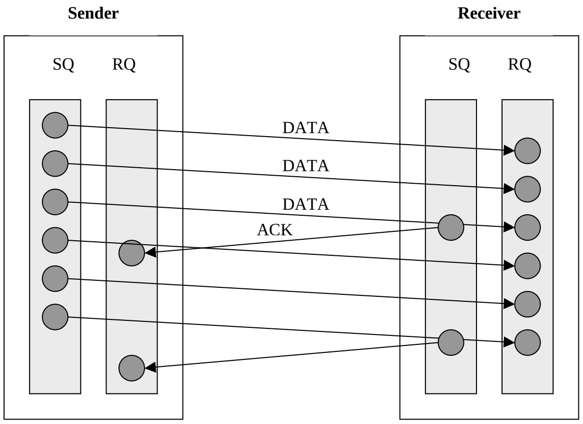

Figure 10: After Delayed Acknowledgments

These delayed acknowledgments bring about an improvement in the latency too. The reason for this is, when the number of credits given is small, half of the total descriptors posted are acknowledgment descriptors. So, when the message arrives, the tag matching at the NIC takes extra time to walk the list that includes all the acknowledgment descriptors. This time was calculated to be about 550 ns per descriptor. However, with the increase in the number of credits given, the fraction of acknowledgment descriptors decreases, and thus reducing the effect of the time required for tag matching.

6.4 EMP Unexpected Queue

EMP supports a facility for unexpected messages. The user can post a certain number of unexpected queue descriptors, and when the message comes in, if a descriptor is not posted, the message is put in the unexpected queue and when the actual receive descriptor is posted, the data is copied from this temporary memory location to the user buffer. The advantage of this unexpected queue is that the descriptors posted in this queue are the last to be checked during tag matching, which means that access to the more time-critical pre-posted descriptors is faster. The only disadvantage with this queue is the additional memory copy which occurs from the temporary buffer to the user buffer.

In our substrate, we use this unexpected queue to accommodate the acknowledgment buffers. The memory copy cost is not a concern, since the acknowledgment messages do not carry data payload. Further, there is the additional advantage of removing the acknowledgment messages from the critical path.

These enhancements have been incorporated in the substrate and are found to give a significant improvement in the performance.

7 Experimental Test-Bed and Performance Results

For the purpose of this paper, the experimental test-bed used included 4 Pentium III 700MHz Quads, each with a Cache Size of 1MB and 1GB main memory. The interconnect was a Gigabit Ethernet network with Alteon NICs on each machine connected using a Packet Engine switch. The linux kernel version used was 2.4.18.

7.1 Evaluation of Different Alternatives

This section gives the performance evaluation of the basic substrate without any performance enhancement and shows the advantage obtained incrementally with each performance enhancement technique.

In Figure 11 the basic performance given by the substrate for data streaming sockets is labeled as DS and that for datagram sockets is labeled as DG. DS_DA refers to the performance obtained by incorporating Delayed Acknowledgments as mentioned in Section 6.3. DS_DA_UQ refers to the performance obtained with both the Delayed Acknowledgments and the Unexpected Queue option turned on (Section 6.4). For this experiment, for the Data Streaming case, we have chosen a credit size of 32 with each temporary buffer of size 64KB. With all the options turned on, the substrate performs very close to raw EMP. The Datagram option performs the closest to EMP with a latency of 28.5 μs (an overhead of as low as 1 μs over EMP) for 4-bytes messages. The Data Streaming option with all enhancements turned on, is able to achieve up to 37 μs.

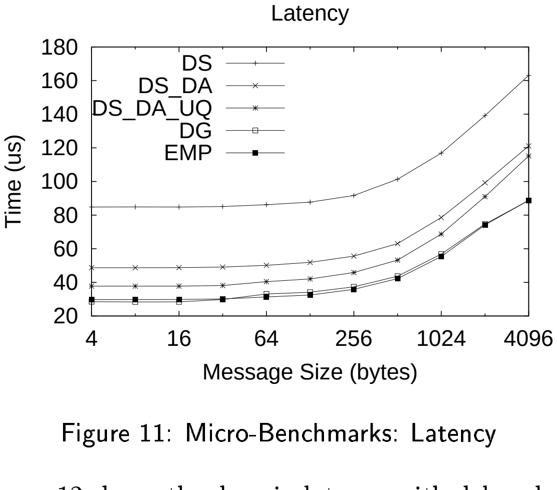

Figure 11: Micro-Benchmarks: Latency

Figure 12 shows the drop in latency with delayed acknowledgment messages. The reason for this is the decrease in the amount of tag matching that needs to be done at the NIC with the reduced number of acknowledgment descriptors. For a credit size of 1, the percentage of acknowledgment descriptors would be 50%, which leads to an additional tag matching for every data descriptor. However, for a credit size of 32, the percentage of acknowledgment descriptors would be 6.25%, thus reducing the tag matching time.

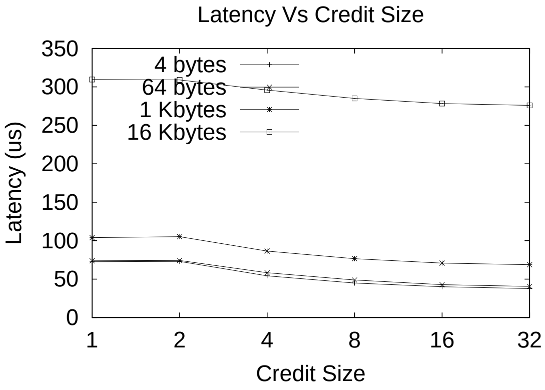

Figure 12: Micro-Benchmarks: Latency variation for Delayed Acknowledgments with Credit Size

The bandwidth results have been found to stay in the same range with each performance evaluation technique.

7.2 Latency and Bandwidth Comparisons

Figure 13 shows the latency and the bandwidth achieved by the substrate compared to TCP. The Data Streaming label corresponds to DS_DA_UQ (Data Streaming sockets with all performance enhancements turned on).

Again, for the data streaming case, a credit size of 32 has been chosen with each temporary buffer of size 64 Kbytes. In default, TCP allocates 16 Kbytes of kernel space for the NIC to use for communication activity. With this amount of kernel space, TCP has been found to give a bandwidth of about 340 Mbps. However, since the modern systems allow much higher memory registration, we changed the kernel space allocated by TCP for the NIC to use. With increasing buffer size in the kernel, TCP is able to achieve a bandwidth of about 550 Mbps (after which increasing the kernel space allocated does not make any difference). Further, this change in the amount of kernel buffer allocated does not effect the

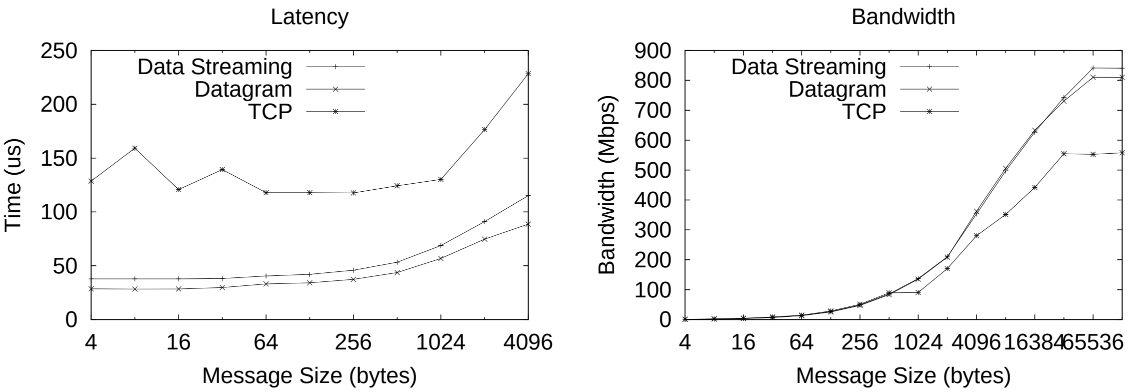

Figure 13: Micro-Benchmarks Results

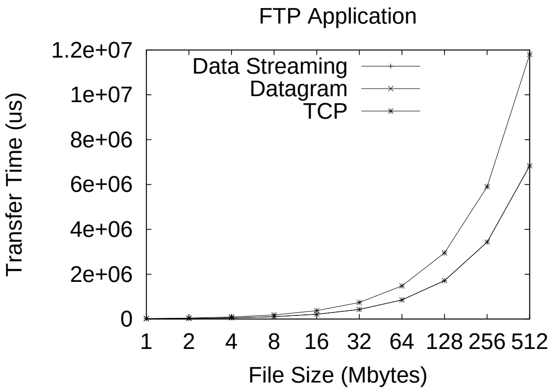

Figure 14: FTP Performance

latency results obtained by TCP to a great extent.

The substrate is found to give a latency as low as 28.5 μs for Datagram sockets and 37 μs for Data Streaming sockets achieving a performance improvement of 4.2 and 3.4 respectively, compared to TCP. The peak bandwidth achieved was above 840Mbps.

7.3 FTP Application

We have measured the performance of the standard File Transfer Protocol (ftp) given by TCP on Gigabit Ethernet and our substrate. To remove the effects of disk access and caching, we have RAM disks for this experiment.

Figure 14 gives the performance achieved by our substrate compared to TCP over Gigabit Ethernet for the FTP application.

The performance has been evaluated for both data streaming sockets and datagram sockets and both have been found to perform significantly better than TCP over Gigabit Ethernet. The application is not able to achieve the peak bandwidth illustrated in section 7.2, due to the File System overhead.

There is a minor variation in the bandwidth achieved by the data streaming and the datagram options in the standard bandwidth test. The overlapping of the performance achieved by both the options in ftp application, is also attributed to the file system overhead.

Note that this application requires both a socket read as well as a file read, thus requiring the substrate to be compatible with UNIX sockets. We have attained it by overloading the function name-space as mentioned in section 5.4

7.4 Web Server Application

We have measured the performance obtained by the Web Server application for a 4 node cluster (with one server and three clients). The experiment was designed in the following manner – The server keeps accepting requests from the clients. The clients connect to the server and send in a request message (which can typically be considered a file name) of size 16 bytes. The server accepts the connection and sends back a message of size S bytes to the client. We have shown results for S varying from 4 bytes to 8 Kbytes. Once the message is sent, the connection is closed (as per HTTP/1.0 specifications). However, this was slightly modified in the HTTP/1.1 specifications, which we also discuss in this section.

A number of things have to be noted about this application. First, the latency and the connection time results obtained by the substrate in the micro-benchmarks play a dominant role in this application. For connection management, we use a data message exchange scheme as mentioned earlier. This gives an inherent benefit to the Sockets-over-EMP scheme since the time for the actual request is hidden, as the connection message descriptors are pre-posted.

However, the substrate also has a disadvantage due to the pre-posting and cleaning up of the descriptors. In this application, per connection, only one descriptor would be used, since it is known that each client sends in just one request message. With a credit size of 32, a lot of time would be wasted in the posting and garbage collection of all the descriptors (which are eventually left unused). In this experiment, we have used a credit size of 4.

Figure 15 gives the results obtained by the Web Server application following the HTTP/1.0 specifications.

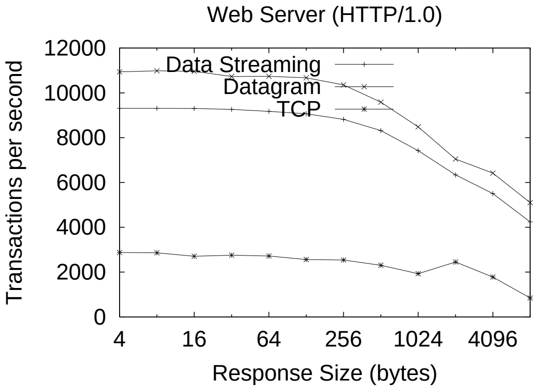

Figure 15: Web Server Average Response Time (HTTP/1.0)

In the substrate, once the "connection request" message is sent by the substrate, the application can start sending the data messages. This reduces the connection time of the substrate to the time required by a message exchange. However, in TCP, the connection time requires intervention by the kernel and is typically about 200 to 250 μs. To cover this disadvantage TCP has, the HTTP 1.1 specifications allow a node to make up to 8 requests on one connection. Even with this specification, our substrate was found to perform better than the base TCP application (Figure 16). In the worst case, if the web server allows infinite requests on a single connection, the web server application boils down to a simple latency test which has been plotted in Section 7.1.

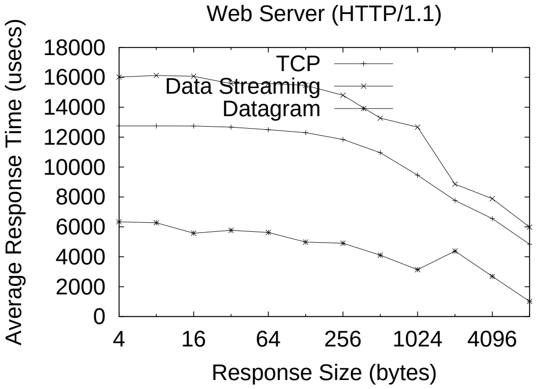

Figure 16: Web Server Average Response Time (HTTP/1.1)

7.5 Matrix Multiplication

We have measured the performance obtained by the matrix multiplication application (written using sockets) for a 4 node cluster.

When the connection requests come in, the application has to know the socket that is connected to any given node. To handle this, we used the select() operation in TCP. Figure 17 shows the performance achieved by the substrate compared to TCP over Gigabit Ethernet for the Matrix Multiplication application.

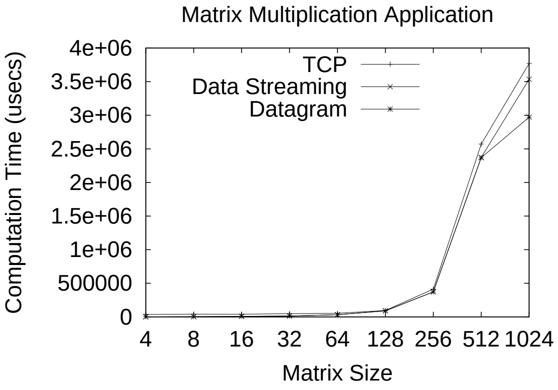

Figure 17: Matrix Multiplication Performance

8 Conclusions and Future Work

Ethernet forms a major portion of the world's networks. Applications written using the sockets library have not been able to take advantage of the high performance provided by Gigabit Ethernet due to the traditional implementation of sockets on kernel based protocols.

In this paper, we have designed and developed a low-overhead substrate to support socket based applications on EMP. For short messages, this substrate delivers a latency of 28.5 μs for Datagram sockets and 37 μs for Data Streaming sockets compared to a latency of 28 μs achieved by raw EMP. Compared to basic TCP, latency obtained by this substrate shows performance improvement up to 4 times. A peak bandwidth of over 840 Mbps is obtained by this substrate, compared to 550 Mbps achieved by basic TCP, a performance improvement by a percentage of up to 53%. For the ftp and Web server applications, compared to the basic TCP implementation, the new substrate shows performance improvement by a factor of 2 and 6, respectively. These results demonstrate that applications written using TCP can be directly run on Gigabit Ethernet-connected cluster with this substrate.

We are currently working on utilizing and evaluating the proposed substrate for a range of commercial applications in the Data center environment. We also plan to develop similar substrate for the emerging InfiniBand interconnect so that a range of applications should be able to take advantage of the low-latency and high-bandwidth associated with interconnects for the next generation clusters.

9 Acknowledgments

We would like to thank all the students working at the Network of Workstations Laboratory at The Ohio State University, for all the valuable suggestions they had given through the course of the project. Without their help, this paper would not have been possible.

References

[1] Infiniband Trade Association. http://www.infinibandta.org.

[2] M. Banikazemi, B. Abali, L. Herger, and D. K. Panda. Design Alternatives for VIA and an Implementation on IBM Netfinity NT Cluster. In *Special Issue of Journal of Parallel and Distributed Computing and Network Based Computing.*

[3] M. Banikazemi, J. Liu, D. K. Panda, and P. Sadayappan. Implementing TreadMarks over VIA on Myrinet and Gigabit Ethernet: Challenges, Design Experience, and Performance Evaluation. In *the Proceedings of International Conference on Parallel Processing '01*, September 2001.

[4] M. Banikazemi, V. Moorthy, L. Hereger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP switch-connected NT clusters. In *the Proceedings of the International Parallel and Distributed Processing Symposium, pages 33-42*, May 2000.

[5] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro, pages 21-28*, February 1995.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network.

[7] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *the Proceedings of Supercomputing '98*, 1998.

[8] Myricom Corporations. The GM Message Passing Systems.

[9] MPI Forum. MPI: A Message Passing Interface. In *the Proceedings of Supercomputing '93*, November 1993.

[10] H. Frazier and H. Johnson. Gigabit Ethernet: From 100 to 1000Mbps.

[11] Giganet Corporations. http://www.giganet.com.

[12] GigaNet Incorporations. cLAN for Linux: Software Users' Guide. 2001.

[13] Jin-Soo Kim, Kangho Kim, and Sung-In Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *the Proceedings of Cluster '01*, October 2001.

[14] M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/ research/FTG/via.

[15] Netgear Incorporations. http://www.netgear.com.

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

[17] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *the Proceedings of Hot Interconnects '01*, August 2001.

[18] L. Prylli and B. Tourancheau. BIP: A New Protocol designed for High Performance Networking on Myrinet. In *the Proceedings of the International Parallel Processing Symposium Workshop on Personal Computer Based Network of Workstations*, 1998.

[19] M. Rangarajan and L. Iftode. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance.

[20] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and Experience with LAPI: A New High Performance Communication Library for the IBM RS/6000 SP. In *the Proceedings of the International Parallel Processing Symposium '98*, March 1998.

[21] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *the Proceedings of the CANPC workshop (held in conjunction with HPCA Conference), pages 91-107*, 1999.

[22] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *the Proceedings of Supercomputing '01*, November 2001.

[23] P. Shivam, P. Wyckoff, and D. Panda. Can User Level Protocols Take Advantage of Multi-CPU NICs? In *the Proceedings of International Parallel and Distributed Processing Symposium '02*, April 2002.

[24] W. Richard Stevens. UNIX Network Programming.

[25] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for Parallel and Distributed Computing. In *the Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[26] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture, pages 256-266*, 1992.

[27] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. In *the Proceedings of Hot Interconnects V*, August 1997.